COMPUTER ASSISTED ANALYSIS OF MULTIPROCESSOR MEMORY SYSTEMS

SeungJoon Park

Technical Report: CSL-TR-96-696
June 1996

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305-4055
e-mail: pubs@shasta.stanford.edu

Abstract

Parallel architecture becomes more and more attractive as the demand for performance increases. One of the most important classes of parallel machines is that of shared memory architectures, which are perceived as easier to program than other parallel architectures. In a shared memory multiprocessor architecture, a *memory model* describes the behavior of the memory system as observed at the user-level. A *cache coherence protocol* aims to conform to a memory model by maintaining consistency among the multiple copies of cached data and the data in main memory. Memory models and cache coherence protocols can be quite complex and subtle, creating a real possibility of misunderstandings and actual design errors. In this thesis, we will present solutions to the problems of specifying memory models and verifying the correctness of cache coherence protocols.

Weaker memory models for multiprocessor systems allow higher-performance implementation techniques for memory systems. However, weak memory models are also very subtle. Hence, it is vital to specify memory models precisely and to verify that the programs running under a memory model satisfy desired properties. Our approach to these problems is to write an executable specification of the memory model using a high-level description language for concurrent systems. This executable description provides a precise specification of the machine architecture for implementors and programmers. Moreover, the availability of automatic verification tools allows users to experiment with the effects of the memory model on small assembly-language routines. Running the verifier can be very effective at clarifying the subtle details of the models and synchronization routines.

Cache coherence protocols, like other protocols for distributed systems, simulate atomic transactions in environments where atomic implementations are impossible. Based on this observation, we propose a verification method which compares a state graph of the implementation with a specification which is also a state graph representing the desired abstract behavior. The comparison is done through an *aggregation function*, which maps the sequence of implementation steps for each transaction to the corresponding transaction step in the specification. An aggregation function supplied by the user is formally proved in full detail to have certain properties using a computer-assisted theorem prover.

The aggregation approach is applied to verification of the cache coherence protocol in the FLASH multiprocessor system. The protocol, consisting of more than a hundred implementation steps, is proved to conform to a reduced description with six kinds of atomic transactions. From the reduced behavior, it is very easy to prove crucial properties of the protocol, including data consistency of cached copies at the user level. The aggregation method is also used to prove that the reduced protocol satisfies a desired memory consistency model.

Key Words and Phrease:
Multiprocessors, Memory models, Cache coherence protocols, Specification, Verification.

Acknowledgments

This work would not have been possible without the help of many people. First among these is my research advisor Professor David Dill. It has been a pleasure and privilege to learn from him how to conduct research and present it. I started to study multiprocessor memory systems at his suggestion. Throughout my years at Stanford, he has always been a constant source of technical feedback and encouragement. I remember working with David on writing papers at some nights, roaming in the dark Stanford Shopping Center to find food for late dinners. I also cannot forget the hearty meal at his home for Thanksgiving with chicken simulating turkey when I was the only student who could not visit home during the holidays.

I would like to thank the other members of my committee, my associate advisor Professor Giovanni De Micheli and Professor Gene Franklin, for their careful reading and helpful comments on my thesis. I am also grateful to Professor Jim Harris and Dr. John Rushby who kindly served on my orals committee.

Working with the fellows in our research group was a joy. Former students, Kenneth Yun and Alan Hu; previous and current officemates, Han Yang, Norris Ip, and Supratik Chakraborty; and other colleagues, Jeffrey Su, Robert Jones, Clark Barrett, and Ulrich Stern; I thank them for friendship and technical assistance, especially for their help on my orals. I also thank Ganesh Gopalakrishnan, a visiting Professor, for his interest and comments on my thesis.

I benefited from many discussions and suggestions from the SPARC design team. Most of the detailed definition of the SPARC memory models was done by a group consisting of Dennis Allison, David Dill, Kourosh Gharachorloo, Paul Loewenstein, Andreas Nowatzyk, and me. Feedback from these individuals was essential in developing the executable memory model in Chapter 2.

I was fortunate to participate in the FLASH multiprocessor project. Mark Heinrich clarified many details in the FLASH protocol which is used in Chapter 4.

I owe a special acknowledgment to the researchers at SRI International: John Rushby, Natarajan Shankar, Sam Owre, Mandayam Srivas, Sree Rajan, and David Cyrluk, who made my days there enjoyable when I was working as a visiting fellow. In particular, I thank Sam Owre for answering numerous questions about their verifier.

Financial support from the following agencies is gratefully acknowledged: the Advanced Research Projects Agency (through NASA grant NAG-2-891), the National Science Foundation (grant MIP-8858807). I am also grateful for my Korean Government Scholarship for Overseas Study.

I would also like to thank Korean friends and senior students who helped me to settle down at Stanford, and professors who taught me during my undergraduate and master years at Seoul National University.

Many thanks are due to my wife Yookyung, who always makes me smile with her amiable nature, for her support while she was also busy finishing her dissertation. Finally, my eternal gratitude goes to my parents for their endless love, care, and belief in me.

SeungJoon Park
May 1996

Contents

Chapter 1

Introduction

1.1 Motivation

Parallel architectures are becoming more and more attractive as the demand for higher performance increases. Many multiprocessors are currently being designed to meet this demand. The scale of parallelism is increasing rapidly, and the use of parallelism is widening as technological improvements reduce costs [57, 29].

For highly parallel architectures to achieve widespread use, they must run a variety of applications efficiently without imposing excessive programming difficulty. From this view, one of the most important classes of parallel architectures is *shared memory architectures*. Shared memory (also called shared address-space) architectures are very attractive for application programmers, because they are perceived as easier to program than other parallel architectures, at least for some applications. In fact, the majority of parallel machines that are sold today are based on shared memory.

Complex systems, particularly those involving parallelism, are difficult to design. One of the major problems is how to avoid design errors resulting from unexpected interactions among system components. Validating the correctness of a design before implementation is essential because it is difficult and expensive to correct errors after a machine is built and the hardware is committed [61, 8, 68, 28].

It is widely believed by designers that system design problems are outstripping current design debugging techniques. Random testing and trace-driven simulations are not sufficient for validation because coverage declines as design complexity increases, so bugs remain undetected by simulations. Moreover, it is not unusual to find bugs even in algorithms and protocols that have been proved correct by hand, because of errors in the proofs.

The general objective of this research is to develop improved methods for debugging and assuring the correctness of high-level multiprocessor designs. We have focussed on large-scale shared-memory multiprocessors as the domain, because there is currently a great deal of interest in designing them [39, 3, 52, 65, 32]. To expedite the validation process and to reveal possible mistakes in human reasoning, *computer assistance* is necessary.

In a shared memory multiprocessor architecture, a *memory model* describes the *behavior* of the memory system observed at the user level. A memory model specifies the semantics of memory operations when multiple processors load and store shared memory locations. The model also provides a programmer-level view of memory transactions ordering. Given a multiprocessor program, a memory model provides sufficient information to determine the set of possible results of the program.

Unfortunately, defining and reasoning about memory models can be very difficult. In this dissertation, we describe methods for defining executable specifications of memory models in a description language that is suitable for verification. The automatic verifier can be used to enumerate all the outcomes of an example program, or to check the correctness of simple synchronization routines.

Most shared memory multiprocessors implement caches which keep multiple copies of data for a given memory location, to provide an illusion of a single shared memory while providing rapid access to data from multiple processors. The use of caches exploits the temporal and spatial locality of memory accesses by the multiprocessors, improving the performance of the memory system. However, dealing with multiple copies of a datum raises a consistency problem: a cached copy of a memory location may not be consistent with other cached copies of the same memory location.

In such implementations, a *cache coherence protocol* aims to maintain consistency among the multiple copies of cached data and the data in main memory. A cache coherence protocol is a lower-level abstraction of a memory system which should conform to the memory model which is a higher-level abstraction of the memory system.

We also present methods for formally verifying cache coherence protocols. One of the methods is to use a finite-state verifier. A more general (but more difficult) approach is to use a general-purpose computer-assisted theorem-prover. We present a new way to use theorem-proving to verify cache coherence protocols, which is based on aggregating implementation steps into high-level transactions.

1.2 Background

This thesis aims to propose better approaches to reasoning about memory models and cache coherence protocols for shared memory architectures, which are two different levels of abstraction of multiprocessor memory systems. This section introduces background for the research.

1.2.1 Memory consistency models

In a shared memory multiprocessor architecture, a memory model is a user-level description of the *behavior* of the memory system. A memory model specifies the semantics of memory operations when multiple processors load and store shared memory locations. In other words, the memory model should be a precise specification of how memory behaves with respect to read and write operations from multiple processors.

Several memory models for shared memory multiprocessor architectures have been proposed. An early model, *sequential consistency* [42], simply requires that multiprocessors simulate atomic reads and writes to a common global memory observing the sequential order defined in a program. This model is relatively easy to understand but has strong constraints which hinder high performance implementations.

Unfortunately, sequential consistency constrains the range of behaviors of the memory system to such a degree that it cannot be implemented efficiently in hardware. Consequently, many less constraining memory models have been proposed, which make fewer guarantees about behavior (we call these *weaker* memory models).

This allows more concurrency in memory system and processor implementations, resulting in improved performance. During the past decade, a lot of effort has been made to design weaker memory models, such as *processor consistency* [27], *weak consistency* [20], *total store ordering*, *partial store ordering* [35], *release consistency* [25], and *relaxed memory order* [66].

A memory model provides sufficient information to determine the results of a program running under the memory system. In other words, given a program, we should be able to know what results are possible and what are impossible from the specified memory model without worrying about its detailed implementation.

To illustrate why it is important to understand memory models, let us consider an example. The following parallel program demonstrates how memory models affect program behavior.

Processor P_0		Processor P_1	
store	$\#1 \rightarrow A$	store	$\#1 \rightarrow B$
load	$B \rightarrow \%r_0$	load	$A \rightarrow \%r_1$

Processor P_0 stores constant value 1 to memory address A, then it loads memory address B to its register $\%r_0$. Processor P_1 does similar instructions. We assume all the registers and memory locations initially contain zero values.

Suppose the program is running on multiprocessors with a memory system which implements the sequential consistency memory model. In this case, either $\%r_0$ or $\%r_1$ should obtain value 1 after the program is executed, because at least one of the two stores must be performed before both of the loads.

However, this result may not be guaranteed if the memory system is based on a weaker memory model which allows more freedom in executing memory instructions. One of the widely-used techniques in implementing efficient memory systems for single processor machines is to use write buffers to delay store transactions while subsequent loads are performed. This technique does not make a visible difference to the user running on a single processor machine. However, the same technique produces different program results when applied to a multiprocessor memory system. For example, suppose each processor issues the corresponding store into its own write buffer. Before any of the stores are performed by the main memory, P_0 loads the memory location B reading the initial value 0 from the main memory, and P_1 also loads the memory location A reading the initial value 0, also from the main memory. Then, the two stores are performed to the main memory. In this case, the result of the program is $\%r_0 = \%r_1 = 0$, which was not allowed by the sequential consistency model.

As we observed from the example, different memory models produce different program results. With simpler memory models, it is relatively easy to program and to reason about results of programs, because their strong requirements allow less diverse executions. However, the strong constraints hinder high performance implementations.

Using weaker memory models allows many efficient implementation techniques in hardware design by exploiting more parallelism. However, weaker memory models are generally very subtle, because understanding the behavior of highly concurrent systems is never easy. Thus, there is a tension between the simplicity of memory models and performance of memory systems.

1.2.2 Cache coherence protocols

In order to improve performance of memory systems, most of multiprocessor architectures use distributed caches for each processor, which keep local copies of main memory. Because data can be found either in memory or in the multiple caches, coherence problems arise when more than one processor's cache holds a copy of a datum at a shared address [57]. Cache coherence is one of the key aspects that is different in the design of memory systems of multiprocessors from that of uniprocessors.

For instance, to build a memory system supporting the sequential consistency, we wish to ensure that when reading a memory location the processor always sees the latest value written to that location. This is simply achieved in uniprocessors, because normally it is only the processor that is reading and writing memory. However, it is not trivial to obtain coherent caches in multiprocessor systems.

Suppose the multiprocessors execute the following memory accesses in order.

1. Processor P_1 loads a memory location A into its cache C_1.

2. Processor P_2 loads a memory location A into its cache C_2.

3. Processor P_2 stores the memory location A by writing a new value into C_2.

When P_2 writes a new value into C_2, the cached value in C_1 and the datum in main memory at location A should be invalidated or updated. Otherwise, future reads to the memory location by a third processor may load the old value in the main memory; and future loads by P_1 may get the old value in C_1.

In shared memory architectures, *cache coherence protocols* maintain the consistency among multiple cached data and data in main memory. The protocols control a number of readable and writable copies of each memory location for multiprocessors in distributed caches. Modification of one copy of a datum may require updating other copies to maintain consistency among them.

1.3 Problems and Related Work

This section presents some of the problems in analyzing multiprocessor memory systems and related work by others.

1.3.1 Specifying and analyzing memory models

Weaker memory models for multiprocessors allow higher-performance implementation techniques for memory systems. However, their behavior may be sometimes counter-intuitive. Therefore, it is vital to specify a memory model precisely.

The precise details of the memory model are crucial to several parties. Obviously, programmers must be aware of the model to write correct and efficient shared memory programs; for example, the model affects the correctness of synchronization routines. Multiprocessor designers should understand the model because the design of the cache coherence scheme must respect the model. Also, processor designers must ensure that processor optimization techniques such as out-of-order issue of memory instructions and register renaming do not violate the model model. Compiler writers may also have to consider the memory model in some optimizations.

Memory model descriptions in English can be ambiguous. Unfortunately, precisely defining the semantics of a memory model often leads to complex specifications that are difficult to understand for typical users such as programmers and hardware builders of computer systems.

Many formal specifications of memory models are written in mathematics. Collier [13] specifies memory models using partial orders and infers the behavior of programs from a set of ordering relations. Gharachorloo [24] and Sindhu and Frailong [62] have used methods similar to Collier's. Another way to specify memory models is using automata. Gibbons et al. [26] use I/O automata, and Hojati et al. [33] use L-automata to specify and reason about memory models.

However, such specifications require that the users infer the behavior of programs from a set of ordering constraints or by reasoning about automata. For those not familiar with such formal notation, it is hard to reason about programs running on multiprocessors by manual computation—and almost impossible for large programs even for experts.

1.3.2 Verifying cache coherence protocols

Cache coherence protocols can be quite complex and subtle, creating a real possibility of design errors, especially for those used in large-scale multiprocessor systems. Formal verification is desirable because the bugs can be quite subtle and hard to capture by simulation. Several coherence protocols have been proposed but few are formally verified [4, 67, 9, 47].

One of the effective ways to validate protocols is using finite-state methods (model checking). Finite-state methods enumerate the states of the reachable state graph of the system, searching for states that violate a specified property (e.g. Murφ [15], SMV [21], SPIN [34], COSPAN [38]).

These methods suffer from the state explosion problem: the number of states for nontrivial numbers of processors and cache lines is very large. For example, the Murφ verifier can barely handle a relatively simple protocol with 3 processors and 2 memory lines using 100 megabytes of memory in the process.

Another problem with the model checkers is that it is very difficult to specify correctness conditions of the protocol using notations such as Murφ or temporal logic. The specification is the corresponding memory model of the protocol so it is required to encode a full memory model in temporal logic.

Symbolic state models proposed by Pong and Dubois [60, 59] reduce the state explosion problem by using symbolic states which abstract away from the exact number of configurations of replicated identical components by recording only whether there are zero, one, or more than zero replicated components. However, there still remains a specification problem of the protocol as in model checking: It is not easy to find a set of properties in their notation, which completely describes the correct behavior of the protocols. Moreover, their method requires the user to write an abstract description of the protocol to be verified, which raises another verification problem that the abstract description and the actual protocol are equivalent.

Another approach to formal verification is computer-assisted theorem-proving. Theorem-provers make available the full power of formal mathematics for proof, so they can routinely deal with problems that cannot yet be solved by any finite-state methods.

However, the major problem with theorem proving is that considerable labor is required. Consequently, previous theorem proving approaches have not been able to verify a problem of the scale of a full multiprocessor cache coherence protocol. The most significant result to date is a manual proof of "lazy caching," a simple and abstract cache coherence algorithm [2, 23, 40].

Overall, the finite-state method and theorem-proving have been applied to simple and small protocol models. However, we expect that the complexity of cache protocols will continue to increase as faster and larger relaxed memory models are implemented. Then verifying cache coherence protocols becomes a serious challenge which must be met with appropriate and efficient techniques.

1.4 Results and Contributions of the Thesis

This thesis will present solutions to the problems of specifying and analyzing memory models, and verifying the correctness of cache coherence protocols.

1.4.1 Executable description and automatic analysis of memory models

We present a new approach for specifying a memory model for multiprocessors. We describe the memory model by giving a maximally general executable description [54, 18], using a high-level description language for concurrent systems called Murφ [15]. Such a description provides a precise specification of the machine architecture, both for programmers and hardware implementors.

The major advantage of using Murφ is that it is also an *automatic verification system*. There is a tool that supports exhaustive checking of all the reachable states of a description for violations of user-specified properties. Using the verifier allows users to experiment with the effects of the memory model on programs being executed in the memory system. Running the verifier can be very effective at clarifying the subtle details of the models.

The Murφ verifier is used for several different kinds of automatic analysis. First of all, we can formally verify the synchronization routines for a memory model which are the most frequently used and important programs of the multiprocessor memory system. Murφ also allows the printing of the state of a system at user-specified points while exploring the reachable states. This feature has been used to list all of the possible register values that can occur when an example program terminates. Consequently, we can obtain a complete list of possible program results for a given multiprocessor program running in a specified memory model. If we obtain an unexpected output, the verifier can also be used to generate an execution producing a trace to the specific output. This feature helps programmers to understand the memory behavior and to correct their programs.

We developed an executable memory model during the process of defining the RMO (Relaxed Memory Order) memory model of the SPARC Architecture Version 9 [66]. The RMO is a generalization of the previous SPARC Version 8 memory models, TSO (Total Store Ordering) and PSO (Partial Store Ordering) [18].

Developing an executable model of RMO in Murφ greatly enhanced our understanding and confidence in the design for several reasons. First, writing a precise description pointed out ambiguities and inconsistencies, even without executing the description. Second, we were able to analyze the possible outcomes of illustrative examples and synchronization programs rapidly and automatically, when there was a question about the implications of a change in detail of the specification. Third, we could verify the examples in the SPARC Architecture Manual Version 9, which increased our confidence that there were no errors in the code examples associated with the memory models [54]. Overall, our approach has been extensively used in the design procedure, and it was very helpful for the SPARC-V9 design team.

The executable description should be a maximally general implementation of the axiomatic specification so that it can be used as an *equivalent specification*. We have done a high-level formal proof of the equivalence that the executable specification generates all the possible behavior that is allowed by the axiomatic specification, and that executions generated by the executable description are allowed by the axiomatic specification.

1.4.2 Verification method for cache coherence protocols

We present a new approach for using a computer-assisted theorem-prover to verify the correctness of protocols and distributed algorithms. The method aims to overcome the finiteness constraint of model checking and to exploit advantages of theorem-proving in verifying cache coherence protocols.

The method compares a state graph of the implementation with a specification which is also a state graph representing the desired abstract behavior. The steps in the specification correspond to atomic transactions, which are not atomic in the implementation.

The method relies on an *aggregation function*, which is a kind of an abstraction function that aggregates the steps of each transaction in the implementation into a single atomic transaction in the specification [55]. We present an easy and systematic way to find such an aggregation function. The method substantially reduces the amount of labor required, hence significantly extends capability of computer-assisted theorem-proving for cache coherence protocols.

Owing to the generality obtained by the use of logic as a formalism, we have been able to validate protocols with an arbitrary number of processors. Our method has successfully verified the safety properties of the FLASH cache coherence protocol [39, 30]. For several years, we believed that proving the correctness of protocols of the complexity of the FLASH cache coherence protocol was well beyond the capability of a general-purpose theorem prover. The aggregation method has broken through this barrier.

The method also solves the specification problem. The aggregation renders a reduced model of the implementation, which can serve as a specification, if none exist.

The proposed verification procedure is not only for cache coherence protocols but also has been applied to other protocols, which are simple but non-trivial: majority consensus algorithm for multiple copy databases [64, 37], and a distributed list protocol [17].

1.5 Overview of the Thesis

Chapter 2 presents techniques for writing executable memory models in a high-level description language for concurrent systems. The techniques are applied to the Relaxed Memory Order of SPARC Version 9 Architecture as well as the earlier, simpler models TSO and PSO. This chapter also demonstrates several ways to use an automatic verification tool for analyzing the memory model: verification of synchronization routines, generation of complete lists of possible program results, and generation of an execution trace for a specific program result.

Chapter 3 explains the aggregation method for verification of cache coherence protocols and similar distributed algorithms using a computer-assisted theorem prover. The method is illustrated on the examples, a distributed list protocol and a majority consensus algorithm for multiple copy databases.

Chapter 4 presents verification of the FLASH cache coherence protocol. The protocol is briefly described, and a finite-state method is used to verify some properties of the protocol. Next, the aggregation method presented in Chapter 3 is applied to verify the protocol with arbitrary numbers of processors.

Finally, Chapter 5 summarizes the thesis and discusses possibilities for future research.

Chapter 2

Reasoning About Memory Models

In this chapter, the Murφ description language and verification system for finite-state concurrent systems is applied to the problem of specifying a family of multiprocessor memory models in the SPARC Architecture Manual Version 9. This chapter describes the memory models and their encoding in the Murφ description language, and presents several techniques using its automatic verifier for analysis of programs running under the specified memory models.

The description language allows for a straightforward executable description of the memory model which can be used as a specification for programmers and machine architects. The automatic verifier can be used to generate all possible outcomes of small assembly-language multiprocessor programs in a specified memory model, which is very helpful for understanding the subtleties of the model. The verifier can also check the correctness of assembly language programs including synchronization routines.

Section 2.1 discusses general problems of specifying abstract memory models for multiprocessors. Section 2.2 explains the intuition behind memory models of SPARC-V9 architecture. Section 2.3 presents the logical specification of Relaxed Memory Order of SPARC-V9. Section 2.4 demonstrates how to write an executable specification using the description language. Section 2.5 presents techniques to verify synchronization routines and to analyze finite state programs using the automatic verifier. We also present some interesting findings from the verification and the analysis. Section 2.6 contains a formal proof that the executable specification is equivalent to the logical specification.

2.1 Executable Specification of Multiprocessor Memory Models

In a shared memory multiprocessor architecture, a *memory model* specifies the semantics of memory operations when multiple processors load and store shared memory locations. The precise details of this model are crucial to several parties: programmers, multiprocessor system designers, processor designers, compilers, and hardware implementors.

Several memory models for shared memory multiprocessor architecture have been proposed. Weaker memory models are attractive because they allow better more concurrency in memory system and processor implementations, resulting in improved performance. However, weaker memory models are generally very subtle. Even sequential consistency can be counter-intuitive at times. Hence, it is vital to specify a memory model precisely.

Our approach to these problems is to describe the memory model by giving a *maximally general* executable description [54], using a high-level description language for concurrent systems called Murφ [15]. Such a description provides a precise specification of the machine architecture, both for implementors and programmers. The major advantage of using Murφ is that it is also an *automatic formal verification system*. There is a tool that supports exhaustive checking of all the reachable states of a description for violations of user-specified properties. Murφ also allows the printing of the state of a system at user-specified points while exploring the reachable states; this feature can be used, for example, to list all of the possible register values that can occur when an example program terminates.

The approach here is different from that used by Collier [13], who infers the behavior of programs from a set of ordering relations, which are not necessarily easy to convert into an executable form. Gharachorloo [24] and Sindhu and Frailong [62] have used methods similar to Collier's. Our method more closely resembles that of Gibbons, et al. [26], who give I/O automata specifications of memory models. The primary differences here are the description languages, and more importantly, our emphasis on support for automatic analysis, while verification with I/O automata is generally by hand.

It is important to note that the executable description is a maximally general implementation [33] which could be regarded as a *formal specification*. In other words, all the execution traces generated by the executable model are legal under the logical specification *and* all the legal execution traces are generated by the executable model. To ascertain this, we present formal proofs, which have been confirmed by automatic theorem provers, that our executable specification in Murφ is equivalent to the logical specification.

We developed an executable memory model during the process of defining the Relaxed Memory Order (RMO) model of the SPARC Architecture Manual Version 9 [66]. RMO is a generalization of the previous SPARC Version 8 memory models, TSO (Total Store Ordering) and PSO (Partial Store Ordering). Intuitively, TSO liberalizes sequential ordering by allowing the performance of stores to be delayed relative to subsequent loads; PSO additionally allows stores to be delayed relative to other stores; and RMO further allows loads to be delayed relative to subsequent loads and stores. In [18], we developed an executable model for TSO and PSO; however, the executable model of RMO is not a simple generalization of the earlier description.

Developing an executable model of the protocol in Murφ greatly enhanced our understanding and confidence in the design for several reasons. First, writing a precise description points out ambiguities and inconsistencies, even if the description is not executed. Second, we were able to analyze the possible outcomes of illustrative examples and synchronization programs rapidly and automatically, when there was a question about the implication of a change in detail of the specification. Third, we could verify the examples in the architecture manual, which increased our confidence that there were no errors in the code examples associated with the memory models.

2.2 SPARC Memory Models

This section explains the intuition behind the SPARC-V9 memory models. It is not intended to be exhaustive or to be a tutorial on the models. For more information, see the SPARC Architecture Manual Version 9 [66].

Figure 2.1 illustrates the intuition behind the memory models. Note that this is a fictitious description that bears no relation to a reasonable *implementation* of a memory model—it is only intended to capture a programmer-level view of the possible behaviors of memory operations. There is a set of processors, $P_1, P_2, \ldots, P_n$, each of which has its own cache, and an *abstract reorder box*. Each processor executes instructions in the natural order specified in the program, called *program order*. Instructions may appear to occur in some order other than program order, due to various implementation techniques, such as local caching or out-of-order instruction execution in the processor implementation. This reordering is modeled in Figure 2.1 by attaching a *reorder box* to each processor.

Each reorder box is also connected to a common global *memory*. The memory arbitrarily chooses one of the reorder boxes, chooses an instruction from the reorder box subject to ordering constraints that are specified as part of the particular memory model. It then executes actions which depend on the instruction, such as updating memory locations or processor registers. The actions for each instruction are executed atomically—other actions in the system cannot interfere with them. An instruction is said to be *performed* when it is executed by the memory.

It bears repeating that this is not intended to resemble any implementation of a memory system. It is merely a fiction that explains the functional behaviors of programs—without regard to performance. Indeed, the "memory" is doing almost all of the work of executing instructions, including modifying processor registers and even doing arithmetic—in early models, the memory performed much more memory-like functions, which is why we use the term. The lack of a clear separation between processor and memory models is a little awkward, but it seems that ordering constraints from the processor inherently affect the memory model, especially in liberal models such as RMO, so we have found it necessary to generalize the memory in this

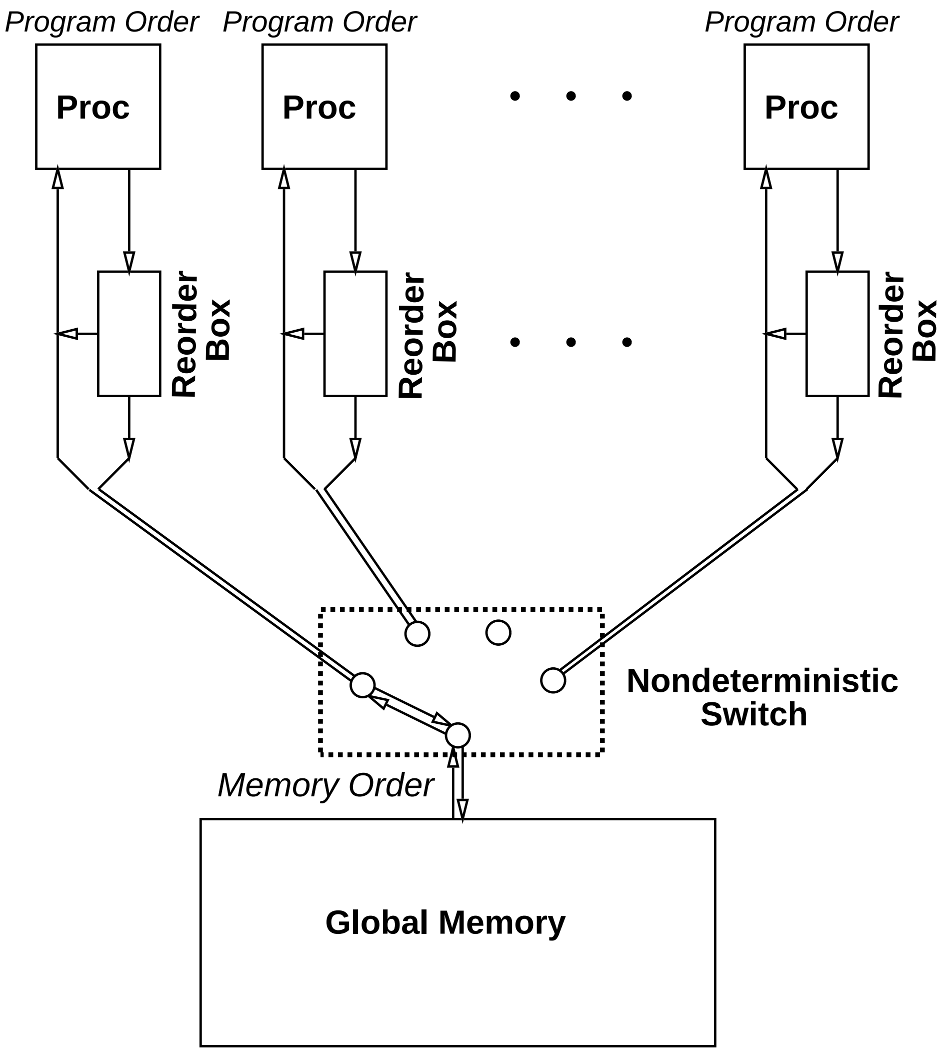

Figure 2.1: An abstract memory model for multiprocessors

way.

All SPARC implementations must support the standard memory model, called *Total Store Ordering* (TSO) which is also the most restrictive model. In TSO, store operations from a processor P_i must be performed by the memory in program order. However, a load issued by P_i may return a value recently stored by P_i before the store is performed by the global memory. Hence, a processor P_j may load an old value *after* P_i has loaded a new value. This last scenario would not be allowed under sequential consistency; TSO is weaker than sequential consistency. Equivalently, any synchronization code that is guaranteed to work properly in TSO will also work properly in a sequentially consistent model, but not the reverse.

The SPARC architecture defines a weaker model called *Partial Store Ordering* (PSO). PSO is similar to TSO except that it allows stores from a processor to be performed without regard to the program order unless they are to the same address. TSO and PSO were the memory models of the SPARC Version 8 architecture.

In order to allow more latitude in multiprocessor implementations, a new model called *Relaxed Memory Order* (RMO) was defined in the SPARC Version 9 architecture. In the TSO and PSO models, a load operation must be performed before any stores that follow it in program order. However, in many cases, RMO allows a store to be performed before a load even if the load occurred first in program order.

2.3 Logical Specification of Relaxed Memory Order

This section is a condensed description[1] of the logical specification of the memory model in the SPARC Architecture Manual Version 9. The logical specification is not executable. In essence, given an instruction trace from each processor consisting of the sequence of instructions and the results of interactions with the memory system, it determines whether the instruction trace is compatible with the memory model. This

[1] The change we made from the SPARC manual is that we do not differentiate memory transactions from instructions. We believe that such a distinction is not necessary at this level of specification.

is the style of specification used by Collier, Gharachorloo, and Sindhu and Frailong [13, 24, 62].

In the remainder of this chapter, X, Y, and Z refer to memory instructions. X_A^n denotes a memory instruction X on processor n that reads or writes memory address A. The processor index and memory address are specified only if needed. Predicates $L(X)$ and $S(X)$ are true if X is a load or a store instruction, respectively. $L(Y)$ and $S(Y)$ can be true simultaneously, when Y is an atomic load/store.

A *program order* is a partial order of instructions that is an interleaving of total orders, one for each processor: instructions associated with the same processor are always program-ordered, while instructions from different processors are never program-ordered. Program order represents the sequence of instructions as issued by each processor. We write $A <_p B$ when instruction A precedes instruction B in program order.

A *memory order* is a total order of all the memory instructions from the processors. Each memory model defines a set of *ordering rules* which constrain legal memory orders. Many memory orders may be consistent with a given program order. This multiplicity of orders reflects nondeterminism in the memory model, and yields nondeterministic results when multiprocessor programs are executed. The choice of a particular global memory order determines the values returned by loads. We write $A <_m B$ when instruction A precedes instruction B in a particular memory order; also, in this case, we say "A is performed before B."

The SPARC-V9 architecture has a special *memory barrier* instruction (*membar*). It explicitly enforces additional constraints on the memory order of certain types of memory instructions preceding and following the *membar*. For instance, `membar{L<S}` requires that all the loads preceding the membar in program order precede the stores following it. The predicate $M(X, Y)$ holds when $X <_p Y$ and X and Y are ordered by a membar of the corresponding type. For example, in the instruction sequence of $L <_p S <_p$ `membar{L<L,S<S}` $<_p L' <_p S'$, $M(L, L')$ and $M(S, S')$ hold. Therefore, $S <_m S' <_m L <_m L'$ is a legal memory ordering—if there are no other constraints, but $S' <_m S <_m L <_m L'$ is not.

A program in a weak memory model can be made to behave like the same program in a stronger model by inserting membars between appropriate instructions. To simulate PSO under RMO, we may insert `membar{L<L,L<S}` immediately following every load, disabling the freedom of RMO to delay the execution of loads until after the following memory instructions.

2.3.1 Ordering rules

There are some times when ordering constraints from a processor must necessarily constrain the memory order. For example, an instruction loading a register cannot be performed after an instruction storing the resulting register value back to memory; a store cannot be performed before a preceding branch is resolved. The SPARC-V9 memory model defines a *dependence order* (denoted by $<_d$) which captures the data dependence and control dependence relations among instructions, as one step in the specification of the constraints on memory order. Dependence order is determined from program order as follows.

$A <_d B$, if $A <_p B$ and at least one of the following is true.

(d1) A and B are control dependent and $S(B)$

(d2) A writes a register read by B

(d3) A stores a memory location loaded by B

Precise rules can be defined so that each dependence can be determined between every pair of instructions in a sequence by inspecting the sequence. Rule (d1) requires a branch instruction to a following store. Rule (d2) orders two instructions, if the preceding writes a register which is read by the following. Note that some branch instructions read a special register CCR (condition code register) to decide branching, so a test instruction that write a CCR is dependence ordered to such a branch instruction that read the CCR. Rule (d3) orders a store to a following load which reads the memory location written by the store.

Caution is required in the definition of dependence order, because it constrains memory order. If dependence order is too strict, it may unnecessarily constrain the range of legal processor implementations. There are two specification issues that should be mentioned. First, the definition (d2) does *not* define a dependence when two instructions write the same register, or when an instruction reads a register then another writes it, in order to allow register renaming in the processor implementation. Second, rule (d1) is defined so that there is a control dependence when an instruction affects a branch which is followed by a store, but *not* when the following instruction is a load. This ensures that the processor is allowed to do speculative execution of loads after a branch, before the branch has been decided. Both of these decisions affect the executable description, which must include register renaming and speculative execution of loads if it is to be maximally general.

A particular memory total order $<_m$ is legal if $X_A <_m Y_B$ whenever one or more of the following conditions holds.

(m1) $X_A <_d Y_B$ and $L(X_A)$

(m2) $M(X_A, Y_B)$

(m3) $A = B$ and $X_A <_p Y_B$ and $S(Y_B)$

Rule (m1) says that if two instructions are data dependent ($<_d$) and the first is a load, then they should be performed in order ($<_m$). Preceding stores may be delayed even if there is a data dependence between them and following instructions. Therefore, rule (m1) allows the implementation to use store buffers. Rule (m2) describes the ordering constraint explicitly imposed by membars. Rule (m3) requires that stores to the same address be performed in program order. The rule also orders a load and a following store to the same address, a constraint that is not captured by dependence order. This constraint is necessary to ensure that a single processor behaves as though instructions were performed in program order.

2.3.2 Value axiom

While the ordering rules constrain the performance order of memory instructions, the following axiom defines the value returned by a load:

$$\text{Value } (L_A) = \text{Value } (\text{ Maximum } S_A \text{ under } <_m \text{ from the set of } \{ S_A <_m L_A \} \cup \{ S_A <_p L_A \}).$$

Given a particular memory order, it implies that the value returned by a load is that of the latest store with respect to the memory order that is performed by the shared memory before the load ($\{S_A <_m L_A\}$), or that precedes the load in program order ($\{S_A <_p L_A\}$). Note that the store in the latter case should be the one issued by the same processor which issued the load, since $<_p$ does not order instructions from different processors.

2.4 Executable Specification of Relaxed Memory Order

The executable specification is intended to be maximally general—not only should it conform to the logical specification, it should generate every possible result that is allowed under the specification. Hence, it is more difficult in some ways to write the executable specification than to describe a particular multiprocessor, because a multiprocessor does not have to take advantage of every degree of freedom allowed by the logical specification. On the other hand, the executable model does not have to represent an efficient or practical solution, so it is much easier to design in that sense.

2.4.1 Murφ description language and verifier system

Murφ is a high-level description language for modeling finite-state asynchronous concurrent systems. There is an automatic verifier for Murφ which generates all of the reachable states of the system while checking for deadlock and other error conditions. Murφ can also check liveness and fairness properties (e.g. progress). The syntax of Murφ is derived from various standard programming languages, especially Pascal and C.

Murφ allows the declaration of familiar data types, including subranges of integers, arrays, records, and user-defined enumerations. Additionally, procedures, functions, and global variables can be declared. The operational part of the language is based on *iterated guarded commands*—which was inspired by Misra and Chandy's Unity language [10]. A *state* of the described concurrent system is an assignment to each global variable with a value in the range of the declared type. A Murφ program consists of a collection of *rules*. Each rule has a *condition*, which is Boolean expression referring to the global variables, and an *action*, which is a statement that modifies the values of the variables, yielding a new state.

Execution of a Murφ program begins with one of a set of initial states specified by the user. Then the following loop is executed forever: some rule whose condition is satisfied by the current state is chosen and its action evaluated, yielding a new current state. If there are no rules whose conditions are true, the execution halts. Although the action may be a compound statement consisting of a sequence of smaller statements, conditionals, and loops, it is executed *atomically*—no other rule can be executed before the action completes.

When several rule conditions are true at the same time, a choice is made arbitrarily, resulting in several possible executions. The Murφ verifier tries them exhaustively by depth-first or breadth-first search.

One essential construct in Murφ is the *ruleset*, which is used to describe a collection of rules that vary over a parameter. A ruleset can be thought of as nondeterministically selecting a value for the parameter from a set.

Several types of errors can be detected in a Murφ description. There is an *error* statement that can appear in an action. *Invariant* Boolean expressions may also be specified; if the invariant is false in any reachable state, an error is reported. The system can detect *deadlock states*, which are states that have no other states as successors. Finally, Murφ can check many common liveness and fairness properties using a subset of linear-time temporal logic.

If a problem of any type is detected, the verifier prints out a *diagnostic trace*, which is a sequence of states that leads to a state exhibiting the problem. In addition to the error traces, it is possible to print out the values of specified variables using `put` commands. This capability is used to obtain all the possible results of test programs.

2.4.2 RMO description in Murφ

The executable specification follows the intuition of Figure 2.1. It describes reordering boxes, global memory with a nondeterministic switch, and necessary part of processors. There are shared variables for all of the state of the system, including the processor registers, the memory, and the contents of the reorder boxes. Here, we provide excerpts from the description.

State of the model

In the first part of Figure 2.2, constants are declared for the number of processors, size of memory, number of registers, size of reorder boxes, and so on. These constants are the only declarations needed to be changed to have a larger or smaller-sized system, since the description is scalable. For verification, these constants should be kept very small in order to bound the size of the state space that must be explored by the verifier. Constants such as `A`, `B`, `r1`, etc. are used to represent memory addresses and registers in the program symbolically.

Types are declared using those constants. For example, `Processor` is declared to be a subrange of integers, which are used as identifiers for processors. `Instruction` is an enumeration, which may include all kinds of instructions used in the program being modeled. `IssueIndex` is a pointer to an instruction in a reorder box and `ReorderBoxType` is a record representing a reorder box of each processor. It consists of a counter and an array of instruction records. The counter holds the number of instruction records in the reorder box. Each instruction record contains all the information of an instruction: the instruction type, memory address operand, register operand, constant operand, and so on.

Global states of the executable model are represented by a set of state variables shown in Figure 2.3. The first variable, `Memory`, models the global memory by an array of `Value` indexed by `Address` type. The control state in a processor is modeled using global variables: program counter `PC` and `nPC` (next `PC`), and condition code register `CCR`. The use of the program counters will be explained in the next section.

```
Const
  ProcessorNum   : 3;
  AddressNum     : 3;
  RegisterNum    : 3;
  TempNum        : 6;
  ValueNum       : 5;
  MaxPC          : 10;
  ReorderBoxSize : 6;
  A : 1;
  B : 1;
  r1: 1;    -- other constants are omitted

Type
  Processor : 0 .. ProcessorNum - 1 ;
  Address   : 0 .. AddressNum - 1 ;
  TempIndex : 0 .. TempNum - 1;   -- for register renaming
  Value     : 0 .. ValueNum - 1;  -- range of data
  Instruction: Enum{Iload, Istore, Ildstore, Iswap, Imov, Icmp, Itst, Imembar,
                  IbeY, IbeN, IbneY, IbneN }; -- other instructions are omitted
  Label      : 0 .. MaxPC ;
  IssueIndex : 0 .. ReorderBoxSize - 1 ;
  ReorderBoxType : Record
    Count : 0 .. ReorderBoxSize;  -- number of instructions in the box
    Ar    : Array[IssueIndex] of
              Record  Instr : Instruction;
                      Addr  : Address;
                      Temp  : TempIndex;  -- and so on
                      MemBit: Array [Enum{ b0, b1, b2, b3 }] of Boolean;
                              -- used for membar instructions
              End;
  End;
  -- other types are omitted
```

Figure 2.2: Constant and type declarations for the executable memory model in Murφ

```
Var
  Memory    : Array [Address] of Value;
  PC        : Array [Processor] of Label;
  nPC       : Array [Processor] of Label;        -- for anulled branch
  CCR       : Array [Processor] of Boolean;      -- condition code register
  Registers : Array [Processor] of Array [Register] of TempIndex;
  Temps     : Array [Processor] of Array [TempIndex] of Value;
  ReorderBox: Array [Processor] of ReorderBoxType;

  -- for speculation branch
  memPC     : Array [Processor] of Label;
  memnPC    : Array [Processor] of Label;
  memReg    : Array [Processor] of Array[Register] of TempIndex;
```

Figure 2.3: State variable declarations for the executable memory model in Murφ

This description is based on a register renaming scheme, so `registers` are a per-processor array of temporary indices, which are pointers to temporaries in a register pool, and `Temps` are real locations where the values for registers are kept.

The reorder boxes are represented as an array `ReorderBox` of `ReorderBoxType` indexed by `Processor`. A reorder box queues up every instruction from its processor in program order.

The description also implements speculation on branches. The states of PCs and registers are saved in `memPC` and `memReg` at the time of issue of a speculative branch; these are used to restore the PCs and registers in case the speculation turns out to be incorrect.

Procedures

There are individual processes for the processors in Figure 2.1 and for the memory. Only one process may execute at a time. The processes modeling the processors issue individual instructions by inserting them at the tail of a reorder box queue, so that instructions in a reorder box are always in program order.

For each class of instruction, there is a procedure in the Murφ description that issues the instruction by inserting it in the reorder box. For example, `Load_init()` in Figure 2.4 inserts a load instruction with its operands after all previously-issued

```
Procedure Load_init(p:Processor; a:Address; r:Register);
Begin
  PutInBox(p, Iload, a, GetTemp(p,r));
  PC[p] := nPC[p]; nPC[p] := nPC[p] + 1;
End;

Procedure Store_init(p:Processor; r:Register; a:Address);
Begin
  PutInBox(p, Istore, a, Registers[p][r]);
  PC[p] := nPC[p]; nPC[p] := nPC[p] + 1;
End;
```

Figure 2.4: Procedure 'Load_init' and 'Store_init'

instructions in the reorder box. This issuing procedure also handles register renaming, so that the instructions in the reorder box refer to temporary registers, not to register names. The function `GetTemp(p,r)` returns a new temporary location for register r in processor p. Procedure `PutInBox()` puts the instruction into a reorder box by copying it at the tail of instruction queue of the reorder box. Finally, the next PC value is copied to the current PC, then incremented.

Procedures for other instructions simply copy the instruction into reorder boxes. When a store instruction is issued, the procedure `Store_init()` puts its arguments into the corresponding reorder box. When a branch instruction is issued, it is performed immediately if the CCR is already set by a test. Otherwise, a nondeterministic prediction of the branch direction is made. The branch is then issued based on this prediction. There are two issuing procedures for each branch instruction: *e.g.*, `BeY_init()` and `BeN_init()` for `be` instruction ("branch on equal").

There is also a procedure to perform each class of instruction type. These procedures are executed by the global memory, and do all of the work of the instructions. For instance, the procedure `Store_perf()` in Figure 2.5 performs a store instruction by writing the contents of the register (assigned temporary) into the memory location.

Performing a load is more involved because the value axiom should be satisfied determining the value to be loaded. The procedure `Load_perf()` shown in Figure 2.5 is executed when a load is performed to read the specified memory address.

```
Procedure Store_perf(p:Processor; t: TempIndex; a:Address);
Begin Memory[a] := Temps[p][t];  End;


Procedure Load_perf(p:Processor; i:IssueIndex; a:Address; t:TempIndex);
-- Read the most recent value of address A from the view of processor[p].
Begin
  Temps[p][t] := Memory[a];
  For j:IssueIndex Do
  Alias J:ReorderBox[p].Ar[j] do
    If ( j<i & J.Addr = a ) Then
      -- if there is a store preceding in program order but not performed,
      -- then get the value being stored.
      If (  J.Instr = Istore | J.Instr = Iswap )
        Then Temps[p][t] := Temps[p][J.Temp];
      Elsif ( J.Instr=Ildstore ) Then Temps[p][t] := 1;  End; --If
    End; --If
  End; --Alias
  End; --For
End;
```

Figure 2.5: Procedure 'Load_perf' and 'Store_perf'

If the prediction on a branch turns out to be incorrect when the branch instruction is performed, the state of the registers and program counter is restored to what it was when the branch was issued, and the speculative instructions are canceled. Since the logical specification allows speculative execution and ignores anti-dependences from register usage, it is necessary to include speculative execution to ensure that the executable specification generates every legal program result.

Functions

The description attempts to provide the most direct possible translation from the ordering rules given in the logical description. We write functions for the low-level predicates that it uses. Functions `Is_load(p,i)` and `Is_store(p,i)` check whether the instruction at position i in the reorder box for processor p is a load or a store instruction. Similarly, `Is_branch` and `Is_CCR` check whether the instruction is a branch or a test that writes a CCR.

The function `Membared(p,i,j)` is shown in Figure 2.6. It returns a Boolean formula which is true if and only if the two memory instructions at entry i and j are ordered through a membar. Each *or*'ed expression checks if there is a memory barrier of the corresponding type in between the two instructions. The `Alias` command is used to define an abbreviation.

```
Function Membared(p:Processor; i,j:IssueIndex):Boolean;
Begin
  Alias R:ReorderBox[p] do
  Return
    (Is_load(p,i) & Is_load(p,j) &
     Exists k:IssueIndex do      -- bit:b0 corresponds to the type of {L<L}
       i<k & k<j & R.Ar[k].Instr = Imembar & R.Ar[k].MemBit[b0] end)
  |
    (Is_store(p,i) & Is_load(p,j) &
     Exists k:IssueIndex do      -- bit:b1 corresponds to the type of {S<L}
       i<k & k<j & R.Ar[k].Instr = Imembar & R.Ar[k].MemBit[b1] end)
  |
    (Is_load(p,i) & Is_store(p,j) &
     Exists k:IssueIndex do      -- bit:b2 corresponds to the type of {L<S}
       i<k & k<j & R.Ar[k].Instr = Imembar & R.Ar[k].MemBit[b2] end)
  |
    (Is_store(p,i) & Is_store(p,j) &
     Exists k:IssueIndex do      -- bit:b3 corresponds to the type of {S<S}
       i<k & k<j & R.Ar[k].Instr = Imembar & R.Ar[k].MemBit[b3] end);
  End;
End;
```

Figure 2.6: Function 'Membared'

In order to define function `Depend()`, we first define function `Direct_Depend()` in Figure 2.7. This function checks whether two instructions are dependence ordered, as define in Section 2.3. At first, it ensures the instruction at i precedes the one at j in program order ($X <_p Y$). The rest of the Boolean expressions correspond to the rules (d1) through (d3). The first *or*'ed expression translates the rule (d1). The next expression for the rule (d2) calls `Reg_Depend()`, which returns true if the preceding instruction is writing the same register that is read by the following instruction—strictly speaking, the same *temporary* location in the register renaming scheme. The dependence through the condition code register is checked separately. The last part of the Boolean expression directly translates the rule (d3).

```
Function Direct_Depend (p:Processor; i,j:IssueIndex):Boolean;
Begin
  Return ( i < j ) &
       (
         ( Is_branch(p,i) & Is_store(p,j) )
       | ( Reg_Depend(p,i,j) | ( Is_CCR(p,i) & Is_branch(p,j) ) )
       | ( Is_store(p,i) & Is_load(p,j) & mAddress(p,i) = mAddress(p,j) ) );
End;


Function Reg_Depend ( p:Processor;  i,j:IssueIndex):Boolean;
-- Return true if instruction at 'i' writes a register read by the one at 'j'.


Function Depend (p:Processor; i,j:IssueIndex):Boolean;
-- Instructions at 'i' and 'j' are dependent through transitivity?
Begin
  If ( i >= j ) Then Return false;
  Elsif Direct_Depend(p,i,j) Then Return true;
  Elsif
    Exists k:IssueIndex Do
       ( i < k ) & ( k < j ) & Depend(p,i,k) & Depend(p,k,j)
    End --Exists
    Then Return  true;
  Else Return false;
  End; --If
End;
```

Figure 2.7: Function 'Direct_Depend' and 'Depend'

Dependence order is the transitive closure of `Direct_Depend()`. The function `Depend(p,i,j)` in Figure 2.7 computes this using the function `Direct_Depend()` and calling itself recursively. It returns true if and only if the instructions at i and j in reorder box p are dependence ordered, by checking if there exists an instruction in-between i and j through which the dependence order is transitive.

Rules

There are two collections of Murφ rules which implement the diagram in Figure 2.1. The first collection of rules describes the behavior of the model that *issues* instructions into reorder boxes. The other collection of rules *performs* the instruction in the reorder boxes. We will postpone the explanation of the first kind of rules to the next section. Here, we show the latter kind of Murφ rules implementing the memory order.

The global memory process nondeterministically selects a processor and an instruction in the processor, which is executed if it is legal to do so. The ordering rules are enforced by the conditions of Murφ rules that decide whether an instruction is legal to perform. An instruction is legal to perform if and only if the ordering rules allow the existence of a memory order in which the instruction can be the minimum under a memory order of all the instructions currently in the reorder box. Each ordering rule from the logical description is translated as directly as possible to a Murφ function, which checks whether the ordering rule is satisfied.

When an instruction can legally be performed, the memory performs it. This involves doing all of the computation associated with the instruction, including ALU operations and updating registers and/or memory. Then it is removed from the reorder box.

In essence, a particular memory order is gradually constructed as the specification executes. The instructions that have been performed are memory ordered and those remaining in the reorder boxes have not yet been ordered. The constraints on nondeterministic choices involved in selecting the next instruction ensure that every legal memory order can be generated.

The rule for the memory order constraints is in Figure 2.8. This rule can also be thought of as implementing the behavior of the memory. The rule is embedded in parameterized rulesets that nondeterministically choose a reorder box and an instruction index. It performs the instruction at that reorder box index only if that instruction is allowed to appear first among all the instructions in the reorder box, according to the memory ordering rules.

The condition of the rule is a conjunction of several Boolean expressions. The first *and*'ed expression ensures that the chosen index i of the reorder box p is not an empty slot. The second expression requires every membar instruction to remain in the reorder box until all the previous instructions are executed and removed from the box.

```
Ruleset p:Processor Do
Ruleset i:IssueIndex Do
  -- For all instructions, ReorderBox[p].Ar[i], check this.
  Alias R : ReorderBox[p];
       op : R.Ar[i].Instr Do
    Rule "Execute one of the instructions in minimal set"
       ( i <  R.Count )      -- instruction at i is valid one
     & ( op=Imembar -> i=0 )    -- the membar no longer needs to be here
     & -- Rule (m1)
       ( Forall j:IssueIndex Do
           j<i -> ! ( Depend(p,j,i) & Is_wr_reg(p,j) )
         EndForall  )
     & -- Rule (m2)
       ( Forall j:IssueIndex Do
           j<i -> ! Membared(p,j,i)
         EndForall )
     & -- Rule (m3)
       ( Is_store(p,i) ->  Forall j:IssueIndex Do
                            ! ( j<i & ( Is_store(p,j) | Is_load(p,j) )
                            & mAddress(p,j) = mAddress(p,i) )
                           EndForall )
    ==>

    Begin
    -- The chosen instruction ReorderBox[p].Ar[i] is allowed to be performed.
      Switch op
        Case Iload    : Load_perf(p, i, R.Ar[i].Addr, R.Ar[i].Temp);
        Case Ildstore: Ldstore_perf(p, i, R.Ar[i].Addr, R.Ar[i].Temp);
        Case Istore   : Store_perf(p, R.Ar[i].Temp, R.Ar[i].Addr);
        -- Others are omitted.
        -- Do appropriate action for each class of instruction type.
      End; -- switch
      ArrangeBox(p,i);   -- Dequeue the performed instruction.
    End; --Rule
  End; --Alias
End; --Ruleset on IssueIndex
End; --Ruleset on Processor
```

Figure 2.8: RMO ordering constraints in Murφ

The rest of the three Boolean expressions correspond to the ordering rules (m1), (m2), and (m3), respectively. The conditions are not direct translation of the rules, but they ensure that there is no preceding memory instruction at index j which is memory ordered ($<_m$) to the one at i. The predicate $L(X)$ in rule (m1) is replaced by the function checking if the instruction is writing to a register, because reorder boxes deal with all kinds of instructions including non-memory instructions, while the axioms in the previous section aim to enforce orders among memory instructions only. The next expression corresponding to (m2) asserts that there is no preceding instruction at entry j which is ordered through a membar to the one at i. The expression corresponding to (m3) asserts that the rule is satisfied.

If the condition of the rule is true, the chosen instruction is performed by executing the corresponding procedure according to the instruction type. For example, `Store_perf()` is executed for a store and `Load_perf()` is executed for a load. Procedure `ArrangeBox()` is called to remove the instruction record from the reorder box. This procedure also shifts the remaining instruction records in the reorder box to make it compact and accommodate more instructions which are not issued yet.

As explained in Section 2.3, TSO and PSO models can be easily defined from RMO with additional constraints. The executable description can be extended to accommodate TSO and PSO models with the conditions in Figure 2.9 appended to the condition of the Murφ rule for RMO model in Figure 2.8. The constants `TSOmodel` or `PSOmodel` are set initially to simulate the TSO or PSO model.

One subtle point is the avoidance of starvation: the logical description requires that the memory order include every instruction. This implies that the memory must eventually perform every memory instruction in every reorder box. Not only is this necessary to conform to the logical specification, it is essential to avoid starvation in synchronization routines. This requirement is handled in Murφ by requiring, in an infinite computation, that the oldest instruction at index 0 in every reorder box be performed infinitely often (instructions not satisfying this requirement can be performed an arbitrary finite number of times between oldest instructions). Since the oldest instruction is always legal to perform, according to the ordering rules, this ensures that every instruction is eventually performed. Murφ descriptions have a facility

```
-- Additional constraint of TSO and PSO: { L < L } and { L < S }
  (
    ( TSOmodel | PSOmodel ) & ( Is_load(p,i) | Is_store(p,i) )
    ->
    Forall j:IssueIndex Do
      (j<i) -> ! Is_load(p,j)
    End --Forall
  )
&
-- Additional constraint of TSO: { S < S }
  (
    TSOmodel & Is_store(p,i)
    ->
    Forall j:IssueIndex Do
      (j<i) -> ! Is_store(p,j)
    End --Forall
  )
```

Figure 2.9: Additional constraints for TSO and PSO

for specifying fairness assumptions which is used to implement this requirement.

Since Murφ can only deal with finite-state processes, various memory structures must be bounded. Furthermore, the number of states grows exponentially with many parameters, so even quantities that are bounded in all implementations, such as the number of registers in a processor, are bounded much more sharply in the Murφ description. Bounded quantities include: the number processors, memory values, memory locations, registers, and reorder boxes.

If the Murφ program is considered without the bounds, it is equivalent to the logical specification. The executable specification in Murφ not only conforms to the logical specification but also generates all the possible behaviors allowed under the specification. We have proved this equivalence using a theorem prover, and the outline of the proof is shown in Section 2.6. With the bounds, however, the executions of the Murφ program may be a subset of the executions allowed by the logical specification. For some programs, however, it is easy to see that small bounds on all parameters allow sufficient resources to enumerate *all* of the possibilities. For larger descriptions, we must trade generality for the ability to verify automatically a bounded description.

2.5 Automatic Analysis and Verification

Using the Murφ verifier allows several kinds of automatic analysis and verification of programs executing in the specified memory models. This section shows techniques to generate complete lists of program results, to generate a specific execution trace for a questioned output, and to verify synchronization routines.

2.5.1 Analyzing test programs with an automatic verifier

When developing the RMO model, it was very helpful to be able to find all of the possible results of small example multiprocessor programs. The automatic verifier Murφ finds all of the reachable states of the system, so it can list complete list of the possible outputs very easily. When ordering rules are changed, it is simple to change the executable specification—since the translation is so direct—and run the test programs through the verifier to find out the consequences.

To make the description fully executable, we need to model *programs* running on the processors in addition to the memory system. This can be accomplished by adding rules for the processor description which specify which instruction to issue, as a function of the current PC value [18, 54]. This translation is easy but could be tedious. So we have implemented a simple program that translates assembly-language programs into the appropriate Murφ rules, which are then combined with the executable specification to yield a Murφ description of that particular program running in the memory model.

Suppose we test the program at the top of Figure 2.10. This program can be automatically translated to the rules in Figure 2.11. The first rule corresponds to the first load of P_0 (Processor 0) and the next rule to the last store of P_1. For readability, we have given symbolic names to memory locations and registers by defining them as constants. The register `V3_1` of P_1 contains a constant value 3.

In order to obtain a list of outcomes of a test program, we have added another rule shown in Figure 2.11 which prints out the state of the registers and shared memory when the program terminates after executing all the instructions. Since Murφ does

```
Processor 0              Processor 1

ld  A, %r1               ld  C, %rx
st #1, B                 membar #LoadLoad #LoadStore
st #2, C                 ld  B, %ry
                         membar #LoadStore
                         st #3, A
```

```
--TSO----------------------------------------------
        A:3 B:1 C:2 r1(0):3 rx(1):0 ry(1):0
        A:3 B:1 C:2 r1(0):0 rx(1):0 ry(1):0
        A:3 B:1 C:2 r1(0):0 rx(1):0 ry(1):1
        A:3 B:1 C:2 r1(0):0 rx(1):2 ry(1):1
--PSO----------------------------------------------
        A:3 B:1 C:2 r1(0):3 rx(1):0 ry(1):0
        A:3 B:1 C:2 r1(0):0 rx(1):0 ry(1):0
        A:3 B:1 C:2 r1(0):0 rx(1):0 ry(1):1
        A:3 B:1 C:2 r1(0):0 rx(1):2 ry(1):0
        A:3 B:1 C:2 r1(0):0 rx(1):2 ry(1):1
--RMO----------------------------------------------
        A:3 B:1 C:2 r1(0):3 rx(1):0 ry(1):0
        A:3 B:1 C:2 r1(0):0 rx(1):0 ry(1):0
        A:3 B:1 C:2 r1(0):3 rx(1):0 ry(1):1
        A:3 B:1 C:2 r1(0):0 rx(1):0 ry(1):1
        A:3 B:1 C:2 r1(0):3 rx(1):2 ry(1):0
        A:3 B:1 C:2 r1(0):0 rx(1):2 ry(1):0
        A:3 B:1 C:2 r1(0):3 rx(1):2 ry(1):1
        A:3 B:1 C:2 r1(0):0 rx(1):2 ry(1):1
---------------------------------------------------
```

Figure 2.10: An example test program and the corresponding set of possible results generated by the automatic verifier

```
Rule  -- a rule corresponding to the load of processor 0
  PC[0]=0 & ReorderBox[0].Count < ReorderBoxSize
==>  begin Load_init(0, A, r1_0); end;

Rule  -- a rule corresponding to the store of processor 1
  PC[1]=4 & ReorderBox[1].Count < ReorderBoxSize
==>  begin Store_init(1, V3_1, A); end;

Rule "Print out the states when the program terminates"
  PC[0]=3 & PC[1]=5 &
  Forall p:Processor Do ReorderBox[p].Count = 0 End
==>
Begin
  Put " A:"; Put Memory[A];  -- prints the value
  Put " B:"; Put Memory[B];
  Put " ry(1):"; Put Temps[1][Registers[1][ry_1]];
  -- others are omitted
End;
```

Figure 2.11: Murφ rules for assembly language programs

exhaustive searching, each result through every possible interleaved performance ordering is caught by the printing rule and printed out. Indeed, each possible result is printed many times because it occurs in many different executions, but the results are then post-processed to eliminate duplicates.

Figure 2.10 shows the results obtained by running the verifier on the example program under the various SPARC memory models—this is actual program output. We assume an initial value of zero in every memory location and register. Each line lists the contents of the relevant memory locations and registers for a different terminating state of the program. Since all the memory operations of P_1 are ordered by membar instructions, they should be performed in the program order. However, the memory operations in P_0 can be reordered as long as they satisfy the ordering constraints of each memory model. Indeed, all of the 6 permutations are possible in RMO, because there are no dependences among the 3 instructions. The result shows that RMO allows more behaviors than PSO, which allows more than TSO, as expected.

The previous example is constructed artificially in order to show different behaviors of the three memory models. Here is another example which shows subtle behavior of RMO memory model. Again we assume every memory location and register has an initial value of zero when the program starts.

```
         Processor 0                Processor 1

1.       ld   [A], %r1       5.     ld   [B], %r0
2.       st   #1 , [A]              membar #LoadStore
3.       ld   [A], %r2       6.     st   #2 , [A]
4.       st   %r2, [B]
```

One of the results reported by the Murφ verifier is `%r1` $= 2$ in P_0 and `%r0` $= 1$ in P_1. The output may be unexpected because the first load of P_0 in the program returns a more recent value of the memory at address A (`%r1` $= 2$) than the second load of the address (`%r2` $= 1$). However, the result can be obtained when the memory instructions are performed in the memory order $3 <_m 4 <_m 5 <_m 6 <_m 1 <_m 2$. This memory order is legal because there is no ordering constraint between 2 and 3, nor between 1 and 3, since anti-dependence is not considered in rule (d2) above.

If a user obtains an unexpected outcome of a test program, a trace can be generated which shows how the outcome can occur, using a simple trick. An invariant is added which asserts that the unexpected state does not occur. When the verifier finds the state, a counter-example trace will be generated automatically that gives the sequence of rules and intermediate states leading from an initial state to the state in question.

2.5.2 Verifying synchronization routines

In many large-scale concurrent programs, the low-level synchronization code—which may even be generated by a compiler—is the only part that depends on the details of the memory model; this code can be carefully crafted to work in a particular memory model, then used elsewhere by programmers who need not be deeply familiar with its internals.

The SPARC Architecture Manual gives several assembly language routines for standard synchronization paradigms, including spin locks in two versions: one using the load-store instruction and one using swap, produce-consumer with a bounded buffer, and Dekker's algorithm. There are different versions for each algorithm for each memory model. Figure 2.12 shows the assembly language code for a spin lock using a load-store instruction. This is taken verbatim from the manual, except that two instructions have been inserted at the label `crit`, to improve error detection.

```
--------------------------------------------------------------
retry:  ldstub  [lock], %l0     -- load-store
        tst     %l0             -- branch if 0
        be      out
        nop
loop:   ldub    [lock], %l0     -- load
        tst     %l0             -- branch if non-0
        bne     loop
        nop
        ba,a    retry           -- jump
out:    membar  #LoadLoad #LoadStore
--------------------------------------------------------------
crit:   stub    #1, [CM(i)]     -- store to a special
        stub    #0, [CM(i)]     -- location of each processor
--------------------------------------------------------------
unlock: membar  #StoreStore     -- RMO, PSO only
        membar  #LoadStore      -- RMO only
        stub    %g0, [lock]     -- store value zero
--------------------------------------------------------------
```

Figure 2.12: Assembly language program for spin lock synchronization

In the synchronization code, the "lock held" condition is kept in a specific memory location `lock`. A nonzero value of the lock represents that the lock is held by some process, while a zero value means that the lock is free. An instruction `ldstub` loads a specified memory location and stores a nonzero value to the memory, atomically. The conditional branch `be` is taken if the special register `CCR` set by `tst` is zero. The following instruction (in this case, `nop`) is executed even if the branch is taken because, the SPARC has delayed branching. Note that the `membars` enforce ordering between memory instructions in the critical section and others in the synchronization routine.

As shown in the spin lock code, we added two store instructions in critical region; one stores a constant value 1 to a critical memory location and the other also stores 0 in the location. The invariant below is used to check the mutual exclusion property of the spin lock when there are two processors.

```
Invariant  "Mutual Exclusion of Memory Access in Critical location"
     ! (Memory[CM0] = 1 & Memory[CM1] = 1);
```

Extending this to more processors is straightforward. The invariant also ensures that a lock is not released too early, before the writes to the lock-protected location are completed. The verifier also checks for deadlocks.

The spin lock example described was computationally the most difficult of all the examples we tried in SPARC Architecture Manual Version 9, although all examples required the same order-of-magnitude time and space. When the spin lock example was modeled with 2 processors and a reorder box size of 6, the verifier explored 37,736 states in less than 10 minutes. The time is not proportional to the number of states because a state may be visited several times—depending on the number of incoming edges in the state graph—and because the amount of time for each rule varies with the complexity of the rules.

This spin lock is subject to starvation. It is possible for P_0 to be denied the lock forever even when P_1 releases the lock infinitely often, because P_1 happens to be holding the lock whenever P_0 tests it. For this reason, Murφ reports a violation of the property,

```
Liveness Eventually Memory[CM0] = 1 ;
```

even though each process is assumed in the description to release its lock infinitely often. However, Murφ finds no violation of the weaker property that "at least one process gets the lock infinitely often,"

```
Liveness Eventually ( Memory[CM0]=1 | Memory[CM1]=1 ) ; .
```

Ultimately, no unexpected behavior was found in the synchronization routines when combined with the appropriate models. Also, as expected, the TSO routine failed when combined with the PSO and RMO memory models.

The state explosion problem, which is a central problem in finite-state verification, has not been an issue in this effort, because of the small size of the assembly language routines. However, it would become a problem for verification of larger programs.

2.6 Formal Proof of Equivalence

In this section, we prove that the executable description in Murφ, described in Section 2.4, is equivalent to the logical specification in Section 2.3 which is a slightly modified version of the one in the SPARC-V9 Architecture Manual, if the number of processors and the size of reorder boxes in the Murφ program are unbounded. The proof has been checked with PVS theorem prover which is developed by Computer Science Laboratory at SRI International [53]. The proof here follows the PVS proof, but has been rewritten for human-readability.

The proof consists of two parts: 1) The set of legal memory orders allowed under the logical specification is equivalent to the set of memory orders generated by the executable description (Theorem 1 and Theorem 2 in this section); 2) the value axiom in the logical specification is correctly implemented in the executable description (Theorem 3 in this section).

We represent a memory order (as defined in the logical specification of RMO) as a finite sequence of memory instructions, $m = \langle m_i \rangle$, indexed by natural numbers. A memory instruction m_i is issued by processor $proc(m_i)$, and it accesses memory location $addr(m_i)$. As defined in Section 2.3, $x <_p y$ is true if x precedes y in program order, and $x <_d y$ is true if x precedes y in dependence order. The predicate $M(x, y)$ is used to represent that $x <_p y$ and x and y are ordered by a memory barrier. The predicate $S(x)$ is true if x is a store instruction, and $L(x)$ is true if x is a load instruction.

The Murφ model of RMO generates a set of *executions*, which are sequences of states. Each state is mapped to the next in the sequence by a Murφ rule from the description. There are two sets of Murφ rules in the executable description.

- *Issue rules* issue an instruction by inserting it at the tail of instruction queue of a reorder box, as shown in Figure 2.11

- *Performance rules* perform an instruction in a reorder box if it is legal, as shown in Figure 2.8

The event of a rule issuing instruction m_i is written by $Issue(m_i)$; the event of performing m_i is written by $Perform(m_i)$. The memory order of an execution is defined to be the order in which the $Perform(m_i)$ events appear in the execution.

It is convenient to refer to the *time* at which a rule fires in an execution. Formally, the time of $Perform(m_i)$ is always i; the time of $Issue(m_i)$ is defined to be the time of the next $Perform(m_j)$ (in other words, j) in the execution. The time of a state is similarly the time of the next instruction performance. We define $itime(m_i)$ to be the time when m_i is issued by $Issue(m_i)$. So, in the sequence,

$$s_0 \xrightarrow{Issue(m_1)} s_1 \xrightarrow{Issue(m_0)} s_2 \xrightarrow{Perform(m_0)} s_3 \xrightarrow{Issue(m_2)} s_4 \xrightarrow{Perform(m_1)} \ldots,$$

$itime(m_0)$ and $itime(m_1)$ are both 0, and $itime(m_2)$ is 1.

A memory instruction is performed only after being inserted in a reorder box, and all instructions in reorder boxes must eventually be performed. So, $itime(m_i) \leq i$. We define $r_k(m_i)$ to be the index of m_i in the reorder box (of $proc(m_i)$) at time k. Note that $r_k(m_i)$ is *defined* (denoted as $def(m_i)$) for k when $itime(m_i) \leq k \leq i$, and *undefined* otherwise.

The following properties always hold on an execution of the Murφ program.

Property 1 *A memory order m in the Murφ program satisfies the following properties.*

$$\forall\, i,j: \quad m_i <_p m_j \;\Rightarrow\; itime(m_i) \leq itime(m_j) \tag{2.1}$$

$$\forall\, i,k: \quad itime(m_i) \leq k \leq i \;\Leftrightarrow\; def(r_k(m_i)) \tag{2.2}$$

$$\begin{aligned} \forall\, i,j,k: \quad & m_i <_p m_j \;\wedge\; def(r_k(m_i)) \;\wedge\; def(r_k(m_j)) \\ & \Leftrightarrow\; r_k(m_i) < r_k(m_j) \;\wedge\; proc(m_i) = proc(m_j) \end{aligned} \tag{2.3}$$

2.6.1 The set of legal memory orders is equivalent to the set of generated memory orders

For a formal proof, we define memory orders in two different views: *legal* memory orders and *generated* memory orders. The first part of the equivalence proof is to show that the set of legal memory orders allowed under the logical specification is equivalent to the set of memory orders generated by the executable description. To prove this, we consider two implications: 1) if a memory order is generated then it is legal; 2) if a memory order is legal then it is generated.

The next definition is a straightforward translation of the specification of Section 2.3 into logic.

Definition 1 *A memory order m (a sequence of memory instructions) is* <u>*legal*</u> *if and only if*

$$\begin{aligned}
\forall\, i,j: \quad & (\ m_i <_d m_j \ \wedge\ L(m_i)\ ) \\
\vee \quad & M(m_i, m_j) \\
\vee \quad & (\ addr(m_i) = addr(m_j) \ \wedge\ m_i <_p m_j \ \wedge\ S(m_j)\ ) \\
\Rightarrow \quad & i < j.
\end{aligned}$$

Definition 2 in the following captures a property of every execution of the Murφ description.

Definition 2 *A memory order m (a sequence of memory instructions) is* <u>*generated*</u> *by the executable description if and only if*

$$\begin{aligned}
\forall\, i: \quad & def(r_i(m_i)) \\
\wedge \quad & \forall n \text{ such that } proc(m_n) = proc(m_i) \ \wedge\ r_i(m_n) < r_i(m_i): \\
& \quad \neg(\ m_n <_d m_i \ \wedge\ L(m_n)\ ) \\
& \quad \wedge\ \neg M(m_n, m_i) \\
& \quad \wedge\ (\ S(m_i) \Rightarrow addr(m_n) \neq addr(m_i)\ ).
\end{aligned}$$

Now, we prove that if a memory order is generated then it is legal.

Theorem 1 (Necessary) *If a memory order m is generated by the executable description, then it is legal under the logical specification.*

Proof: The proof is by contradiction. Assume that the generated memory order m is not legal. Then there exist i and j which violate Definition 1, that is,

$$\begin{array}{lll} \exists\, i,j \text{ such that } i \geq j : & & (\ m_i <_d m_j \ \wedge\ L(m_i)\ ) \\ & \vee & M(m_i, m_j) \\ & \vee & (\ addr(m_i) = addr(m_j) \ \wedge\ m_i <_p m_j \ \wedge\ S(m_j)\ ). \end{array}$$

Let us consider each of the disjuncts separately. For the first, suppose

$$\exists\, i,j :\ i \geq j \ \wedge\ m_i <_d m_j \ \wedge\ L(m_i).$$

By the definition of dependence order in Section 2.3, $m_i <_d m_j$ implies $m_i <_p m_j$. By (2.1), $itime(m_i) \leq itime(m_j)$. Definition 2 implies that $def(r_i(m_i))$ and $def(r_j(m_j))$, so, by (2.2), we obtain

$$itime(m_i) \leq itime(m_j) \leq j \leq i.$$

Using (2.2) again, we also know $def(r_j(m_i))$. By (2.3), this implies $proc(m_i) = proc(m_j)$ and $r_j(m_i) < r_j(m_j)$. Instantiating $i = j$ and $n = i$ in Definition 2 yields

$$\neg(\ m_i <_d m_j \ \wedge\ L(m_i)\ ),$$

which contradicts the assumption. The other two cases are similar. Therefore, if m is generated by the executable model, it is legal under the logical specification. $\square$

Next, we prove the implication on the reverse direction.

Theorem 2 (Sufficient) *If a memory order m is legal under the logical specification, then it is possibly generated by the executable description.*

Proof: The theorem requires that m satisfy Definition 2. The first condition of the definition, $def(r_i(m_i))$, can be satisfied from (2.2) if all the instructions are issued by $Issue(m_i)$ at time 0, i.e., $\forall i : itime(m_i) = 0$.

The latter condition should hold for any m_i and m_n in a same reorder box such that $r_i(m_n) < r_i(m_i)$. Any m_n residing in a reorder box at time i will be performed later than time i, therefore, $i < n$. If we instantiate Definition 1 with $i = n$ and $j = i$, the contraposition implies the theorem.□

2.6.2 The value axiom of the logical specification is correctly implemented in the executable description

In addition to defining legal memory orders, RMO specifies what value should be returned when a memory location is read by a load operation. From the view of loading processor, the value axiom finds the most recent store to a specified memory location at the time of performance of the load. The following lemma formalizes the relation between a load and the corresponding store specified in the value axiom.

Axiom 1 (Value Axiom) *For a load m_l in memory order m, m_s is the corresponding store defined by the value axiom if and only if $m_s \in \mathcal{S}$ and $\forall m_j \in \mathcal{S} : j \leq s$, where $\mathcal{S} \equiv \{\ m_j \mid S(m_j) \ \wedge\ addr(m_j) = addr(m_l) \ \wedge\ (j < l \ \vee\ m_j <_p m_l)\ \}$.*

In the executable description, this axiom is implemented in the procedure `Load_perf` as shown in Figure 2.5, which is executed atomically at the time of performance of a load instruction. `Load_perf` is written using a for-loop in the Murφ program to search for relevant stores in a reorder box; for the proof, the linear search is rewritten in recursive form so that automatic theorem provers can handle it.

Load_perf$(a : \mathit{address},\ p : \mathit{processor},\ n : \mathit{index}) =$
if $n > 0$
then let m_r be an instruction such that $\mathit{proc}(m_r) = p \ \wedge\ r_l(m_r) = n - 1$
/* instruction at index $n - 1$ of reorder box p */
if $S(m_r) \ \wedge\ \mathit{addr}(m_r) = a$
then m_r
else *Load_perf*$(a, p, n - 1)$
else $S_{M[a]}$ /* the store whose value is in the global memory */

Let m_l be a load instruction performed at time l. The procedure *Load_perf*$(\mathit{addr}(m_l), \mathit{proc}(m_l), r_l(m_l))$ is called to compute the value used to update the loaded register. We need to prove that the *Load_perf* returns the store specified in the value axiom.

We infer a property of global memory in the executable description. The executable model performs store instructions in memory order. Therefore, the store whose value is in memory location $M[a]$ at time l is the most recent store to A preceding m_l.

Axiom 2 (Global Memory) *At the time of performance of m_l, the global memory at address a, $M[a]$, contains the value of the store m_s such that $m_s \in \mathcal{S}_M$ and $\forall m_j \in \mathcal{S}_M : j \leq s$, where $\mathcal{S}_M \equiv \{\ m_j \mid S(m_j) \ \wedge\ \mathit{addr}(m_j) = a \ \wedge\ j < l\ \}$.*

To deal with the recursion in *Load_perf*, we prove an invariant. The following lemma says that *Load_perf*(a, p, n) executed at time l returns the store m_s to memory location a which is either the store performed most recently in memory order, or the store issued most recently before the instruction at index n by processor p (the same processor that issued the load), whichever is performed latest. In the lemma, $\mathcal{S}_n$ is the set of all stores to a that have been performed before l, or that are issued before the instruction $m_{n'}$ of index n.

Lemma 1 *If m_s is a store returned by Load_perf(a, p, n) at time l, then $m_s \in \mathcal{S}_n$ and $\forall m_j \in \mathcal{S}_n : j \leq s$, where $\mathcal{S}_n \equiv \{\ m_j \mid S(m_j) \wedge \mathit{addr}(m_j) = a \wedge (j < l \vee m_j <_p m_{n'})\ \}$ for $\forall m_{n'}$ such that $\mathit{proc}(m_{n'}) = p \ \wedge\ r(m_{n'}) = n$.*

Proof: The proof is by induction on n. When $n = 0$, the procedure returns $S_{M[a]}$, the store whose value is in the global memory location at time l. From Property 1, it can be proved that every instruction preceding $m_{n'}$ in program order has been performed by time l, that is, $m_j <_p m_{n'} \Rightarrow j < l$. This implies $\mathcal{S}_n = \mathcal{S}_M$, so Lemma 1 holds when $n = 0$ from Axiom 2.

As an induction step for $n > 0$, assume that Lemma 1 holds for $n - 1$. There are two cases depending on whether the if-condition in the procedure is true or false. If the instruction m_s at index $n - 1$ is a store to address a, then the procedure returns m_s; otherwise, it returns $\mathit{Load\_perf}(a, p, n - 1)$.

When the if-condition is true, $S(m_s) \wedge \mathit{addr}(m_s) = a$ must be true. Furthermore, $r_l(m_s) = n - 1 < r_l(m_{n'}) = n$, so by (2.3) we have $m_s <_p m_{n'}$. Hence, $m_s \in \mathcal{S}_n$.

Next, we show that $\forall m_j \in \mathcal{S}_n : j \leq s$. Let m_j be an arbitrary store in $\mathcal{S}_n$. If $j \leq l$, (2.2) implies $l \leq s$, so $j \leq s$. If $j > l$, by the definition of $\mathcal{S}_n$, we know that $m_j <_p m_{n'}$. So, by (2.3), $r(m_j) < r(m_{n'}) = n$. If $r(m_j) = n - 1$, then $m_j = m_s$. If $r(m_j) < n - 1$, we can see that $j < s$ by the following reasoning. By Theorem 1,

$$\forall\, j, s : S(m_j) \;\wedge\; S(m_s) \;\wedge\; \mathit{addr}(m_j) = \mathit{addr}(m_s) \;\wedge\; m_j <_p m_s \;\Rightarrow\; j < s. \qquad (2.4)$$

By (2.3) and (2.4), we obtain $j < s$, completing the case where the if-condition is true.

When the if-condition is false, the condition $\neg\, (S(m_r) \wedge \mathit{addr}(m_r) = a)$ implies $\mathcal{S}_n = \mathcal{S}_{n-1}$. From the induction hypothesis, the store m_s returned by $\mathit{Load\_perf}(a, p, n - 1)$ satisfies that $m_s \in \mathcal{S}_{n-1}$ and $\forall m_j \in \mathcal{S}_{n-1} : j \leq m_s$.

Therefore, Lemma 1 holds.□

Theorem 3 *The procedure $\mathit{Load\_perf}(\mathit{addr}(m_l), \mathit{proc}(m_l), r_l(m_l))$ returns the correct store for load m_l as defined in the value axiom.*

Proof: Substitute m_l for $m_{n'}$ in Lemma 1.□

Chapter 3

Reduction By Aggregating Distributed Transactions

This chapter presents a new approach to verify the correctness of protocols and distributed algorithms. The method compares a state graph of the implementation with a specification which is a state graph representing the desired abstract behavior. The steps in the specification correspond to atomic transactions, which are not atomic in the implementation.

The method relies on an *aggregation function*, which is a type of abstraction function that aggregates the steps of each transaction in the implementation into a single atomic transaction in the specification. The key idea in defining the aggregation function is that it must complete atomic transactions which have committed but are not finished.

This chapter illustrates the method on simple but nontrivial examples: a distributed list protocol and a majority consensus algorithm for multiple copy databases. Section 3.1 introduces the idea of aggregation and related work by others. Section 3.2 presents the verification method. Section 3.3 illustrates the method on a distributed list protocol, which is a fragment of distributed cache coherence protocols. Section 3.4 applies the method to another example, a majority consensus algorithm for multiple copy databases.

3.1 The Basic Idea

Protocols for distributed systems often simulate atomic transactions in environments where atomic implementations are impossible. This observation can be exploited to make formal verification of protocols and distributed algorithms using a computer-assisted theorem-prover much easier than it would otherwise be [55]. Indeed, the techniques described below have been used to verify safety properties of significant examples: the cache coherence protocol for the FLASH multiprocessor which is currently being designed at Stanford [30, 39], a majority consensus algorithm for multiple copy databases [64, 37], and a distributed list protocol [17].

The method proves that an implementation state graph is consistent with a specification state graph that captures the abstract behavior of the protocol, in which each transaction appears to be atomic. The method involves constructing an abstraction function which maps the distributed steps of each transaction to the atomic transaction in the specification. We call this *aggregation*, because the abstraction function reassembles the distributed transactions into atomic transactions.

This method addresses the primary difficulty with using theorem proving for verification of real systems, which is the amount of human effort required to complete a proof, by making it easier to create appropriate abstraction functions. Although our work is based on using the PVS theorem-prover from SRI International [53], the method is useful with other computer-assisted theorem-provers, or manual proofs.

Although finite-state methods (e.g. [51, 15, 34, 38]) can solve many of the same problems with even less effort, they are basically limited to finite-state protocols. Finite-state methods have been applied to non-finite-state systems in various ways, but these techniques typically require substantial pencil-and-paper reasoning to justify. Moreover, it is not obvious how to apply these extension to the examples of this chapter. Theorem-provers make sure that such manual reasoning is indeed correct, in addition to making available the full power of formal mathematics for proof, so they can routinely deal with problems that cannot yet be solved by any finite-state methods.

For our method to be applicable, the description must have an identifiable set of transactions. Each transaction must have a unique *commit point* [31], at which a state change first becomes visible to the specification. The most important idea in the method is that the aggregation function can be defined by completing transactions that have committed but not yet completed. In general, the steps to complete separate transactions are independent, which simplifies the definition of this function. In our experience, this guideline greatly simplifies the definition of an appropriate aggregation function.

The same idea of aggregating transactions can be applied to reverse-engineer a specification where none exists, because the specification with atomic transactions is usually consistent with the intuition of the system designer. We extract a specification model which performs transactions atomically at their commit steps in the implementation, and does nothing at other steps. The extracted specification provides an illusion that the transactions take effect instantaneously at the commit steps in the implementation.

If the extracted specification is not considered as a complete specification, or is not obviously correct, it can instead be regarded as a model of the protocol having an enormously reduced number of states. The amount of reduction is much more than other reduction methods used in model checking, such as partial order reduction, mainly because the reduced system is based on the only state variables relevant to the specification, without variables such as local states and communications buffers.

3.1.1 Related work

The idea of using abstraction functions to relate implementation and specification state graphs is very widely used, especially when manual or automatic theorem-proving is used [49, 41] (indeed, whole volumes have been written on the subject [14].) The idea has also been used with finite-state techniques [38, 16].

Ladkin, et al. [40] have used a refinement mapping [1] to verify a simple caching algorithm. Their refinement mapping hides some implementation variables, which may have the effect of aggregating steps if the specification-visible variables do not change. Our aggregation functions generalize on this idea by merging steps even when specification-visible variables change more than once.

A more limited notion of aggregation than ours is found in [43, 44], where a state function undoes or completes an unfinished process. The method only aggregates sequential steps within a local process, while our method aggregates steps across distributed components. The idea of an aggregated transaction has been used to prove a protocol for data base systems [58], where aggregation is obtained in a local process by showing the commutativity of actions from simple syntactic analysis.

In program verification, proofs can be simplified by pretending that a statement is atomic if its execution contains at most one access of a shared variable. This is the so-called "single-action rule" [48, 19, 45]. The single-action rule is generalized in [46]. This method classifies program statements as "left-movers" or "right-movers" depending on their commutativity properties. Using these properties, the statements are permuted to obtain a coarser-grained version of the program, for which safety properties can be checked.

We claim several advances over this earlier work. First, the problem is cast in the form of finding an abstraction function (the aggregation function) from an implementation state graph to a specification state graph. Abstraction functions have several advantages: for example, the functional composition of two abstraction functions is also an abstraction function. Second, our aggregation functions can hide some implementation variables, so the specification description can have fewer state variables than the implementation. This simplifies the proof, since many of the changes made by implementation steps are invisible to the specification. Finally, our method allows the use of an invariant on the implementation state space. Some implementation steps may only be commutative for states satisfying the invariant, so this increases the power of the verification method.

Cohen used an idea similar to aggregation to prove global progress properties by combining progress properties of local processes [12]. The idea of how to construct our aggregation function was inspired by a method of Burch and Dill for defining abstraction functions when verifying microprocessors [7].

3.2 The Verification Method

The verification method begins with a description in higher-order logic of the state graph of the implementation of a distributed computation, and a logical description of the state graph of the specification. The implementation description contains a set of *state variables*, which are partitioned into *specification variables* and *implementation variables*. The set Q of *states* of the implementation is the set of assignments of values to state variables. The description of the implementation also includes a logical formula defining the relation between a state and its possible successors. The relation is represented by a set of functions, $\mathcal{F} : 2^{Q \to Q}$, each of which maps a given implementation state to its next state. The implementation is nondeterministic if this set has more than one function.

The description of the specification state graph is similar. A specification state is an assignment of values to the *specification variables* of the implementation (implementation variables do not appear in the specification). Also, every state in the specification has a transition to itself. We call these *idle* transitions. The idle transitions are necessary for following implementation steps that do not change specification variables. We adopt the convention that components of the specification are primed, so the set of states of the specification is Q', the set of functions is $\mathcal{F}'$, etc.

The verification method is based on the usual notion of an abstraction function. The function, which we call *aggr*, maps implementation states to specification states and must satisfy a commutativity property

$$\forall q \in Q \quad \forall N \in \mathcal{F} \quad \exists N' \in \mathcal{F}' : \quad aggr(N(q)) = N'(aggr(q)). \tag{3.1}$$

The most interesting part of the method is how the aggregation idea is used to define this function.

The method relies on the notion that there is a set of *transactions* which the computation is supposed to implement, which are atomic at the specification level—meaning that a transaction occurs during a single state transition in the specification, but non-atomic at the implementation level. Indeed, the transactions in the implementation may involve many steps that are executed in several different components of the implementation.

The method requires that each transaction in the implementation have an identifiable *commit step*. Intuitively, when tracing through the steps of a transaction, the commit step is the implementation step that first causes a change in the specification variables. Implementation states that occur before the transaction or during the transaction but before the commit step are called *pre-commit states* for that transaction. The transaction is *complete* when the last specification variable change occurs as part of the transaction. The states after the commit step but before the completion of the transaction are called *post-commit states* for the transaction. A state where every committed transaction has completed is called a *clean* state.

Formally, all of the above concepts can be derived once the post-commit states are known for each transaction. The pre-commit states for the transaction are the states that are not post-commit; the commit step for an transaction is the transition from a pre-commit state to a post-commit state for that transaction; and the completion step is the transition from a post-commit state to a pre-commit state. A state is clean if it is a pre-commit state for *every* transaction.

An aggregation function consists of two parts: a *completion function* which changes the state as though the transaction had completed, and a *projection* which hides the implementation variables, leaving only the specification variables.

Once a purported aggregation function has been defined, the user must prove that it meets the commutativity requirement (3.1). The proof consists of a sequence of standard steps, many of which are or could be automated.[1] The initial $\forall q$ and $\forall N$ can be eliminated automatically by *Skolemization*,[2] which is substituting a new symbolic constant for q throughout (when we Skolemize in this presentation, we will not change the name of the quantified variable). This yields a subgoal of the form

$$(N \in \mathcal{F}) \Rightarrow \exists N' \in \mathcal{F}' : aggr(N(q)) = N'(aggr(q)). \tag{3.2}$$

The set of implementation steps $\mathcal{F}$ will often be defined with a logical formula of

[1] We base this comment on our use of the PVS theorem prover, but we believe the same basic method would be used with others.

[2] named after a logician, Thoralf Skolem.

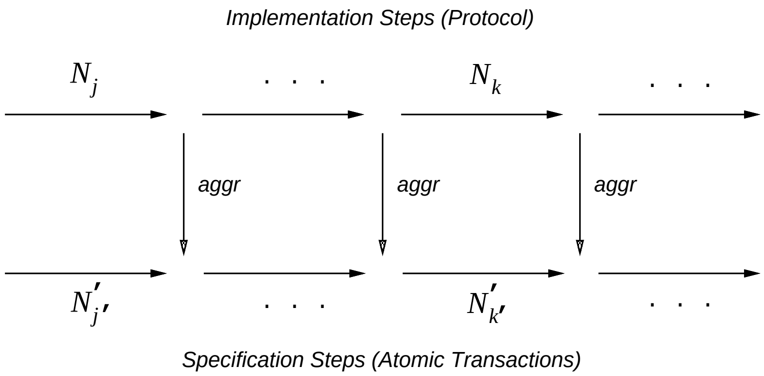

Figure 3.1: Step simulation using an aggregation function

the general form $\exists \boldsymbol{p} : N = N_1(\boldsymbol{p}) \vee N = N_2(\boldsymbol{p}) \vee \ldots$, where $\boldsymbol{p}$ is a tuple of parameters (perhaps ranging over an unknown number of components), and each N_j is a different kind of implementation step. Since the $\exists \boldsymbol{p}$ is in the antecedent of an implication, it can be Skolemized automatically, and the resulting disjunction can be proved by proving a collection of subgoals

$$(N = N_j(\boldsymbol{p})) \Rightarrow \exists N' \in \mathcal{F}' : aggr(N(q)) = N'(aggr(q)). \tag{3.3}$$

The existential quantifier $\exists N'$ can be eliminated by the user by manually substituting the definition of the appropriate function for N'. Given j and $\boldsymbol{p}$, the user must supply proper instantiations j' and $\boldsymbol{p}'$ such that the resulting subgoals

$$aggr(N_j(\boldsymbol{p})(q)) = N'_{j'}(\boldsymbol{p}')(aggr(q)) \tag{3.4}$$

are provable. Figure 3.1 shows the step simulation using the aggregation function.

The number of subgoals is equal to the number of transition functions in the implementation. In most cases, the required specification step $N'_{j'}(\boldsymbol{p}')$ is the idle transition; indeed, the only non-idle step is that which corresponds to the commit step in the implementation. We have no global strategy for proving these theorems, although most are very simple.

The above discussion omits an important point, which is that not all states are worthy of consideration. Theorem (3.1) will generally not hold for some absurd states that cannot actually occur during a computation. Hence, it is usually necessary to provide an *invariant* predicate, which characterizes a superset of all the reachable states. If the invariant is Inv, Theorem (3.1) can then be weakened to

$$\forall q \in Q \quad \forall N \in \mathcal{F} \quad \exists N' \in \mathcal{F}' : \quad Inv(q) \Rightarrow aggr(N(q)) = N'(aggr(q)). \tag{3.5}$$

In other words, $aggr$ only needs to commute when q satisfies the Inv.

Use of an invariant incurs some additional proof obligations. First, we must prove that the initial states of the protocol satisfy Inv, and second, that the implementation transition functions all preserve Inv.

For a proof by aggregation to be meaningful, the user must appropriately identify the transactions, and must associate commit points in the implementation with the proper transactions. For example, an aggregation function that maps all implementation states to a single specification state, and all implementation steps to the idle specification step could be easily conducted, but would not be meaningful [22]. A less likely way to get meaningless results would be to map commit steps to the wrong transactions.

3.3 The Distributed List Protocol

We illustrate the concepts of the previous section on a small but somewhat nontrivial example, which we call the "distributed list protocol" [17]. The protocol is an abstraction of part of a multiprocessor cache coherence protocol, which maintains a singly-linked list of processors which share a cache line.

The finite-state techniques we have applied do not scale especially well for this protocol. We have tried explicit state methods (specifically our Murφ verifier) with techniques such as symmetry reduction, reversible rule reduction [36], and special verification methods for parameterized families of protocols, as well as BDD-based techniques [6]. None of these methods has allowed us to verify systems with more than about 5 list cells, because we do not have a good way of compressing or abstracting states containing linked lists. However, using the method described here, we have verified the protocol for arbitrary or even infinite numbers of list cells.

3.3.1 The transactions of the protocol

The protocol maintains a circular, singly-linked list of list cell processes, called *cells*. There is a special process called *head cell* which is always in the list. Cells not in the list may request to be added to the list, and cells in the list may request to be removed. The cells communicate by sending messages over a network that is reliable, but does not preserve the sending order of messages.

The network is modeled as a state variable which is a set of messages. Every message used in the protocol has a field *src* that contains the index of the sending cell, and a field *dst* that contains the address of the cell to which it was sent. Additional fields, *old* and *new*, are used in some message types to hold the indices of other cells.

Every cell has state variables for its control state, *state*, and the index of the next cell in the list *next*. When a cell is not in the list, its *next* variable contains the index of the cell itself. The *next* variable of each cell is a specification variable, because the list structure is important for the correctness of the protocol. The variable *state* is an implementation variable. There are also variables associated with the cells to hold messages that are in transmission.

A cell, other than the head cell, can perform two types of transactions: *add* and *delete*. There is an add_i transaction and a $delete_i$ transaction for each cell i in the protocol (i.e., if there are n cells, there are $2n$ transactions, not 2 transactions). In the following, let i be the index of the cell initiating the transaction.

An add_i transaction can occur when cell i is not in the list, and when the state of cell i is *normal*. The cell i will be added at the head of the list. The transaction consists of three steps:

1. Cell i sends an *add* message to the head cell; and cell i changes its state to *w_head* ("wait for *head* message").

2. The head cell sends a *head* message containing the *next* value of the head cell to cell i. Then the head cell stores i in its *next* variable.

3. When cell i receives the *head* message, it stores the value in the message into its *next* variable. Cell i then changes its state back to *normal*.

The specification state variables consist of the collection of *next* pointers of the cells. The add_i transaction in the specification inserts cell i at the front of the list, updating the *next* variables of the head cell and cell i in a single atomic step.

The commit step for the add_i transaction occurs in step 2, which is the first point where a specification variable is modified (*next* of the head cell). Step 1 only modifies implementation variables *state* and *network*, so it begins and ends in pre-commit states for add_i. The state between step 2 and 3 is a post-commit state. Step 3 completes the transaction; it is the point where a specification variable changes for the last time in the transaction. Hence, the state following step 3 is again a pre-commit state for add_i.

A $delete_i$ transaction can occur when a cell's *next* points to a cell other than i (meaning i is in the list) and its *state* is *normal*. The problem with deleting in a distributed singly-linked list is that there is no easy way for cell i to determine its predecessor in the list, which is unfortunate since *next* of the predecessor must be changed to point to the *next* of cell i.

The solution to this problem is to have another message *pred* which circulates around the list at all times[3]. When cell i receives the *pred* message, it can determine its predecessor by examining the *src* field of the message. So, the steps of the $delete_i$ transaction are:

1. Cell i changes its *state* to *w_pred* ("wait for *pred* message").

2. When cell i receives a *pred* message, it sends a *chnext* message ("change next") to the source of the *pred* message, which is usually the predecessor of i in the list. The *chnext* message has i in its *old* field and the *next* of cell i in its *new* field. Cell i changes *state* to *w_delack* ("wait for delete-acknowledgment").

3. When a cell j receives the *chnext* message there are several possible cases. The subtleties of these rules handle difficult scenarios, such that new cells are added

[3]There is another version of distributed list protocol, in which *pred* message is generated only when necessary.

between cell i and the predecessor, after cell i's receipt of the *pred* message and before the receipt of the *chnext* message.

(a) If the *state* of cell j is not *normal* or *w_pred*, the *chnext* message remains in the network (progress occurs when some other message arrives at cell j).

(b) Otherwise, if the *old* field of the message matches the *next* variable of cell j, the cell changes its *next* to be the *new* of the *chnext* message (*next* of i). Then cell j sends a *delack* message to cell i (*src* of the *chnext* message). Cell j then sends a *pred* message to its *next* cell.

(c) Otherwise, cell j forwards the *chnext* message to its *next* cell. In this case, the cell receiving the *chnext* message is the head cell and one or more new cells were inserted at the head of the list while cell i was being deleted, so the predecessor of cell i is now somewhere further down the list. The true predecessor will eventually receive the *chnext*, causing the case (b) to occur.

4. When cell i receives a *delack*, it changes its *next* variable to i, and changes *state* to *normal*.

The commit step of the $delete_i$ transaction is in case (b) of step 3 above. Step 3 may be repeated several times because of case (c) before a commit occurs, so a state immediately following step 3(c) is a pre-commit state. Step 4 completes the transaction.

The specification handles the delete transaction atomically by removing cell i from the list in the obvious way: it sets the *next* of the predecessor of i to the *next* of i, then sets *next* of i to i.

The *pred* message circulates around the list constantly except when it temporarily disappears during processing of a *chnext* during a delete transaction, so each cell has rules for propagating it. However, processing a *pred* message never affects a specification variable, so there are no transactions associated with it. It is necessary to reason about the processing of *pred* messages during the proof of invariants (discussed below), and also for liveness properties (which are not discussed in this thesis).

Step	**Condition**	**Action**
Initiate Add	$i \neq$headptr $\wedge$ next[i]=i $\wedge$ state[i]=normal	Send *add*⟨src=i⟩ to headptr state[i] := w_head
Process *add*	*add* sent to headptr	Send *head*⟨new=next[headptr]⟩ to *add.src* next[headptr] := *add.src*
Process *head*	*head* sent to i	next[i] := *head.new* state[i] := normal
Initiate Delete	$i \neq$headptr $\wedge$ next[i]$\neq i$ $\wedge$ state[i]=normal	state[i] := w_pred

Process *pred*	*pred* sent to i	if state[i]=normal:
		 Send *pred*⟨src=i⟩ to next[i]
		 if state[i]=w_pred:
		 state[i] := w_delack,
		 Send *chnext*⟨old=i, new=next[i]⟩
		 to *pred.src*

Process *chnext*	*chnext* sent to i $\wedge$ *chnext.old*$\neq$next[i] $\wedge$ state[i]$\in${normal,w_pred}	Send *chnext* to next[i]
Process *chnext*	*chnext* sent to i $\wedge$ *chnext.old*=next[i] $\wedge$ state[i]$\in${normal,w_pred}	next[i] := *chnext.new* Send *delack* to *chnext.old* Send *pred*⟨src=i⟩ to *chnext.new*
Process *delack*	*delack* sent to i	next[i] := i, state[i] := normal

Table 3.1: Formal description of Distributed List Protocol

The above description of the protocol traces through individual transactions. It is easier to make sure that a description is complete if the behavior is described for each component, not each transaction (and, indeed, the above description is not complete). Table 3.1 gives the rules of cell behavior in pseudo-code on a per-cell basis.

In the table, the action of a step is executed if its condition holds. Each process consumes the message that triggers it. A message consists of a record with fields *src, new, old.* When a message is created, we use $m\langle f{=}a'\rangle$ to denote that message m has value a' for its record field *f.* We use $m.f$ to refer to the value of field f in message m. State variables for cells are kept in the arrays, *state* and *next.*

3.3.2 The aggregation function

Here, we define the aggregation function *aggr* for the distributed list example. The key question is how to complete all committed transactions in the current state, especially since the number of cells, and hence the number of committed transactions, is unknown. The general strategy, which has worked for our larger examples as well, is to define a per-component completion function, which is then generalized to a completion function for all of the cells in the system. This is possible because the post-commit steps of different nodes are generally independent.

It is quite simple to complete a committed transaction for a particular cell. If a *head* message destined for cell i exists, an add_i transaction must be completed by simulating the effect of cell i processing the *head* message it receives at the end of the transaction. This processing changes *next* to point to the value of the *new* field in the message. Changes to implementation variables, such as removing messages from the network, can be omitted from the completion function as they do not affect the corresponding specification state. All of this computation is done solely in cell i, without the involvement or interference of other cells. If there is a *delack* message for cell i, a $delete_i$ transaction must be completed by setting *next* to i. Otherwise, the completion function does nothing. Figure 3.1 shows completion of an add transaction.

It is easy to generalize the completion function for one cell to a completion function for all of the cells because the completions do not interact. The global implementation state is an array of cell state records, indexed by the cell indices. Let $cc(q[i])$ be a completion function for cell i, which modifies the state variables for i in the record $q[i]$, and returns a new record of the state variables as modified by the completion of the transaction.

If $cc(q[i])$ completes committed transactions on node i, the completion function for all nodes is $\lambda q.\lambda i.cc(q[i])$. When this function is supplied a state q, it returns $\lambda i.cc(q[i])$,[4] which is an array of the completed node states, i.e., the desired clean global state. The aggregation function is simply the completion function, followed by a projection which eliminates all implementation variables.

3.3.3 Extracting specification

Reverse engineering of a specification can be illustrated on the distributed list protocol. Indeed, we had to do this because there exists no formal specification. Given only an implementation description, the first step is to identify the specification variables. In the distributed list protocol, we decided that they were the *next* variables for the cells. The next step is to trace through a transaction: 1) concatenating the implementation steps, 2) simplifying by substituting values forward through intermediate assignments, 3) eliminating statements that only change implementation variables.

For an add_i transaction in the protocol, the sequence of steps is "initiate add," "process *add*," and "process *head*." The result obtained by the procedure is

$$\begin{aligned} \mathit{Atomic\_Add}(i)\text{: } & \text{if } i \neq \text{headptr} \wedge \text{next}[i] = i \text{ then} \\ & \text{next}[i] := \text{next}[\text{headptr}];\ \text{next}[\text{headptr}] := i. \end{aligned}$$

Similarly, a $delete_i$ transaction corresponds to the sequence of steps, "initiate delete," "process *pred*," "process *chnext*," and "process *delack*." The atomic transaction obtained by aggregation is

$$\begin{aligned} \mathit{Atomic\_Delete}(c, i)\text{: } & \text{if } i \neq \text{headptr} \wedge \text{next}[i] \neq i \wedge \text{next}[c] = i \text{ then} \\ & \text{next}[c] := \text{next}[i];\ \text{next}[i] := i. \end{aligned}$$

[4] The notation may be a bit confusing. $\lambda i.cc(q[i])$ is a function, which when applied to a particular value of i, say i_0, returns $cc(q[i_0])$, which is the completed state for node i_0. This is effectively the same as indexing into an array of completed node states.

Implementation step at node i	Specification step
Initiate_Add(i)	ε
Process_add(i, add)	$Atomic\_Add(add.src)$
Process_head(i, $head$)	ε
Initiate_Delete(i)	ε
Process_pred(i, $pred$)	ε
Process_chnext_forward(i, $chnext$)	ε
Process_chnext_commit(i, $chnext$)	$Atomic\_Delete(i,\ chnext.old)$
Process_delack(i, $delack$)	ε

Table 3.2: Correspondence of specification steps with implementation steps in the distributed list protocol

With the two atomic transactions and idle steps in the specification, we instantiate the subgoals, Equation (3.4), for each implementation steps. The proper instantiation for the proof is shown in Table 3.2.

3.3.4 The invariant

The proofs of the subgoals (3.4) corresponding to each row in Table 3.2 are simple. PVS can handle them almost automatically. Among the eight subgoals, four have been proved automatically for any state q. The remaining four subgoals cannot be proved without first proving an invariant. Our invariant includes the following properties.

- The head cell is always in *normal* state.
- If a cell is in *normal* or *w_pred* state, there is no *add* message from the cell, *delack* message to the cell, or *chnext* message with *old* field equal to the cell.
- If there is an *add* message from or *head* message to a cell i, then the *next* of the cell is i.
- In a *chnext* message, the *next* of the cell contained in the *old* field of the message must be the same as the *new* field of the message.
- There is at most one message in the network for each transaction currently in progress, and there must be no more than one *pred* message in the network.

For example, the second subgoal from Table 3.2 can be proved only if the last assertion is true.

The only manual step occurs when proving subgoals of the form $(\forall j : Inv(j)) \Rightarrow Q(i)$, where i is a cell index, which requires eliminating the $\forall j$ by substituting i for j to obtain $Inv(i) \Rightarrow Q(i)$; the proof can be completed automatically.

Part of the reason that the proof is simple is that we have chosen to represent the network in a non-obvious way. We observe that there is at most one message pertaining to any particular transaction at any time. So the network can be represented with one variable per cell (sometimes associated with the source, sometimes with the destination), plus a single variable for the *pred* message. Hence, instead of proving that there is only one message of a certain type in the network for cell i at any time, we register an error whenever a message in a variable for the network is about to be overwritten, and verify that no error occurs. The description can read a message by accessing the variable instead of choosing a message from a set of messages, which is a bit more difficult to deal with in PVS.

3.4 Majority Consensus Algorithm for Distributed Multiple Copy Databases

In this section, we apply the verification method to another example, a majority consensus algorithm for distributed multiple copy databases [64]. This algorithm is based on a commit-point while updating distributed database so our aggregation method is suitable for the verification. This algorithm has also been proved in [37].

3.4.1 The algorithm

The algorithm intends to query and update a distributed database while ensuring consistency. The data in the database is replicated so that each site contains a copy of each data element in the database. The database copy at each site is accessible only through a database managing process (DBMP) which resides at that site. Application processes (APs) submit requests to query the current value of a data element or to update a data element. Each access to the database is completed by a DBMP acting on behalf of the initiating AP.

A problem is to ensure that the database is consistent. The consistency maintained by the algorithm requires that different copies of data elements contain the same value except for possible delays in the propagation of changes, and preserve some relations among data elements within a database. Since several conflicting requests for updates to a data element can be issued by different APs, the algorithm must choose which request to grant in consistent ways.

For instance, consider a simple database duplicated at sites A and B that includes data elements x, y, and z, which all initially have the value 1 in both copies. Further assume that the relation, $x+y+z=3$, must be preserved for the database. Consider two updates U1 and U2:

U1: $x := 0,\ y := 2$

U2: $y := 0,\ z := 2,$

each of which preserves the relation based on the initial database state. If U1 and U2 are both applied, regardless of the order of application, the relation of the database will be violated. Hence one of the requests must be rejected in order to preserve the relation of the database. In other words, the update request that gets rejected must be refused because it is based on information made obsolete by the request that gets accepted. The other requirement of consistency will be destroyed if update U1 is accepted in site A and update U2 is accepted in site B. Therefore, all sites should make the same decision for concurrently initiated conflicting updates.

The algorithm uses a voting scheme to decide which update request to grant. The main idea of the algorithm is to pass each update request around among the DBMPs. Each DBMP votes whether to accept the request or not. An update request is granted only if a majority of the DBMPs accept it; otherwise it is rejected. Intuitively, this procedure ensures consistency because for any two consecutive update requests that are granted, there should be at least one DBMP that accepts both of them in some order, ensuring that they do not conflict.

The database consists of a collection of named elements. Each named element has a value and a timestamp associated with it. The timestamp of an element represents the time that the element received its current value. Timestamps are used in two ways: 1) during updates synchronization to ensure the preservation of internal exclusion locks, and 2) in the procedure followed by a DBMP when it applies an update to its database copy.

To query the database, an AP sends a query request to a DBMP. The DBMP acts upon the request by querying its copy of the database and returning the results to the requesting AP.

Performing updates is somewhat more involved. In general, an AP initiates an update by first performing a computation to generate new values for certain database elements using database values obtained by one or more queries, and then submitting an update request to a DBMP which cooperates with the other DBMPs to perform the update. The update procedure can be decomposed into the following sequence of steps.

1. *Query Database*: The AP queries the database to obtain data element values to use in its update computation. The DBMP responding to the query supplies the value stored in its copy of the database as well as the timestamp of the value.

2. *Submit Update Request*: The AP computes and constructs an update request, then submits the request to a DBMP.

3. *Synchronize Update*: The DBMP set cooperates to decide to accept or reject the request. Each DBMP participating in the decision executes the same voting procedure, which is explained below in detail. If the request is accepted, the decision is sent to all sites so that it can be reflected to all the replicated copies.

4. *Apply Update*: When a DBMP receives an accepted decision, it updates its copy if the update is not already obsolete.

5. *Notify AP*: A DBMP informs the AP how the request was resolved.

The consensus algorithm consists of a voting rule and a resolution rule, which constrain DBMP behavior in step 3 so that no two conflicting concurrent update requests can be accepted. They insure that mutual exclusion is achieved for possibly conflicting concurrent updates.

The following voting rule is the basis for concurrency control.

3a. Compare the timestamps for the old data in the request with the corresponding timestamps in the local database copy.

3b. Vote "reject" if the data in the request is obsolete.

3c. Vote "okay" and mark the request as pending if the data is current and there is no other pending request.

3d. Vote "pass" if the data is current but the request has lower priority than the pending one. (Update with more recent timestamp of the old value has higher priority; if the timestamps of the old values are same, update with more recent timestamp of the new value has higher priority.)

3e. Otherwise, defer voting.

After voting, a DBMP uses the following request resolution rule to check whether its vote resolved the request. The basic idea is that the request should be accepted if a majority of the DBMPs have voted okay.

3f. If a vote is "okay" and majority consensus exists, accept the request and notify all DBMPs and the AP that the request is accepted.

3g. If a vote is "reject," reject the request and notify all DBMPs and the AP that the request is rejected.

3h. If a vote is "pass" and majority consensus is no longer possible, reject the request and notify all DBMPs and the AP that the request is rejected.

3i. Otherwise, forward the request with accumulated voting results to other DBMPs which have not voted.

3.4.2 Finding an aggregation function

The algorithm implements a single kind of transaction *update*. However, there can be a number of such transactions being processed in the system, initiated by each AP.

The specification model may query and update multiple copies of the database *atomically* satisfying mutual consistency. Therefore, the specification consists of an atomic transaction *Query & Update* and idle steps. A natural choice is to consider the specification variables to be the local copies of the distributed database.

The transaction is implemented in a number of steps as described above. The commit step of the transaction is the step which resolves the request as accepted.

Our aggregation function should finish all the committed transactions in the system. For this algorithm, finishing the committed transactions corresponds to the steps which process the accepted decisions of requests by updating the local copies. In terms of a specific database site i, the aggregation function simulates processing all the accepted decisions sent to DBMP i. Although there are possibly a number of accepted decisions pending to be processed, the aggregation function is simply equivalent to updating the local database of i to the value with the most recent timestamp, because processing an accepted decision in step 4 updates the copy, only if the decision is more recent than the local copy.

3.4.3 Assignments of specification steps

The atomic transaction *Query & Update* corresponds to the commit step "Decide Accept" in the implementation. Simply, idle steps correspond to all the other implementation steps. Table 3.3 shows the correspondences between the implementation and specification steps, each of which represents a subgoal in the proof.

Rejected requests do not modify any copy of the database. Therefore, a rejected update maps to idle step in the specification.

Implementation Step	Specification Step
Query	ε
Submit Update	ε
Vote Okay	ε
Vote Reject	ε
Vote Pass	ε
Decide Accept	*Query & Update*
Decide Reject	ε
Process Decision	ε

Table 3.3: Correspondence of specification steps with implementation steps in the majority consensus algorithm

3.4.4 Proof

The eight correspondences except for the steps "Decide Accept" and "Process Decision," are proved without any invariant because the steps do not change the specification variables. The remaining two correspondences can be proved with the following properties in the invariant. Let $max(i)$ be the value with the maximum timestamp of all accepted decisions sent to DBMP i and the local copy in DBMP i.

- For every site i, $max(i)$ should be the same.
- When an update request is accepted, the old value in the update is same as $max(i)$.

Using the invariants, the algorithm is proved to conform to the specification consisting of the atomic transaction and idle steps.

3.5 Summary

To make it easy to apply the verification procedure, we list typical sequences of implementation steps in Table 3.4. The three types of sequences which modify distributed data are commonly observed in protocols. For each type of concrete transaction, correspondence of atomic transactions are shown as well as its aggregation

Transaction		Implementation step	Specification step	In *aggr*
TYPE-I	I_1	Send Request	ε	
	I_2^*	Process Request; Send Reply	Atomic Transaction	I_3^*
	I_3^*	Process Reply	ε	
TYPE-II	I_1	Send Request	ε	
	I_2	Forward Request	ε	
	I_3^*	Process Request; Send Reply	Atomic Transaction	I_4^*
	I_4^*	Process Reply	ε	
TYPE-III	I_1^*	Update; Send Request	Atomic Transaction	I_2^*
	I_2^*	Process Request	ε	

Table 3.4: Applying the aggregation method to typical transactions in distributed protocols

function *aggr*. Protocol actions which modify specification variables are marked with stars.

Chapter 4

Reasoning About Cache Coherence Protocols

This chapter presents verification of a directory-based cache coherence protocol developed for the Stanford FLASH multiprocessors. The protocol is briefly described, and a finite-state method is used to verify some properties of the protocol.

Next, the aggregation method presented in Chapter 3 is applied to verify the protocol with arbitrary numbers of processors. The coherence protocol consisting of more than a hundred different kinds of implementation steps has been reduced to a specification with six kinds of atomic transactions. Based on the reduced behavior, it is very easy to prove crucial properties of the protocol including data consistency of cached copies at the user level. Moreover, the reduced model allows us to write a simple executable memory model of the protocol. The aggregation method is also used to prove that the reduced protocol satisfies a desired memory consistency model.

Section 4.1 discusses various methods applied to the verification of cache coherence protocols. Section 4.2 describes the FLASH cache coherence protocol in two ways: in terms of transactions, and per-node based steps. Section 4.3 presents verification of the protocol using the Murφ finite-state verifier. Section 4.4 applies our aggregation method to the same problem. Using the reduced model, section 4.5 proves that a specific mode supported by the protocol implements sequential consistency memory model. Finally, section 4.6 presents a simple executable memory model for each mode in the protocol to help the users to analyze their programs running in the FLASH multiprocessors.

4.1 Verification of Cache Coherence Protocols

In shared-memory multiprocessor architectures, cache coherence protocols maintain consistency of multiple copies of cached data. The protocols control a number of readable and writable copies of each memory line for multiprocessors. Modification of one copy of a datum may require updating of other copies to maintain consistency among them. Several coherence protocols have been proposed for distributed multiprocessor architectures but few are formally verified [4, 5, 67, 9, 47].

Formal verification is very important because there could be subtle design errors as the complexity of protocols increases, especially for large-scale multiprocessor systems. Finite-state methods (e.g. [15, 21, 34, 38]) have been used to validate some cache coherence protocols, including Gigamax [50], IEEE Futurebus+ [11], and SCI cache coherence protocol [63]. Finite-state methods can solve many verification problems with little effort. However, they are basically limited to finite-state protocols. The finite-state techniques we have applied do not scale especially well for the implementation-detailed cache coherence protocols. For example, Murφ verifier can barely handle the protocols with 3 processors and 2 memory lines, using 100 megabytes of memory in the process.

Symbolic state models proposed by Pong and Dubois [60, 59] use symbolic states which abstract away from exact number of configurations of replicated identical components by recording only whether there are zero, one, or more than zero replicated components. However, Pong & Dubois' method is still limited to finite size systems for protocols involving linked lists or data forwarding, and there remains a specification problem of the protocol as in model checking: It is not easy to find a set of properties, say in temporal logic or in their notation, which completely describes the correct behavior of the protocols. Moreover, Pong & Dubois' method requires the user to write an abstract description of the protocol to be verified, which raises another verification problem: Are the abstract description and the actual protocol are equivalent?

Theorem-proving can avoid the above problems owing to the generality obtained by the use of logic as a formalism. It supports verification of non-finite systems as well as hierarchical verification. However, the major problem with theorem proving is that considerable amount of human effort is required. Consequently, previous theorem proving approaches have not been able to verify a problem of the scale of a full multiprocessor cache coherence protocol [40]. However, the aggregation functions introduced in Chapter 3 reduce the required effort to a much more reasonable level.

4.2 FLASH Cache Coherence Protocol

This section informally describes the cache coherence protocol used in the Stanford FLASH multiprocessor [39, 30]. The cache coherence protocol is directory-based so that it can support a large number of distributed processing nodes. Each cache line-sized block in memory is associated with *directory header* which keeps information about the line. For a memory line, the node on which that piece of memory is physically located is called *home*; the other nodes are called *remote*. The home maintains all the information about memory lines in its main memory in the corresponding directory headers.

The system consists of a set of nodes, each of which contains a processor, caches, and a portion of global memory of the system. The distributed nodes communicate using asynchronous messages through a point-to-point network. The state of a cached copy is in either *invalid*, *shared* (readable), or *exclusive* (readable and writable).

4.2.1 Informal description of the protocol

If a read miss occurs in a processor, the corresponding node sends out a GET request to the home (this step is not necessary if the requesting processor is in the home). Receiving the GET request, the home consults the directory corresponding to the memory line to decide what action the home should take. If the line is *pending*, meaning that another request is already being processed, the home sends a NAK (negative acknowledgment) to the requesting node. If the directory indicates there is a dirty copy in a remote, then the home forwards the GET to that node. Otherwise, the home grants the request by sending a PUT to the requesting node and updates the directory properly. When the requesting node receives a PUT reply, which returns the requested memory line, the processor sets its cache state to *shared* and proceeds to read.

For a write miss, the corresponding node sends out a GETX request to the home. Receiving the GETX request, the home consults the directory. If the line is *pending*, the home sends a NAK to the requesting node. If the directory indicates there is a dirty copy in a third node, then the home forwards the GETX to that node. If the directory indicates there are shared copies of the memory line in other nodes, the home sends INVs (invalidations) to those nodes. At this point, the protocol depends on which of two modes the multiprocessor is running in: EAGER or DELAYED. In EAGER mode, the home grants the request by sending a PUTX to the requesting node; In DELAYED mode, this grant is deferred until all the invalidation acknowledgments are received by the home. If there are no shared copies, the home sends a PUTX to the requesting node and updates the directory properly. When the requesting node receives a PUTX reply which returns an exclusive copy of the requested memory line, the processor sets its cache state to *exclusive* and proceeds to write.

During the read miss transaction, an operation called sharing write-back is necessary in the following "three hop" case. This occurs when a remote processor in node R_1 needs a shared copy of a memory line an exclusive copy of which is in another remote node R_2. When the GET request from R_1 arrives at the home H, the home consults the directory to find that the line is dirty in R_2. Then H forwards the GET to R_2 with the source of the message *faked* as R_1 instead of H. When R_2 receives the forwarded GET, the processor sets its copy to *shared* state and issues a PUT to R_1. Unfortunately, the directory in H does not have R_1 on its sharer list yet and the main memory does not have an updated copy when the cached line is in the shared state. The solution is for R_2 to issue a SWB (sharing write-back) conveying the dirty data to H with the source *faked* as R_1. When H receives this message, it writes the data back to main memory and puts R_1 on the sharer list. Figure 4.1 shows the processing

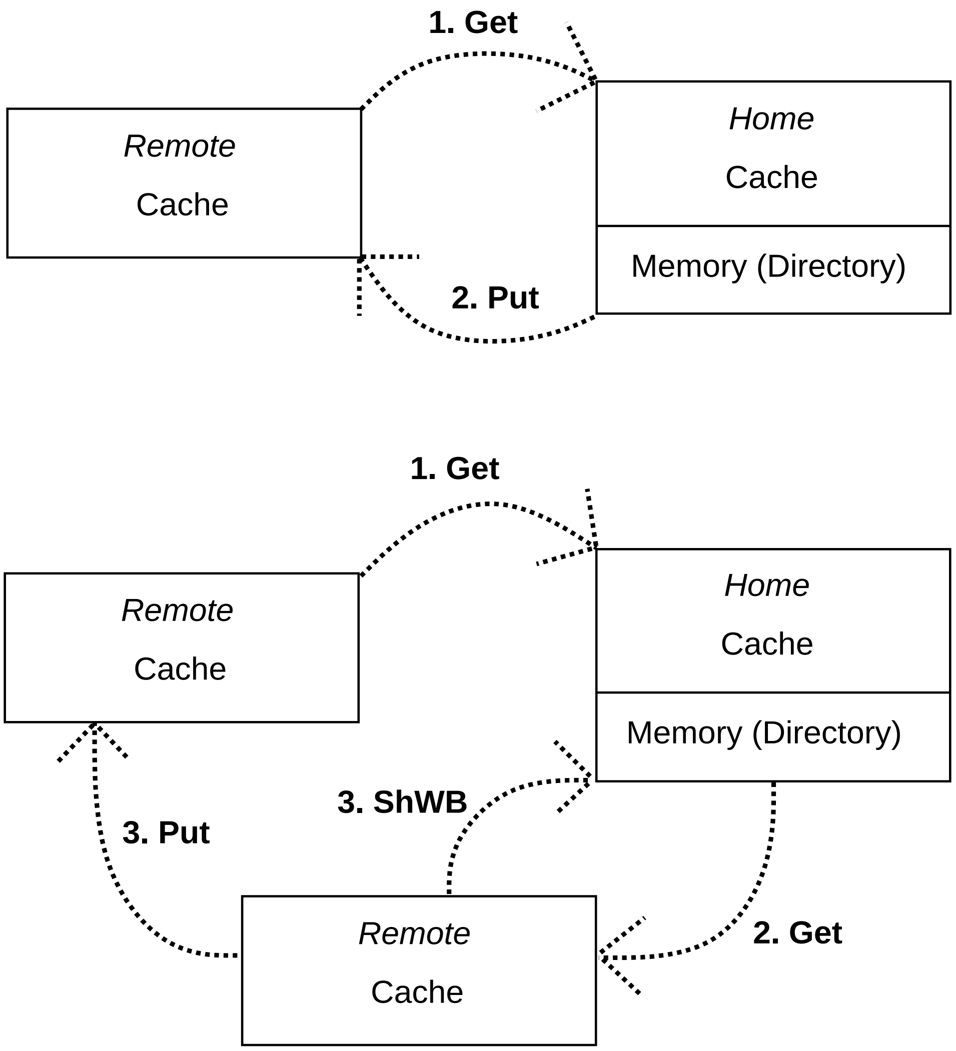

Figure 4.1: Processing a read miss (GET request) in the FLASH protocol

a read miss in the protocol.

When a remote receives an INV, it invalidates its copy and then sends an acknowledgment to the home. There is a subtle case with an invalidation. A processor which is waiting for a PUT reply may get an INV before it gets the shared copy of the memory line, which is to be invalidated if the PUT reply is delayed. In such a case, the requested line is marked as invalidated, and the PUT reply is ignored when it arrives.

A valid cache line may be replaced to accommodate other memory lines. A shared copy is replaced by issuing a replacement hint to the home, which removes the remote from its sharers list. An exclusive copy is written back to main memory by a WB (write-back) request to the home. Receiving the WB, the home updates the line in main memory and the directory properly.

The above description of the protocol traces through individual transactions. However, the formal description of the protocol is written for each component, not each transaction, to make sure that the description is complete. Appendix A presents an English version of the formal description of the FLASH protocol in EAGER mode.

4.2.2 Detailed description of the protocol

Each cache line-sized block in main memory is associated with a directory header which keeps information about the line. The directory header consists of several boolean flags: Local, Dirty, Pending, Head_Valid, and List; and two pointers to other nodes: Head_Pointer and Sharer_List; and a number of sharers in Real_Pointers. The Local bit indicates if the local processor contains a cached copy of the line in either shared or exclusive state. The Dirty bit is set if the home thinks that there is a dirty copy of the line in the system. The Pending bit is set if the current request for the memory line is being processed by a third node. The Head_Valid bit indicates whether the Head_Pointer contains a valid pointer to a node. The Head_Pointer entry is simply a cache pointer that is stored in the directory header as an optimization. It keeps a pointer to a remote cache with a dirty copy if there is one, or one of the nodes with a shared copy. The List bit indicates whether Sharer_List contains one or more pointers. Sharer_List is represented abstractly as a set[1] of pointers to the nodes that have a shared copy of the memory line. Real_Pointers contains the count of the number of sharers in the Sharer_List. This count excludes the Head_Pointer and is mainly used to count invalidation acknowledgments.

The FLASH protocol consists of a set of rules which are called *handlers*. Each handler is prefixed with NI (Network Interface) or PI (Processor Interface) to indicate where the requests are generated from. PI handlers are initiated by a requesting processor and NI handlers are initiated by a message from the network. The additional notation 'Local' or 'Remote' indicates whether the processing node is the home of the requested memory address or not. In the following, some handlers of the protocol for processing a read miss are listed; the rest of the handlers are shown in Section 4.7.

- **PI.Local.Get:** this handler describes actions of the home when the local processor needs a shared copy of a memory line. If Pending[2], the local processor is NAKed. Otherwise, if Dirty, the home sends a GET request to Head_Pointer and Pending is set. Otherwise, the data in main memory is copied into the local cache (in *shared* state) and Local is set.

- **NI.Remote.Get:** this handler describes actions of a remote receiving a GET request. If the cached data is in the *exclusive* state, it is changed to *shared* and the node sends a PUT reply to the source (and also SWB to the home if the source is not the home). Otherwise, the node sends a NAK to the source and a NAKC to the home.

- **NI.Remote.Put:** The shared copy is put into the cache.

- **NI.SharingWriteback:** this handler describes actions of the home receiving a SWB. Dirty and Pending are reset, List is set, Real_Pointers is incremented, the source is added to Sharer_List, and the data is written back into main memory.

[1]The FLASH protocol uses a linked list for sharers (within the home) by dynamic pointer allocation.

[2]I.e., the Pending bit is set in the directory.

4.3 Verification Using A Finite State Method

Most present verification of cache coherence protocols uses finite-state methods. The methods model a cache coherence protocol as a finite state machine and check the correctness of the protocol by exploring the state space. This section presents the application of the Murφ system to verification of the FLASH protocol. The description language Murφ was explained in Section 2.4.

4.3.1 Data structure and global state

Some of the state variables and the corresponding types used in the description of the FLASH protocol are shown in Figure 4.2. Type `Header` is a record consisting of all the information in directory headers. A processor state `ProcState` consists of state variables representing caches, and main memory and their directory headers in the node. The messages are represented by two types: `Request` and `Reply`.

The global state of the system consists of a state variable `Procs` which is an array of processor states for each node, and state variables for the network. The point-to-point network is modeled as an array of incoming message queues, one for each node; each queue is again an array of messages. The protocol maintains requests and replies in separate queues, so `QNet` and `PNet` models them respectively. The state variable `QNet` is an array of queues for request messages, and `PNet` is an array of queues for replies.

4.3.2 Transition relations in rulesets

Each handler of the protocol in Section 4.2.2 is written as a Murφ rule, which is embedded in a ruleset. Each ruleset is parameterized to describe the action for arbitrary processors. To have a complete description of the system, we must also describe the behavior of processors which initiate a request when a cache miss occurs. Such actions are included in PI handlers.

For instance, the ruleset corresponding to PI.Local.Get is shown in Figure 4.3. The ruleset is a direct translation of the handler. The condition `Qspaces` of the

```
Type
  Header : Record
      Pending : Boolean;          -- is an operation pending on this line ?
      Local   : Boolean;          -- is the line locally cached ?
      Dirty   : Boolean;          -- is the data held dirty ?
      Head    : Boolean;          -- is the head pointer used ?
      List    : Boolean;          -- is the head link used ?
      HPtr    : Proc;             -- head pointer : processor id
      Real    : 0..NumProcessor;  -- number of linked pointers
  End;

  ProcState: Record
    Cache : Array[Proc] of Array[Address] of
              Record
                Wait  : Boolean;      -- waiting for cc_put(x)
                Invalid : Boolean;    -- outstanding request is invalidated
                State : CacheState;
                Value : Value;
              End;
    Memory: Array[Address] of Value;  -- main memory for each processor
    DH    : Array[Address] of Header; -- directory header
    -- Record for the shared link is omitted.
  End;

  Request: Record
    src :  Proc;         -- real src of request
    Mtype: RequestType;
    SRC :  Proc;         -- src of request (may be faked)
    Node:  Proc;         -- node id of memory address
    Addr:  Address;
    Data:  Value;
  End;
  -- Record 'Reply' is similar.

Var  -- State Variables
  Procs: Array[Proc] of ProcState;
  QNet : Array[Proc] of Record Count  : 0..QueueSize;
                               Message: Array[Queue] of Request; End;
  PNet : Array[Proc] of Record Count  : 0..QueueSize;
                               Message: Array[Queue] of Reply; End;
```

Figure 4.2: Global variables and type declarations for FLASH protocol

```
Ruleset src : Proc Do                           -- for any home
Ruleset addr : Address Do                       -- for any address
Alias Cache : Procs[src].Cache[src][addr] do  -- src is the home
Alias Dir : Procs[src].DH[addr] Do              -- directory of the line
  Rule "PI Local Get"
      Cache.State = Invalid  & !Cache.Wait    -- read miss occurred
    & !Dir.Pending        -- if pending, NAK to processor in the home
    & Qspaces             -- there is enough space in the queue
  ==>
  Begin
    Assert !Dir.Local "PI Local Get: L = A0";
    If Dir.Dirty Then
      Assert Dir.Head & ! Dir.List & Dir.Real=0 "PI Local Get: case D=1";
      Dir.Pending := true;
      Cache.Wait := true;
      Send_Request(src, Dir.HPtr, Get, src, src, addr, void);
    Else
      Dir.Local := true;
      Cache.Value := Procs[src].Memory[addr]; -- send CC_Put
      Cache.State := Shared;
    End;
  End;
End; End; End; End;
```

Figure 4.3: The ruleset corresponding to PI.Local.Get

rule requires the rule be enabled only when there is enough space in the network. This is because the description is for a finite state system, where the size of the incoming buffer should be finite. The procedure `Send_Request()` sends a message to the destination by putting it in the incoming buffer of the destined node.

4.3.3 Specification of the protocol

The following properties are specified and checked by the Murφ verifier.

S1. For any memory line, there is at most one exclusive cached copy.

S2. If a shared cached copy contains a value different from that in main memory, there should be a message, INV, SWB, or PUT for the memory line waiting to be processed.

S3. If a processor owns an exclusive cached copy of a memory line, then there is no outstanding request for the line from the processor.

S4. There is no message with destination and source for a same node.

S5. For any directory header, if List is false and Real_Pointers > 0, then Pending is true.

S6. For any directory header, if Head is false, then List is false.

S7. For any directory header, if List is true, then Real_Pointers > 0.

L1. For any directory header, if Pending is set, then it should be reset eventually.

L2. For any cache line in a processor, if the processor is waiting for a reply, then it should get it eventually.

Safety properties S1 to S3 are for the consistency of cached copies. The property S4 is a simple assertion on the messages in the network. The properties S5 to S7 are assertions on the state in the directory headers, which are also specified in the protocol documentation. A couple of liveness properties are shown in L1 and L2.

# of *proc*'s	# of *addr*'s	Data included	Size of queue	# of states	Time consumption	Memory consumption
2	1	No	3	476	2 sec	5.7 Kbyte
2	1	Yes	3	2.6 K	13 sec	36 Kbyte
3	1	No	3	53 K	7 min	1.1 Mbyte
3	1	Yes	3	549 K	1.5 hr	13 Mbyte
3	1	Yes	4	715 K	2.5 hr	21 Mbyte
3	1	Yes	5	800 K	3.7 hr	30 Mbyte
4	1	No	3	$\gg$ 2.8 M	$\gg$ 11 hr	$\gg$ 100 Mbyte
3	2	No	3	$\gg$ 2.7 M	$\gg$ 5.3 hr	$\gg$ 100 Mbyte

Table 4.1: Time and memory space required for the state exploration using Murφ

As mentioned before, it is not easy to find complete correctness conditions for cache coherence protocols, because it requires encoding a full memory model in a given specification language. The above properties might be a subset of complete specification. This problem will disappear when we apply the aggregation method in the next section.

4.3.4 State explosion problem

The safety and liveness properties above are checked by the Murφ verifier. Running the automatic verifier has been very useful and effective to capture errors rapidly. Many description errors were found and fixed by verifying with Murφ.

However, due to the finiteness constraint of the state exploration method, the system that can be completely explored turned out to be quite small—no more than 3 processors. Table 4.1 shows time and space consumed for the verification of finite-sized models of the protocol in EAGER mode. The numbers are obtained running the verifier on SPARC Station 20 with symmetry reduction features. As shown in the table, the state space increases exponentially as the number of components of the system increases. Therefore, the method is limited to small models of the protocol, with partial specifications.

Two common simplifications are used. First, the description includes only a small number of processors and memory addresses. Unfortunately, this could be a serious limitation because errors can go undetected in such small-scale models. Since the actions in the protocol do not interact between different memory addresses, simplifying the model with a single memory address would be reasonable. However, the small number of processors may hide a number of possible bugs. The simplified model with 2 processors are not so meaningful because some of transactions such as three hop cases involve at least three processors. The model with 3 processors still may not be enough, because a fourth processor whose request is rejected could stimulate behavior that is not seen with 3 processors.

The small size of the incoming message queues is another limitation. Because some of actions generate two messages to a node, a message queue size of 3 may restrict consecutive actions to a specific node, potentially resulting in overlooked errors.

The other simplifications are that the values of cached copies are not tracked and the property of data consistency is formulated on the states of caches only. Either of these simplifications could lead to inaccurate verification results. To avoid such finiteness constraints and prove the protocol with unlimited number of sources, we apply the aggregation method in the next section.

4.4 Verification By Aggregating Distributed Transactions

Although Murφ is useful for debugging protocols, the previous section has revealed some deficiencies: state explosion problem and specification problem. These problems are solved by using the aggregation method of Chapter 3.

Using the aggregation method, we have formally verified the protocol at the level of its formal description [56]. The protocol consisting of more than a hundred different implementation steps has been reduced to a model with only six kinds of atomic transactions. Based on the reduced atomic behavior, it is very easy to reason about the protocol, checking safety properties and data consistency of cached copies.

In the following, we illustrate how the protocol is reduced to an atomic model by an aggregation function. The detailed proofs are confirmed by a theorem prover and some techniques to simplify the proof are presented.

4.4.1 Extracting reduced model of the protocol

Verification requires two descriptions of same behavior: an implementation and a specification. Sometimes, there is an *a priori* specification as in the memory model verification in the next section. However, in most practical instances, there is only an implementation. In such cases, we extract a reduced model of the implementation using aggregation. The reduced model captures concise behavior of the implementation and serves as a specification.

Recall from Chapter 3 that to use the aggregation method, we first decide which state variables should be considered specification variables. In cache coherence protocols, the consistency of multiple copies of a memory line is a function of the values and states of cached copies, and the corresponding value in main memory. Therefore, the specification variables should be the state variables representing the data and states of cached copies and the data in main memory.

We construct a reduced model of the protocol, which we use for a specification. The reduced model is a much simpler version of the protocol which reads and writes only the specification variables. The specification steps update the values and states of cached copies in multiple nodes *atomically.*

The reduced model of the protocol is shown in Table 4.2. Atom-WB invalidates an exclusive copy and writes back the data to main memory atomically. Atom-INV simply invalidates a shared copy. There are two kinds of transactions for a read miss: Atom-Get-1 corresponds to the transaction that the home grants a shared copy to the requester when there is no dirty copy of the memory line; Atom-Get-2 corresponds to the transaction that a node with an exclusive copy grants a shared copy. For the transaction for a write miss, Atom-GetX-1 sends an exclusive copy of a memory line from the home if there are no other copies in remotes; Atom-GetX-2 transfers an exclusive ownership from a dirty node to the requester.

	Condition	Atomic Action
Atom-WB (p, a)	cache[p][a].state=exclusive	cache[p][a].state := invalid memory[a] := cache[p][a].data
Atom-INV (p, a)	cache[p][a].state=invalid $\vee$ cache[p][a].state=shared	cache[p][a].state := invalid
Atom-Get-1 (p_2, a)	$\neg\exists i$: cache[i][a].state=exclusive	cache[p_2][a].state := shared cache[p_2][a].data := memory[a]
Atom-Get-2 (p_1, p_2, a)	cache[p_1][a].state=exclusive $\wedge$ $p_1 \neq p_2$	memory[a] := cache[p_1][a].data cache[p_1][a].state := shared cache[p_2][a].state := shared cache[p_2][a].data := cache[p_1][a].data
Atom-GetX-1 (p_2, a)	$\neg\exists i$: cache[i][a].state=exclusive $\wedge$ ($\neg\exists i$: cache[i][a].state=shared $\wedge$ $i \neq p_2$)$*$	cache[p_2][a].state := exclusive cache[p_2][a].data := memory[a]
Atom-GetX-2 (p_1, p_2, a)	cache[p_1][a].state=exclusive $\wedge$ $p_1 \neq p_2$	cache[p_1][a].state := invalid cache[p_2][a].state := exclusive cache[p_2][a].data := cache[p_1][a].data

* additional constraint for DELAYED mode.

Table 4.2: Reduced model of the FLASH protocol obtained by aggregation of distributed transactions

4.4.2 Commit steps

To define the aggregation function *aggr*, we should first identify the commit steps of each transaction in the protocol. The transaction for a read miss begins with sending a GET request to the home. Depending on the directory state of the memory line, the request may be forwarded to a remote which contains a dirty copy of the line. These steps do not modify the specification variables, so they are pre-commit steps of the transactions. The transaction for a write miss is similar.

The commit step occurs when the home, or a remote with an exclusive copy, sends a PUT or PUTX reply, granting the request. In each case, the state of the cache line or main memory in the granting node is modified. Any future request for the memory line is processed as if the committed reply had been processed by the requesting node, even if that has not actually happened. For instance, if a GETX request arrives at the home from R_1 right after a grant of an exclusive ownership to R_2, the home forwards the GETX to R_2 regardless of whether the PUTX sent to R_2 has arrived there or not. If a request is NAKed, then there is no change in specification variables by the transaction, so, in effect, no action occurs.

The write-back transaction begins with invalidating an exclusive copy and sending a WB request to the home. This is the commit step of the transaction because the state of cached data, a part of the specification variables, is already updated at this moment and the write-back request can not be denied by the home. The invalidation transaction is similar to this case.

4.4.3 Aggregation function

Once a transaction is committed, the aggregation function *aggr* simulates the post-commit steps of the transaction to complete it. The post-commit steps in the protocol are the steps that process a PUT and SWB for a read miss, and that process a PUTX for a write miss, and that process a WB for a write-back. Therefore, to complete all the committed transactions, the *aggr* should process all the messages of types PUT, PUTX, WB, and SWB.

The key question is how to complete all committed transactions in the current state, especially since the number of distributed nodes, and hence the number of committed transactions, is unknown. The same strategy as in the distributed list protocol in Chapter 3 works for the FLASH as well. We first define a *per-node* completion function for a node indexed by variable i; the per-node function is then *generalized* to define a completion function for all of the nodes in the system.

It is quite simple to complete a committed transaction for a particular node. If a PUT message destined for node i exists, the transaction for a read miss in node i must be completed by simulating the effect of node i processing the PUT message it receives at the end of the transaction: putting the data in the message into its cache and setting the state to *shared*. The transaction for a write miss is similarly completed by processing a PUTX to node i. If node i is the home, there are two more kinds of messages possibly generated at commit steps: SWB and WB. Note that there exists at most one message of the four types destined to a particular node at any time.

This processing changes values and states of cached copies, and values in main

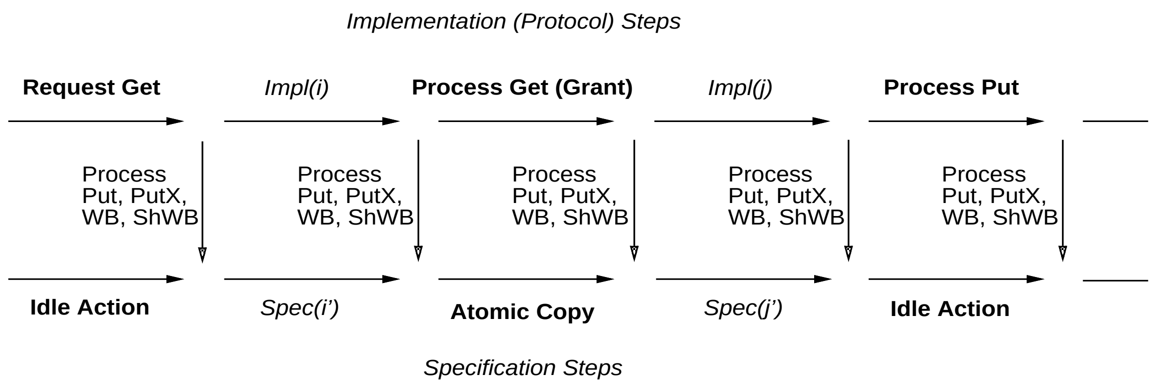

Figure 4.4: Step simulation for processing a GET request in the FLASH protocol

memory. Changes to implementation variables, such as removing messages from the network, and resetting the waiting flag in the processor can be omitted from the completion function, as they do not affect the corresponding specification state. All of this computation is done solely in node i, without the involvement or interference of other nodes.

As shown in Chapter 3, it is easy to generalize the per-node completion function to a completion function for all of the nodes because the completions do not interact. The completion functions are simply performed in parallel.

4.4.4 Specification steps

The specification steps corresponding to implementation steps are simply idle transitions for pre-commit steps and post-commit steps. The only non-idle actions are those which correspond to the commit steps of actions. Figure 4.4 shows the step simulation for processing a GET request. The steps in boldface correspond to the transaction and they may be interleaved with the steps for other transactions.

A complete assignment of atomic actions of the reduced model to the implementation steps of the protocol is shown in Table 4.3. Each pair corresponds to a subgoal (3.4) in Section 3.2. The condition of an atomic action should be true at the corresponding commit step in the implementation, which is included in the invariant of the system.

Protocol Step at Node p	**Atomic Transaction (Specification)**
PI.Local.Get.else	ε
PI.Local.Get.put	Atom-Get-1(*home*)
PI.Remote.Get (at node p)	ε
PI.Local.GetX.else	ε
PI.Local.GetX.putx	Atom-GetX-1(*home*)
PI.Remote.GetX (at node p)	ε
PI.Local.PutX	Atom-WB(*home*)
PI.Remote.PutX (at node p)	Atom-WB(p)
PI.Local.Replace	Atom-INV(*home*)
PI.Remote.Replace (at node p)	Atom-INV(p)
NI.NAK	ε
NI.NAK.Clear	ε
NI.Local.Get.else	ε
NI.Local.Get.put	Atom-Get-1(GET.src)
NI.Local.Get.put.ex[1]	Atom-Get-2(*home*, GET.src)
NI.Local.Get.put.inv[2]	Atom-Get-1(GET.src); Atom-INV(GET.src)
NI.Local.Get.put.ex.inv	Atom-Get-2(*home*, GET.src); Atom-INV(GET.src)
NI.Remote.Get.else (at node p)	ε
NI.Remote.Get.put (at node p)	Atom-Get-2(p, GET.src)
NI.Remote.Get.put.inv[2] (at node p)	Atom-Get-2(p, GET.src); Atom-INV(GET.src)
NI.Local.GetX.else	ε
NI.Local.GetX.putx	Atom-GetX-1(GETX.src)
NI.Local.GetX.putx.ex[1]	Atom-GetX-2(*home*, GETX.src)
NI.Remote.GetX.else (at node p)	ε
NI.Remote.GetX.putx (at node p)	Atom-GetX-2(p, GETX.src)
NI.Local.Put	ε
NI.Remote.Put	ε
NI.Local.PutXAcksDone	ε
NI.Remote.PutX	ε
NI.Inv (at node p)	Atom-INV(p)
NI.InvAck	ε
NI.WB	ε
NI.FAck	ε
NI.ShWB	ε
NI.Replace	ε

Table 4.3: Correspondence of protocol steps with atomic transactions (EAGER mode)

In the FLASH protocol, some handlers perform commit steps in some cases and not in others. In order to establish the necessary correspondence between implementation steps and specification steps in a proof of property (3.4), we need to split these handlers into multiple transition functions, each of which either always commit or never commit. For example, the PI.Local.Get handler simply NAKs the local processor if the requested line is pending (pre-commit step), or sends a request to a remote if there is a dirty copy (pre-commit step), otherwise, it updates the state and data of the local cache which are specification variables (commit step). In the first two cases, the reduced model should take idle transitions, but in the last case, an Atom-Get transaction should be taken.

The PI.Local.Get handler is decomposed into two different transition functions PI.Local.Get.else and PI.Local.Get.put with disjoint enabling conditions, where the first includes the pre-commit steps, and the latter corresponds to the commit step. Other handlers are decomposed in the same manner, if necessary. In Table 4.3, the protocol steps named with suffix 'ex' (with superscript 1) correspond to the decomposed handlers when the home holds an exclusive copy. The protocol steps named with suffix 'inv' (with superscript 2) correspond to the decomposed handlers when the requesting node is invalidation marked. Note that these decompositions do not change the original protocol implementation.

Table 4.3 lists all the transition functions of the protocol in EAGER mode and the corresponding atomic transactions of the reduced model. The atomic transactions are listed with properly instantiated parameters. The table for DELAYED mode would be the same as Table 4.3 except that ownership transfer (GetX-Atom) corresponds to the protocol step which processes the last invalidation acknowledgment.

4.4.5 Invariant

As mentioned in Section 3.2, we need an invariant which contains several assertions to prove the subgoals. The subgoals corresponding to pre-commit steps are simply proved to be valid because the specification variables are not modified at all. PVS can handle them automatically. However, some of the other subgoals need some assertions about the system to satisfy the commutativity requirement. The theorem prover guides the user to find such assertions.

To check those assertions, we write an invariant which is the logical "and" of the assertions, and prove that it is preserved by every step of the protocol. If the invariant is not strong enough to be preserved by all the implementation steps, we need to strengthen it. Although not intellectually difficult, this was the most time-consuming part of the proof.

The invariant we eventually derived includes the following assertions.
For each memory line:

- There is at most one exclusive copy.
- There is at most one message to each node of type PUT, PUTX, WB, or SWB.
- If a node contains an exclusive copy, then there is no PUT to the home and no PUTX, WB, or SWB to any node.
- If there is a PUTX message being processed, then there is no PUT to the home and no WB or SWB, and no other PUTX to any node.
- A node is waiting for a PUT reply if there is a GET request from the node, a PUT reply to the node, or an invalidation marked.
- A node is waiting for a PUTX reply if there is a GETX request from the node, or a PUTX reply to the node.
- If Dirty in the directory header is false, there is no exclusive copy, no PUT to the home, and no PUTX, WB, or SWB to any node.
- If Pending in the directory header is false, then there is no PUT, PUTX, SWB, FWAK, GET, GETX to the home, no forwarded GET or GETX, no NAKC or INV to any node.
- The cache state in the home is in *invalid* if Local is false, or Pending is true and Dirty is false.

4.4.6 Tricks for using a theorem prover

The tricks used in the distributed list protocol also work for the FLASH protocol. To make the computer-assisted proofs fast, we have chosen to represent the network in a non-obvious way. We observe that there is at most one request/reply message for a memory line pertaining to any particular node at any time. So the network can be represented with one variable per node per memory line (sometimes associated with the source, sometimes with the destination) for relevant kinds of messages. Hence, instead of proving that there is only one message of a certain type in the network for node i at any time, we register an error whenever a message in a variable is about to be overwritten, and verify that no error occurs.

4.5 Delayed Mode Conforms to Sequential Consistency Memory Model

As mentioned before, the FLASH protocol supports two memory model modes: EAGER and DELAYED. The difference between the two modes lies in when the reply is sent for a GETX request of a processor trying to write. In EAGER mode the reply can be sent before all the invalidation acknowledgments have been collected, while DELAYED mode only sends the reply after invalidation acknowledgments have been collected. Therefore, EAGER mode supports a more aggressive memory model which grants exclusive ownership when there are still old copies valid for reads. This difference is visible to users and may affect the correctness of synchronization code.

In this section, we show that the DELAYED mode implements the sequential consistency memory model [42], if the processors execute instructions in a sequential order one at a time, stalling at each cache miss [26]. For the proof, we use the aggregation method again. This time, the reduced behavior of DELAYED mode shown in Table 4.2 is considered the implementation instead of the specification as in the proof of Section 4.4, and the specification is a state graph that models a collection of processors doing atomic loads and stores. The composition of two aggregation functions is an aggregation function, so this also implies the existence of an aggregation

Delayed Memory Model	**Sequential Consistency**
Load_Delayed	*Load_SC* register[p][r] := memory[a]
Store_Delayed	*Store_SC* memory[a] := register[p][r]
Atomic Actions of DELAYED Mode in Table 4.2	ε

Table 4.4: DELAYED mode conforms to Sequential Consistency memory model

	Condition	**Action**
Load_Delayed	cache[p][a].state = shared $\vee$ cache[p][a].state = exclusive	register[p][r] := cache[p][a].data
Store_Delayed	cache[p][a].state = exclusive	cache[p][a].data := register[p][r]
Atomic Actions of DELAYED Mode in Table 4.2	See Table 4.2	See Table 4.2

Table 4.5: Reduced model of the FLASH protocol in DELAYED model

function from the full protocol to a sequential consistency memory model.

The sequential consistency memory model is specified in the right column of Table 4.4. The model consists of two transactions Load_SC and Store_SC which read and write data between the registers and main memory, atomically. The specification variables model the main memory and registers. The caches are now implementation variables, which are not visible to the memory model specification.

In order to model registers in the implementation, we add a couple of steps to the reduced model which load and store a cached copy respectively. The step Load_Delayed in Table 4.5 simulates a processor loading a memory location by reading a cached datum into a designated register if the copy is in a shared or an exclusive state. The step Store_Delayed simulates a processor storing a memory location by writing a datum into a cache line if it has an exclusive ownership of the memory line.

The commit step of the load transaction in the protocol is Load_Delayed and that of the store transaction is Store_Delayed. The aggregation function should simulate a delayed update of main memory by immediately writing back an exclusive copy, if it exists. Table 4.4 shows correspondence of specification steps with each step of the reduced model for DELAYED mode. All the rest six steps correspond to idle transitions.

The proof involves proofs of property (3.4) for eight implementation transition functions with the following invariant of the system: if a cached line is in shared state, then main memory has a same data as in the cache and there is no exclusive copy; and there exists at most one exclusive copy. It is easy to see that the invariant is true in the system which consists of the eight transitions of the reduced model for DELAYED mode.

What have we really proved? The composition of the two aggregation functions, from FLASH to the reduced model to sequential consistency, may be extended inductively to sequences of steps. If a multiprocessor program is executed on FLASH, the execution will contain interleaved steps of various memory transactions. This function maps a sequence of steps on FLASH to a sequence of high-level memory transactions (and idle steps) in our model of sequential consistency. Since the aggregation function preserves the specification variables for the memory and processor registers unchanged between transactions, the visible result of a terminating program on FLASH is guaranteed to be the same as the result on the sequential consistency model.

4.6 Executable Memory Models

We have proved that the DELAYED mode implements the sequential consistency memory model. However, there does not exist a well-defined memory model for EAGER mode, though we know that EAGER mode supports a weaker memory model than sequential consistency. Moreover, the different behavior between the memory models is important to the users, especially to programmers, because the outputs of programs could be different depending on the modes the multiprocessor is running in.

In Chapter 2, we have developed executable descriptions of memory models, derived from axiomatic specifications of memory models. We can apply the same technique for this protocol using the reduced behavior of the FLASH protocol in Table 4.2. The executable description automatically generates all the possible outcomes of test programs so that we can analyze the programs running on the two different modes of the protocol.

We write the executable model using a high-level description language for finite-state concurrent systems called Murφ. The description consists of a set of rules, each of which has an enabled condition and atomic action statements. Execution of a Murφ program begins with one of a set of initial states specified by the user. Then the following loop is executed forever: some rule whose condition is satisfied by the current state is chosen and its action evaluated, yielding a new current state. If there are no rules whose conditions are true, the execution halts. When several rule conditions are true at the same time, a choice is made arbitrarily, resulting in several possible executions. The Murφ verifier tries them exhaustively by depth-first or breadth-first search. It can print out the value of system variables at user-specified points while exploring all the reachable states of the system.

We present a simple test program which shows different behavior between the two modes of the protocol.

```
Proc[0] :   st #1, A;  ld  B, %r1;
Proc[1] :   st #1, B;  ld  A, %r2;
```

The following is excerpted from the Murφ description for the above test program.

```
Rule  -- Proc[0] does  < st #1, A >
pc[0] = 0 & cache[0][A].state = exclusive  -- condition to store
==> begin store(0, 1, A); end;  -- stores the value into memory

Rule  -- Proc[0] does  < ld  B, %r1 >
pc[0] = 1 & cache[0][B].state != invalid   -- condition to load
==> begin load(0, B, r1); end;  -- loads the data in memory to the register

-- Other rules are omitted.
```

```
Rule
'condition that pc[0], pc[1] are in final state'
==> begin 'print out memory and registers'; end;
```

The list of all possible outcomes of the test program generated by the description is shown below. As expected, the output of DELAYED mode is equivalent to that of the sequential consistency memory model. The output of EAGER mode is a superset of the that of DELAYED mode; the first output of EAGER mode is not possible in DELAYED mode. This confirms that EAGER mode supports a weaker memory model than sequential consistency. The results of other test programs demonstrates that the memory model with FLASH protocol in EAGER mode is as weak as the PSO SPARC memory model.

```
EAGER:: A:1 B:1 r1:0 r2:0
EAGER:: A:1 B:1 r1:0 r2:1    DELAY:: A:1 B:1 r1:0 r2:1
EAGER:: A:1 B:1 r1:1 r2:0    DELAY:: A:1 B:1 r1:1 r2:0
EAGER:: A:1 B:1 r1:1 r2:1    DELAY:: A:1 B:1 r1:1 r2:1
```

4.7 Detailed description of FLASH protocol (EAGER mode)

This section contains a complete list of the handlers of the FLASH protocol in EAGER mode.

- **PI.Local.GetX:** this handler describes actions of the home when the local processor needs an exclusive copy. If Pending, the processor is NAKed. Otherwise, if Dirty, the home sends a GETX request[3] to Head_Pointer and Pending is set. Otherwise, the data in main memory is copied into the local cache (in *exclusive* state) and Local and Dirty are set. In the last case, if Head_Valid,

[3]The original protocol uses a different request UPGRADE for an exclusive copy, rather than using GETX, when the cache has a shared copy. The reason is to enhance performance by avoiding unnecessary data transfer. However, the two requests are processed in the same manner except whether the reply contains the cached data or not. We did not model the UPGRADE request in the verified description.

which indicates there are shared copies in remote nodes, the home sends INVs to Head_Pointer and the nodes in Sharer_List, Pending is set, Head_Valid is reset, and the number of invalidations is written in Real_Pointers.

- **PI.Remote.Get(X):** this handler describes actions of a remote node when the processor needs a shared (or an exclusive) copy. The remote sends a GET (or GETX) request to the home.

- **PI.Local.PutX:** this handler writes back a cached exclusive copy in the home. Dirty is reset (and Local, if not Pending) and the cached copy to the main memory is written back.

- **PI.Remote.PutX:** this handler writes back a cached exclusive copy in a remote. The remote sends a WB request to the home.

- **PI.Local.Replacement:** this handler replaces a shared copy in the home. Local is reset.

- **PI.Remote.Replacement:** this handler replaces a shared copy in a remote. The remote sends a RPL request to the home.

- **NI.NAK:** this handler describes actions of a node receiving a NAK reply. The processor clears its waiting flag and invalidation mark.

- **NI.NAKC:** this handler describes actions of the home receiving a NAK clear (NAKC). Pending is reset.

- **NI.Local.Get:** this handler describes actions of the home receiving a GET request from a remote. If Pending, the home sends a NAK to the source. Otherwise, if Dirty and not Local, Pending is set and the home forwards the GET to Head_Pointer with source faked as the original requester. Otherwise, if Dirty and Local, then writes back the exclusive copy in the local cache to main memory, sends a PUT reply to the source, and Dirty is reset, Head_Valid is set, and Head_Pointer is set to the source. Otherwise, the home sends a PUT reply to the source; If Head_Valid, List is set, Real_Pointers is incremented, the source is added to Sharer_List. If not Head_Valid, Head_Valid is set, Head_Pointer is set to the source.

- **NI.Local.GetX:** this handler describes actions of the home receiving a GETX request from a remote. If Pending, the home sends a NAK to the source. Otherwise, if Dirty and not Local, then Pending is set and the home forwards the GETX to Head_Pointer with source faked as the original requester. Otherwise, if Dirty and Local, the home sends a PUTX reply to the source with the exclusive data from the local cache, and Local is reset, Head_Valid is set, and Head_Pointer is set to the source. Otherwise, the home sends a PUTX to the source with the data in main memory.

  In the last case, if not Dirty and Head_Valid, Dirty is set, List is reset, and if Head_Pointer is not equal to the source, Pending is set, and the home sends an INV to the Head_Pointer, and Head_Pointer is set to the source. If Local, invalidates the local copy, and if List, the home send INVs to all the nodes in Sharer_List and set Real_Pointers to the number of invalidations. Otherwise, if not Dirty and not Head_Valid, Head_Valid and Dirty are set, Local is reset, and Head_Pointer is set to the source.

- **NI.Remote.GetX:** this handler describes actions of a remote node receiving a GETX request. If the cached data is in *exclusive* state, it is invalidated and the node sends a PUTX reply to the source (and a forward acknowledgment FWAK to the home if the source is not the home). Otherwise, the node sends a NAK to the source and a NAKC to the home.

- **NI.Local.Put:** this handler processes a PUT reply to the home. Local is set, Dirty and Pending are reset, and the shared copy is put into the local cache.

- **NI.Local.PutX:** this handler processes a PUT reply to the home. Local is set, Head_Valid and Pending are reset, and the exclusive copy is put into the local cache.

- **NI.Remote.PutX:** The exclusive copy is put into the cache.

- **NI.Inval:** Receiving a INV, the remote invalidates the cached copy and sends an INVAK to the home. If the node was waiting for a PUT(examining its waiting flag), it marks the line invalidated.
- **NI.InvalAck:** this handler describes actions of the home receiving an INVAK. Real_Pointers is decremented. If it reaches to zero, Pending is reset (and Local if not Dirty).
- **NI.Writeback:** this handler describes actions of the home receiving a WB request. Dirty and Head_Valid are reset and the data is written back into the main memory.
- **NI.ForwardAck:** this handler describes actions of the home receiving a FWAK. Pending is reset. If Dirty, Head_Pointer is set to the source.
- **NI.Replacement**: this handler describes actions of the home receiving a RPL. The source is removed from Sharer_List if found and Real_Pointers is adjusted.

Chapter 5

Conclusion

This chapter summarizes the thesis and proposes possible lines of future research.

5.1 Summary

5.1.1 Executable memory models

As an alternative formal specification of multiprocessor memory models, we wrote an executable model using a simple general-purpose description language for concurrent systems. We presented some techniques for writing the executable description in Murφ and using its automatic verifier to analyze programs running under the specified memory model.

The description provides a precise specification of the machine architecture, both for hardware implementors and programmers. We believe that this type of executable description strikes an appropriate balance between formality and understandability by programmers and machine architects.

Moreover, the availability of an automatic verification tool allows users to experiment with the effects of the memory model on small assembly-language routines. Also, as we have learned in this experiment, developing an executable description and running the verifier can be very effective at clarifying the subtle details of the models and synchronization routines.

We participated in defining Relaxed Memory Order of SPARC Version 9 Architecture. We wrote an executable memory model and used the proposed automatic analysis techniques during the design procedure. Our approach to memory model specification and analysis turned out to be very helpful to the SPARC-V9 design team.

5.1.2 Verification of cache coherence protocols

We proposed a verification method for cache coherence protocols and distributed algorithm. The method provides an easy and systematic way to find an *aggregation function* used for verification. The method substantially reduces the amount of labor required, so it significantly extends the capability of computer-assisted theorem-proving for cache coherence protocols. Owing to the generality of the higher-order logic we used as a formalism, we have been able to validate protocols with an arbitrary number of processors.

The method has been successfully applied to the verification of the FLASH directory-based cache coherence protocol, which is too large and complicated to prove using a finite-state method. The protocol consisting of more than a hundred implementation steps has been reduced to a specification with six kinds of atomic transactions. Based on the reduced behavior, it is very easy to prove crucial properties of the protocol including data consistency of cached copies at the user level. Moreover, the reduced model allows us to write a simple executable memory model of the protocol. The aggregation method is also used to prove that the reduced protocol conforms to the corresponding memory consistency model.

For several years, we had believed that proving the correctness of protocols of the complexity of the FLASH cache coherence protocol was well beyond the capability of a general-purpose theorem prover. The aggregation method has broken through this barrier.

The proposed verification procedure is not only for cache coherence protocols but also has been applied to other protocols, which are simple but non-trivial: a majority consensus algorithm for multiple copy databases, and a distributed list protocol.

5.2 Future Research

There are many memory models besides those we have formalized. It would be useful to try writing executable specifications for them, to see if the same methods used for the SPARC models work well for them.

The aggregation method as described can be applied to many protocols, we have only tried a few. It may need to be generalized and many generalizations are conceivable: multiple commit points and reversing transactions instead of completing.

The aggregation method can be further automated to even greater advantages. We want more automation in defining an aggregation function, finding invariants of a system, and detailed proofs. From this and many other efforts, it has become clear that finding invariants is the most time consuming part of many verification problems. More computer assistance is needed, especially for large problems.

We have not considered the important problem of proving liveness properties here. However, showing liveness using strong fairness assumption is not difficult, because the implementation steps for each action in the protocol are successively enabled in a sequence. We plan to find more systematic ways to prove liveness properties.

Bibliography

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

[2] Y. Afek, G. Brown, and M. Merritt. Lazy caching. *ACM Transactions on Programming Language and Systems*, 15(1):182–205, January 1993.

[3] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and performance. In *Proc. 22nd International Symposium on Computer Architecture*, 1995.

[4] J. Archibald and J. Baer. An economic solution to the cache coherence problem. In *Proc. 11th International Symposium on Computer Architecture*, pages 355–362, June 1984.

[5] J-L. Baer and C. Girault. A petri net model for a solution to the cache coherence problem. In *Proceedings of the First Conference on Supercomputing Systems*. IEEE Computer Society, 1985.

[6] J. Burch, E. Clark, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2), June 1992.

[7] Jerry Burch and David Dill. Automatic verification of pipelined microprocessor control. In *Computer Aided Verification, 6th International Conference, CAV'94*, pages 68–80, June 1994.

[8] Paolo Camurati and Paolo Prinetto. Formal verification of hardware correctness: Introduction and survey of current research. *Computer*, 21(7):8–19, July 1988.

[9] L. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 27(12):1112–1118, December 1978.

[10] K. Mani Chandy and Jayadev Misra. *Parallel Program Design — a foundation.* Addison-Wesley, 1988.

[11] E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K. McMillan, and L. Ness. Verification of the Futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6(2):217–232, March 1995.

[12] Ernest Cohen. *Modular progress proofs of asynchronous programs.* PhD thesis, University of Texas at Austin, 1993.

[13] William W. Collier. *Reasoning about Parallel Architectures.* Prentice Hall, 1992.

[14] J. de Bakker, W. de Roever, and G. Rozenberg, editors. *Stepwise Refinement of Distributed Systems. Models, Formalisms, Correctness: LNCS 430.* Springer-Verlag, 1990.

[15] D. Dill, A. Drexler, A. Hu, and C. Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design: VLSI in Computers.* IEEE Computer Society, 1992.

[16] D. Dill, A. Hu, and H. Wong-Toi. Checking for language inclusion using simulation relation. In *Computer Aided Verification, 3rd International Workshop*, pages 255–265, July 1991.

[17] David Dill, 1994. Private communication.

[18] David Dill, Seungjoon Park, and Andreas Nowatzyk. Formal specification of abstract memory models. In *Research on Integrated Systems: Proceedings of the 1993 Symposium*, pages 38–52. MIT Press, March 1993.

[19] Thomas Doeppner, Jr. Parallel program correctness through refinement. In *Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 155–169, January 1977.

[20] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *Proceedings for the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[21] Ásgeir Eiríksson and Ken McMillan. Using formal verification/analysis methods on the critical path in system design: A case study. In *Computer Aided Verification, 7th International Conference, CAV'95*, pages 367–380, July 1995.

[22] Steven German, 1996. Private communication.

[23] Rob Gerth. Verifying sequentially consistent memory problem definition. Eindhoven University of Technology, April 1993.

[24] K. Gharachorloo, S. Adve, A. Gupta, J. Hennessy, and M. Hill. Programming for different memory consistency models. *Journal of parallel and distributed computing*, 15(4):399–407, August 1992.

[25] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings for the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[26] P. Gibbons, M. Merritt, and K. Gharachorloo. Proving sequential consistency of high-performance shared memories. In *Proc. 3rd ACM Symposium on Parallel Algorithms and Architectures*, pages 292–303, July 1991.

[27] James R. Goodman. Cache consistency and sequential consistency. Technical Report 61, SCI Committee, March 1989.

[28] Aarti Gupta. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1:151–238, 1992.

[29] Anoop Gupta, 1996. Private communication.

[30] Mark Heinrich. *The FLASH Protocol.* Internal document, Stanford University FLASH Group, 1993.

[31] Maurice Herlihy and Jeannette Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Language and Systems*, 12(3):463–492, July 1990.

[32] M. Hill, J. Larus, S. Reinhardt, and D. Wood. Cooperative shared memory: Software and hardware for scalable multiprocessors. *ACM Transactions on Computer Systems*, November 1993.

[33] R. Hojati, R. Mueller-Thuns, P. Loewenstein, and R. Brayton. Automatic verification of memory systems which service their requests out of order. In *Proceedings CHDL '95*, 1995.

[34] Gerard Holzmann. *Design and validation of computer protocols.* Prentice Hall, 1991.

[35] SPARC International. *The SPARC Architecture Manual Version 8.* Prentice Hall, 1992.

[36] C. Norris Ip and David Dill. State reduction using reversible rules. In *Proceedings of 33rd Design Automation Conference*, June 1996.

[37] Bengt Jonsson. Compositional specification and verification of distributed systems. *ACM Transactions on Programming Language and Systems*, 16(2):259–303, March 1994.

[38] Robert Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton, 1994.

[39] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proc. 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[40] P. Ladkin, L. Lamport, B. Olivier, and D. Roegel. Lazy caching: An assertional view. *Distributed Computing*, 1996. To appear.

[41] S. Lam and A. Shankar. Protocol verification via projection. *IEEE Transactions on Software Engineering*, 10(4):325–342, July 1984.

[42] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[43] Leslie Lamport. An assertional correctness proof of a distributed algorithm. *Science of Computer Programming*, 2:175–206, 1982.

[44] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Language and Systems*, 5(2):190–222, April 1983.

[45] Leslie Lamport. A theorem on atomicity in distributed algorithms. *Distributed Computing*, 4:59–68, 1990.

[46] Leslie Lamport and Fred Schneider. Pretending atomicity. Technical Report 44, SRC Digital, 1989.

[47] D. Lenosky, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proc. 17th International Symposium on Computer Architecture*, pages 148–159, May 1990.

[48] Richard Lipton. Reduction: a method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, December 1975.

[49] N. Lynch. I/O automata: A model for discrete event systems. In *22nd Annual Conference on Information Science and Systems*, March 1988. Princeton University.

[50] K. McMillan and J. Schwalbe. Formal verification of the gigamax cache-consistency protocol. In *Proceedings of the International Symposium on Shared*

Memory Multiprocessing, pages 242–251. Information Processing Society of Japan, 1991.

[51] Ken McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993. Boston.

[52] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, W. Radke, and S. Vishin. The S3.mp Scalable Shared Memory Multiprocessor. In *ICPP*, 1995.

[53] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[54] Seungjoon Park and David Dill. An executable specification, analyzer and verifier for RMO (Relaxed Memory Order). In *Proc. 7th ACM Symposium on Parallel Algorithms and Architectures*, pages 34–41, July 1995.

[55] Seungjoon Park and David Dill. Protocol verification by aggregation of distributed transactions. In *Computer Aided Verification, 8th International Conference, CAV'96*, July 1996.

[56] Seungjoon Park and David Dill. Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In *Proc. 8th ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[57] David Patterson and John Hennessy. *Computer architecture A Quantitative approach.* Morgan Kaufmann Publishers, Inc., 1990.

[58] D. Peled, S. Katz, and A. Pnueli. Specifying and proving serializability in temporal logic. In *Proc. 6th Annual IEEE Symposium on Logic in Computer Science*, pages 232–244, July 1991.

[59] Fong Pong. *Symbolic State Model: A New Approach for the Verification of Cache Coherence Protocols.* PhD thesis, University of Southern California, 1995.

[60] Fong Pong and Michel Dubois. The verification of cache coherence protocols. In *Proc. 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 11–20, 1993.

[61] H. Sharangpani and M. Barton. Statistical analysis of floating point flaw in the Pentium processor. Technical report, Intel Corporation, November 1994.

[62] Pradeep S. Sindhu, Jean-Marc Frailong, and Michel Cekleov. Formal specification of memory models. Technical Report CSL-91-11, Xerox Palo Alto Research Center, December 1991.

[63] Ulrich Stern and David Dill. Automatic verification of the sci cache coherence protocol. In *Correct Hardware Design and Verification Methods. IFIP WG 10.5 Advanced Research Working Conference, CHARME95*, pages 21–34. Springe-Verlag, 1995.

[64] Robert Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[65] David Warren and Seif Haridi. The Data Diffusion Machine – A scalable shared virtual memory multiprocessors. In *Proc. of the 1988 International Conference on 5th Generation Computer Systems*, pages 943–952, December 1988.

[66] David Weaver and Tom Germond, editors. *The SPARC Architecture Manual Version 9.* Prentice Hall, 1994.

[67] W. C. Yen and W. L. Yen. Data coherence problem in a multicache system. *IEEE Transactions on Computers*, 34(1), January 1985.

[68] Michael Yoeli, editor. *Formal Verification of Hardware Design.* IEEE Computer Society Press, 1990.