FORMAL DESIGN AND VERIFICATION METHODS FOR SHARED MEMORY SYSTEMS

by

Ratan Prasad Nalumasu

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

The University of Utah

May 1999

Copyright © Ratan Prasad Nalumasu 1999

All Rights Reserved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Ratan Prasad Nalumasu

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Ganesh Gopalakrishnan

John B. Carter

Al Davis

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Ratan Prasad Nalumasu in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Ganesh Gopalakrishnan
Chair, Supervisory Committee

Approved for the Major Department

Robert Kessler
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

Many modern hardware and software systems are designed as a collection of components that run concurrently in order to achieve higher performance. These components employ sophisticated protocols for coordinating their actions. The correctness of these protocols is *critical* for the overall correctness of the system. Traditional debugging techniques such as simulation are increasingly unable to cover all aspects of the protocols. As a result, formal methods, especially *model checkers* that examine all possible scheduling of the events in the protocol, have gained considerable attention both from academia and industry.

This dissertation shows how formal methods can be tailored to a particular domain to address concerns specific to the domain, and in doing so obtain algorithms that perform better on the protocols that occur in the narrower domain. The domain chosen is shared memory system design and verification.

The contributions of the dissertation are:

1. a partial order reduction algorithm, called *two phase*, to improve the effectiveness of the model checkers,
2. a *refinement procedure* that synthesizes detailed distributed shared memory protocols from high-level specifications, and
3. a testing based approach, called *test model checking*, that can be used to verify if a given shared memory system correctly implements a given formal memory model.

The two phase algorithm is more effective than the current partial order reduction algorithms on a number of protocols, including the protocols that occur in shared memory design. The refinement technique shows how formal methods can exploit domain specific knowledge to support a high-level specification and validation of protocols followed by an automatic synthesis of a detailed implementation. The test model checking approach shows how limitations of model checking can be overcome by combining model checking with traditional testing methods.

To amma and nana

CONTENTS

ABSTRACT iv

LIST OF FIGURES ix

LIST OF TABLES xi

ACKNOWLEDGMENTS xii

CHAPTERS

1. INTRODUCTION 1

1.1 Formal Methods 1
  1.1.1 Formal verification techniques 2
    1.1.1.1 Model checking 2
    1.1.1.2 Theorem proving 6
  1.1.2 Formal design derivation techniques 6
1.2 Shared Memory System Design 7
  1.2.1 Shared memory multiprocessor organization 8
  1.2.2 Memory model verification 9
1.3 Contributions of the Dissertation 10
  1.3.1 Partial order reductions 11
  1.3.2 Design derivation technique 11
  1.3.3 Memory model verification 12
1.4 Organization of the Dissertation 12

2. A PARTIAL ORDER REDUCTION ALGORITHM WITHOUT THE PROVISO 14

2.1 Chapter Overview 14
2.2 Introduction 14
2.3 Related Work 18
2.4 Definitions and Notation 18
  2.4.1 Linear temporal logic and Büchii automaton 21
2.5 Basic DFS and Proviso Based Partial Order Reduction Algorithms 21
  2.5.1 Efficacy of partial order reductions 23
2.6 The Two Phase Algorithm 24
  2.6.1 Notation 25
  2.6.2 Correctness of the two phase algorithm 26
2.7 On-the-fly Model Checking 30
  2.7.1 Selective caching 33
  2.7.2 Combining on-the-fly model checking and selective caching with two phase 33
2.8 Experimental Results 34

2.9 Concluding Remarks 36

3. DERIVING EFFICIENT CACHE COHERENCE PROTOCOLS THROUGH REFINEMENT 37

3.1 Chapter Overview 37
3.2 Introduction 37
3.3 Related Work 39
3.4 Cache Coherency in Distributed Systems 40
  3.4.1 Complexity of DSM protocol design 41
  3.4.2 Communication model 41
  3.4.3 Methodology 42
  3.4.4 Process structure 42
  3.4.5 Forward progress 43
3.5 The Refinement Procedure 44
  3.5.1 Refining the remote node 44
  3.5.2 Refining the home node 46
  3.5.3 Request/reply communication 48
3.6 Correctness of the Refinement Procedure 49
  3.6.1 PVS proof of correctness 50
    3.6.1.1 Construction of *abs* 51
    3.6.1.2 Construction of *aug* 52
  3.6.2 Proof of forward progress 53
3.7 Refinement of an Example Protocol 54
  3.7.1 Model checking efficiency 57
3.8 Buffer Requirements and Fairness 57
3.9 Concluding Remarks and Future Directions 58

4. FORMAL MEMORY MODELS 60

4.1 Chapter Overview 60
4.2 Introduction 60
4.3 Program and Execution 61
4.4 Sequential Consistency 61
4.5 Coherency 62
4.6 Parallel Random Access Memory 63
4.7 Processor Consistency 64
4.8 Linearizability 65
4.9 Memory Model Verification Problem 66
4.10 Concluding Remarks 68

5. MEMORY MODEL VERIFICATION 69

5.1 Chapter Overview 69
5.2 Introduction 69
5.3 Related Work 72
5.4 Overview of ARCHTEST 73
5.5 Test Model Checking 75
  5.5.1 Assumptions about memory systems realized in hardware 76
  5.5.2 Creation of test automata 76
  5.5.3 Abstracting Test 1 77

5.5.4 Abstracting Test 2 78
5.5.5 Abstracting Test 3 79
5.6 Case Studies 81
5.6.1 Sequential consistency and serial memory protocol 81
5.6.2 Serial memory and lazy caching 81
5.6.3 Runway 83
5.6.4 VIS verification results 84
5.6.5 PV and SPIN verification results 85
5.7 Complete Tests Based on the Test Model Checking Approach 86
5.7.1 Verifying (CMP, RO, WOS) 87
5.7.1.1 One address test for (CMP, RO, WOS) 88
5.7.1.2 Two address test for (CMP, RO, WOS) 92
5.7.2 Verifying (CMP, POS) 93
5.7.3 Verifying (CMP, POS, WA) 94
5.7.4 Application of the complete tests to the Runway model 99
5.8 Concluding Remarks 100

6. CONCLUDING REMARKS AND FUTURE DIRECTIONS 101
6.1 Contributions 101
6.2 Extensions 102
6.2.1 Partial order reductions 102
6.2.2 Protocol synthesis 102
6.2.3 Memory model verification 103
6.3 The Future of Formal Verification 103
6.3.1 More efficient algorithms 103
6.3.2 Refinement 104

APPENDICES

A. FORMAL MODEL OF A SHARED MEMORY SYSTEM 105

B. COMPLETE TESTS FOR MEMORY MODEL VERIFICATION 119

REFERENCES 137

LIST OF FIGURES

1.1 State explosion problem 3

1.2 Symmetry reductions and partial order reductions 4

1.3 Typical multiprocessor configuration 9

2.1 Basic idea behind the partial order reduction algorithms 15

2.2 Basic depth first search algorithm 22

2.3 Proviso based partial order reduction algorithm 23

2.4 A trivial system, and its optimal reduced graph, and the reduced graph generated by `dfs_po` 24

2.5 Two phase algorithm 25

2.6 Exetending a transition sequence while preserving LTL-X properties 28

2.7 Reordering two transitions in a sequence while presreving LTL-X properties 29

2.8 An on-the-fly model checking algorithm 32

3.1 Examples of communication states in the home node and remote nodes . . . 43

3.2 The commute diagram 50

3.3 Home node of the migratory protocol 55

3.4 Remote node of the migratory protocol 55

3.5 Refined home node of the migratory protocol 56

3.6 Refined remote node of the migratory protocol 56

4.1 Abstracting away all instructions other than memory instructions 61

4.2 Operational semantics of SC 62

4.3 Examples for sequential consistency and coherency 62

4.4 An execution that is not coherent 63

4.5 Operational semantics of PRAM 63

4.6 An execution that is not PRAM, but coherent 64

4.7 An execution that is coherent and PRAM but not PC 65

4.8 Operational semantics of linearizability 65

4.9 An ambiguous execution that is not PRAM 67

4.10 Unambiguous executions that are not PRAM 68

5.1 ∀CTL* specification of sequential consistency for lazy caching protocol 73
5.2 Test 1: A test to check for (CMP, RO, WOS) 75
5.3 Test 2: A test to check for (CMP, RO, WOS, WA) 75
5.4 Abstraction of Test 1 78
5.5 Test automata for Test 2 79
5.6 Test 3: A test and corresponding test automaton for (CMP, POS) 80
5.7 Complete test for (CMP, RO, WOS) using one address 88
5.8 Complete test for (CMP, RO, WOS) using two addresses 92
5.9 Complete test for (CMP, POS) using one address 94
5.10 Complete test for (CMP, POS) using one addresses 94
5.11 Complete test for (CMP, POS, WA) using one address 95
5.12 Complete test for (CMP, POS, WA) using two addresses 98
A.1 Structure of a component 106
A.2 Example executions of CMP, RO, and WOS 114
B.1 Computation of G_c from G_R 123
B.2 Example construction of G_c 124
B.3 An example circuit violating CMP, RO, WOS, and its transformation 130
B.4 A CMP, POS, WA violation 132
B.5 (CMP, POS, WA) violation: Case 1 133
B.6 (CMP, POS, WA) violation: Case 2 134
B.7 (CMP, POS, WA) violation: Case 3 134
B.8 (CMP, POS, WA) violation: Case 4 135
B.9 (CMP, POS, WA) violation: Case 5 135

LIST OF TABLES

2.1 Number of states visited and the time taken in seconds by the `dfs_po` algorithm and `Twophase` algorithm on various protocols 34

3.1 The actions of the remote node 45

3.2 Actions taken by the home node 47

3.3 Verification of rendezvous and asynchronous protocols. 58

5.1 Serial memory transaction rules 82

5.2 Gerth's version of the lazy caching protocol 82

5.3 Memory model verification using VIS 84

5.4 Runway memory model verification using SPIN and PV 86

5.5 Application of complete tests to Runway memory model 99

ACKNOWLEDGMENTS

This thesis would not have been possible without the guidance and encouragement of Ganesh Gopalakrishnan throughout my graduate career at University of Utah. Ganesh has spent countless hours coaching me on how to conduct research and how to report the results. I am also greatful for the numerous discussions I had with John Carter about subtle differences between various memory models, as well as the design decisions behind contemporary shared memory systems. I have also greatly benefited from the discussions I had with Al Davis regarding conducting and presentation of research. I have learned much from his insight into the industrial research.

Bob Kurshan has explained the basics of the model-checking, which formed the basis of this dissertation. The numerous discussions I had with Gerald Holzmann taught me how to separate good ideas from bad. My work has also benefited from the insightful comments from Ken McMillan and Norris Ip and the discussions with Francisco Corella regarding the temporal logics. David Dill and Paliath Narendran also have contributed with their comments.

I would also like to thank the UV group, Ravi Hosabettu, Rajnish Ghughal, Abdel Mokkedem, and Mike Jones. In particular, the discussions I had with Ravi and Abdel provided me with a more balanced view of the theorem proving.

Ravi Kuramkote and Chen-Chi Kuo have spent a lot of time explaining the design decisions made as part of Avalanche project which in the end helped me to pick the shared memory system design as the topic for my dissertation. Ravi, Chen-Chi, and Mohamed Dekhil have listened patiently while I vented my frustrations, sometimes for hours, at my apparent lack of progress in my research.

I would especially like to thank my family—my parents, sisters, grandparents, and uncles—for their love and support throughout my life. Last, but not least, I would like to thank every one in the CS front office, especially Colleen Hoopes, for going out of their way to help me throughout my stay here.

CHAPTER 1

INTRODUCTION

1.1 Formal Methods

With the increasing complexity of the hardware and software systems, formal verification of such systems is an important practical need. Many modern hardware and software systems are designed as a collection of components that run concurrently or in parallel in order to achieve higher performance. Each of these components themselves may still further be a collection of subcomponents running concurrently. For example, in a multiprocessor system the processors run concurrently with each other. Each processor in turn may contain a cache controller that runs concurrently with the execution unit of the processor. The execution unit may be pipelined with different stages of the pipeline running concurrently. The components at each level employ sophisticated protocols for coordinating their actions to provide a higher-level functionality: a cache controller provides a consistent view of the shared memory and a pipelined execution unit implements the instruction set architecture (ISA).

Even though most protocols follow a simple high level algorithm, the detailed implementation is usually difficult to understand. For example, a typical high level specification of a cache controller is to invalidate all other cached copies in the system atomically before allowing a write by the local processor. Such a capability for performing atomic actions is seldom available. As a result, the atomic step must be broken down, or *refined*, into several smaller actions each of which can be implemented atomically. This division of larger *logically* atomic actions into smaller *physically* atomic actions introduces race conditions that are not present in the high level protocol. In the cache controller example, the protocol must deal with the possibility that two controllers may wish to invalidate each other simultaneously. If proper care is not taken, such *race conditions* can result in an incorrect operation of the system ("safety violation"), the system coming to a complete halt ("deadlock"), or system not making any meaningful forward progress ("livelock").

The most commonly used approach to debugging the protocols is to run them using simulated environments, random test vectors, test vectors generated from the structure of the protocol, or test vectors collected over long periods of time. Such simulation methods (or testing methods) are quite successful at finding many kinds of bugs. However, they suffer from the disadvantage that they are not guaranteed to find all bugs in the protocol. As the complexity of a protocol increases, the coverage obtained by simulation techniques decreases. Since simulation is also much slower than running the same test vectors on the final product, the test vectors are usually limited in number and length, hence may not cover many aspects of the protocol. This limitation may cause simulation to miss many corner cases. Another short coming of simulation methodology is that it is not applicable until much later into the design cycle. Due to these two disadvantages, there has been considerable industrial interest in formal techniques that provide higher quality-assurances by complementing simulation techniques. Such formal techniques can be divided into two broad classes: formal verification techniques and formal design derivation techniques.

1.1.1 Formal verification techniques

Formal verification techniques refers to the class of techniques that answer whether a given protocol (sometimes referred to as system or model) satisfies a given property. These techniques can be further classified into model checking [18] and theorem proving [5].

1.1.1.1 Model checking

Model checking refers to the problem of deciding whether a given protocol P "models" or satisfies a property ϕ, where ϕ is expressed in a suitable logic such as linear temporal logic (LTL) or computation tree logic (CTL) [87]. Linear temporal logic is explained in Section 2.4.1. The notation $P \models \phi$ is used to indicate that P models ϕ. When P is a finite state system, one can develop automatic procedures to decide the truth value of $P \models \phi$. In this dissertation, a model checker refers to a tool that implements such an *automated* algorithm.

Model checkers differ from simulators in two respects. First, the model checkers support a richer property language than simulators. For example, consider the property that when a component generates a request for a resource, it always receives a response from the resource manager. This property cannot be expressed in simulators due to the unbounded time difference between when a request is sent and a reply is received; but this can be expressed in the logic supported by most model checkers. Second, model checkers operate on a reduced model and cover the model completely. In contrast, simulators operate on a complete model but cover the model only partially, as driven by the test vectors or test bench. Due to these differences, model checkers complement the traditional simulators and have made considerable inroad into industry [7, 27, 51, 80].

Model checkers construct a graph of all reachable state of P starting from its initial state and check if the property ϕ holds in the graph. Depending on how the reachable states are stored, they can be classified into two classes: explicit enumeration based [26, 52] and implicit enumeration based [12, 67]. An explicit enumeration based model checker stores the reachable states in a *hash table* as they are visited. An implicit enumeration based model checker stores the reachable states in *a symbolic form* such as a boolean expression using binary decision diagrams (BDDs) [8, 9]. It has been observed that for many data intensive circuits and hardware descriptions, implicit enumeration based algorithms perform better than explicit enumeration algorithms, while for control intensive protocols explicit enumeration based algorithms perform better.

Model checkers suffer from state explosion problem: typically the number of reachable states grows exponentially as the size of the system. A trivial system with two components each containing three states is shown in Figure 1.1(a). Each state of an individual component is between 0 and 2, and each state of the composite system is a pair (i, j) where i is the state of P_1 and j is the state of P_2. The state graph of this system contains nine states, as shown in Figure 1.1(b). A straightforward generalization shows that when this trivial system contains n components, its state graph contains 3^n states.

Two popular methods of dealing with the state explosion are symmetry reductions [16, 17, 53, 69, 79, 80] and partial order reductions [34, 37, 49, 55, 70–72, 84–86]. Symmetry reductions refer to the class of algorithms that attempt to reduce the number of reachable

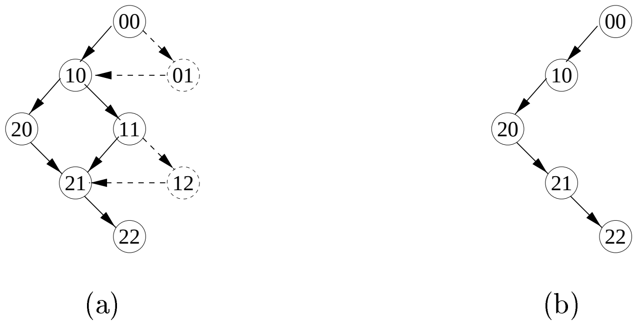
(a) (b)

Figure 1.1. State explosion problem

states when two states are "identical" under a given symmetry relation on the states by expanding only one of the two states. In the above example, if the property ϕ does not distinguish between the two components, i.e., if P_1 and P_2 are assumed to be identical components, then state (i, j) and (j, i) can be treated as equivalent. Figure 1.2(a) shows a possible state graph generated by a symmetry reduction algorithm for the system in Figure 1.1(a). When the algorithm generates the state (0, 1), it notices that an equivalent state—namely (1,0)—has already been explored and hence does not explore (0, 1). Similarly, the state (1,2) is not explored as (2, 1) is already explored. This state graph contains 3 fewer states than the full state graph in Figure 1.1(b): (0, 1) and (1, 2) are generated, but are not part of the state graph, and (0, 2) is never generated (but its equivalent state (2,0) is explored). When the protocol contains n identical components, symmetry reductions can reduce the number of reachable states by a factor of $\mathit{factorial}(n)$.

Partial order reductions, on the other hand, reduce the size of the graph by exploiting the fact that, in realistic protocols, many transitions commute with each other. These algorithms select a subset of transitions at each state and expand only those transitions instead of expanding all transitions. The algorithms ensure that the selected transitions are sufficient to preserve the truth value of ϕ. Using the protocol in Figure 1.1, *without requiring* P_1 *and* P_2 *to be identical under* ϕ, a possible graph generated by a partial order reduction algorithm is shown in Figure 1.2(b). This graph is obtained by observing that every transition of P_1 commutes with every transition of P_2. Hence the algorithm can select P_1's transitions until there are no more P_1 transitions, and after that it can select P_2's transitions. Partial order reduction algorithms can reduce the number of reachable states of a protocol by an exponential in the number of components in the protocol.

One of the contributions of this dissertation is a new partial order reduction algorithm called two phase, presented in Chapter 2. This algorithm is implemented in an explicit



(a) (b)

Figure 1.2. Symmetry reductions and partial order reductions

state enumeration based model checker called Protocol Verifier ("PV"). On many practical cases, the two phase algorithm generates a much smaller graph than the previous partial order reduction algorithms. Another advantage of the two phase algorithm is that it supports *selective caching*, which the current partial order reduction implementations do not support. This can further reduce the number of states stored in the hash table.

Designer of a protocol is interested in a number of properties such as if the protocol is free of deadlocks, if an assertion such as mutual exclusion condition always holds, if forward progress is guaranteed under all circumstances, or if forward progress is guaranteed when the scheduler is fair. The model checker accepts a finite state model of the protocol and one such property and checks if the property is true or not of the model. If the property does not hold, then the model checker generates an *error trace.*

If the property is either deadlock freedom or an assertion, the error trace is a finite sequence of states that shows how the initial state can reach either a deadlocked state or a state where the assertion fails. If the property of interest is forward progress guarantee either under all circumstances or under a fair scheduler, the error trace cannot be a finite sequence: one cannot conclude from a finite sequence whether progress can be made in the future. Hence, the trace must be an infinite sequence where no progress is made. In a finite state system, infinite sequences appear as *loops*; i.e., the error trace shows how the initial state can reach a state, say, S_1, and how S_1 can reach itself by means of a loop $S_1 S_2 \ldots S_n S_1$ where no progress is made along this loop.

A property is said to be a safety property if its violation can be shown by a finite error trace. Otherwise, it is said to be a liveness property. Deadlock freedom and assertions are safety properties, and the forward properties are liveness properties.

Automated model checking methods suffer from two limitations. The first limitation, state explosion, has been already mentioned. The second limitation is that the specification language—the logic in which ϕ is expressed—is not very powerful. These logics allow one can express such properties as deadlock freedom, assertions, and forward progress, but *not high level requirements* such as "cache controller implements a coherent view of the shared memory." As a result, such requirements need to be broken into several smaller properties. In some cases the semantic gap between the high level requirement and the specification language is so high that the set of properties is not exactly equivalent to the requirement; i.e., the set of properties is stronger or weaker than the requirement. In some cases, this limitation is a real concern.

Another contribution of the dissertation is a hybrid approach called *test model checking* that shows how, in some restricted contexts, this limitation can be overcome. Test model checking is a hybrid of testing and model checking approaches and is presented in Chapter 5. This approach can be used to verify whether a shared memory system correctly implements a high level requirement such as sequential consistency.

1.1.1.2 Theorem proving

Theorem proving usually refers to the class of verification techniques where the proof is done by a human, possibly with the help of a *theorem prover*. Since the proof is a manual process, it is *not* limited to finite state systems. Some modern theorem provers such as PVS [73] and Isabelle [76] implement powerful decision procedures. Unlike model checkers, theorem provers typically have a very rich specification language; hence it is easy to express high level requirements of a protocol in a theorem prover's specification language. Another strength of theorem provers is that an algorithm can be proved to be correct once, thereby avoiding case by case verification. However, the proofs for even the simplest of the theorems can be very tedious and involved, even when powerful decision procedures are used. As a result, the theorem proving techniques are not yet as widely used in the industry as model checkers. In some safety critical environments, such as NASA [13], theorem provers have been used with considerable success.

Another contribution of the thesis is a design derivation technique (explained below) for distributed shared memory system protocols. This technique has been verified using PVS.

1.1.2 Formal design derivation techniques

Given the inability of traditional simulation methods to cover the design adequately, state explosion problem of model checking, and the tedium involved in using theorem proving, automated procedures for developing protocols are growing in importance. *Refinement* procedures, which are defined in this dissertation to be those that accept high level protocol specifications, apply provably correct transformations or *refinement rules* on them to yield detailed implementations of protocols that run efficiently and have modest resource requirements. Such procedures enable correctness proofs of protocols to be carried out with respect to high level specifications, which can considerably reduce the proof effort. For example, a high level specification of a cache coherence protocol typically contains 10s of transitions, whereas an equivalent detailed implementation contains 100s of transitions [74,75]. In addition, the state space of the implementation protocol is much larger than the high level specification, as the former contains state related to message transmission that is not normally present the later. As a result, it is much more efficient to verify the high level protocol using either a model checking or a theorem prover. Once the refinement rules are shown to be sound, the detailed protocol implementations produced by those refinement rules need not be verified. Chapter 3 presents a refinement technique that can be used in the design of distributed shared memory protocols.

The formal techniques discussed so far are *generic*, in the sense that they are applicable to a number of domains. These techniques can be specialized to a particular domain to obtain more efficient algorithms or to address concerns specific to the domain. In this dissertation, formal verification and design derivation techniques are applied to shared memory system domain.

1.2 Shared Memory System Design

Memory system design is an ideal candidate for applying formal methods. The performance of the memory subsystem is usually one of the major limiting factors of any computer system's performance [81], typically referred to as the memory bottleneck problem. Due to technology differences between the processor fabrication and main memory fabrication, the performance gap between the two is increasing exponentially: memory latency has been improving at a rate of 7% per year, whereas the processor performance has been improving at 55% per year [43]. The current trend of connecting multiple processors together to form a multiprocessor aggravates the imbalance further as the memory needs to serve multiple fast processors. One way to reduce this gap is to implement main memory by employing the same semiconductor fabrication processes as used in implementation of the processors. Unfortunately, this is not an economical option. The speed mismatch between the main memory and processors, hence, is expected to grow at the current rate for foreseeable future. Current architectures address this problem by employing a hierarchical memory organization, based on *principle of locality.* The basic idea is to organize the memory in a hierarchy such that a level closer to the processor is faster (and more expensive per bit) but smaller than a level further away. If the memory is accessed with sufficient locality in time and space, then the most frequently used data would reside in the fastest memory. Two-level memory hierarchy (1-level cache) and three-level memory hierarchies (2-level caches) are the most common organizations at the current time. Most modern caches also have builtin support for multiprocessors, for example, by replicating the tags, and providing snooping on a shared bus.

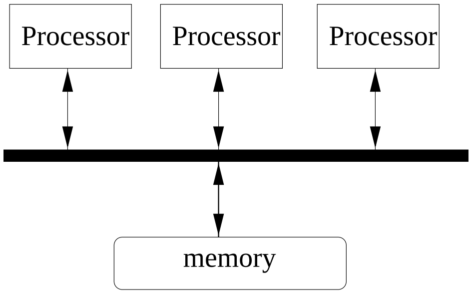
The memory subsystem is responsible for maintaining the consistency between the data stored in caches and the main memory. If the protocol used by cache controller is not very efficient, the performance of a machine suffers badly [81]; hence a conservative design is usually not a viable option. Efficient solutions to consistency problem can be found with relative ease in uniprocessor system as all other components that access the main memory such as disk controllers are under the processor control. In a multiprocessor, however, such is not the case, as two or more processors may attempt to access the data simultaneously resulting in race conditions.

To summarize, shared memory protocols tend to be *complex* for the sake of efficiency. However, their correctness is *critical* for the correctness of the entire system. Hence, these protocols are ideal candidates for benefiting from formal methods.

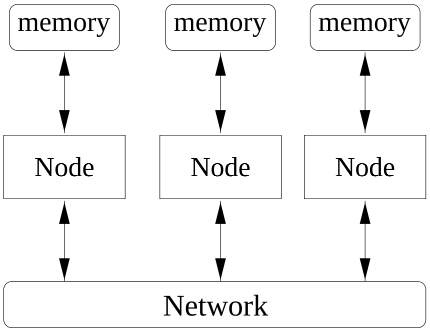
1.2.1 Shared memory multiprocessor organization

Two most common methods of building shared memory multiprocessors are to connect the processors and main memory using a shared bus to form a symmetric multiprocessor (SMP) as shown in Figure 1.3(a), or using a switch or a network to form a distributed shared memory (DSM) system as shown in Figure 1.3(b). In Figure 1.3(b), *Node* indicates that each of the building boxes themselves may be a SMP node. The bandwidth and latency of the shared bus in SMP systems becomes a bottleneck around 10 processors. Hence for larger configurations, DSM systems are normally used.

A typical transaction in these systems starts with a cache miss, followed by arbitration for the bus and/or network port, transmission of a request, and waiting for the response either from the memory or from another processor. This sequence of actions can take 100s of clock cycles, which in a modern pipeline microprocessor could mean 100s of wasted instruction slots. To reduce this penalty, some modern multiprocessors implement *relaxed* memory models; i.e., instead of stalling the instruction issue on a cache miss, the processor would continue to issue other instructions. This means that if the processor has two memory instructions and the first one resulted in a cache miss, it could still issue and *complete* a later memory instruction. This situation does not cause any trouble in uniprocessor machines, as no other component may observe the data; hence caches can be viewed as an *invisible* optimization. In multiprocessors, however, such an out-of-order execution of memory instructions may be visible to programs running on other



(a) SMP system



(b) DSM system

Figure 1.3. Typical multiprocessor configuration

processors. In other words, the organization of the memory system may expose one or more *architectural features* to the application program. Since the behavior of the memory is a basic contract between the hardware designers and the programmers, it is important to ensure that the protocols did not *unintentionally* expose an architectural feature that was supposed to be hidden.

1.2.2 Memory model verification

Formal memory models, which are defined in this dissertation to be the models that define how memory accesses may be reordered (explained in detail in Chapter 4), determine whether a given concurrent program can produce a given output. The memory model verification problem is to determine whether a shared memory system correctly implements a given formal memory model. This problem appears not only in the context of shared memory system, but also in the context of memory bus and I/O bus design, multithreaded language and compiler design, database system design, and out-of-order execution system design. Unfortunately, despite the central importance of this problem and the large body of formal methods research in this area, there is still no single formally based method that the designer of a realistic multiprocessor system can use on his/her detailed design model *without unduly increasing the design time.* The reasons are that (a) due to the considerable effort involved in using theorem provers, they increase the design time and (b) the requirements of a formal memory model cannot be expressed in the logics provided by model checkers [4]; hence considerable manual effort is needed before model checkers can be used [41].

Chapter 5 presents a verification method called *test model checking* that addresses this deficiency by formally adapting and *extending* an architectural testing method called ARCHTEST [21] to the realm of model checking. ARCHTEST is an *incomplete* testing method in that it is not guaranteed to detect all violations of formal memory models. In contrast, test model checking is a *complete* method in that it is guaranteed to find all violations of formal memory model. ARCHTEST's methodology is meant to be applied to *real machines*, whereas test model checking is mainly meant to be applied to high level models of the memory system *early in the design cycle.*

1.3 Contributions of the Dissertation

The dissertation is based on the hypothesis that specializing formal methods for a particular domain leads to *efficient* verification techniques applicable to the designs arising in the domain, as well as increases the *applicability* of formal methods. In other words, domain specific formal methods can solve verification problems that could not be solved in a pure theoretic setting. The hypothesis is demonstrated in the domain of shared memory systems by making the following key contributions.

- A new partial order reduction called *two phase* that typically generates far fewer states than comparable algorithms and is especially effective in memory protocols. This algorithm shows how the efficiency of model checkers can be improved by the domain specific heuristics.

- A design derivation algorithm for designing cache coherence protocols for implementing distributed shared memory and its proof of correctness using a theorem prover. This algorithm shows that a derivation algorithm targeted for a domain can transform a high level specification into an efficient implementation.

- A verification technique to verify whether a given shared memory protocol implements a formal memory model such as sequential consistency [58,59]. This technique shows that even though complex properties such as sequential consistency cannot be expressed in the logics provided by model checkers [4], by adopting testing methods into the realm of model checking and defining a formal model of a multiprocessor, the limitation can be effectively eliminated.

1.3.1 Partial order reductions

As already mentioned, partial order reductions combat the state explosion by not visiting some of the intermediate states. The basic idea behind these algorithms is that, in most realistic protocols, there are many transitions that "commute" with each other. Hence it is sufficient to explore those transitions in any *one order* to preserve the truth value of the temporal property under consideration. When such transitions are explored in only one order, many intermediate states are not generated; hence the graph constructed is smaller, which helps reducing both the time and space demands of a model checker. In other words, instead of exploring all transitions from a given state, a partial order reduction algorithm explores only a *subset* of transitions that are sufficient to preserve the truth value of the temporal properties under consideration, postponing exploration of the rest of the transitions to the successors of the state. However, care must be taken to ensure that no transition is postponed forever, commonly referred to as *ignoring problem*. Previous implementations solved the ignoring problem by using a condition called *proviso* (explained further in Chapter 2). Chapter 2 shows that in a large number of practical examples, especially those arising in memory protocol verification domain, the provisos cause all existing partial order reduction algorithms to be ineffective. Chapter 2 also presents a new partial order algorithm called two phase that does not use proviso. Two phase, in all practical cases, outperforms the proviso based partial order reduction algorithms. Another advantage of the two phase is that, it naturally supports *selective caching*, which can further reduce the memory and time requirements.

1.3.2 Design derivation technique

Many protocols are sufficiently complicated that, even with partial order reductions, they cannot be analyzed completely for required properties. This problem is readily apparent in many shared memory protocols, as these protocols tend to be complicated so as to hide the large difference between the memory response time and the processor speed; hence it is difficult to verify such shared memory protocols. Chapter 3 presents a protocol *refinement* procedure that accepts a high level specifications of distributed shared memory (DSM) protocols and apply provably correct transformations on them to yield detailed implementations of protocols. The efficiency of such synthesized detailed implementations is comparable to that of a hand-written protocols. In addition, the synthesized implementations also have modest resource requirements. Such a refinement procedure enables correctness proofs of protocols to be carried out with respect to high level specifications, which considerably reduces the proof effort. Chapter 3 also presents a PVS proof that the refinement procedure preserves safety properties and a manual proof that the procedure preserves forward progress ("liveness") properties.

1.3.3 Memory model verification

As already explained, the logic supported by most model checkers is too weak to express that a given shared memory system correctly implements a given formal memory model. Chapter 5 presents a novel technique called test model checking to address this problem. Test model checking *adopts* and *extends* a formal testing method called ARCHTEST, to the realm of model checking. The major differences between ARCHTEST and test model checking are as follows:

1. ARCHTEST is an incomplete method in that not all violations may be caught, whereas test model checking is a complete method and
2. the tests of ARCHTEST cannot be used until much later into the design cycle, whereas test model checking can be used much earlier in the design cycle.

The basic idea behind the test model checking is to construct a program and a simple safety property for the formal memory model such that when the program is run on the model, a model checker would detect the violation of the safety property if and only if the model does not implement the formal memory model. The advantage of using a testing approach to solving the memory verification problem is that even though the logics of model checkers cannot express the property that the memory system conforms to the formal memory, they can easily express the safety condition associated with the program.

1.4 Organization of the Dissertation

The chapters in this dissertation are mostly self-contained facilitated by the fact that the three presented techniques can be understood independent of each other. Chapter 2 describes a new partial order reduction algorithm called two phase and presents its correctness. Chapter 3 presents a refinement algorithm to transform a high level shared memory protocol into a detailed implementation with modest buffer requirements. Chapter 4 summarizes various formal memory models and presents background into graph theory to be used in test model checking. Chapter 5 presents how an implementation can be verified to test whether it conforms to a given formal shared memory model. Chapter 6 provides concluding remarks. Finally, Appendices A and B present proofs of theorems used in Chapter 5.

CHAPTER 2

A PARTIAL ORDER REDUCTION ALGORITHM WITHOUT THE PROVISO

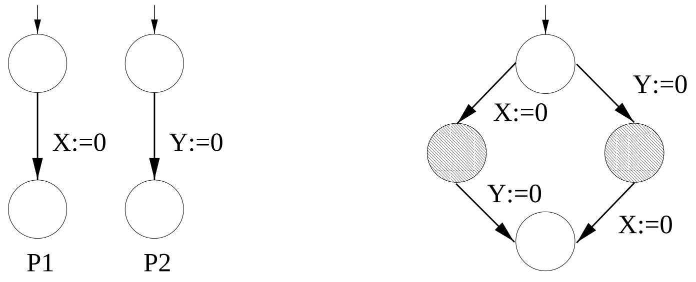
2.1 Chapter Overview

This chapter presents a partial order reduction algorithm called *two phase* that greatly increases the size of the model that a model checker can handle. The algorithm is also shown to preserve all stutter free linear time temporal logic formulae.

2.2 Introduction

With the increasing scale of software and hardware systems and the corresponding increase in the number and complexity of concurrent protocols involved in their design, formal verification of concurrent protocols is an important practical need. Automatic verification of finite state systems based on explicit state enumeration methods [18,26,47,52] has shown considerable promise in real-world protocol verification problems and have been used with success on many industrial designs [27,51]. Using most explicit state enumeration tools, a protocol is modeled as a set of concurrent processes communicating via shared variables and/or communication channels [26,52]. The tool generates the state graph represented by the protocol and checks for the desired temporal properties on that graph. A common problem with this approach is that state graphs of most practical protocols are quite large and the size of the graph often increases exponentially with the size of the protocol, commonly referred to as *state explosion*.

The interleaving model of execution used by these tools is one of the major causes of state explosion. This is shown through a simple example in Figure 2.1. Figure 2.1(a) shows a system with two processes P1 and P2 and Figure 2.1(b) shows the state space of this example. If the property under consideration does not involve at least one of the variables X and Y, then one of the two shaded states need not be generated, thus saving



(a) Simple system (b) State graph

Figure 2.1. A simple system and its state graph

one state. A straightforward extension of this example to n processes would reveal that an interleaving model of execution would generate 2^n states where $n+1$ would suffice.

Partial order reductions attempt to bring such reductions by exploiting the fact that in realistic protocols there are many transitions that "commute" with each other, and hence it is sufficient to explore those transitions in any *one order* to preserve the truth value of the temporal property under consideration. In essence, from every state, a partial order reduction algorithm selects a *subset* of transitions to explore, whereas a normal graph traversal such as depth first search (DFS) algorithm would explore all transitions. Partial order reduction algorithms play a very important role in mitigating state explosion, often reducing the computational and memory cost by an exponential factor. This chapter presents a new partial order reduction algorithm called two phase, that in most practical cases outperforms existing implementations of the partial order reductions. The algorithm is implemented in a tool called PV ("Protocol Verifier") that finds routine application in our research.

To our knowledge, so far there have been only two partial order reduction algorithms which have implementations: the algorithm presented in [49, 78] and the algorithm presented in [35]. The algorithm in [49, 78] is implemented in the explicit state enumeration model checker SPIN and in implicit state exploration tools VIS and COSPAN [3, 55, 72]. The algorithm in [35] is implemented in PO-PACKAGE tool. Both these algorithms solve the ignoring problem by using *provisos*, whose need was first recognized by Valmari [85]. Provisos ensures that the subset of transitions selected at a state do not generate a state that is in the stack maintained by the DFS algorithm. If a subset of transitions satisfying this check cannot be found at a state s, then all transitions from s are executed by the DFS algorithm. The provisos used in the two implementations differ slightly.

The PO-PACKAGE algorithm (and also the algorithm presented in [48]) requires that at least one of the selected transitions do not generate a state in the stack, whereas SPIN algorithm requires the stronger condition that no selected transition generates a state in the stack. The stronger proviso is sufficient to preserve all stutter free linear time temporal logic (LTL-X) formulae (safety and liveness), whereas the weaker proviso preserves only stutter free safety properties [48, 49, 77, 78].

It is observed that in a large number of practical examples arising in reactive systems, such as validation of directory based coherence protocols and server-client protocols, the provisos cause all existing partial order reduction algorithms to be ineffective [70]. As an example, on *invalidate*, a distributed shared memory protocol described later, SPIN aborts its search by running out of memory after generating more than 270,000 states when limited to 64MB memory usage. PO-PACKAGE also aborts its search after generating a similar number of states. In *invalidate*, there are many opportunities for partial order reductions to reduce the complexity; hence, protocols of this complexity ought to be easy for on-the-fly explicit enumeration tools to handle—an intuition confirmed by the fact two phase, a partial order reduction that does not use the proviso, finishes comfortably on this protocol. In fact, as showed in Section 2.8, in all nontrivial examples, two phase outperforms proviso based algorithms. Two phase is implemented in a model checker called PV ("Protocol Verifier").

The first major difference between two phase and other partial order reduction algorithms is the way the algorithms expand a given state. Other partial order reduction algorithms attempt to expand each state visited during the search using a subset of enabled transitions at that state. To address the ignoring problem, the algorithms use a proviso (or a condition very similar to the provisos). Two phase search strategy is completely different: when it encounters a new state x, it first expands the state using only *deterministic* transitions in its first phase resulting in a state y. (Informally, deterministic transitions are the transitions that can be taken at the state without effecting the truth property of the property being verified.) Then in the second phase, y is expanded *completely.* The advantage of this search strategy is that it is not necessary to use a proviso. As the results in Section 2.8 show, this often results in a much smaller graph.

The second major difference is that two phase naturally supports *selective caching in conjunction with on-the-fly model checking.* An explicit enumeration search algorithm typically saves the list of visited states in a hash table ("cached"). Since the number of visited states is large, it would be beneficial if not all visited states need to be stored in the hash table, referred to as selective caching. On-the-fly model checking means that the algorithm finds if the property is true or not as the state graph of the system is being constructed (as opposed to finding only after the graph is completely constructed). It is difficult to combine the on-the-fly model checking algorithm, partial-order reductions, and selective-caching together due to the need to share information among these three aspects. [50] showed that previous attempts at combining proviso based algorithms with the on-the-fly algorithm presented in [23] have been erroneous. However, thanks to the simplicity of the first phase of two phase algorithm, it can be combined easily with the on-the-fly algorithm presented in [23]. Also two phase lends itself to be used in conjunction with a simple but effective selective-caching strategy.

To summarize, the contributions of this chapter are as follows:

1. A new partial order reduction algorithm called two phase that does not use the proviso,
2. A proof of correctness of two phase,
3. A selective caching scheme that can be quite naturally integrated with two phase, and
4. An evaluation of performance of the algorithm compared to other implementations using the PV model checker.

The rest of the chapter is organized as follows. Section 2.4 presents definitions and background. Section 2.5 presents the basic depth first search algorithm, the partial order reduction algorithm presented in [78] (algorithms in [35, 49, 85] are very similar), and the two phase algorithm, as well as a proof that the two phase preserves all LTL-X properties. Section 2.7 presents the on-the-fly model checking algorithm presented in [23] and discusses on how it can be combined with two phase. This section also presents a selective caching strategy and shows how it can be combined with two phase. Section 2.8 compares the performance of [78] algorithm (implemented in Spin) with that of two phase (implemented in PV) and provides a qualitative explanation of the results. Finally, Section 2.9 provides concluding remarks.

2.3 Related Work

Lipton [64] suggested a technique to avoid exploring the entire state graph to find if a concurrent system deadlocks. Lipton notes that execution of some transitions can be postponed as much as possible (*right movers*) and some transitions can be executed as soon as possible (*left movers*) without affecting the deadlocks. Partial order reductions can be considered as a *generalization* of this idea to verify richer properties than just deadlocks.

Valmari [85,86] has presented a technique based on *stubborn sets* to construct a reduced graph to preserve the truth value of all stutter free LTL formulae. The algorithm in [85] uses a general version of the proviso mentioned above. The algorithm in [86] does not use the proviso, but avoids the ignoring problem by choosing stubborn sets that *always* include all transitions that affect the LTL formulae. [35–37] present a partial order theory based on traces to preserve safety properties, using a slight variation of the proviso, implemented in PO-PACKAGE. [78] presents a partial order reduction algorithms based on *ample* sets and the strong proviso. [49] presents an algorithm very similar to and based on the algorithm presented in [78], implemented in SPIN [52]. The algorithm in [78] is discussed in Section 2.5. Since the implementations of the two algorithms are similar, whenever one algorithm fails to bring much reductions, so does the other.

The version of the proviso discussed earlier (first appeared in [77]) is shown to be sufficient to preserve all liveness properties. In [85] a more general condition for correctness is given: if (a) every elementary loop in the reduced graph contains at least one state where all global transitions (visible transitions in their terminology) are expanded, (b) at every state s, if there is an enabled local transition, then the set of transitions chosen to be expanded at s contains at least one local transition then the reduced graph preserves all LTL-X formulae on global transitions. Two phase does not use the provisos; instead it uses *deterministic* transitions to bring the reductions. Two phase has been previously reported in [70, 71].

2.4 Definitions and Notation

A process oriented modeling language with each process maintaining a set of local variables that only it can access is assumed. The value of these local variables form the *local state* of the process. For convenience, each process is assumed to contain a distinguished local variable called program counter ("control state"). A concurrent system or simply system consists of a set of processes, a set of global variables, and point-to-point channels of finite capacity to facilitate communication among the processes. The global state, or simply the "state" of the system, consists of local states of all the processes, values of the global variables, and the contents of the channels. $\mathcal{S}$ denotes the set of all possible states ("syntactic state") of the system, obtained simply by taking the Cartesian product of the range of all variables (local variables, global variables, program counters, and the channels) in the system. The range of all variables (local, global, and channels) is assumed to be finite, hence $\mathcal{S}$ is also finite.

Each program counter of a process is associated with a finite number of transitions. A transition of a process P can read/write the local variables of P, read/write the global variables, send a message on the channel on which it is a sender, and/or receive a message from the channel for which it is a receiver. A transition may not be enabled in some states (for example, a receive action on a channel is enabled only when the channel is nonempty). If a transition t is enabled in a state $s \in \mathcal{S}$, then it is uniquely defined. Nondeterminism can be simulated simply by having multiple transitions from a given program counter. t, $t^{'}$ are used to indicate transitions, $s \in \mathcal{S}$ to indicate a state in the system, $t(s)$ to indicate the state that results when t is executed from s, P to indicate a sequential process in the system, and pc(s,P) to indicate the program counter (control state) of P in s, and pc(t) to indicate the program counter with which the transition t is associated.

local: A transition (a statement) is said to be *local* if it does not involve any global variable.

global: A transition is said to be *global* if it involves one or more global variables. Two global transitions of two different processes may or may not commute, whereas two local transitions of two different processes commute.

internal: A control state (program counter) of a process is said to be *internal* if all the transitions associated with it are *local* transitions.

unconditionally safe: A *local* transition t is said to be *unconditionally safe* if, for all states $s \in \mathcal{S}$, if t is enabled (disabled) in $s \in \mathcal{S}$, then it remains enabled (disabled) in $t^{'}(s)$ where $t^{'}$ is any transition from another process. Note that if t is an unconditionally safe transition, by definition it is also a *local* transition. From this observation, it follows that executing $t^{'}$ and t in either order would yield the same state, i.e., t and $t^{'}$ commute. This property of commutativity forms the basis of the partial order reduction theories.

Note that channel communication statements are *not unconditionally safe*: if a transition t in process P attempts to read and the channel is empty, then the transition is disabled; however, when a process Q writes to that channel, t becomes enabled. Similarly, if a transition t of process P attempts to send a message through a channel and the channel is full, then t is disabled; when a process Q consumes a message from the channel, t becomes enabled.

conditionally safe: A *conditionally safe* transition t behaves like an *unconditionally safe* transition in some of the states characterized by a *safe execution condition* $p(t) \subseteq \mathcal{S}$. More formally, a local transition t of process P is said to be *conditionally safe* whenever, in state $s \in p(t)$, if t is enabled (disabled) in s, then t is also enabled (disabled) in $t^{'}(s)$ where $t^{'}$ is a transition of process other than P. In other words, t and $t^{'}$ commute in states represented by $p(t)$.

Channel communication primitives are *conditionally safe.* If t is a receive operation on channel c, then its safe execution condition is "c is not empty." Similarly, if t is a send operation on channel c, then its safe execution condition is "c is not full."

safe: A transition t is *safe* in a state s if t is an *unconditionally safe* transition or t is *conditionally safe* whose safe execution condition is true in s, i.e., $s \in p(t)$.

deterministic: A process P is said to be *deterministic* in s, written *deterministic(P, s)*, if the control state of P in s is *internal*, all transitions of P from this control state are *safe*, and exactly one transition of P is enabled.

independent: Two transitions t and $t^{'}$ are said to be independent of each other iff at least one of the transitions is *local*, and they belong to different processes.

The partial order reduction algorithms such as [35, 49, 78, 85] use the notion of *ample set* based on *safe* transitions. The two phase algorithm, on the other hand, uses the notion of *deterministic* to bring reductions. The proof of correctness of the two phase algorithm uses the notion of *independent* transitions.

2.4.1 Linear temporal logic and Büchii automaton

A LTL-X formulae is a LTL formulae without the next time operator X. Formally, system LTL-X (*linear-time logic without next time operator* or stutter free LTL) is defined from atomic propositions $p_1 \ldots p_n$ by means of boolean connectives, $\Box$ ("always"), $\Diamond$ ("eventually"), and U ("until") operators. If $\alpha = \alpha(0) \ldots \alpha(\omega)$ is an infinite sequence of states that assign a truth value to $p_1 \ldots p_n$, ϕ a LTL-X formulae, then the satisfaction relation $\alpha \models \phi$ is defined as follows:

$$
\begin{array}{ll}
\alpha \models p_i & \text{iff } \alpha(0) \models p_i \\
\alpha \models \phi_1 \wedge \phi_j & \text{iff } \alpha \models \phi_1 \text{ and } \alpha \models \phi_2 \\
\alpha \models \neg\phi & \text{iff } \neg(\alpha \models \phi) \\
\alpha \models \Box\phi & \text{iff } \forall i \geq 0 : \ \alpha(i) \ldots \alpha(\omega) \models \phi \\
\alpha \models \Diamond\phi & \text{iff } \exists i \geq 0 : \ \alpha(i) \ldots \alpha(\omega) \models \phi \\
\alpha \models \phi_1 U \phi_2 & \text{iff } \exists i \geq 0 : \ \alpha(i) \ldots \alpha(\omega) \models \phi_2 \\
 & \quad \text{and } \forall 0 \leq j < i : \ \alpha(j) \ldots \alpha(\omega) \models \phi_1
\end{array}
$$

If M is a concurrent system, then $M \models \phi$ is true iff for each sequence α generated by M from the initial state, $\alpha \models \phi$.

Büchii automaton [87] are nondeterministic finite automata with an acceptance condition to specify which infinite word (ω-word) is accepted by the automaton. Formally, a Büchii automaton is a tuple $A = (Q, q_0, \Sigma, \Delta, F)$ where Q is the set of the states, q_0 is the initial state, Σ is the input, $\Delta \subseteq Q \times \Sigma \times Q$, and $F \subseteq Q$ is the set of final states. A *run* of A on an ω-word $\alpha = \alpha(0)\alpha(1)\ldots$ from Σ^ω is an infinite sequence of states $\sigma = \sigma(0)\sigma(1)\ldots$ such that $\sigma(0) = q_0$ and $(\sigma(i), \alpha(i), \sigma(i+1)) \in \Delta$. The sequence α is accepted by A iff at least one state of F appears infinitely often in σ.

The model checking problem, $M \models \phi$, may be viewed as an *automata-theoretic verification* problem, $L(M) \subseteq L(\phi)$ where $L(M)$ and $L(\phi)$ are languages accepted by M and the linear-time temporal formulae ϕ respectively. If an ω automaton such as the Büchii automaton $A_{\neg\phi}$ accepts the language $\overline{L(\phi)}$, the verification problem of $L(M) \subseteq L(\phi)$ can be answered by constructing the state graph of the synchronous product of M and $A_{\neg\phi}$, $S = M \otimes A_{\neg\phi}$. If any strongly connected components of the graph represented by S satisfies the acceptance condition of $A_{\neg\phi}$ then and only then ϕ is violated in M [54].

2.5 Basic DFS and Proviso Based Partial Order Reduction Algorithms

Figure 2.2 shows the basic depth first search (DFS) algorithm used to construct the full state graph a protocols. V_f is a hash table ("visited") used to cache all the states that are already visited. Statement $\boxed{1}$ shows that the algorithm expands *all* transitions

```
model_check()
{
    /* No states are visited */
    $V_f$ := $\phi$;
    /* No edges are visited */
    $E_f$ := $\phi$;
    dfs(InitialState);
}
```

```
dfs(s)
{
    $V_f$ := $V_f$ + {s};
    $\boxed{1}$ foreach enabled transition t in s do
        $\boxed{2}$ $E_f$ := $E_f$ + {(s,t,t(s)};
        if t(s) $\notin$ $V_f$ then
            dfs(t(s));
        endif
    endforeach
}
```

Figure 2.2. Basic depth first search algorithm

from a given state. Statement $\boxed{2}$ shows how the algorithm constructs the state graph of the system in E_f.

Partial order reduction based search algorithms attempt to replace $\boxed{1}$ by choosing a subset of transitions. The idea is that if two transitions t and $t^{'}$ commute with each other in a state s and if the property to be verified is insensitive to the execution order of t and $t^{'}$, then the algorithm can explore $t(s)$, postponing examination of $t^{'}$ to $t(s)$. Of course, care must be exercised to ensure that no transition is postponed forever, commonly referred to as the *ignoring problem*. The algorithm in [49, 78] is shown as `dfs_po` Figure 2.3. As already mentioned, this algorithm is implemented in SPIN. This algorithm also uses `ample(s)` to select a subset of transitions to expand at each step. When `ample(s)` returns a proper subset of enabled transitions, the following conditions must hold: (a) the set of transitions returned commute with all other transitions, (b) none of the transitions result in a state that is currently being explored (as indicated by its presence in `redset` variable maintained by `dfs_po`).

The intuitive reasoning behind the condition (b) is that, if two states s and $s^{'}$ can reach each other, then without this condition s might delegate expansion of a transition to $s^{'}$ and vice versa; hence without this condition the algorithm may never explore that transition at all. Condition (b), sometimes referred to as *reduction proviso* or simply *proviso*, is enforced by the highlighted line in `ample(s)`. If a transition, say t, is postponed at s, then it must be examined at a successor of s to avoid the ignoring problem. However, if $t(s)$ itself being explored (i.e., $t(s)$ $\in$`redset`), then a circularity results if $t(s)$ might have postponed t. To break the circularity, `ample(s)` ensures that $t(s)$ is not in `redset`. As Section 2.7.1 shows later, the dependency of `ample` on `redset` to evaluate the set of transitions has some very important consequences when on-the-fly model checking algorithms are used.

```
dfs_po(s)
{
    /* Record the fact that s is partly
       expanded in redset */
    redset := redset + {s};
    $V_r$ := $V_r$ + {s};
    /* ample(s) uses redset */
    $\boxed{1}$ foreach transition t
             in ample(s) do
        $\boxed{2}$ $E_r$:=$E_r$+{(s,t,t(s)};
        if t(s) $\notin$ $V_r$ then
            dfs_po(t(s));
        endif;
    endforeach;
    /* s is completely expanded.  So
       remove it from redset */
    redset := redset-{s};
}
```

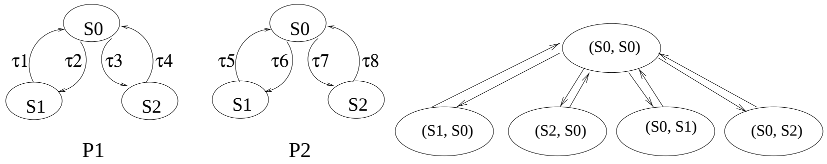
```
ample(s)
{
  for each process P do
    acceptable := true;
    T := all transitions t of P
      such that pc(t) = pc(s,P);
    foreach t in T do
      if(t is global) or
        (t is enabled and
          $\boxed{\text{(t(s) } \in \text{ redset)}}$) or
        (t is conditionally safe
             and s $\notin$ p(t)) then
          acceptable := false;
      endif
    endforeach;
    if acceptable and T has at least
          one enabled transition
      return enabled transitions in T;
    endif;
  endforeach;

  /* No acceptable subset of
     transitions is found */
  return all enabled transitions;
}
```

Figure 2.3. Proviso based partial order reduction algorithm

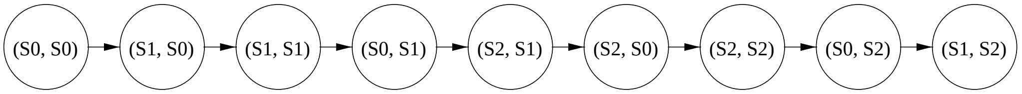
2.5.1 Efficacy of partial order reductions

The partial order reduction algorithm shown in Figure 2.3 can reduce the number of states by an exponential factor [49, 78]. However, in many practical protocols, the reductions are not as effective as they can be. The reason can be traced to the use of proviso. This is motivated using the system shown in Figure 2.4. Figure 2.4(a) shows a system consisting of two sequential processes P1 and P2 that do not communicate at all; i.e., $\tau_1 \ldots \tau_4$ commute with $\tau_5 \ldots \tau_8$. The total number of states in this system is 9. The optimal reduced graph for this system contains 5 states, shown in Figure 2.4(b).

Figure 2.4(c) shows the state graph generated by the partial order reduction algorithm in Figure 2.3. This graph is obtained as follows. The initial state is <s0,s0>. `ample(<s0,s0>)` may return either $\{\tau_1, \tau_3\}$ or $\{\tau_5, \tau_7\}$. Without loss of generality, assume that it returns $\{\tau_1, \tau_3\}$, resulting in states <s1,s0> and <s2,s0>. Again, without loss of generality, assume that the algorithm chooses to expand <s1,s0> first, where transitions $\{\tau_2\}$ of P_1 and $\{\tau_5, \tau_7\}$ of P_2 are enabled. τ_2(<s1,s0>) = <s0,s0>, and when dfs_po(<s1,s0>) is called, `redset`={<s0,s0>}. As a result ample(<s1,s0>) cannot return $\{\tau_2\}$; it returns $\{\tau_5, \tau_7\}$. Executing τ_5 from (<s1,s0>) results in <s1,s1>, the third state in the figure. Continuing this way, the graph shown in Figure 2.4(c) is obtained. Note that this system



(a) System (b) Optimal graph



(c) Graph by `dfs_po`

Figure 2.4. A trivial system, and its optimal reduced graph, and the reduced graph generated by `dfs_po`

contains all 9 reachable states in the system, thus showing that a proviso based partial order reduction algorithm might fail to bring appreciable reductions. As confirmed by the examples in Section 2.8, the algorithm may not bring much reductions in realistic protocols also.

2.6 The Two Phase Algorithm

As the previous contrived example, the size of the reduced graph generated by a proviso based algorithm can be quite high. This is true even for realistic reactive systems. In most reactive systems, a transaction typically involves a subset of processes. For example, in a server-client model of computation, a server and a client may communicate without any interruption from other servers or clients to complete a transaction. After the transaction is completed, the state of the system is reset to the initial state. If the partial order reduction algorithm uses the proviso, state resetting cannot be done as the initial state will be in the stack until the entire reachability analysis is completed. Since at least one process is not reset, the algorithm generates unnecessary states, thus increasing the number of states visited, as already demonstrated in Figure 2.4. Section 2.8 will demonstrate that in realistic systems also the number of extra states generated due to the proviso can be high.

The proposed alogirithm, two phase is shown as `Twophase` in Figure 2.5. This algorithm does not use provis, thus does not generated the extra states. In the first phase (`phase1`), `Twophase` executes deterministic processes resulting in a state `s`. In the second

```
model_check()
{
  $V_r$ := $\phi$;
  $E_r$ := $\phi$;
  /* fe (fully expanded) is used in proof */
  fe := $\phi$;
  Twophase();
}
```

```
phase1(in)
{
  s := in;
  list := {s};
  path := {};
  foreach process P do
    while (deterministic(s, P))
      /* Let t be the only enabled
         transition in P */
      olds := s;
      s := t(olds);
      path := path + {(olds, t, s)};
      if (s $\in$ list)
         goto NEXT_PROC;
      endif
      $\boxed{1}$ list :=  list + {s};
      endwhile;
    NEXT_PROC: /* next process */
  endforeach;
  return(path, s);
}
```

```
Twophase(s)
{
    /* Phase 1 */
    (path, s) := phase1(s);

    /* Phase 2: Classic DFS */
    if s$\notin V_r$ then
      /* fe is used in proof */
      $\boxed{1}$ $V_r$ := $V_r$ + all states in path;
      $\boxed{2}$ $E_r$ := $E_r$ + path;
      $\boxed{3}$ fe := fe + {s};
      foreach enabled transition t do
        $\boxed{3}$ $E_r$ := $E_r$ + (s, t, t(s));
        if t(s) $\notin$ $V_r$ then
          Twophase(t(s));
        endif;
      endforeach;
    else
      $\boxed{1'}$ $V_r$ := $V_r$ + all states in path;
      $\boxed{2'}$ $E_r$ := $E_r$ + path;
    endif;
}
```

Figure 2.5. Two phase algorithm

phase, *all* enabled transitions at `s` are examined. The two phase algorithm outperforms SPIN (and PO-PACKAGE) when the proviso is invoked often; confirmed by the examples in Section 2.8. Note that `phase1` is *more general* than coercening of actions. In coercening of actions, two or more actions of a given process are combined together to form a larger "atomic" operation. In `phase1`, actions of multiple processes are executed.

2.6.1 Notation

If G=(V,E) is a graph, then a sequence of G is of the form $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \ldots$, where each s_i is in V, and (s_i, t_i, s_{i+1}) is in E. A sequence may be finite or infinite. σ, ρ, σ_1, ρ_1 etc. are used to denote sequences. If $\sigma = s_1 \xrightarrow{t_1} \ldots\ s_i \xrightarrow{t_i} \ldots s_j \ldots$ is a sequence in G, $\sigma(i) \ldots \sigma(j)$ indicates the subsequence $s_i \xrightarrow{t_i} s_{i+1} \ldots s_j$, and $\sigma(i) \ldots \sigma(\text{inf})$ indicates the subsequence of σ starting from s_i till the end of σ (if σ is finite, this is equivalent to $s_i \xrightarrow{t_i} s_{i+1} \ldots\ s_n \xrightarrow{t_n} s_{n+1}$ where $s_n \xrightarrow{t_n} s_{n+1}$ is the last transition of σ). □

2.6.2 Correctness of the two phase algorithm

To show that the graph generate by the two phase algorithm, G_r=(V_r, E_r) in `Twophase`, satisfies a LTL-X property ϕ iff the graph generated by `dfs`, G_f=(V_f, E_f), also satisfies ϕ, it is required to show that every sequence in G_r is "represented" in G_f and vice versa. However, for G_r to exist at all, `Twophase` must be terminate.

LEMMA 2.1 (TERMINATION) All calls made to `phase1` and `Twophase` terminate.

Proof: In `phase1`, a new state is added to `list` every time the `while` loop is executed. Since the number of states in the system is finite, the loop terminates; hence so does `phase1`. Similarly, at least one new state is added to V_r every time `Twophase` is called recursively. Hence these calls also terminate. □

From the construction, it is clear that G_r is a subgraph of G_f. Hence all paths in G_r are also paths in G_f, hence if G_f satisfies ϕ so does G_r. The rest of the section shows that if G_f violates ϕ, then so does G_r. Let σ be a path in G_f starting from the initial state that reveals the violation of ϕ. The construction below shows how to transform σ successively obtaining "equivalent" sequences σ_1, ... $\sigma_n = \rho$, where ρ is a sequence of transitions in G_r that shows the violation of ϕ. To do so, first it is needeed to establish that from every state $x \in V_r$, there is a path to a state $y \in V_r$ where y is completely expanded. Note that when a state y is completely expanded, `Twophase` adds y to `fe` on line $\boxed{3}$.

LEMMA 2.2 (REACHFE) If x is a state in V_r, then there is a finite sequence $\Pi_x \in G_r$, of length zero or more such that Π_x takes x to a state $y \in$ `fe`. In addition, if $(s, t, s^{'})$ is a transition in Π_x where t belongs to process P, then P is deterministic in s.

Proof: The proof is by constructing Π_x that satisfies the lemma. x is added to V_r either on line $\boxed{1}$ or $\boxed{1'}$ in `Twophase`. The argument shows that the lemma holds by a simple induction on the order in which the states are added to V_r.

Induction basis: During the first call of `Twophase`, V_r is empty; hence the *then* clause of the outermost *if* statement is executed. At this time, all states in `path` are added to V_r, and `s` is completely expanded by the *foreach* statement. Then for every state x in `path`, the lemma holds with y =`s`, with Π_x being a subpath of `path` starting from x.

Induction hypothesis: Assume that the lemma holds for states added to V_r during the first i calls of `Twophase`.

Induction Step: x is added to V_r in $i+1$ th call of `Twophase`. There are two cases to consider:

Case i: x is added to V_r on $\boxed{1}$. This case is similar to the induction basis: the lemma holds with $y =$`s` and Π_x is a subpath of `path` from x to `s`.

Case ii: x is added to V_r on $\boxed{1'}$ (in the *else* clause). In this case `s` is already in V_r. By induction hypothesis, there is a finite sequence, $\Pi_{\mathtt{s}}$ from `s` to y where y is in `fe`. Let p be the subpath of `path` from x to `s`. The lemma holds with Π_x being concatenation of p and $\Pi_{\mathtt{s}}$. $\square$

NOTE 2.1 If $\sigma = s_1 \xrightarrow{t_1} s_2 \ldots$ is a (finite or infinite) sequence in G_f, l is a *local* transition of process P, no transitions of P are in σ, and l is enabled at s_1, then $\sigma' = s_1 \xrightarrow{l} {s_1}' \xrightarrow{t_1} s_2' \ldots$ is a sequence in G_f obtained by prepending l to σ, and σ and σ' satisfy the same set of LTL-X formulae on the *global* transitions.

NOTE 2.2 If σ and σ' are two sequences in G_f starting from x and the sequence of transitions in σ' is a permutation of the sequence of transitions in σ such that only consecutive independent transitions are reordered, then σ and σ' satisfy the same set of LTL-X formulae on the *global* transitions.

LEMMA 2.3 Let $p = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \ldots s_m \xrightarrow{t_m} s_{m+1} \ldots s_n$ be a subsequence of Π_x for some $x \in V_r$ and t_m be the first transition of some process P in p. If ρ is a (finite or infinite) sequence in G_f starting from s_1 and does not contain t_m then ρ contains *no* transitions from P. (This implies that t_m is independent of all transitions in ρ.)

Proof: The proof is by contradiction. Assume that the lemma is false, i.e., ρ contains a transition u of P such that $u \neq t_m$. From the assumptions that $s_m \xrightarrow{t_m} s_{m+1}$ is in Π_x, t_m and u belong to the same process, and u is executed in ρ it is clear that

O1 u and t_m are safe at s_m,

O2 t_m is enabled at s_m and u is disabled at s_m,

O3 u continues to be disabled from every state in a sequence starting from s_m until at least t_m is executed,

O4 u is executed in ρ after some finite number of transitions, and

O5 none of the transitions in $t_1 \ldots t_{m-1}$ belong to P.

The following construction transforms an initial segment of $\rho_0 = \rho$ successively into ρ_1, $\rho_2 \ldots \rho_{m-1}$ such that

C1 p and ρ_i are identical upto the first i transitions and

C2 if u is executed at some state in ρ_i then it is also executed at some state (possibly different) in ρ_{i+1}.

By construction p and ρ_i are identical up to the first i transitions. Now ρ_{i+1} is constructed from ρ_i such that ρ_{i+1} and Π_x are identical up to first $i+1$ transitions and if u is executed in some state in ρ_i then it is also executed in some state in ρ_{i+1}. Finally the proof will show that u is not executed in all states of ρ_{m-1}, which implies that it is not executed in any state of $\rho_0 = \rho$, leading to a contradiction with (**O4**) above. There are two cases to consider.

Case 1: (Figure 2.6) t_{i+1} does not appear in ρ_i at all. ρ_{i+1} is constructed by simply inserting t_{i+1} at the appropriate position as shown in the Figure 2.6, i.e., $\rho_{i+1} = \rho_i(1) \ldots \rho_i(i) t_{i+1} \rho_i i + 1 \ldots \rho_i(\text{inf})$. If u is in ρ_i then it will also be in ρ_{i+1}.

Case 2: (Figure 2.7) t_{i+1} appears in ρ_i, at position j (by construction $j > i$), i.e., $\rho_i(j) = t_{i+1}$ ρ_{i+1} is obtained by moving t_{i+1} such that it is executed from s_{i+1}; i.e., $\rho_{i+1} = \rho_i(1) \ldots \rho_i(i) t_{i+1} \rho_i(i+1) \ldots \rho_i(j-1) \rho_i(j+1) \ldots \rho_i(\text{inf})$. (Since t_{i+1} is a *local* transition and is enabled at s_{i+1}, this reordering is allowed.) If u is in ρ_i, then it is also in ρ_{i+1}.

At the end of the construction, the first $m-1$ transitions of ρ_{m-1} are $s_1 \xrightarrow{t_1} s_2 \ldots \xrightarrow{t_{m-1}} s_m$, t_m is not in ρ_{m-1}, and u is disabled at every state after s_m in ρ_{m-1} (observation

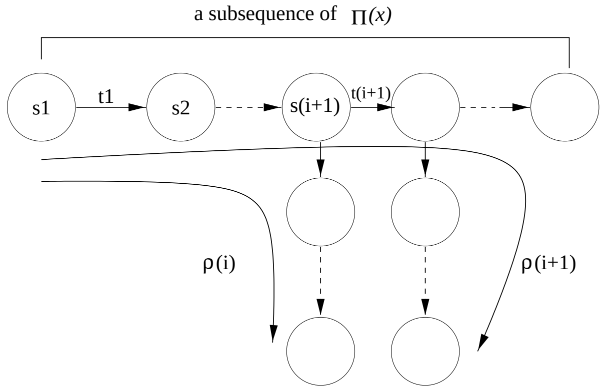


Figure 2.6. ρ_{i+1} is obtained from ρ_i by adding the t_{i+1} to ρ_i

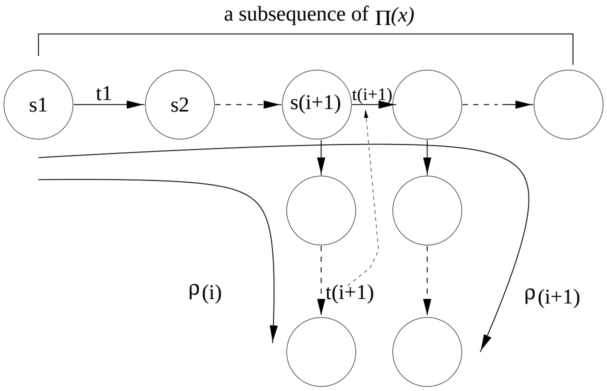


Figure 2.7. ρ_{i+1} is obtained from ρ_i by moving t_{i+1} into the appropriate position

O3). In other words, u is not in ρ_{m-1}. From **C2**, one can conclude that u is not in $\rho_0 = \rho$, which contradicts (**O4**). □

LEMMA 2.4 Let σ be a (finite or infinite) sequence from a state x in G_f. If x is also in V_r, then there is a sequence ρ from x in G_r that satisfies exactly the same set of LTL-X formulae on global transitions as σ.

Proof: The proof is by constructing a ρ that satisfies the lemma. This construction is very similar to the construction in Lemma 2.3. The construction is by transforming σ successively in $\sigma_1, \sigma_2 \ldots$ such that at each step, the validity of LTL formulae are not affected, and the last sequence is ρ (if σ is infinite the construction is also infinite). If σ contains no transitions (i.e., $\sigma = x$), then ρ is equal to σ. Otherwise, let $\sigma = x\xrightarrow{a}y \ldots \sigma(\text{inf})$; i.e., let the first transition be a.

Case 1: x is either expanded by `Twophase` in phase 2 or x is expanded in phase 1 by transition a. From the algorithm it is clear that $y \in V_r$. In this case, ρ also starts with a and $\rho(2) \ldots \rho(\text{inf})$ is obtained by this construction from y and $\sigma(2) \ldots \sigma(\text{inf})$.

Case 2: x is expanded in phase 1 by transition b_1 different from a. Let Π_x (as given by Lemma 2.2) be the finite sequence $(x = s_1)\xrightarrow{b_1} \ldots\ s_j\xrightarrow{b_j}s_{j+1}$.

Case 2.1: a is in $\{b_1 \ldots b_j\}$. Let t be the smallest $1 \leq t \leq j$ such that $b_t = a$. In this case let sequence p be $(x = s_1)\xrightarrow{b_1} \ldots\ s_{t-1}\xrightarrow{b_{t-1}}s_t$. (Construction continues at "Case 2 (Contd)" below.)

Case 2.2: a is not in $\{b_1 \ldots b_j\}$. In this case, let t be $j + 1$, and p be Π_x (i.e., $p = (x = s_1)\xrightarrow{b_1} \ldots \xrightarrow{b_{t-1}}s_t$). (Construction continues at "Case 2 (Contd)" below.)

Case 2 (Contd): By construction, $p = (x = s_1)\xrightarrow{b_1} \ldots\ s_{t-1}\xrightarrow{b_{t-1}}s_t$ is in G_r and $s_t\xrightarrow{a}a(s_t)$

is in G_r. By a direct application of Lemma 2.3 (with $\rho = a$), all transitions in p are independent of a. Now $\sigma_1, \sigma_2 \ldots \sigma_{t-1}$ are constructed such that σ_i and p are identical up to the first i transitions. (Since p is in G_r, the $\sigma_i(1) \ldots \sigma_i(i+1)$ is also in G_r.) Let σ_0 be σ. σ_{i+1} is obtained from σ_i as follows.

Case 2.a: b_{i+1} does not occur in $\sigma_i(i+1) \ldots \sigma_i(\mathrm{inf})$. From Lemma 2.3, b_{i+1} is independent of all transitions in $\sigma_i(i+1) \ldots \sigma(\mathrm{inf})$. σ_{i+1} obtained by inserting b_{i+1} into σ_i at position $i+1$; i.e., $\sigma_{i+1} = \sigma_i(1) \ldots \sigma_i(i) b_{i+1} \sigma_i(i+1) \ldots \sigma_i(\mathrm{inf})$. From Note 2.1, σ_i and σ_{i+1} satisfy the same set of LTL-X formulae on global transitions. (Construction continues at "Case 2 (Contd)" below.)

Case 2.b: b_{i+1} *first* appears in $\sigma_i(i+1) \ldots \sigma_i(\mathrm{inf})$ at l th position. Again from Lemma 2.3, b_{i+1} is independent of all transitions in $\sigma_i(i+1) \ldots \sigma_i(l-1)$. In this case, σ_{i+1} is obtained from σ_i by moving b_{i+1} from lth position to the $i+1$th position; i.e., $\sigma_{i+1} = \sigma_i(1) \ldots \sigma_i(i) b_{i+1} \sigma_i(i+1) \ldots \sigma_i(l-1) \sigma_i(l+1) \ldots \sigma_i(\mathrm{inf})$. By Note 2.2, σ_i and σ_{i+1} satisfy the same set of LTL-X formulae on global transitions. (Construction continues at "Case 2 (Contd)" below.)

Case 2 (Contd): By construction, the first $t-1$ transitions of σ_{t-1} are also transitions of p and the t th transition of σ_{t-1} is a. The initial segment of ρ will be the first t transitions of σ_{t-1}; i.e., $(x = s_1) \xrightarrow{b_1} s_2 \ \ldots \ s_{t-1} \xrightarrow{b_{t-1}} s_t \xrightarrow{a} a(s_t)$. From the construction of p, it is clear that this segment is in G_r. $\rho(t+2) \ldots \rho(\mathrm{inf})$ is obtained by recursively applying this construction to the sequence $\sigma_{t-1}(t+2) \ldots \sigma_k(\mathrm{inf})$ from the state $a(s_{t+1})$. □

THEOREM 2.1 Let ϕ be a LTL-X formulae on global transitions. ϕ holds in G_f from the initial state iff it holds in G_r generated by `Twophase`.

Proof: If ϕ is true in G_f, then since G_r is a subgraph of G_f, it is also true in G_r. If ϕ is false in G_f, let σ be a sequence starting from initial state that shows the violation. Since initial state is added to V_r, by the above lemma, a $\rho \in G_r$ can be constructed that reveals the violation of ϕ. □

2.7 On-the-fly Model Checking

A model checking algorithm is said to be on-the-fly if it examines the state graph of the system as it builds the graph to find the truth value of the property under consideration. If the truth value of the property can be evaluated by inspecting only a subgraph, then the algorithm need not generate the entire graph. Since state graph of many protocols is quite large, an on-the-fly model checking algorithm might be able to find errors in protocols that are otherwise impossible to analyze.

As discussed in Section 2.4.1, the model checking problem $M \models \phi$ can be equivalently viewed as answering the question if the graph represented by $S = M \otimes A_{\neg\phi}$, the synchronous product of the model M and the Büchii automaton representing $\neg\phi$, does not contain any paths satisfying the acceptance condition of $A_{\neg\phi}$. The algorithms `dfs` and `dfs_po` are not on-the-fly model checking algorithms since they construct the graph in E_f or E_r, which must be analyzed later to find if the acceptance condition of the Büchii automaton $A_{\neg\phi}$ is met or not. Note that E_f and E_r holds the information about the edges traversed as part of the search.

The condition that there is an infinite path in E (E_f or E_r) that satisfies the acceptance condition of $A_{\neg\phi}$ can be equivalently expressed as there is a strongly connected component (SCC) in the graph that satisfies the acceptance condition. Tarjan [83] presented a DFS based on-the-fly algorithm to compute SCCs *without storing any edge information.* Since space is at a premium for most verification problems, not having to store the edge information can be a major benefit of using this algorithm. This algorithm uses one word overhead per state visited and traverses the graph twice.

Coucoubetis et al. presented an on-the-fly model checking algorithm in [23]. This algorithm, shown in Figure 2.8, can be used to find if a graph has at least one infinite path satisfying a Büchii acceptance condition. Note that whereas Tarjan's algorithm can find all strongly connected components that satisfy the acceptance condition of $A_{\neg\phi}$, the algorithm in [23] is guaranteed to find only one infinite path satisfying the acceptance condition. Since presence of such an infinite path implies that the property is violated, it is usually sufficient to find one infinite path. The attractiveness of the algorithm in [23] comes from the fact that it can be implemented with only one bit per state compared to one word per state in the case of Tarjan's algorithm. This algorithm, shown in Figure 2.8, consists of two DFS searches, `dfs1` and `dfs2`. The outer dfs, `dfs1`, is very similar to `dfs`, except that instead of maintaining E_f, the algorithm calls an inner dfs, `dfs2`, after an accept state is fully expanded. `dfs2` finds if that accept state can reach itself by expanding the state again. If the state can reach it self, then a path violating ϕ can be found from the stack needed to implement `dfs1` and `dfs2`.

This figure assumes that full state graph is being generated. To use it along with partial order reductions, statements labeled $\boxed{1}$ in `dfs1(s)` and `dfs2(s)` can be appropriately

```
model_check()
{
    V1 := $\phi$; V2 := $\phi$;
    dfs1(InitialState);
}

/* outer dfs */
dfs1(s)
{
    V1 := V1 + {s};
    $\boxed{1}$ foreach enabled transition t do
        if t(s) $\notin$ V1 then
            dfs(t(s));
        endif;
    endforeach;
    $\boxed{2}$ if s is an accept state and
        /* Call nested dfs */
        s $\notin$ V2 then
            seed := s;
            dfs2(s);
        endif;
    endif;
}
```

```
/* inner dfs */
dfs2(s)
{
    V2 := V2 + {s};
    $\boxed{1}$ foreach enabled transition t do
        if t(s)=seed then error();
        elseif t(s) $\notin$ V2 then
            dfs2(t(s));
        endif;
    endforeach;
}
```

Figure 2.8. An on-the-fly model checking algorithm

modified to use the transitions in `ample(s)` (when used in conjunction with `dfs_po`) or with the search strategy of two phase. Earlier attempts at combining this on-the-fly model checking algorithm with the `dfs_po` have been shown to incorrect in [50]. The reason is that `ample(s)` depends on `redset`; hence when a state `s` is expanded on lines indicated by $\boxed{1}$ in `dfs1` and `dfs2`, `ample(s)` might evaluate to different values. If `ample(s)` returns the different set of transitions in `dfs1` and `dfs2`, even if an accept state s is reachable from itself in the graph constructed by `dfs1`, `dfs2` might not be able to prove that fact. Since the information in `redset` is different for `dfs1` and `dfs2`, `ample(s)` may indeed return different transitions, leading to an incorrect implementation. [50] solves the problem using the following scheme: `ample(s)` imposes an ordering on the processes in the system. When `ample(s)` cannot choose a process, say P_i, in `dfs1` due to the proviso, they choose `ample(s)` to be equal to all enabled transitions of s. In addition, one bit of information is recorded in `V1` to indicate that s is completely expanded. When `s` is encountered as part of `dfs2`, this bit is inspected to find if `ample(s)` must return all enabled transitions or if it must return a subset of transitions *without requiring the proviso*. This strategy reduces the opportunities for obtaining effective reductions, but it is deemed a good price to pay for the ability to use the on-the-fly model checking algorithm.

Thanks to the independence of `phase1` on global variables, including V_r, when `phase1(s)` is called in `dfs2`, the resulting state is exactly same as when it is called in `dfs1`. Hence the on-the-fly model checking algorithm can be used easily in conjunction with two phase. In Section 2.7.2, it is argued that the combination of this on-the-fly model checking algorithm, the selective caching technique can be used *directly* with two phase.

2.7.1 Selective caching

Both `Twophase` and `dfs_po`, when used in conjunction with the above on-the-fly model checking algorithm, obviate the need to maintain E_r. However, memory requirements to hold V_r, for most practical protocols, can be still quite high. Selective caching refers to the class of techniques where instead of saving every state visited in V_r, only a subset of states are saved.

There is a very natural way to incorporate a selective caching into `Twophase`. Instead of adding all states of `path` to V_r (line $\boxed{1}$ in `Twophase`) only `s` can be added. This guarantees that a given state always generates the same subgraph beneath it whether it is expanded as part of outer dfs or inner dfs; hence the above on-the-fly model checking algorithm can still be used. Adding s instead of `list` also means that the memory used for `list` in `phase1` can be reused. Even the memory required to hold the intermediate variable `list` can be reduced: the reason for maintaining this variable is only to ensure that the `while` loop terminates. This can be still guaranteed if instead of adding `s` to `list` unconditionally, it is added only if "`s<olds`," where $<$ is any total ordering on $\mathcal{S}$. PV uses bit-wise comparison as $<$.

2.7.2 Combining on-the-fly model checking and selective caching with two phase

When the selective caching technique is combined with two phase, the execution goes as follows: a given state is first expanded by `phase1`, then the resulting state is added to V_r and fully expanded. In other words, V_r contains only fully expanded states, which implies that the state graph starting a given state is the same in `dfs1` and `dfs2` of the on-the-fly algorithm. Hence, the on-the-fly algorithm and selective caching can be used together with two phase.

2.8 Experimental Results

As already mentioned, `Twophase` outperforms the proviso based algorithm `dfs_po` (implemented in SPIN) when the proviso is invoked often, confirmed by the results in Table 2.1. This table shows results of running `dfs_po` and `Twophase` (with and without selective caching enabled) on various protocols. The column corresponding to `dfs_po` shows the number of states entered in V_r and the time taken in seconds by the SPIN. The column "all" column in `Twophase` shows the number of states in V_r and the time taken in seconds when `Twophase` is run *without* the selective caching. The "Selective" column in `Twophase` shows the number of states entered in V_r *or* `list` and time taken in seconds when `Twophase` is run with the selective caching. All verification runs are conducted on an Ultra-SPARC-1 with 512MB of DRAM.

Contrived examples: B5 is the system shown in Figure 2.4(a) with 5 processes. W5 is a contrived example to show that `Twophase` does not always outperform the `dfs_po`. This system has no deterministic states; hence `Twophase` degenerates to a full search, whereas `dfs_po` can find significant reductions. *SC* is a server/client protocol. This protocol consists of n servers and n clients. A client chooses a server and requests for a service. A service consists of a two round trip messages between server and client and some local computations. `dfs_po` cannot complete the graph construction for $n = 4$, when the memory is limited to 64MB; when the memory limit is increased to 128MB it generates 750k states.

Table 2.1. Number of states visited and the time taken in seconds by the `dfs_po` algorithm and `Twophase` algorithm on various protocols

Protocol	dfs_po	Twophase	
		all	Selective
B5	243/0.34	11/0.33	1/0.3
W5	63/0.33	243/0.39	243/0.3
SC3	17,741/4.6	2,687/1.6	733/1.4
SC4	749,094/127	102,345/41.0	47,405/21.9
Mig	113,628/14	22,805/2.6	9,185/1.7
Inv	961,089/37	60,736/5.2	27,600/3.0
Pftp	95,241/11.0	187,614/30	70,653/19
Snoopy	16,279/4.4	14,305/2.7	8,611/2.4
WA	4.8e+06/340	706,192/31	169,680/21
UPO	4.9e+06/210	733,546/32	176,618/21
ROWO	5.2e+06/330	868,665/44	222,636/32

DSM protocols: *Mig* and *inv* are two cache coherency protocols used in the implementation of distributed shared memory (DSM) using a directory based scheme in Avalanche multiprocessor [14]. In a directory based DSM implementation, each cache line has a designated node that acts as its *home*—a node that is responsible for maintaining the coherency of the line. When a node needs to access the line, if it does not have the required permissions, it contacts the home node to obtain the permissions. Both *mig* and *inv* have two cache lines and four processes; two processors act as home nodes for the cache lines and the other two processors access the cache lines. Both algorithms can complete the analysis of *mig* within 64MB of memory, but on *inv*, `dfs_po` requires 128MB of memory `Twophase` on the other hand finishes comfortably generating a modest 27,600 states (with selective caching) or 60,736 states (without selective caching) in 64MB.

Protocols in SPIN **distribution:** *Pftp* and *snoopy* protocols are provided as part of SPIN distribution. On *pftp*, `dfs_po` generates fewer states than `Twophase` without state caching. The reason is that there is very little determinism in this protocol. Since `Twophase` depends on determinism to bring reductions, it generates a larger state space. However, with state caching, the number of states in the hash table goes down by a factor of 2.7. On *snoopy*, even though `Twophase` generates fewer states, the number of states generated `dfs_po` and `Twophase` (without selective caching) is very close to obtain any meaningful conclusion. The reason for this is twofold. First, this protocol contains some determinism, which helps `Twophase`. However, there are a number of deadlocks in this protocol. Hence, the proviso is not invoked many times. Hence the number of states generated is very close.

Memory model verification examples: *WA*, *UPO*, and *ROWO* test the interaction of PA (Precision Architecture from Hewlett-Packard) memory ordering rules with the runway bus protocol [10,39]. Runway is a high-performance split-transaction bus designed to support cache coherency protocols required to implement a symmetric multiprocessor (SMP). These three protocols consist of two HP PA models connected to the runway bus, executing read and write instructions. These property of interest is whether the PA/runway system correctly implements memory consistency rules called write atomicity (WA), uniprocessor ordering (UPO), and read-order, write-order (ROWO) [21]. On these protocols, the number of states saved by `dfs_po` is approximately 25 times larger than the number of states saved by `Twophase` (with selective caching).

2.9 Concluding Remarks

This chapter presented a new partial order reduction algorithm two phase that does not use the proviso and formally proved that it preserves all LTL-X properties on global variables. The chapter also showed how the algorithm can be combined with an on-the-fly model checking algorithm. Since the algorithm does not use the proviso, it outperforms previous algorithms on protocols where the proviso is invoked often. The two phase algorithm also naturally lends itself to be used in conjunction with a simple yet powerful selective caching scheme. The algorithm is implemented in a model checker called PV.

CHAPTER 3

DERIVING EFFICIENT CACHE COHERENCE PROTOCOLS THROUGH REFINEMENT

3.1 Chapter Overview

This chapter describes a syntax-directed *refinement* procedure to transform a high level distributed shared memory (DSM) protocol into an equivalent implementation. As the complexity of the protocols increases, importance of such refinement procedures also increases. The procedure is shown to be correct using an automated theorem prover.

3.2 Introduction

With the growing complexity of concurrent systems, automated procedures for developing protocols are growing in importance. This chapter presents a protocol *refinement* procedure—a procedure that accepts high level specifications of protocols and applies provably correct transformations on them to yield detailed implementations of protocols that run efficiently and have modest buffer resource requirements—in the context of distributed shared memory (DSM) protocols. Such procedures enable correctness proofs of protocols to be carried out with respect to high level specifications, which can considerably reduce the proof effort. Once the refinement rules are shown to be sound, the detailed protocol implementations need not be verified. Also, if the refinement rules apply for a family of protocols, then case-specific proofs can be avoided.

DSM systems have been widely researched in the academia as the next logical step in parallel processing [14,56,63]. High-end workstation manufacturers also have introduced DSM systems lately [25,62] thus providing added confirmation to the growing importance of DSM. A central problem in DSM systems is the design and implementation of distributed cache coherence protocols for shared cache lines using *message passing* [43]. These protocols are notoriously difficult to design correctly, especially in distributed systems, where the nodes implement the protocols using *message passing* [43]. The present-day approach to this problem consists of specifying the detailed interactions possible between computing nodes in terms of low level requests, acknowledges, negative acknowledges, and dealing with “unexpected” messages. Difficulty of designing these protocols is compounded by the fact that verifying such low level descriptions invites state explosion (when done using model checking [26, 28]) or tedious (when done using theorem-proving [74,75]) even for simple configurations. Often these low level descriptions are model checked for specific resource allocations (e.g., buffer sizes); it is often not known what would happen when these allocations are changed. Protocol refinement can help alleviate this situation considerably. This chapter presents a protocol refinement procedure which can be applied to derive a large class of DSM cache protocols.

Most of the problems in designing DSM cache coherence protocols are attributable to the apparent lack of atomicity in the implementation behaviors. Although some of the designers of these protocols may begin with a simple atomic-transaction view of the desired interactions, such a description is seldom written down. Instead, what gets written down as the “highest level” specification is a detailed protocol implementation which was arrived at through *ad hoc* reasoning of the situations that can arise. In this chapter, the rendezvous construct of CSP [45] is used as the specification language to allow the designers to capture their initial atomic-transaction view. The atomic-transaction protocol is then subjected to syntax-directed translation rules to modify the rendezvous communication primitives of CSP into asynchronous communication primitives yielding an efficient detailed implementation. The atomic-transaction view is referred to as *rendezvous protocol* and the detailed implementation is referred to as *asynchronous protocol*. Empirical results in Section 3.7 show that the rendezvous protocols are several orders of magnitude more efficient to model check than their corresponding detailed implementations. In addition, this section also shows that in the context of a state of the art DSM machine project called the Avalanche [14], the refinement procedure can automatically produce protocol implementations that are comparable in *quality* to hand-designed asynchronous protocols, where quality is measured in terms of (1) the number of *request*, *acknowledge*, and *negative acknowledge* (nack) messages needed for carrying out the rendezvous specified in the given specification, and (2) the buffering requirements to guarantee a precisely defined and practically acceptable progress criterion.

Rest of the chapter is organized as follows. Section 3.3 presents related past work done on the derivation of protocols in related domains. Section 3.4 presents the structure of typical DSM protocols in distributed systems. Section 3.5 presents the syntax-directed translation rules along with an important optimization called *request/reply* that constitute the refinement procedure Section 3.6 presents a proof that the refinement rules always produce correct result. This section also presents the proof using PVS [73]. Section 3.7 presents an example protocol developed using the refinement rules and also the efficiency of model checking the rendezvous protocol compared to the efficiency of model checking the asynchronous protocol. Section 3.8 presents a discussion on buffering requirements of the refined protocol and its impact on the forward progress made by the asynchronous protocol. Finally, Section 3.9 concludes the chapter.

3.3 Related Work

Chandra et al. [15] use a model based on continuations to help reduce the complexity of specifying the coherency protocols. The specification can then be model checked and compiled into an efficient object code. The motivation behind the approach is that a given cache controller acts on a number of cache lines simultaneously; hence for correct operation, the controller must continue to act on messages for an address a while an operation for a different address b is in progress. To facilitate this, their compiler provides a framework where operations on a given address can be suspended and resumed as the messages are sent and received for that address. After suspending an operation, the cache controller can act on a new message. In this approach, the specification is still at a low level; hence model checking of the specification remains expensive. Rendezvous communication can be modeled, for example, by manually splitting the rendezvous into a request and a reply. However, since the compiler can only suspend and resume a protocol action, such a rendezvous construct is not very useful. In other words, their approach cannot support rendezvous well as the transient states introduced by their compiler cannot adequately handle unexpected messages. Hence the designer has to explicitly state how each race condition is to be handled (the compiler also provides a default action to be taken for all race conditions that are not explicitly stated—usually buffering the message or raising an error). In contrast, using the refinement approach, user writes the rendezvous protocol using only the rendezvous primitive, verifies the protocol at this level with great efficiency and compiles it into an efficient asynchronous protocol or object code.

The idea of refinement closely resembles that of Buckley and Silberschatz [11]. Buckley and Silberschatz consider the problem of implementing rendezvous using message when the processes use generalized input/output guard. However, since the focus of their problem is for implementation in software, efficiency is not a primary concern. Their solution is too expensive for DSM protocol implementations. In contrast, the refinement procedure presented here focuses on a star configuration of processes with suitable syntactic restrictions on the high level specification language, so that an efficient asynchronous protocol can be automatically generated.

Gribomont [42] explored the protocols where the rendezvous communication can be simply replaced by asynchronous communication without affecting the processes in any other way. In contrast, the refinement rules *change* the processes by replacing each rendezvous communication action by a sequence of asynchronous communication actions. Lamport and Schneider [61] have explored the theoretical foundations of comparing atomic transactions (e.g., rendezvous communication) and split transactions (e.g., asynchronous communication), based on left and right movers [64], but have not considered specific refinement rules that form the heart of the refinement procedure.

3.4 Cache Coherency in Distributed Systems

In directory based cache coherent multiprocessor systems, the coherency of each line of shared memory is managed by a CPU node, called *home* node, or simply *home*.[1] All nodes that may access the shared line are called *remote* nodes. The home node is responsible for managing access to the shared line by all nodes without violating the coherency policy of the system.

The remote nodes and home node engage in the following activity. Whenever a remote node R wishes to access the information in a shared line, it first checks if the data are available (with required access permissions) in its local cache. If so, R uses the data from the cache. If not, it sends a request for permissions to the home node of the line. The home node may then contact some other remote nodes to revoke their permissions in order to grant the required permissions to R. Finally, the home node grants the permissions (along with any required data) to R. As can be seen from this description, a remote node interacts only with the home node, while the home node interacts with all the remote nodes. This suggests that one can restrict the communication topology of interest to a

---

[1] The home for different cache lines can be different. Usually, the protocol is specified per each cache line; hence the refinement procedure also derives protocols focusing on one cache line.

star configuration, with the home node as the hub, without loosing any descriptive power. This decision helps synthesize more efficient asynchronous protocols, as the rest of the chapter shows.

3.4.1 Complexity of DSM protocol design

As already pointed out, most of the problems in the design of DSM protocols can be traced to lack of atomicity. For example, consider the following situation. A shared line is being read by a number of remote nodes. When one of these remote nodes, say R1, wishes to modify the data, it sends a request to the home node for write permission. The home node then contacts all other remote nodes that are currently accessing the data to revoke their read permissions and then grants the write permission to R1. Unfortunately, it is incorrect to *abstract* the entire sequence of actions consisting of contacting all other remote nodes to revoke permissions and granting permissions to R1 as an atomic action. This is because when the home node is in the process of revoking permissions, a different remote node, say R2, may wish to obtain read permissions. In this case, the request from R2 must be either nacked or buffered for later processing. To handle such *unexpected* messages, the designers introduce intermediate states, also called *transient* states, leading to the complexity of the protocols. On the other hand, as the rest of the chapter shows, if the designer is allowed to state the desired interactions using an atomic view, it is possible to *refine* such a description using a refinement procedure that introduces transient states appropriately to handle such unexpected messages.

3.4.2 Communication model

The network that connects the nodes in the systems is assumed to provide *reliable, point-to-point in-order delivery* of messages. This assumption is justified in many machines, e.g., DASH [63] and Avalanche [14]. Network also assumed to have an infinite buffering, in the sense that the network can always accept new messages to be delivered. Without this assumption, the asynchronous protocol generated may deadlock. Unfortunately, this assumption is not satisfied in some networks. A solution to this problem that is orthogonal to the refinement procedure is given by Hennessy and Patterson [43]. They divide the messages into two categories: *request* and *acknowledge.* A *request* message may cause the recipient to generate more messages in order to complete the transactions, while an *acknowledge* message does not. The authors argue that if the network always accepts *acknowledge* messages (as opposed to all messages in the case of a network with infinite buffer), such deadlocks are broken. As Section 3.5 shows, asynchronous protocol has two *acknowledge* messages: ack and nack. Guaranteeing that the network always accepts these two *acknowledge* messages is beyond the scope of the refinement procedures.

3.4.3 Methodology

The high level specification language uses rendezvous communication primitive of CSP [45] to simplify the DSM protocol design. In particular, it uses direct addressing scheme of CSP, where every input statement in process `Q` is of the form `P?msg(v)` or `P?msg`, where `P` is the identity of the process that sent the message, `msg` is an *enumerated constant* ("message type") and `v` is a variable (local variable of `Q`) which would be set to the contents of the message, and every output statement in `Q` is of the form `P!msg(e)` or `P!msg` where `e` is an expression involving constants and/or local variables of `Q`. When P and Q rendezvous by P executing `Q!m(e)` and Q executing `P?m(v)`, P is said to be the active process and Q to be the passive process in the rendezvous.

The rendezvous protocol written using this notation is verified using either a theorem prover or a model checker for desired properties. Then the protocol is refined using the rules presented in Section 3.5 to obtain an *efficient* asynchronous protocol that can be implemented directly, for example in microcode.

3.4.4 Process structure

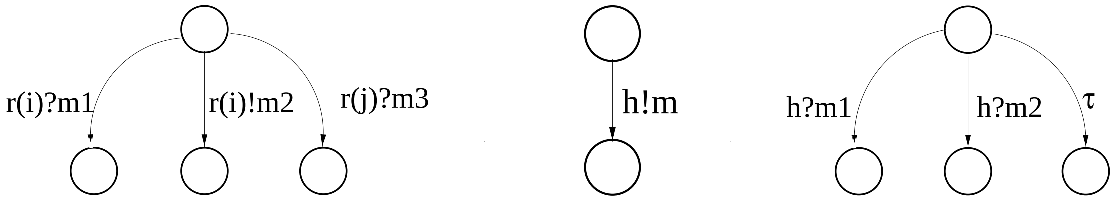
The states of processes in the rendezvous protocol are divided into two classes: *internal* and *communication*. When a process is in an internal state, it cannot participate in rendezvous with any other process. However, such a process will eventually enter a communication state where rendezvous actions are offered (this assumption can be syntactically checked). The refinement procedure introduces *transient* states where all unexpected messages are handled.

The i^{th} remote node is denoted by r_i and the home node by h. For simplicity, all the remote nodes are assumed to follow the same protocol and that the only form of communication between processes (in both asynchronous and rendezvous protocols) is through messages; i.e., other forms of communication such as global variables are not available.

As discussed before, the communication topology is restricted to a star. Since the home node can communicate with all the remote nodes and behaves like a *server* of remote-node requests, it is natural to allow generalized input/output guards in the home node protocols (e.g., Figure 3.1(a)). In contrast, the remote nodes are restricted to contain only input nondeterminism; i.e., a remote node can either specify that it wishes to be an active participant of a single rendezvous with the home node (e.g., Figure 3.1(b)) or it may specify that it is willing to be a passive participant of a rendezvous on a number of messages (e.g., Figure 3.1(c)). Also, as shown in Figure 3.1(c), τ guards are allowed in the remote node to model autonomous decisions such as cache evictions. These decisions, empirically validated on a large number of real DSM protocols, help synthesize more efficient protocols. Finally, it is assumed that no fairness conditions are placed on the nondeterministic communication options available from a communication state, with the exception of the forward progress restriction imposed on the entire system (described below).

3.4.5 Forward progress

Assuming that there are no τ loops in the home node and remote nodes, the refinement procedure guarantees that at least one of the refined remote nodes makes forward progress, if forward progress is possible in the rendezvous protocol. Notice that forward progress is guaranteed for some remote node, not for every remote node. This is because assuring forward progress for each remote node requires allocating too much buffer space at the home node. If there are n remote nodes, to assure that every remote node makes progress, the home node needs a buffer that can hold n requests. This is both impractical and not scalable as n in DSM machines can be as high as a few thousands. In contrast, to guarantee forward progress for at least one remote node, a buffer that can hold two messages suffices, as shown later in Section 3.5. Incidentally, assuring forward progress for each individual remote node corresponds to strong fairness whereas assuring forward progress for at least one remote node corresponds to weak fairness [66].



(a) Home node (b) Remote node (c) Remote node

Figure 3.1. Examples of communication states in the home node and remote nodes

3.5 The Refinement Procedure

The refinement procedure systematically refines the communication actions in h and r_i by inspecting the syntactic structure of the processes. The technique is to split each rendezvous into two halves: a request for the rendezvous and an acknowledgment (ack) or negative acknowledgment (nack) to indicate the success or failure of the rendezvous. At any given time, a refined process is in one of three states: *internal*, *communication*, or *transient*. Internal and communication states of the refined process are same as in the corresponding unrefined process in the rendezvous protocol. Transient states are introduced by the refinement procedure in the following manner. Whenever a process P has `Q!m(e)` as one of the guards in a communication state, P sends a request to Q and awaits in a transient state for an ack/nack or a request for rendezvous from Q. In the transient state, P behaves as follows:

R1. If P receives an ack from Q, the rendezvous is successful. P changes its state as given by the high level specification.

R2. If P receives a nack from Q, the rendezvous has failed. P goes back to the communication state and tries the same rendezvous or a different rendezvous.

R3. If P receives a request from Q, the action taken depends on whether P is the home node or a remote node. If P is a remote node (and Q is then the home node), P simply ignores the message. (This is because, as discussed in the next sentence, P "knows" that Q will get its request that is tantamount to a nack of Q's own request.) If P is the home node, it goes back to the communication state as though it received a nack ("implicit nack") and processes Q's request in the communication state.

The rules R1–R3 govern how the remote node and home node are refined, as will now be detailed.

3.5.1 Refining the remote node

Every remote node has a buffer to store one message from the home node. When the remote node receives a request from the home node, the request would be held in the buffer. When a remote node is at a communication or transient state, its actions are shown in Table 3.1. The rows of the table are explained below.

Table 3.1. The actions of the remote node

Row	State	Buffer contents	Action
C1	Communication (Active)	empty	(a) Request for rendezvous (b) goto transient state
C2	Communication (Active)	request	(a) delete the request (b) Request home for rendezvous (c) goto transient state
C3	Communication (Active)	request	Ack/nack the request
T1	Transient	ack	Successful rendezvous
T2	Transient	nack	go back to the communication state
T3	Transient	request	Ignore the request

Note: After each action, the message in the buffer is removed.

C1 When a remote node is in a communication state where it wishes to be an active participant of a rendezvous, and no request from home node is pending in the buffer, it sends a request for rendezvous to home, goes to a transient state, and awaits for an ack/nack or a request for rendezvous from home node.

C2 This row is similar to C1, except that there is a request from home is pending in the buffer. In this case also, the remote sends a request to home and goes to a transient state. In addition, the request in the buffer is deleted. As explained in R3, when the home receives the remote's request, it acts as though a nack is received (implicit nack) for the deleted request.

C3 When the remote node is in a communication state, and it is passive in the rendezvous, it waits for a request for rendezvous from home. If the request satisfies any guards of the communication state, it sends an ack to the home and changes state to reflect a successful rendezvous. If not, it sends a nack to home and continues to wait for a matching request. In both cases, the request is removed from the buffer.

T1, T2 If the remote node receives an ack, the rendezvous is successful. The state of the process is appropriately changed to reflect the completion of the rendezvous. If, on the other hand, the remote node receives a nack from the home, it is because the home node does not have sufficient buffers to hold the request. In this case, the remote node goes back to communication state, retransmits the request, and reenters the transient state.

T3 As explained in R3, if the remote node receives a request from home, it simply deletes the request from buffer and continues to wait for an ack/nack from home.

3.5.2 Refining the home node

The home node has a buffer of capacity k messages ($k \geq 2$). All incoming messages are entered into the buffer when there is space, with the following exception. The last buffer location (called the *progress buffer*) is reserved for an incoming request for rendezvous that is known to complete a rendezvous in the current state of the home. If no such reservation is made, a livelock can result. For example, consider the situation when the buffer is full and none of the requests in the buffer can enable a guard in the home node. Due to lack of buffer space, any new requests for rendezvous must be nacked, thus the home node can no longer make progress. In addition, when the home node is in a transient state expecting an ack/nack from r_i, an *additional* buffer need to be reserved so that a message (ack, nack, or request for rendezvous) from r_i can be held. This buffer location is referred to as *ack buffer*.

When the home is in a communication or transient state, the actions taken are shown in Table 3.2. The rows of this table are explained below.

C1 If the home is in a communication state and it can accept one or more requests pending in the buffer, then the home finishes rendezvous by arbitrarily picking one of these messages.

C2 If no requests pending in the buffer can satisfy any guard of the communication state and one of the guards of the communication state is $r_i!m_i$, then home node sends a request for rendezvous to r_i and enters a transient state. As described above, before sending the message, it also reserves an additional buffer location, ack buffer, so that forward progress can be assured. This step may require the home to generate a nack for one of the requests in the buffer in order to free the buffer location. Also note that condition (c) states that no request from r_i is pending in the buffer. The rationale behind this condition is that, if there is a request from r_i pending, then r_i is at a communication state with r_i being the active participant of the rendezvous. Due to the syntactic restrictions placed on the description of the remote nodes, r_i cannot satisfy any requests for rendezvous in this communication state. Hence it is wasteful to send any request to r_i in this case.

Table 3.2. Actions taken by the home node

Row	State	Condition	Action
C1	Communication	buffer contains a request from r_i that satisfies a rendezvous	(a) an ack is sent to r_i (b) delete request from buffer
C2	Communication	(a) no request in the buffer satisfies any required rendezvous (b) home node can be active in a rendezvous with r_i on m_i (i.e., $r_i!m_i$ is a guard in this state) (c) no request from r_i is pending in buffer	(a) ack buffer is allocated (if not enough buffer space a nack may be generated) (b) a request for rendezvous is sent to r_i (c) goto transient state
T1	Transient	ack from r_i	rendezvous is completed
T2	Transient	nack from r_i	rendezvous failed. Go back to the communication state and send next request. If no more requests left, repeat starting with the first guard.
T3	Transient	(a) request from r_i (b) waiting for ack/nack from r_i	treat the request as a a nack plus a request
T4	Transient	(a) request from $r_j \neq r_i$ has arrived (b) waiting for ack/nack from r_i (c) buffer has $>$ 2 free entries	enter the request into buffer
T5	Transient	(a) request from $r_j \neq r_i$ has arrived (b) waiting for ack/nack from r_i (c) buffer has 2 free entries (d) the request can satisfy a guard in the communication state	enter the request into progress buffer
T6	Transient	request from r_j has arrived (all cases not covered above)	nack the request

T1 When the home is in transient state, if it receives an ack, the rendezvous is successful. The state of the home is modified to reflect the completion of the rendezvous.

T2 When the home is in transient state, if it receives a nack the rendezvous failed. Hence the home goes back to the communication state. From the communication state, it checks if any new request in the buffer can satisfy any guard of the communication state. If so, an ack is generated corresponding to that request, and that rendezvous is completed. If not, the home tries the next output guard of the communication state. If there are no more output guards, it starts all over again with the first output guard. The reason for this is that, even though a previous attempt to rendezvous has failed, it may now succeed, because the remote node in question might have changed its state through a τ guard in its communication state.

T3 When the home is expecting an ack/nack from r_i, if it receives a request from r_i instead, it uses the implicit nack rule, R3. It first assumes that a nack is received;

hence it goes to the communication state. From this state all the requests, including the request from r_i, are processed as in row T2.

T4 If the home receives a request from r_j, when it is expecting an ack/nack from a different remote r_i, and there is sufficient room in the buffer, the request is added to the buffer.

T5 When the home is in a transient state and has only two buffer spaces, if it receives a message from r_j, it adds the request to buffer according to the buffer reservation scheme; i.e., the request is entered into the progress buffer iff the request can satisfy one of the guards of the communication state. If the request cannot satisfy any guards, it would be handled by row T6.

T6 When a request for rendezvous from r_j is received and there is insufficient buffer space (all cases not covered by T4 and T5), home sends a nack to r_j. r_j would retransmit the message.

3.5.3 Request/reply communication

The generic scheme outlined above replaces each rendezvous action with two messages: a request and an ack. In some cases, it is possible to avoid ack message. An example is when two messages, say `req` and `repl` are used in the following manner: `req` is sent from the remote node to home node for some service. The home node, after receiving the `req` message, performs some internal actions and/or communications with other remote nodes and sends a `repl` message to the remote node. In this case, it is possible to avoid exchanging ack for both `req` and `repl`. If statements `h!req(e)` and `h?repl(v)` always appear together as `h!req(e); h?repl(v)` in remote node, and r_i`!repl` always appears *after* r_i`?req` in the home node, then the acks can be dropped. This is because whenever the home node sends a `repl` message, the remote node is always ready to receive the message, hence the home node does not have to wait for an ack. In addition, a reception of `repl` by the remote node also acts as an ack for `req`. Of course, if the remote node receives a nack instead of `repl`, the remote node would retransmit the request for rendezvous.

This scheme can also be used when `req` is sent by the home node and the remote node responds with a `repl`. In this case, of course, after receiving `req`, the remote node performs local actions only (i.e., no rendezvous actions) and responds with a `repl`.

3.6 Correctness of the Refinement Procedure

The following argument shows that the refinement is correct by analyzing the different scenarios that can arise during the execution of the asynchronous protocol. The argument is divided into two parts: (a) all rendezvous that happen in the asynchronous protocol are allowed by the rendezvous protocol and (b) forward progress is assured for at least one remote node. Note that the forward progress is not assured for any given remote node due to buffer considerations (Section 3.4.5).

The rendezvous is finished in the asynchronous protocol when the remote node executes rows C1, C3, or T1 of Table 3.1 and the home node executes rows C1 or T1 of Table 3.2. To see that all the rendezvous are in accordance with the rendezvous protocol, consider what happens when a remote node is the active participant in the rendezvous (the case when the home node is the active participant is similar). The remote node r_i sends out a request for rendezvous to the home h and starts waiting for an ack/nack. There are three cases to consider.

1. h does not have sufficient buffer space. In this case the request is nacked. In this case, no rendezvous has taken place.

2. h has sufficient buffer space and it is in either an internal state or a transient state where it is expecting an ack/nack from a different remote node, r_j. In this case, the message is entered into the h's buffer. When h enters a communication state where it can accept the request, it sends an ack to r_i, completing the rendezvous. Clearly, this rendezvous is allowed by the rendezvous protocol. If h has to send a nack to r_i later to make some space in buffer by row C2, r_i would retransmit the request, in which case no rendezvous has taken place.

3. h has sent a request for rendezvous to r_i and is waiting for an ack/nack from r_i in a transient state. (This corresponds to R3 of page 44.) In this case, r_i simply ignores the request from h. h knows that its request would be dropped. Hence it treats the request from r_i as a combination of nack for the request it already sent and a request for rendezvous. Thus, this case becomes exactly like one of the two cases above; hence, h generates an ack/nack accordingly (if an ack is generated it would be allowed by the rendezvous protocol).

As can be seen from this case analysis, an ack is generated only in case 2. In this case the rendezvous is allowed by the rendezvous protocol.

3.6.1 PVS proof of correctness

The above argument is formalized with the help of PVS [73] and proved that the refinement rules are *safety preserving*; i.e., if the a transition is taken in the refined protocol, then it is allowed in the original rendezvous protocol. Such proofs are normally done by establishing a "commuting diagram" as shown in Figure 3.2. A_1 and A_2 are two states in the asynchronous protocol, and abs is a function that maps a state in asynchronous protocol into a state in the rendezvous protocol. If the asynchronous protocol has a transition that takes A_1 to A_2, then the rendezvous protocol must have a transition that takes $R_1 = abs(A_1)$ to $R_2 = abs(A_2)$. If S_a represents the set of states in the asynchronous protocol, S_r represents the set of states in the rendezvous protocol, $\rightarrow_a$ represents the set of transitions of the asynchronous protocol, and $\rightarrow_r$ represents the set of transitions of the rendezvous protocol, the figure can be expressed as:

$$\forall A_1, A_2 \in S_a \quad : \quad A_1 \rightarrow_a A_2 \Rightarrow abs(A_1) \rightarrow_r abs(A_2). \tag{3.1}$$

This equation cannot be used directly with the refinement procedure because some of the moves made by asynchronous protocol are *invisible*; in other words, for some asynchronous transitions, $abs(A_1) = abs(A_2)$. Hence the following equation can be established.

$$\begin{tikzcd} A1 \arrow[r, dashed, "abs"] \arrow[d, "\text{asynchronous transition}"'] & R1 \arrow[d, "\text{rendezvous transition}"] \\ A2 \arrow[r, dashed, "abs"] & R2 \end{tikzcd}$$

Figure 3.2. The commute diagram

$$\forall A_1, A_2 \in S_a \quad : \quad A_1 \rightarrow_a A_2 \Rightarrow abs(A_1) = abs(A_2) \ \vee \ abs(A_1) \rightarrow_r abs(A_2). \tag{3.2}$$

Establishing Equation 3.2 constitutes only a partial proof. For example, if *abs* maps *every* state in S_a to the same state in rendezvous protocol, then the equation holds; i.e., the equation can be made to hold vacuously by artificially making $abs(A_1) = abs(A_2)$ for ever A_1 and A_2. Hence the full proof requires establishing that *abs* is not a vacuous function by establishing existence of a function *aug* that maps a state of the rendezvous protocol to a state of the asynchronous protocol satisfying the following conditions, as done in [82].

$$aug(R_i) = A_i \tag{3.3}$$

$$\forall R \in S_r \quad : \quad abs(aug(R)) = R \tag{3.4}$$

$$\forall R_1, R_2 \in S_r \quad : \quad R_1 \rightarrow_r R_2 \Rightarrow aug(R_1) \rightarrow_a^+ aug(R_2) \tag{3.5}$$

R_i and A_i are the initial states of the rendezvous protocol and the asynchronous protocols respectively, and $\rightarrow_a^+$ represents a sequence of one or more transitions from $\rightarrow_a$. Equation 3.3 states that *aug* maps the initial state of the rendezvous protocol to the initial state of the asynchronous protocol, Equation 3.4 states that *abs* is inverse of *aug*, and Equation 3.5 states that every transition of the rendezvous protocol is imitated by a sequence of transitions in the asynchronous protocol. In other words, Equation 3.2 shows that every transition allowed under asynchronous protocol is also allowed under the rendezvous protocol, Equation 3.5 shows that every transition of rendezvous protocol is mimicked by a sequence of transitions in the asynchronous protocol, and Equations 3.3 and 3.4 are sanity checks to ensure that *aug* and *abs* are consistent with each other.

3.6.1.1 Construction of *abs*

To construct *abs* satisfying Equation 3.2, it is necessary to characterize S_a. One possible characterization of S_a is simply obtained from the syntactic description; i.e., S_a

contains all reachable as well as unreachable states. Using such a simple characterization, it is not possible to construct a *abs* satisfying Equation 3.2 other than the trivial function that maps every state in S_a to the same state. Hence the following inductive invariant is used as S_a.

1. At any given time there is at most one ack towards any node.
2. Every remote node has at most one pending rendezvous transaction at any time; i.e., no remote node sends more than one request for rendezvous to home until a response (ack or nack) is received from home.
3. The home node has at most one pending rendezvous transaction at any given time; i.e., home node never sends a request for rendezvous until a response (ack, nack, or implicit nack) is received for the last rendezvous request.

Using these constraints, *abs* can be constructed as follows.

1. All requests for rendezvous in the medium and buffers are discarded by *abs*. If a request for rendezvous from a process P is discarded, the state of P is modified from transient state back to the communication state; i.e., *abs* modifies the system as though the request was never sent.
2. If there is an ack towards a process P, the ack is discarded, and the state of P is modified to the state which P would attain after consuming the ack.
3. All nacks in the medium and buffers are also discarded. If a nack sent to P is discarded, the state of P is changed from transient state back to the communication state.

Using the higher-order functions available in PVS, it is shown that $\rightarrow_a$ as defined by Tables 3.1 and 3.2, along with the above *abs* function satisfies Equation 3.2. The proof is reported in [68].

3.6.1.2 Construction of *aug*

Given a state $R \in S_r$, $aug(R)$ is obtained by adding empty communication channels to R. The initial state of the asynchronous protocol, A_i, is defined as $aug(r_i)$; hence proving Equation 3.3 is trivial. Similarly, proving Equation 3.4 is also trivial as it involves simply expanding the definitions of abs and aug. To prove Equation 3.5, the following strategy is used. If the transition that takes R_1 to R_2 is an internal transition, then the same transition shows that $aug(R_1) \rightarrow_a^+ aug(R_2)$. If the transition is a rendezvous transition with a remote node r_i as the active participant and the home node, h, as the passive participant, then the sequence (a) r_i sending a request for rendezvous, (b) h receiving the request and sending an ack, and (c) r_i receiving the ack shows $aug(R_1) \rightarrow_a^+ aug(R_2)$. Similarly, if the transition is a rendezvous transition with a remote node r_i as the passive participant and the home node, h, as the active participant, then the sequence (a) h sending a request for rendezvous, (b) r_i receiving the request and sending an ack, and (c) h receiving the ack shows $aug(R_1) \rightarrow_a^+ aug(R_2)$.

3.6.2 Proof of forward progress

To see that at least one of the remote nodes makes forward progress, observe that when the home node h makes forward progress, one of the remote nodes also makes forward progress. Since no process may stay in internal states forever, from every internal state h eventually reaches a communication state from which it may go to a transient state. Note that because of the same restriction, when h sends a request to a remote node, the remote would eventually respond with an ack, nack, or a request for rendezvous. If any forward progress is possible in the rendezvous protocol, the following argument shows that h would eventually leave the communication or the transient state by the following case analysis.

1. h is in a communication state, and it completes a rendezvous by row C1 of Table 3.2. Clearly, progress is being made.

2. h is in a communication state, and conditions for row C1 and C2 of Table 3.2 are not enabled. h continues to wait for a request for rendezvous that would enable a guard in it. Since a buffer location is used as progress buffer, if progress is possible in the rendezvous protocol, at least one such request would be entered into the buffer, which enables C1.

3. h is in a communication state, and row C2 of Table 3.2 is enabled. In this case, h sends a request for rendezvous and goes to transient state. Cases below argue that it eventually makes progress.

4. h is in a transient state and receives an ack. By row T1 of Table 3.2, the rendezvous is completed, hence progress is made.

5. h is in a transient state and receives a nack (row T2 of Table 3.2) or an implicit nack (row T3 of Table 3.2). In response to the nack, the home goes back to the communication state. In this case, the progress argument is based on the requests for rendezvous that h has received while it was in the transient state and the buffer reservation scheme. If one or more requests received enable a guard in the communication state, at least one such request is entered into the buffer by rows T4 or T5. Hence an ack is sent in response to one such request when h goes back to the communication state (row C1), thus making progress. If no such requests are received, h sends request for rendezvous corresponding to another output guard (row C2) and reenters the transient state. This process is repeated until h makes progress by taking actions in C1 or T1. If any progress is possible, eventually either T1 would be enabled (since h keeps trying all output guards repeatedly), or C1 would be enabled (since h repeatedly enters communication state repeatedly from T2 or T3 and checks for incoming requests for rendezvous). Hence, unless the rendezvous protocol is deadlocked, the asynchronous protocol makes progress.

3.7 Refinement of an Example Protocol

The effectiveness of the synthesis procedure is demonstrated by applying it to the rendezvous specification of the migratory protocol of avalanche. (The architectural team of Avalanche had previously developed the asynchronous migratory protocol without using the refinement rules described in this chapter.) The protocol followed by the home node is shown in Figure 3.3 and the protocol followed by the remote nodes is shown in Figure 3.4. Initially the home node starts in state F (free) indicating that no remote node has access permissions to the line. When a remote node r_i needs to read/write the shared line, it sends a `req` message to the home node. The home node then sends a `gr` (grant) message to r_i along with the data. In addition, the home node also records the identity of r_i in a variable `o` (owner) for later use. Then the home node goes to state E (exclusive). When the owner no longer needs the data, it may relinquish the line (`LR` message). As a result of receiving the `LR` message, the home node goes back to F. When the home node is in E, if it receives a `req` from another remote node, the home node revokes the permissions from the current owner and then grants the line to the new requester. To

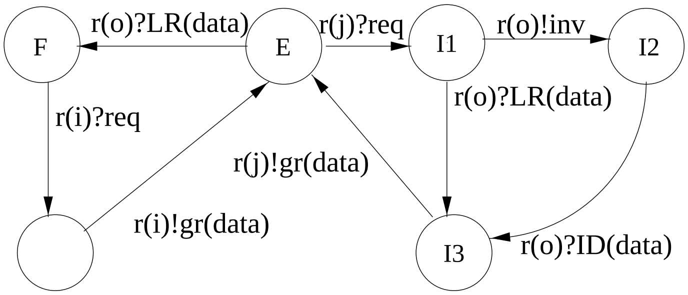


Figure 3.3. Home node of the migratory protocol

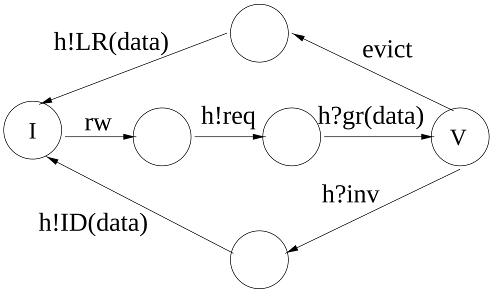


Figure 3.4. Remote node of the migratory protocol

revoke the permissions, it either sends an `inv` (invalidate) message to the current owner `o` and waits for the new value of the data (obtained through `ID` (invalid done) message) or waits for a `LR` message from `o`. After revoking the permissions from the current owner, a `gr` message is sent to the new requester, and the variable `o` is modified to reflect the new owner.

The remote node initially starts in state I (invalid). When the CPU tries to read or write (shown as `rw` in the figure), a `req` is sent to the home node for permissions. Once a `gr` message arrives, the remote node changes the state to V (valid) where the CPU can read or write a local copy of the line. When the line is evicted (for capacity reasons, for example), a `LR` is sent to the home node. Or, when another remote node attempts to access the line, the home node may send an `inv`. In response to `inv`, an `ID` (invalid done) is sent to the home node and the line reverts back to the state I.

To refine the migratory protocol, note that the messages `req` and `gr` can be refined using the request/reply strategy. This is because the remote node after sending a `req` message immediately waits for a `gr` message from the home node. The home node, on the other hand, after receiving a `req` message, either sends a `gr` message (resulting in state change from F to E) or may have to contact a remote node and then send a `gr` message (resulting in a state change from E back to E, via E-I1-I3-E or E-I1-I2-I3-E). Similarly, the messages `inv` and `ID` can be refined using request/reply, except that in this case `inv` is sent by the home node, and the remote node responds with an `ID`. By following the request/reply strategy, a pair of consecutive rendezvous such as r_i?`req`; r_i!`gr` or r_i!`inv`; r_i?`ID` (data) takes only two messages as in Figures 3.5 and 3.6.

The refined home node is shown in Figure 3.5 and the refined remote node is shown in Figure 3.6. In these figures, the operators "??" and "!!" are used instead of "?" and "!" to emphasize that the communication is asynchronous. In both these figures, transient states are shown as dotted circles (the dotted arrows are explained later). As discussed in Section 3.5.2, when the refined home node is in a transient state, if it receives

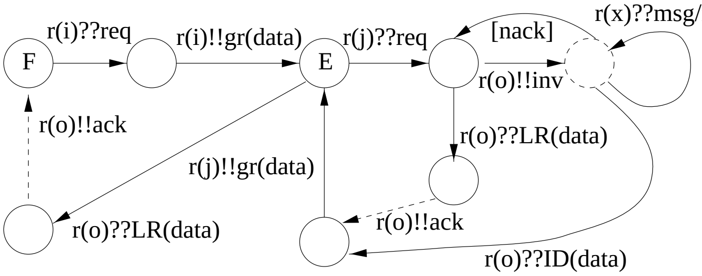


Figure 3.5. Refined home node of the migratory protocol

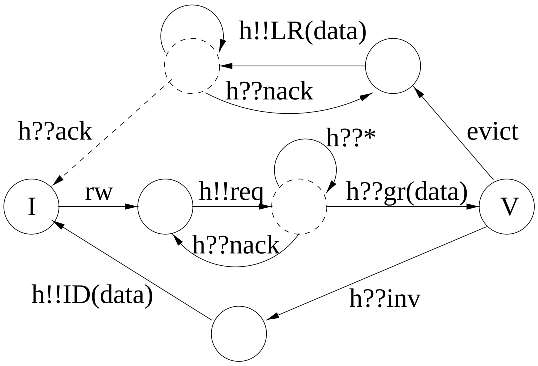


Figure 3.6. Refined remote node of the migratory protocol

a request from the process from which it is expecting an ack/nack, it would be treated as a combination of a nack and a request. This is shown as `[nack]` to imply that the home node has received the nack as either an explicit nack message or an implicit nack. Again, as discussed in Section 3.5.2, when the home node does not have sufficient number of empty buffers, it nacks the requests, irrespective of whether the node is in an internal, transient, or communication state. For the sake of clarity, the figure leaves out all such nacks other than the one on transient state (labeled `r(x)??msg/nack`).

As explained in Section 3.5.1, when the remote node is in a transient state, if it receives a message from the home node, the remote node ignores the message; no ack/nack is ever generated in response to this request. Figure 3.6 shows this as a self loop on the transient states, labeled `h??*`. The asynchronous protocol designed by the Avalanche design team differs from the protocol shown in Figures 3.5 and 3.6 in that in their protocol the dotted lines are τ actions; i.e., no ack is exchanged after an `LR` message.

3.7.1 Model checking efficiency

As can be expected, verifying the rendezvous protocols is much simpler than verifying the asynchronous protocol. The rendezvous and asynchronous versions of the migratory protocol above and invalidate—another DSM protocol used in Avalanche—are model checked using the SPIN [47] model checker. The number of states visited by SPIN and time taken in seconds on these two protocols is shown in Table 3.3. The complexity of verifying the hand designed migratory or invalidate is comparable to the verification of asynchronous protocol. As can be seen, verification of the rendezvous protocol generates far fewer states and takes much less run time than verifying the asynchronous protocol. In fact, the rendezvous migratory protocol could be model checked for up to 64 nodes using 32MB of memory, whereas the asynchronous protocol can be model checked for only two nodes using 64MB of memory.

3.8 Buffer Requirements and Fairness

As mentioned in Section 3.4.5, refinement process preserves forward progress for at least one remote node, but does not guarantee forward progress for any *given* remote node. This means that, it is possible that one of the nodes may starve. For example, a request for a rendezvous from a remote node might be continually nacked by the home node. This problem can be solved if the size of the buffer in the home node is n, where n is the number of the remote nodes. In this case, the home node *never* generates a nack.

Table 3.3. Verification of rendezvous and asynchronous protocols.

Protocol	N	Asynchronous protocol	Rendezvous protocol
Migratory	2	23163/2.84	54/0.1
	4	Unfinished	235/0.4
	8	Unfinished	965/0.5
Invalidate	2	193389/19.23	546/0.6
	4	Unfinished	18686/2.3
	6	Unfinished	228334/18.4

The table shows the number of states visited and time taken in seconds for reachability analysis of the rendezvous and asynchronous versions of the migratory and invalidate protocols. All verifications were limited to 64MB of memory.

If the messages in the home node's buffer are processed in a fair manner, one can show that no remote node is starved.

However, this requires too much memory to be reserved for buffers. For example, in a multiprocessor with 64 nodes, if each node of the multiprocessor acts as home for 1024 lines (a modest number of lines), then each node needs to reserve a total of 64K messages to be used as buffer space. Clearly, it is impractical to reserve such a large amount of space for buffer. Hence, it is impractical to guarantee forward progress per each remote node by *refinement alone.* However, it is usually not difficult to ensure the forward progress when other properties of modern CPUs are considered. A modern CPU can have a small number, say 8, of transactions outstanding. If the home node were to reserve a buffer that can handle 513 messages ($512 = 64 \times 8$ for requests for rendezvous, 1 for ack/nack) and the buffer pool is managed as a resource shared by all the 1024 shared lines, forward progress can be assured per each shared line per each remote node.

3.9 Concluding Remarks and Future Directions

The framework presented in this chapter can be used to specify the protocols implementing distributed shared memory at a high level. These rendezvous protocols can be efficiently verified, for example using a model checker. After such verification, the protocol can be translated into an efficient asynchronous protocol using the refinement rules presented in this chapter. The refinement rules add transient states to handle unexpected messages. The rules also address buffering considerations. To assure that the refinement procedure generates an efficient asynchronous protocol, some syntactic restrictions are placed on the processes. These restrictions, namely enforcing a star configuration and restricting the use of generalized guard, are inspired by domain specific considerations.

The future directions include letting two remote nodes communicate in *asynchronous* protocol so that better efficiency can be obtained. Relaxing the star configuration requirement for the rendezvous protocol does not add much descriptive power. However, relaxing this constraint for the asynchronous protocol may improve efficiency.

CHAPTER 4

FORMAL MEMORY MODELS

4.1 Chapter Overview

This chapter surveys background in formal memory models by presenting the definitions of five popular models: sequential consistency, coherency, parallel random access memory, processor consistency, and linearizability. Examples are used to clarify the differences between various models. These models set the background for the memory model verification problem discussed in Chapter 5.

4.2 Introduction

With the growing interest in the design and implementation of shared memory multiprocessors, the abstraction of a shared memory system—*formal memory model*—is of growing importance. In a traditional uniprocessor system, the abstraction is that each read must return the value written by the most recent write as given by the sequential program. Such a simple definition cannot be used with multiprocessors as a *concurrent program* consists of not a single sequential program, but a collection of sequential programs. As a result, the designers have defined a number of formal memory models over the past three decades. Some of these models allow very efficient implementations but require more effort to program. Others do not allow as efficient implementations, but are easier to program.

The rest of the chapter is organized as follows. Section 4.3 defines a concurrent program and an execution of the concurrent program. Sections 4.4–4.8 explain five formal memory models—sequential consistency (SC), coherency, parallel random access memory (PRAM), processor consistency (PC), and linearizability—and the differences between the five models. Section 4.9 defines the *memory model verification* problem. Finally, Section 4.10 provides concluding remarks.

4.3 Program and Execution

For the purposes of verifying a memory system, a sequential program is abstracted to its memory operations; i.e., all other instructions such as arithmetic operations, branches are removed. As a result, a sequential program with branches, etc. may need to represented by multiple programs without branches. This is shown in Figure 4.1 where sequential program P is represented by P1 or P2 depending on whether the `if` branch or the `else` branch is taken. A rd instruction indicates a read operation (or a LOAD instruction) and a wr instruction indicates a write operation (or a STORE instruction). Predictably, a rd instruction takes an address as an argument and returns a data value. A wr instruction takes an address and a data value as an argument and does not return any value.

A concurrent program is simply an ordered set of sequential programs. This concurrent program is intended to run on a shared memory multiprocessor. A sequential execution (concurrent execution) is similar to a sequential program (concurrent program) where all rd instructions are annotated with a value to indicate the value returned by the instruction. The sequential execution E in Figure 4.1 shows a possible execution of P1. Note that such a sequence of operations is never executed by P itself as wr(b,2) is allowed in P only if rd(a) returned 1. However, due to the abstraction, such information is lost.

4.4 Sequential Consistency

The operational semantics of sequential consistency are provided with the help of Figure 4.2. Each processor executes one sequential program. The memory contains the data for all addresses; the processors themselves do not have any cache. At every "time unit" the memory nondeterministically chooses a processor to connect to. At that time, the processor may complete zero or more memory instructions by reading from and/or writing to the memory.

Formally, an execution is allowed under sequential consistency (SC) [58] if there is a

```
        P                  P1           P2             E

if (a == 1) then       rd(a)        rd(a)          rd(a, 5);
     b = 2;            wr(b,2)      wr(c,3)        wr(b, 2);
else
     c=3;
endif
```

Figure 4.1. Abstracting away all instructions other than memory instructions

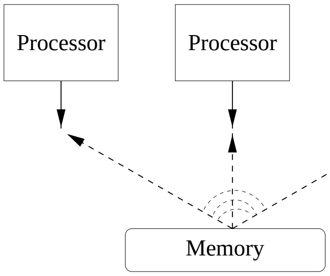


Figure 4.2. Operational semantics of SC

sequence S such that

SC1. S contains all instructions in the execution,

SC2. each rd instruction in S returns the value written by the most recent wr instruction for that address in S; if there is no such wr instruction then the value returned is a predefined constant $\top$, and

SC3. if an instruction i_1 appears before i_2 in some sequential execution, then they also appear in that order in S.

Ex1 in Figure 4.3 shows an execution that is allowed under SC, and Ex2 shows an execution that is *not* allowed under SC. Ex1 can be shown to be SC by the sequence $x_1y_1y_2x_2$. Ex2 cannot be explained by any sequence where both x_1 appears before x_2 and y_1 appears before y_2; thus any sequence that satisfies SC2 violates SC3.

4.5 Coherency

Coherency is a weaker condition than sequential consistency in that it requires the execution be sequentially consistent only one address at a time. Formally, coherency requires that an execution be explained by a set of sequences S_x, S_y ... where $x, y \ldots$ are all the addresses used in the execution such that each S_a satisfies the following conditions:

Coh1. S_a consists of all instructions in the execution that have a as their operand,

Ex1		Ex2	
X	Y	X	Y
x_1 : wr(A,1);	y_1 : rd(A,1);	x_1 : wr(A,1);	y_1 : wr(B,1);
x_2 : rd(B,1);	y_2 : wr(B,1);	x_2 : rd(B,$\top$);	y_2 : rd(A,$\top$);

Figure 4.3. Examples for sequential consistency and coherency

Coh2. each rd in S_a returns the value written by most recent wr instruction in S_a; if there is no such instruction, the value returned is $\top$, and

Coh3. if two instructions i_1 and i_2 appear in that order in some sequential execution and both involve the operand a, then they also appear in that order S_a.

Ex2 of Figure 4.3 is allowed under coherency, as it can be explained by sequences y_2, x_1 for address A and x_2, y_1 for address B. Ex3 of Figure 4.4 shows an execution that is not coherent. Note that coherency is strictly less stringent than SC, hence Ex3 is not SC either.

4.6 Parallel Random Access Memory

The operational semantics of parallel random access memory [65] can be provided with the aid of Figure 4.5. Each processor has its own memory, and they are interconnected by a point-to-point order preserving network. Whenever a processor updates its memory, it notifies all other processors by a message. When a processor receives a notification from another, it updates its memory accordingly to reflect the new value.

Formally, an execution is PRAM if for each sequential execution X there is a sequence S_X such that

PRAM1. S_X contains all instructions from X and all wr instructions from all other sequential executions,

Ex3

X	Y
x_1 : wr(A,1);	y_1 : wr(A,2);
x_2 : rd(A,2);	y_2 : rd(A,1);

Figure 4.4. An execution that is not coherent

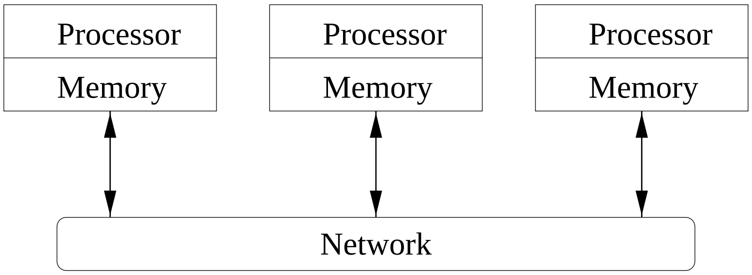


Figure 4.5. Operational semantics of PRAM

PRAM2. each rd instruction in S_X returns the most recent value written for that address in S_X; if there is no such wr instruction, rd returns $\top$, and

PRAM3. if i_1 appears before i_2 in some sequential execution, and if they also appear in S_X, then they appear in that order in S_X.

Ex2 is PRAM as demonstrated by $S_X = x_1x_2y_1$ and $S_Y = y_1y_2x_1$. Ex3 is also PRAM as demonstrated by $S_X = x_1y_1x_2$ and $S_Y = y_1x_1y_2$. Ex4 of Figure 4.6 shows an execution that is not PRAM, but coherent. PRAM and coherent are not comparable: Ex3 is not coherent but PRAM, and Ex4 is not PRAM but coherent. PRAM is also less stringent than SC.

4.7 Processor Consistency

Processor consistency (PC) [38] is defined as a conjunction of coherence and PRAM. Formally, an execution is said to be processor consistent if there are a set of sequences $S_x, S_y \ldots$ and $S_X, S_Y \ldots$, where $x, y \ldots$ are addresses and $X, Y \ldots$ are sequential executions such that

PC1. S_x consists of all instructions for address x,

PC2. S_X consists of all instructions of X and all wr instructions,

PC3. each rd in each sequence (S_x or S_X) returns the value written by the most recent wr instruction for that address in the sequence; if there is no such instruction the value returned is $\top$,

PC4. if two instructions i_1 and i_2 appear in that order in some sequential execution and also appear in S_x (S_X), then they appear in the same order in S_x (S_X), and

PC5. if two instructions i_1 and i_2 appear in both S_x and S_X then they appear in the same order in both sequences.

Ex4

X	Y
x_1 : wr(A,1);	y_1 : rd(B,1);
x_2 : wr(B,1);	y_2 : rd(A,$\top$);

Figure 4.6. An execution that is not PRAM, but coherent

Note that the last condition places a constraint on the structure of S_x and S_X: it is not acceptable to give an explanation for coherency using an explanation that contradicts the explanation of PRAM. Ex5 of Figure 4.7 (taken from [2]) shows an example that is both PRAM and coherent independently but not PC. The reason for this is that S_A is $y_2 z_1 x_1 z_2$ and S_Y is $x_1 x_2 y_1 y_2$; i.e., y_2 occurs before x_1 in S_A but x_1 occurs before y_2 in S_Y. Hence the execution is not PC.

4.8 Linearizability

Linearizability [44] is a popular correctness condition used in databases. The operational semantics of linearizability are provided with the aid of Figure 4.8. A shared memory system consists of a set of processors, where each processor consists of a sequential thread, shown as *process* in the figure, and a coherency manager, shown as *cache* in the figure. The caches maintain the consistency of the data by using the communication medium, shown as *network* in the figure.

The process can communicate with the cache using a *rd*(a) and *wr*(a, d) commands. When the cache completes the command, it issues *ok*(d) to the process in response to a *rd* instruction or *ok* in response to a *wr* instruction. The interface between the process and the cache is strictly *serial*; i.e., once a process issues a *rd* or *wr* command to its cache, it waits until the cache responds with *ok* before issuing another command. For each instruction i, req(i) indicates the time at which the process initiated the instruction

Ex5

X	Y	Z
x_1 : wr(A,1)	y_1 : rd(B,1)	z_1 : rd(A,2)
x_2 : wr(B,1)	y_2 : wr(A,2)	z_2 : rd(A,1)

Figure 4.7. An execution that is coherent and PRAM but not PC

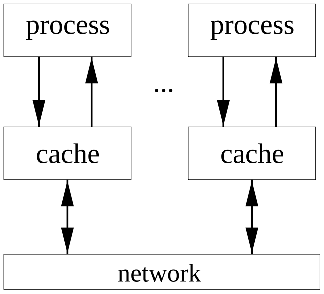


Figure 4.8. Operational semantics of linearizability

at the process/cache interface, and ack(i) indicates the time at which the cache issued *ok* for the instruction.

An execution generated by such a system is said to be linearizable if there is a sequence S such that

L1. S contains all instructions in the execution,

L2. each rd instruction in S returns the value written by the most recent wr instruction for that address in S; if there is no such wr instruction then the value returned is a predefined constant $\top$, and ,

L3. if $\mathrm{ack}(i_1) < \mathrm{req}(i_2)$ for some i_1 and i_2 in S, then i_1 appears before i_2 in S.

Note that condition L3 above is strictly stronger than SC3 due to the serial interface between the process and the cache.

Though linearizability appears to be a very close approximation of SC, it is not widely used to described shared memory systems. The principle reason is that the interface between the process and the cache must be serial, which eliminates many potential optimizations. From a programmer perspective, however, linearizability and SC are indistinguishable, as the program running as part of the process rarely, if ever, has access to the time at which the instructions are started and finished at the process/cache interface.

4.9 Memory Model Verification Problem

A formal memory model determines whether a *given execution* is allowed by the model. In contrast, the memory model verification is not whether a given execution satisfies a formal memory model—it is rather whether *every execution generated by a given model* satisfies the formal memory model. This is a subtle but very important distinction. For example, one can construct an execution using n addresses that violates PRAM, but satisfies PRAM when projected onto any $n-1$ addresses. However, as Theorem 5.2 shows, for most practical models, if the model shows a violation of PRAM with $n > 2$ addresses, then it also shows a violation with two addresses. This distinction is explained with the help of execution Ex6 in Figure 4.9. This execution is not PRAM as shown by the following analysis. In S_X,

1. y_5 must be done before x_1 for rd(C) to return 1,

2. x_2 must be done after x_1 to satisfy PRAM 3, hence it must be done after y_5; the only instruction that can make B to be 1 after y_5 is y_7. In other words, y_7 must be done before x_2, and

3. if y_7 is done before x_2, then y_6 is done before x_3, which implies rd(A) must return 3, which contradicts x_3.

However, this sequence is PRAM when only two addresses are considered at a time. When considering A and B, every sequence that starts with $y_1y_2x_1x_2$ satisfies the PRAM conditions. Similarly, when considering B and C, any sequence that ends with x_1x_2 satisfies the PRAM conditions, and when considering C and A, any sequence that contains $y_5x_1x_3$ satisfies the conditions. The reason for the "anomaly" is that there is an inherent ambiguity in explaining how x_2 could have returned 1 (written by y_2 or y_7), and x_3 could have returned 1 (written by y_1 or y_3). In other words, repeating the instructions wr(B,1) and wr(A,1) leads to an ambiguity that can be resolved only when all three addresses are considered to show that the execution violates PRAM.

If the ambiguity is resolved, for example, by replacing y_3 with wr(A,2) and y_7 by wr(B,3), then all resulting execution can be shown to violate PRAM by considering just two addresses. Note that replacing y_3 and y_7 means that x_3 *may* also need to replaced by rd(A,2) and x_2 by rd(B,3). In other words, there are four combinations of executions that need to be considered.

M1. Both x_2 and x_3 remain unchanged,

M2. x_2 changes to show that rd(B) may return 3, and x_3 does not change,

M3. x_2 does not change, and x_3 changes to reflect that rd(A) may return 2, and

Ex6

X	Y
x_1 : rd(C,1);	y_1 : wr(A,1);
x_2 : rd(B,1);	y_2 : wr(B,1);
x_3 : rd(A,1);	y_3 : wr(A,1);
	y_4 : wr(B,2);
	y_5 : wr(C,1);
	y_6 : wr(A,3);
	y_7 : wr(B,1);

Figure 4.9. An ambiguous execution that is not PRAM

M4. both x_2 and x_3 change as above.

The execution corresponding to M1 is shown by X1 and Y1 and the execution corresponding to M2 is shown by X2 and Y1 in Figure 4.10. The execution X1 and Y1 has a circuit involving only addresses A, and C, and the execution X2 and Y1 has a circuit involving addresses A and B only. The other two cases also reveal violations using only two addresses.

Chapter 5 presents two conditions called *projectable* and *data independence* under which it is sufficient to consider only unambiguous programs to test whether a given model correctly implements PRAM or SC. To our knowledge, these conditions are met by all current concurrent memory systems. Chapter 5 also presents test programs to verify whether a given system is PRAM or SC.

4.10 Concluding Remarks

The difference between verification of a model's and an executions to conformance to a formal memory model is crucial and will form the basis of a technique called test model checking that solves the memory model verification problem. Theorem B.1 shows that if the model uses only certain constructs, to show that the model satisfies any formal memory, it is sufficient to consider its behavior on "unambiguous" programs—concurrent programs where no two write instructions write the same value to the same address. Building on this result, Theorem B.4 shows that to verify that a model conforms to PRAM, it is sufficient to consider only unambiguous programs using one or two addresses only, and Theorem B.5 shows that to verify that a model conforms to SC, it is sufficient only unambiguous programs using no more than N addresses, where N is the number of processors in the model.

Ex7 & Ex8

X1	Y1	X2
x_1 : rd(C,1);	y_1 : wr(A,1);	x_1 : rd(C,1);
x_2 : rd(B,1);	y_2 : wr(B,1);	x_2' : rd(B,3);
x_3 : rd(A,1);	y_3' : wr(A,2);	x_3 : rd(A,1);
	y_4 : wr(B,2);	
	y_5 : wr(C,1);	
	y_6 : wr(A,3);	
	y_7' : wr(B,3);	

Figure 4.10. Unambiguous executions that are not PRAM

CHAPTER 5

MEMORY MODEL VERIFICATION

5.1 Chapter Overview

This chapter presents an *incomplete* verification technique—a verification methodology that may not reveal *all* errors—called test model checking for solving the memory model problem. The chapter also presents two conditions called *projectable* and *data independence* under which the the test model checking can be made *complete*. To our knowledge, these conditions are met by all contemporary memory systems. Appendix A presents a modeling language such that all memory systems expressed in the language meet these conditions. Appendix B proves that the test model checking can be made complete under these two conditions. Section 5.7 uses these results to present complete tests for PRAM and SC.

5.2 Introduction

The fundamentally important problem of verifying whether a given *memory system model* (or "a memory system") provides a *formal memory model* (or "memory model") [1] appears in a number of guises. CPU designers are interested in knowing whether some of the aggressive execution techniques such as speculative issue of memory operations violate sequential consistency; I/O bus designers are interested in knowing the exact semantics of shared accesses provided by split I/O transactions [22]; even language designers of multithreaded languages such as Java that support shared updates [40] are interested in this problem.

Formal verification methods are ideally suited for this problem because (i) the semantics of memory orderings are too subtle to be fathomed through informal reasoning alone; (ii) ad hoc testing methods cannot provide assurance that the desired memory model has been implemented. Unfortunately, despite the central importance of this problem and the large body of formal methods research in this area, there is still no single formally based method that the designer of a realistic multiprocessor system can use on his/her detailed design model to *quickly* find violations in the design. In this chapter we describe such a method called *test model checking.*

Test model checking formally adapts to the realm of model checking a formally based architectural testing method called ARCHTEST. ARCHTEST has been successfully used on a number of commercial multiprocessors [20] by running a suite of test-programs on them. ARCHTEST is an *incomplete* testing method in that it does not, under all circumstances, detect violations of memory orderings [21]. Nevertheless, its tests have been shown to be incisive in practice [20]. Most importantly, the formal theory of memory ordering rules developed by Collier in [21] forms the basis for ARCHTEST, which means that whenever a violation is detected by ARCHTEST, there is a formal line of reasoning leading back to the precise cause.

Being based on ARCHTEST, test model checking is also incomplete. However, none of the (presumed) complete alternatives to date have been shown to be practical for verifying large designs. For example [75] involves the use of manually guided mechanical theorem proving. Even approaches based on *conventional* model checking are impossibly difficult to use in practice. For example, the assertions pertaining to the sequential consistency of lazy caching [30], a simple memory system, expressed in various temporal logics (by Graf [41] in $\forall$CTL*, Clark et al. [19], and Ladkin et al. [57] in TLA [60]) are highly complex. We do not believe that descriptions of this style will scale up. On the other hand, the test model checking method has not only been able to comfortably handle the memory system defined by a state-of-the art symmetric multiprocessor (SMP) memory bus called *Runway* [10, 39] used by Hewlett-Packard in their high-end machines, but also it discovered many subtle bugs in early models describing this bus that *we created.* Our model includes a number of details such as split transactions, out of order transaction completions and even an element of speculative execution. The errors we made in capturing these details could well have been made in an actual industrial context. We believe that with growing system complexity, the role of debugging methods that are effective and are formally based will only grow in significance, regardless of whether the methods are complete or not.

Test model checking also has a number of other desirable features. It involves model checking a *fixed* set of safety properties for each formal memory model, that are *independent* of the actual memory system model being tested. This fixed nature greatly facilitates the use of test model checking *within the design cycle* where debugging is most effective, design changes are frequent, and time-consuming alterations to the properties being verified following design changes would be frowned upon (test model checking will not need such alterations). Also, the formal adaptation of the tests of ARCHTEST made in test model checking can be verified once and for all, thanks to the fixed set of tests used in test model checking (we describe and argue the correctness of these abstractions later). Finally, in test model checking, a memory model is viewed as a collection of simpler ordering rules. For each constituent ordering rule, a specific property is tested on the memory system. We found that this significantly helps compartmentalize errors, as opposed to producing nonintuitive error traces that could result during conventional model checking, which can be very difficult to understand for realistic memory systems.

Test model checking is also a more effective debugger for memory models than ARCHTEST in a formal sense. The tests of ARCHTEST are straight-line programs of length k, one per node. Such programs execute on various nodes of the multiprocessor concurrently. The recommendation accompanying ARCHTEST is that users run the tests for as large a k that is feasible, because then the chances of being scheduled according to different interleavings (by the underlying operating system, memory controller arbiter, etc.) increase. In adapting the tests of ARCHTEST, test model checking gives the effect of choosing $k = \infty$. Thus, we cover *all possible schedules.* The subtle errors detected by test model checking on realistic examples that are reported in Section 5.8 corroborate our intuition that test model checking is indeed an effective debugging tool for memory models.

Two conditions called *projectable* and *data independence* are also presented, which if is true of the model, then the test model checking technique can be made complete. Finally, complete tests for PRAM and SC are presented.

To summarize, the specific contributions of this chapter are as follows:

- the adaptation of a formal testing method for memory models to model checking that can be applied during the design of modern microprocessors whose memory systems can be very complex;
- a formal characterization (and proofs) of *how* the tests of the testing method are abstracted and turned into a fixed set of safety properties that are then model checked;
- experimental results of verifying a state-of-the art SMP memory bus called Runway

using the PV and SPIN model checkers;

- experimental results of verifying three examples (including Runway) using the VIS model checker;
- two conditions called projectable and data independence, and the result that the test model checking can be made *complete* under these two conditions; and
- a set of tests for PRAM and SC that guarantee that a memory system satisfying the *projectable* and *data independence* conditions passes the tests if and only if it correctly implements the set of formal memory model associated with the test.

5.3 Related Work

In [41], abstract interpretation [24] is employed to reduce infinite-system verification to finite $\forall$CTL* model checking. They apply this technique to verify the sequential consistency of lazy caching with unbounded queues. They recognize that to get an exact characterization of sequential consistency involving only the observable event names, one needs full second-order logic [41]. To be able to express sequential consistency in $\forall$CTL*, they give a stronger characterization of sequential consistency. For this stronger characterization, the expression of sequential consistency is very complex, as shown in Figure 5.1 (this figure shows only part of their sequential consistency expression). It is not feasible to use this approach with more realistic and complicated systems such as Runway due to the complexity involved.

A technique very similar to test model checking was proposed in [67] under the section heading "Sequential Consistency." However, this test cannot be a complete test for sequential consistency, as it involves only a single address.

In [75], the authors use a method called *aggregation* on a distributed shared memory coherence protocol used in an experimental multiprocessor, to arrive at a simplified model of system behavior. Their technique involves manual theorem proving. The work in [46] as well as [27] are aimed at verifying that synchronization routines work correctly under various memory models, where the memory models themselves are described using finite-state operational models. They do not address the problem of establishing the memory models provided by detailed memory subsystem designs, which is our contribution. In [32, 33], the authors analyze the problem of deciding whether a given set of traces are sequentially consistent. Our approach differs in two respects. First, we are interested in

$$\begin{array}{ll}
(C1) & \forall (a,d) \in \mathit{address} \times \mathit{datum}\ \forall i \in \mathit{index} : \\
 & \mathit{init} \implies \mathbf{AG}(\mathit{enable}(\mathit{read}_i(a,d)) \implies \mathit{avail}_i(a,d))
\end{array}$$

$$\begin{array}{ll}
(C2) & \forall (a,d), (a,d') \in \mathit{address} \times \mathit{datum}.d \neq d'\ \forall i \in \mathit{index} : \\
 & \mathit{init} \implies \mathbf{AG}((\mathit{avail}_i(a,d) \wedge \mathbf{EF}(\mathit{enable}(\mathit{read}_i(a,d)))) \implies \\
 & \qquad\qquad\qquad \mathbf{A}[\neg \mathit{avail}_i(a,d)\ \mathbf{W}\ \mathbf{AG}(\neg \mathit{avail}_i(a,d))])
\end{array}$$

$$\begin{array}{ll}
(C3) & \forall (a,d) \in \mathit{address} \times \mathit{datum}\ \forall i,k \in \mathit{index} : \\
 & \mathit{init} \implies \mathbf{AG}[\mathit{after}(\mathit{write}_k(a,d)) \implies \mathbf{AF}(\mathit{avail}_i(a,d))]
\end{array}$$

$$\begin{array}{ll}
(S1) & \forall (a,d) \in \mathit{address} \times \mathit{datum}\ \forall i \in \mathit{index} : \\
 & \mathit{init} \implies \mathbf{AG}[\mathit{after}(\mathit{write}_i(a,d)) \implies \\
 & \qquad\qquad\qquad \mathbf{A}(\neg \mathit{enabled}(\mathit{read}_i(a,d))\ \mathbf{W}\ \mathit{avail}_i(a,d))] \\
 & \vdots \\
(S4) & \forall (a,d), (a,d') \in \mathit{address} \times \mathit{datum}.\ d \neq d'\ \forall i,k \in \mathit{index} : \\
 & \mathit{init} \implies \mathbf{A}([\neg \mathit{avail}_i(a,d)\ \mathbf{W}\ (\mathit{avail}_i(a',d') \wedge \neg \mathit{avail}_i(a,d))] \implies \\
 & \qquad\qquad\qquad [\neg \mathit{avail}_k(a,d)\ \mathbf{W}\ \mathit{avail}_k(a',d')])
\end{array}$$

Figure 5.1. ∀CTL* specification of sequential consistency for lazy caching protocol

proving that detailed models of memory systems are correct, whereas they obtain traces (presumably from actual machines) and analyze them for sequential consistency. Second, our method is more useful for CPU designers as it can give feedback during early phases of the design.

[4] showed that the problem of verifying whether a given model implements sequential consistency is undecidable. The proof uses a model that can make decisions based on the data; i.e., the authors show that if the protocol followed by the model can inspect data present in write instructions and data returned by read instructions, then verification of model's conformance to sequential consistency is undecidable. In contrast, we assume that the memory model does not make use data for control purposes (referred to as *data independence* and formalized in Section 5.5.1) and show that the problem is decidable with this assumption if the model is also *projectable.* To our knowledge, this assumption holds in all contemporary memory systems.

5.4 Overview of ARCHTEST

ARCHTEST is based on the theory presented in [21] that formally defines and characterizes architectural *rules* obeyed by memory subsystems of multiprocessors. Although these rules are *elemental*, in realistic memory systems the rules manifest in *compound* form. Obeying a compound rule is tantamount to obeying *all* the constituent elemental rules;

violating a compound rule is tantamount to violating *any* of the constituent elemental rules. There are five crucial elemental ordering rules (these rules are also given equivalent definitions in Appendix B using a different formalization):

Rule of Computation (CMP): This is a basic rule defining how the terminal value of each operand is calculated from the initial values of the operand. When a program is abstracted to its memory instructions only, this condition is equivalent to the conditions SC2, Coh2, PRAM2, and PC3 defined in Chapter 4.

Rule of Program Order by Storage (POS): If a pair of events e_1 and e_2 come in that order in some sequential program P_1, then e_1 occurs before e_2.

Rule of Read Order (RO): If a pair of read events rd(a) and rd(b) come in that order in some sequential program P_1, then a is read before b.

Rule of Write Order by Storage (WOS): If a pair of write instructions wr(a, v_1) and wr(b,v_2) come in that order in a sequential program P_1, then every process (including P_1) would see the write for a before it would see the write for b.

Rule of Write Atomicity (WA): A write operation becomes visible to all sequential programs instantaneously. More precisely, one conceptual *store* S_i is associated with each processor node P_i. For each write operation W, one write event W_i is defined per store S_i. WA requires that there is no i, j and no event e such that e is before W_i but after W_j.

The test of ARCHTEST for the *compound* rule consisting of the elemental rules CMP, RO, and WOS, denoted (CMP, RO, WOS), is shown in Figure 5.2(a). (Figure 5.2(b) will be discussed later.) P_1 executes a sequence of write instructions (intended to check for WOS), and P_2 executes a sequence of read instruction (intended to check for RO). If the memory system correctly realizes (CMP, RO, WOS), then Condition 1 is true:

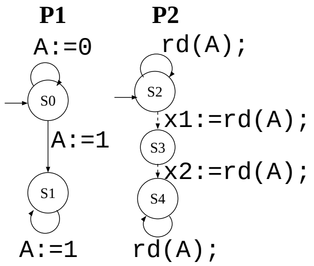
CONDITION 1 (MONOTONIC) The sequence of X values is monotonically increasing, i.e., $\forall i : 1 \leq i < k : X[i] \leq X[i+1]$.

Figure 5.3 shows the test for (CMP, RO, WOS, WA) of ARCHTEST, where the conditions checked are (i) the MONOTONIC condition (suitably modified for arrays U, V, X, Y) and (ii) ATOMIC, shown below:

InitiallyA = 0

$ProcessP_1$	$ProcessP_2$
$L_1 : A := 1;$	$X[1] := A;$
$L_2 : A := 2;$	$X[2] := A;$
$L_3 : A := 3;$	$X[3] := A;$
$\ldots$	$\ldots$
$L_k : A := k$	$X[k] := A;$

(a) Test 1 of ARCHTEST



(b) Test automata for Test 1

Figure 5.2. Test 1: A test to check for (CMP, RO, WOS)

InitiallyA = B = 0

P_1	P_2	P_3	P_4
$L_1 : A := 1;$	$L_{A_1} : U[1] := A;$	$L_{B_1} : X[1] := B;$	$L_1 : B := 1;$
$L_2 : A := 2;$	$L_{B_1} : V[1] := B;$	$L_{A_1} : Y[1] := A;$	$L_2 : B := 2;$
$\ldots$	$L_{A_2} : U[2] := A;$	$L_{B_2} : X[2] := B;$	$\ldots$
$L_k : A := k;$	$L_{B_2} : V[2] := B;$	$L_{A_2} : Y[1] := A;$	$L_k : B := k;$
	$\ldots$	$\ldots$	
	$L_{A_k} : U[k] := A;$	$L_{B_k} : X[k] := B;$	
	$L_{B_k} : V[k] := B;$	$L_{A_k} : Y[k] := A;$	

Figure 5.3. Test 2: A test to check for (CMP, RO, WOS, WA)

CONDITION 2 (ATOMIC) $\forall i, j : 1 \leq i, j \leq k : V[i] \geq X[j] \vee Y[j] \geq U[i]$.

The ATOMIC condition watches for the possibility that a write operation from P_1 and a write operation from P_4 appear to have finished in different orders to P_2 and P_3. Since test programs such as Test 2 are meant to be run on real machines, there cannot be any real guarantees that the particular interleavings that reveal violations (such as for condition WA watched by condition ATOMIC) will indeed happen. To allow for as many interleavings as possible, ARCHTEST recommends that its tests be run for large values of k. With test model checking, we effectively run the tests for $k = \infty$, as will be elaborated shortly.

5.5 Test Model Checking

Test model checking converts the tests of ARCHTEST to corresponding *memory rule test automata* ("test automata") that drive model of the memory system being examined. The CONDITIONs corresponding to each compound memory rule being tested are turned into corresponding *memory rule safety properties* that are checked by the model checker tool. In the remainder of this section, we explain the assumptions under which we formally derive *test automata* as well as *memory rule safety properties*, followed by a description of how test automata as well as memory rule safety properties are derived for specific cases.

5.5.1 Assumptions about memory systems realized in hardware

A memory system is said to be *data independent* if it does not base its control-point settings on the data values themselves, but simply moves the data around. A memory system M is said to be *projectable* if and only if the following condition holds: if E of M using address $\mathcal{A}$, then for every $\mathcal{A}^{'} \subseteq \mathcal{A}$ and the execution $E^{'}$ obtained by projecting E onto the addresses in $\mathcal{A}^{'}$, $E^{'}$ is also an execution of M. The intuition behind the condition is that realistic memory systems do not *loose* a capability to produce an execution when instructions dealing with new addresses are introduced into the concurrent program or when instructions related to some addresses are removed from the concurrent program.

These two conditions are true of all memory systems to our knowledge. Appendix A presents a modeling language where any system expressed in the language satisfies the above condition.

Every model expressed in the modeling language presented in Appendix A satisfies these two conditions.

5.5.2 Creation of test automata

As illustrated in Figure 5.2(b), we obtain test automata for various memory models by finitely abstracting the data used in test of ARCHTEST, using nondeterminism to justify the abstraction. For example, we abstract the specific activities of process P_1 of Figure 5.2(a) into that of (nondeterministically) writing *all possible* ascending values over {0,1}, as shown in P_1 of Figure 5.2(b). Also, since we cannot store infinite arrays in creating process P_2, we turn P_2 and the corresponding memory rule safety property into an automaton that checks that the array values read are monotonically increasing. This, in turn, can be performed using just two consecutive array values $x1$ and $x2$ that are nondeterministically recorded by P_2. Hence, the memory rule safety property we model check for is: P_2 *in final state* $\Rightarrow x2 \geq x1$.

We now provide a justification that these abstractions preserve the memory rule safety properties; i.e., for the a given model, a violation of a condition occurs in a test of ARCHTEST for $k = \infty$ iff the a violation occurs when model checking the memory rule safety property corresponding to the condition when the test automata are used to drive the memory system model. To keep the presentation simple, we formally argue how the test automata finds every violation present in the test of ARCHTEST with $k = \infty$; the opposite direction of *iff*; i.e., how a test of ARCHTEST with $k = \infty$ finds violations found by the test automata is easy to see because the test automata just appears as a "stuttering" of the test of ARCHTEST. For example, the actions of P_1 in Figure 5.2(b) can be viewed as repeating the initialization and then repeating the instruction at label L_1 of P_1 of Figure 5.2(a). Our proof sketches are illustrated on the two tests presented in Section 5.4 and another test described in this section.

5.5.3 Abstracting Test 1

We show that if the test program in Test 1 shows that MONOTONIC is violated, then the test automaton also reveals the error. Since MONOTONIC is violated,

$$\begin{array}{lll} & \exists i : & 1 \leq i < k : X[i] > X[i+1] \\ \Longleftrightarrow & \exists i, \alpha : & 1 \leq i < k : (X[i] > \alpha) \wedge (X[i+1] \leq \alpha) \\ \Longleftrightarrow & \exists i, \alpha : & 1 \leq i < k : (X[i] > \alpha) \wedge \neg(X[i+1] > \alpha) \end{array}$$

The last formula compares $X[i]$ and $X[i+1]$ only to α; hence we can rewrite the test program as shown in Figure 5.4(a) *assuming data independence*, and rewrite the last formulae as

$$\exists i : 1 \leq i < k : X[i] = 1 \wedge X[i+1] = 0$$

Note that in Figure 5.4(a) all reads of A occur in the expression $A > \alpha$. Hence, we can replace every $A := v$ with $A := (v > \alpha)$ and $X[i] := (A > \alpha)$ with $X[i] := A$ without affecting MONOTONIC again, *if data independence holds*, to obtain Figure 5.4(b). Figure 5.4(c) is obtained by simplifying Figure 5.4(b): each $v > \alpha$ evaluates to 0 for $v \leq \alpha$ and 1 otherwise. This figure is generalized to obtain the test automaton in Figure 5.2(b).

$$\begin{array}{ll} \multicolumn{2}{c}{\text{InitiallyA} = 0} \\ Process P_1 & Process P_2 \\ L_1 : A := 1; & X[1] := (A > \alpha); \\ L_2 : A := 2; & X[2] := (A > \alpha); \\ L_3 : A := 3; & X[3] := (A > \alpha); \\ \dots & \dots \\ L_k : A := k & X[k] := (A > \alpha); \end{array}$$

(a)

$$\begin{array}{ll} \multicolumn{2}{c}{\text{InitiallyA} = (0 > \alpha)} \\ Process P_1 & Process P_2 \\ L_1 : A := (1 > \alpha); & X[1] := A; \\ L_2 : A := (2 > \alpha); & X[2] := A; \\ L_3 : A := (3 > \alpha); & X[3] := A; \\ \dots & \dots \\ L_k : A := (k > \alpha) & X[k] := A; \end{array}$$

(b)

$$\begin{array}{ll} \multicolumn{2}{c}{\text{InitiallyA} = 0} \\ Process P_1 & Process P_2 \\ L_1 : A := 0; & X[1] := A; \\ \dots & \dots \\ L_\alpha : A := 0 & X[\alpha] := A \\ L_{\alpha+1} : A := 1 & X[\alpha+1] := A \\ L_{\alpha+2} : A := 1 & X[\alpha+2] := A \\ \dots & \dots \\ L_k : A := 1 & X[k] := A; \end{array}$$

(c)

Figure 5.4. Abstraction of Test 1

Intuitively the automaton finds the violation as follows. P_1 remains in the initial state for α iterations (executing A:=0) and then switches to second state (executing A:=1). Also, P_2 remains in the initial state for $i-1$ iterations and then switches to second state recording $x1$ and then $x2$ (dashed edges show when these variables are recorded). Thus the test automaton's execution is identical to that in Figure 5.4(c) except that the test automaton gives the effect of taking k to ∞. Also notice that $x1$ and $x2$ get the values corresponding to $X[i]$ and $X[i+1]$. Also, corresponding to $X[i] = 1 \wedge X[i+1] = 0$, we have $x1 = 1 \wedge x2 = 0$. Hence the memory rule safety property corresponding to condition MONOTONIC is found violated by the test automaton exactly when Test 1 for $k = \infty$ detects a violation. Note that the nondeterminism employed in constructing test automata enables P_1 and P_2 to *guess* the right value of α and i corresponding to the violation.

5.5.4 Abstracting Test 2

Test automaton for Test 2 is shown in Figure 5.5. In this automaton P_1 and P_4 write all possible ascending sequences of $\{0, 1\}$ in A and B respectively. Each processor *independently* and *nondeterministically* decides to switch from writing 0 to writing 1.

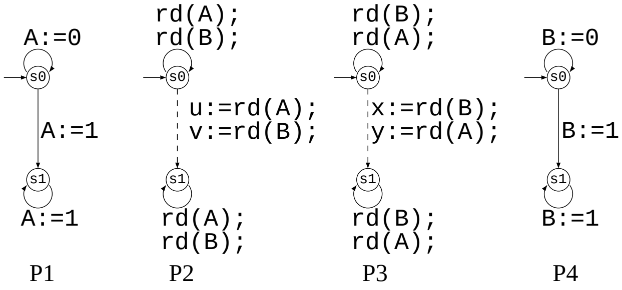


Figure 5.5. Test automata for Test 2

Modifications similar to those in Test 1 are applied to P_2 and P_3 also, to (nondeterministically) decide which $U[i]$,$V[i]$ pair and $X[j]$,$Y[j]$ pair are recorded in u, v and x, y. The memory rule safety property corresponding to condition Atomic is: *P_2 and P_3 in their final states* $\Rightarrow v \geq x \vee y \geq u$. As was explained in Section 5.5.2 for Test 1, our abstraction avoids having to remember the entire extent of the arrays U, V, X, and Y. (In Test 2, one has to check for Monotonic also; this is done similarly to that in Test 1.)

To show that the abstraction preserves Atomic, let Atomic be violated in Test 2 of Archtest. Hence

$$\begin{array}{lll} & \exists i, j : & U[i] > Y[j] \wedge X[j] > V[i] \\ \Longleftrightarrow & \exists, i, j, \alpha, \beta : & Y[j] = \alpha \wedge U[i] > \alpha \wedge V[i] = \beta \wedge X[j] > \beta \end{array}$$

Similar to Test 1, assuming *data-independence*, we have an execution of the test automaton (Figure 5.5) in which P_1, P_2, P_3, P_4 iterates for $\alpha, i-1, j-1, \beta$ times (respectively) in their initial states before switching to their final states. This test automaton execution detects violations of Atomic exactly when Test 2 for $k = \infty$ would. A violation of Atomic happens exactly when $u = 1 \wedge v = 0 \wedge x = 1 \wedge y = 0$.

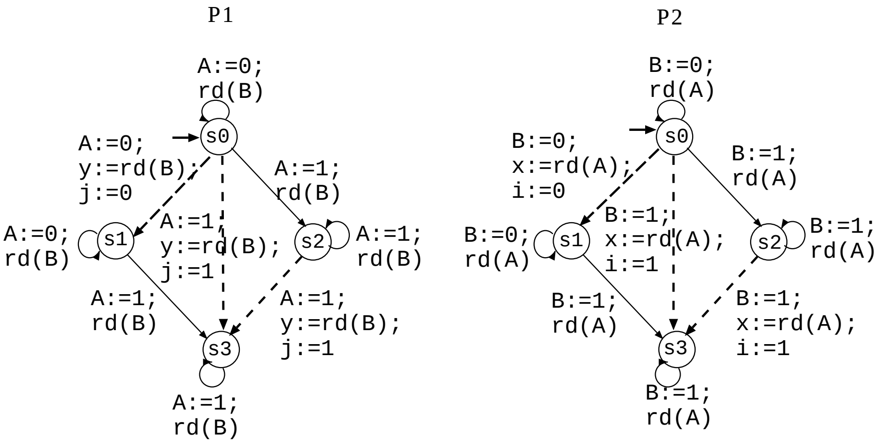
5.5.5 Abstracting Test 3

POS requires that two events of the same process occur in the order specified by the program. Archtest provides the test for the compound rule (CMP,POS) shown in Figure 5.6. Violation of (CMP,POS) is detected if Condition 3 fails:

Condition 3 (PO_Cross) $\forall i, j : 1 \leq i, j \leq k : (X[i] \geq j \vee Y[j] \geq i) \wedge (X[i] \leq j \vee Y[j] \leq i)$.

$$\begin{array}{ll}
\multicolumn{2}{c}{\text{Initially}\mathrm{A} = \mathrm{B} = 0} \\
L_{11} : A := 1; & L_{11} : B := 1; \\
L_{12} : Y[1] := B; & L_{12} : X[1] := A; \\
L_{21} : A := 2; & L_{21} : B := 2; \\
L_{22} : Y[2] := B; & L_{22} : X[2] := A; \\
\ldots & \ldots \\
L_{k1} : A := k; & L_{k1} : B := k; \\
L_{k1} : Y[k] := B; & L_{k1} : X[k] := A;
\end{array}$$

(a) Test 3 of ARCHTEST



(b) Test automata for Test 3

Figure 5.6. Test 3: A test and corresponding test automaton for (CMP, POS)

We obtain the test automaton and the memory rule safety property for Test 3 of Figure 5.6(a) as illustrated in Figure 5.6(b). P_1 executes a pair of instructions: write to A followed by read from B, infinitely often. The value written to A is 0 for some iterations and is nondeterministically changed to 1. P_2 runs similarly. P_1 nondeterministically selects a pair of write followed by read instruction. It assigns the value written to A to j and the value read from B to y. Similarly, processor 2 updates i and x. The dashed edges in Figure 5.6 show when x, y, i, j are updated. The memory rule safety property corresponding to condition PO_Cross is "*P_1 and P_2 in their final states* $\Rightarrow (x \geq j \vee y \geq i) \wedge (x \leq j \vee y \leq i)$." We can show that this abstraction preserves PO_Cross by an argument similar to that for Test 1 and Test 2, as given in [68].

5.6 Case Studies

To demonstrate the effectiveness of our approach, we verified a simplified model of the Runway bus using PV, along with serial memory, lazy caching and a (different) simplified model of the Runway bus using VIS [6], an implicit enumeration based model checker. These three memory systems are described in some detail below, along with some of the subtle ordering violations that we could detect using test model checking.

5.6.1 Sequential consistency and serial memory protocol

From the definition of sequential consistency in Chapter 4, it is clear that it is equivalent to (CMP, POS, WA). ARCHTEST does not provide a single compound test to check for (CMP, POS, WA). However, it provides a test for (CMP, RO, WOS, WA) and a test for (CMP, POS). This combination is exactly equivalent to testing sequential consistency as POS is strictly stronger than both RO and WOS. For every memory system we consider, these two tests are model checked separately and summarized in Table 5.3.

5.6.2 Serial memory and lazy caching

The serial memory protocol for n processors and a memory is shown in Table 5.1. Serial memories are often used to define SC operationally. The lazy caching protocol [30], shown in Table 5.2, also implements sequential consistency and is geared towards a bus based architecture.

In lazy caching, the memory interface still consists of reads and writes; however, caches C_i are interposed between the shared memory Mem and the processors P_i. Each cache C_i

Table 5.1. Serial memory transaction rules

Event	Action or condition
Ri(d, a)	if Mem[a] = d
Wi(d, a)	Mem[a] := a

Table 5.2. Gerth's version of the lazy caching protocol

Event	Allowed if	Action
$\mathsf{R}_i(d, a)$	$C_i(a) = d \wedge Out_i = \{\}$ $\wedge$ no $*$-ed entries in In_i	
$\mathsf{W}_i(d, a)$		$Out_i := append(Out_i, (d, a))$
$\mathsf{MW}_i(d, a)$	$head(Out_i) = (d, a)$	$Mem[a] := d;$ $Out_i := tail(Out_i);$ $(\forall k \neq i \; :: \; In_k := append(In_k, (d, a)));$ $In_i := append(In_i, (d, a, *))$
$\mathsf{MR}_i(d, a)$	$Mem[a] = d$	$In_i := append(In_i, (d, a))$
$\mathsf{CU}_i(d, a)$	$head(In_i)$ is either (d, a) or $(d, a, *)$	$In_i := tail(In_i); \; C_i := update(C_i, d, a)$
CI_i		$C_i := restrict(C_i)$

Initially: $\forall a \;\; Mem[a] = 0$
$\wedge \, \forall i = 1 \ldots n \;\; C_i \subset Mem \wedge In_i = \{\} \wedge Out_i = \{\}$
Fairness: no action other than CI_i can be always enabled but never taken

W—write
R—read
MW—memory write
MR—memory read
CU—cache update
CI—cache invalidate

contains a part of the memory Mem and has two queues associated with it: an out-queue Out_i in which P_i write requests are buffered and an in-queue IN_i in which the pending cache updates are stored. These queues model the asynchronous behavior of write events in a sequentially consistent memory. A write event $W_i(a,d)$ does not have an immediate effect. Instead, a request (d,a) is placed in Out_i. When the write request is taken out of the queue, by an internal memory-write event $MW_i(a,d)$, the memory is updated and a cache update request (d,a) is placed in every in-queue. This cache update is eventually removed by an internal cache update event $CU_j(a,d)$ as a result of which the cache C_j gets updated. Cache evictions are modeled by internal caches invalidate events: CI_i can arbitrarily remove locations from cache C_i. Caches are filled both as the delayed result of write events and through internal memory-read events, $MR(a,d)$. The latter events model the effect of a cache-miss: in that case the read event stalls until the location is copied from the memory. A read event $R_i(a,d)$, predictably, stalls until a copy of location a is present in C_i but also until the copy contains a correct value in the following sense: SC demands that a processor P_i reads the value at a location a that was recently written by P_i unless some other processor updated a in the meantime. Hence, a read event $R_i(a,d)$ cannot occur unless all pending writes in Out_i are processed as well as the cache updates requests from In_i that corresponds to writes of P_i. For this reason, such cache updates requests are marked (with a $\star$).

5.6.3 Runway

Our third example, called Runway, is modeled after a commercial bus used to interconnect processors and memory controller together to form a multiprocessor system. The behavior of this memory system is described in some detail in [39]. The complexity of this protocol stems from many sources, a few of which are elaborated here (see [39] for more details). First, the queues in the clients introduce decoupled execution, leading to a large number of “otherwise equivalent” states. Next, the control mechanism is very complex, owing to many reasons, including: (i) lines can be obtained in various sharing modes such as read-shared-private and read-private; (ii) line states can be eagerly promoted to *private* before the data actually arrives (concurrent dirtying are merged into when the data arrives); (iii) hit after miss situations can be speculatively processed and unrolled when invalidated. Though we did not try to model each of these features in their full glory, we *did* include a modicum of these aggressive features into our models.

5.6.4 VIS verification results

Table 5.3 shows execution time for model checking our Serial memory, lazy caching and Runway models for tests of (CMP, POS) and (CMP,RO,WOS,WA) using VIS. All experiments are conducted on a SPARC Ultra-1 with 512MB memory. Recall that (CMP, POS, WA) implies sequential consistency. The size of the state space and number of nodes in BDDs are also reported. Note the large number of states and small BDD size for lazy caching (compared to Runway) which are respectively due to queues and the low complexity of the control logic. On the other hand, observe that the very complex control logic of Runway model causes the large BDD size, which in turn results in high run-time to finish searching correct models. However, in all our experiments, whenever there was any memory ordering rule violation in our model, test model checking detected it quickly (in the order of minutes). A very desirable feature one can provide in a tool based on test model checking is a *menu* of previously generated test automata for the various compound rules in [21], using which designers can probe their model.

We now summarize an insidious error in our models that has been revealed using test

Table 5.3. Memory model verification using VIS

(CMP,POS)	#states	#bdd nodes	conditions verified	runtime (mn:sec)
serial memory	7229	7145	Vacuity Cond1 and Cond4	00:02 00:09
lazy caching	7.80248e+06	306692	Vacuity Cond1 and Cond4	01:12 36:33
Runway	953675	1657308	Vacuity Cond1 and Cond4	14:23 27h28:30

(CMP,WOS,RO,WA)	#states	#bdd nodes	conditions verified	runtime (mn:sec)
serial memory	21242	10084	Vacuity Cond1 – Cond3	00:04 00:34
lazy caching	1.90736e+06	513655	Vacuity Cond1 – Cond3	02:02 59:33
Runway	985236	1695092	Vacuity Cond1 – Cond3	17:24 40h17:33

Vacuity: Antecedent of **implies** is not always false
Cond1: All values in 0..k
Cond2: Values increase monotonically
Cond3: $V[i] \geq X[j]$ or $Y[j] \geq U[i]$
Cond4: ($X[i] \geq j$ or $Y[j] \geq i$) and ($X[i] <= j$ or $Y[j] <= i$) (POS)

model checking.

Description of an ordering violation: The following error in our model of lazy caching was caught by a violation of Test 4. The error was in the queues used by lazy caching, which were implemented as shift registers. We forgot to shift the $\star$-bit in In_i when the processor P_i receives a cache-update from In_i queue. With this error it is possible that In_i queue is not $\star$-ed when it should be; consequently reads in P_i may bypass writes. This results in a violation of POS. This is a difficult error to find because its detection involves understanding the complex feedback from all components of the protocol to each other (queues, memory, and caches). Moreover, this error is interesting because it violates POS but does not violate WA. This is so because only write-read (WR) order is affected by this error. Our technique effectively caught this error the POS conditions does not pass when we model checked the model for Test 3 (for (CMP,POS)). However, Test 7 for (CMP,RO,WOS,WA) (note that it does not involve POS) passes! This shows the futility of ad hoc testing methods: one could apply subjective criteria to consider a test similar to Test 7 to be sufficiently incisive, when in fact it fails to account for a crucial ordering relation such as POS. The distinctive advantage of a formally based testing method such as ARCHTEST is that it covers various compound memory ordering rules must be apparent.

5.6.5 PV and SPIN verification results

Even though VIS runway models for testing (CMP, POS) and (CMP, RO, WOS, WA) have less than 1e+06 states, verification took over 36 hours for each. The reason for this is that the protocol is sufficiently complicated that there is no good ordering of the variables to obtain a compact BDD representation; hence the number of BDD nodes is high, which leads to a high runtime (and memory usage). Most explicit enumeration model checkers can handle 1e+06 states comfortably and finish in less than an hour on a machine similar to the one used for the VIS experiments. Hence, we developed a Promela model of the protocol. This model explicitly represents all queues, whereas the VIS model abstracted most of the queues. This Promela model is model checked using PV and SPIN model checkers, introduced in Chapter 2. The results are summarized in Table 5.4.

As the Promela model has more details than present in the VIS model, even with partial order reductions present in SPIN, the number of reachable states is approximately five times the number of reachable states in VIS model. The first two rows, (CMP, RO,

Table 5.4. Runway memory model verification using SPIN and PV

Test	SPIN		PV	
	states	runtime (sec)	states	runtime (sec)
(CMP,RO,WOS,WA)	4.8e+06	340	169,680	21
(CMP,POS)	5.2e+06	330	222,636	32

WOS, WA) and (CMP, POS), are the tests presented in Figures 5.5 and 5.6. The runtime required needed to complete PV models is less than six minutes (in contrast, VIS models required 36 hours and 40 hours or runtime respectively). As can be seen from the table, the two phase partial order reduction algorithm in PV generates far fewer states the partial order reduction algorithm in SPIN.

5.7 Complete Tests Based on the Test Model Checking Approach

Test model checking can be made complete if the shared memory system under inspection is *projectable* and *data independent*. Given these conditions, Appendix B proves the following theorems.

THEOREM 5.1 Let M be a shared memory system with N components, and E be an execution of M. (A component is, for all practical purposes, a processor. This notion is formalized in Appendix A.) If E shows that the composite rule (CMP, RO, WOS) is violated, then there is an *unambiguous* execution with no more than two addresses that also reveals that M violates (CMP, RO, WOS).

Proof: See Theorem B.3. □

THEOREM 5.2 Let M be a shared memory system with N components and E be an execution of M. If E shows that the composite rule (CMP, POS) is violated, then there is an *unambiguous* execution with no more than two addresses that also reveals that M violates (CMP, POS).

Proof: See Theorem B.4. □

THEOREM 5.3 Let M be a shared memory system with N components and E be an execution of M. If E shows that the composite rule (CMP, POS, WA) is violated, then there is an *unambiguous* execution with no more than N addresses that also reveals that M violates (CMP, POS, WA).

Proof: See Theorem B.5. □

Using these three theorems, one can obtain complete tests for verifying the three architectural rules, as we now show.

5.7.1 Verifying (CMP, RO, WOS)

From Theorem 5.1, if a model M is verified to be (CMP, RO, WOS) for all *unambiguous* 1-address and 2-address executions, then it is (CMP, RO, WOS) for *every* execution. Since every 1-address execution can be trivially treated as a 2-address execution, it is sufficient to verify that the model is (CMP, RO, WOS) for all 2-address executions only. For practical reasons, however, it is better to conduct the tests separately, as verification of the model for (CMP, RO, WOS) for all 2-address executions without first verifying for 1-address executions would be more complicated. This additional complexity leads to higher memory requirements. In other words, it is better to verify the model's conformance to (CMP, RO, WOS) first for 1-address executions and then use simpler tests to verify its conformance for 2-address executions. For this reason, Section 5.7.1.1 presents a complete automaton for verifying whether a given shared memory system implements (CMP, RO, WOS) when the execution has one address. Section 5.7.1.2 presents a complete test automaton for verifying whether a shared memory system implements (CMP, RO, WOS) when the execution has two addresses *assuming that the shared memory system correctly implements (CMP, RO, WOS) when the execution has only one address.*

These automatons are constructed by making the following observation. A concurrent execution C is allowed under (CMP, RO, WOS) if for each sequential execution X in C there is a sequence S_X such that:

RW1. S_X contains all instructions from X and all wr instructions from all other sequential executions,

RW2. each rd instruction in S_X returns the most recent value written for that address in S_X; if there is no such wr instruction, it returns $\top$, and

RW3. if i_1 appears before i_2 in some sequential execution and they also appear in S_X and both i_1 and i_2 are rd instructions or both are wr instructions, then they appear in that order in S_X.

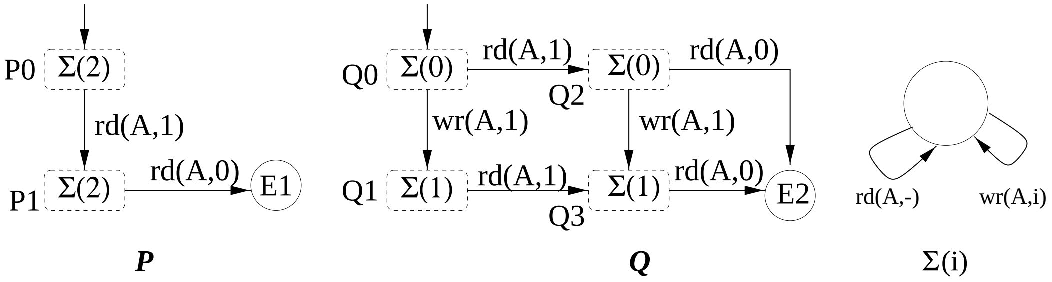
These three conditions are used in Sections 5.7.1.1 and 5.7.1.2 to enumerate all possible violations of (CMP, RO, WOS) and develop a *complete* test covering them. Note that condition RW3 is weaker than PRAM3 in that it does not constraint a rd instruction relative to a wr instruction in any manner.

5.7.1.1 One address test for (CMP, RO, WOS)

Figure 5.7(a) shows a *complete* test for (CMP, RO, WOS) using one address when M has two components. This figure uses $\Sigma(i)$ as a macro for writing i into A repeatedly with interspersed reads as shown in Figure 5.7(b) (the value returned by the rd instruction is ignored). Initial value of A is 0—not shown in the figure.

This automaton is interpreted as follows. Each state in the automaton has an associated action; for example P0 has the nondeterministic action $\Sigma(2)$ associated with it, and Q0 has the nondeterministic action $\Sigma(0)$ associated with it. When the automaton is in the state, it can perform the action associated with the state any number of times (including zero times). Each arc in the automaton also has an action associated with it; for example the action associated with the arc from P0 to P1 is rd(A,1), and the action associated with the arc from Q0 to Q1 is wr(A,1). The automaton may make the transition whenever it can execute the action. In other words, when Q is at Q0, it can write a 1 into A and move to Q1. Similarly, when P is in P0, if it reads a 1 from A, it can move to P1 (however, in this particular case, it need not move to P1, as $\Sigma(2)$ allows P to read any value and remain in P0).

It is easy to see that if P reaches E1 or Q reaches E2 then the model does not implement (CMP, RO, WOS). The argument below shows that if the model has any *1-address execution* violating (CMP, RO, WOS), then there is a run of the automatons such that either P reaches E1 or Q reaches E2. From the definitions of CMP, RO, and



(a) (b)

Figure 5.7. Complete test for (CMP, RO, WOS) using one address

WOS, if a component generates two writes, and they appear to have happened in the opposite order to itself or another component, then the model does not implement (CMP, RO, WOS). Q generates a sequence of writes to A, with every wr(A,1) coming after every wr(A,0). P also generates writes to A, but it always writes a value of 2, to indicate that the value is really a "don't-care." The purpose of these instructions is to obtain the side effects associated with the writes. It is a violation if the write instructions of Q appear in a different order than issued by Q to either P or Q. If P can see the effect of wr(A,1) first and then the effect of wr(A,0)—as witnessed by transitions rd(A,1) transition from P0 to P1 rd(A,0) from P1 to E1—then it reaches state E1, indicating an error. Similarly if Q can see the execute rd(A,1) followed by rd(A,0), it reaches E2, indicating an error. If P and Q are symmetric, then this verification is sufficient to guarantee (CMP, RO, WOS). If the two processes are not symmetric, then the verification must be repeated with the roles of P and Q reversed (i.e., P as the writer of 0 and 1 and Q as the writer of 2).

The above argument only shows that if the automata reaches E1 or E2, then the model violated (CMP, RO, WOS), but does not show that if there is a violation either E1 or E2 is reached. To see that it is indeed the case, we use abstraction as done in Figure 5.4. Let E be an unambiguous (concurrent) execution involving one address x and two sequential executions P_1 and P_2 that shows that the model does not implement (CMP, RO, WOS). In other words, either P_1 or P_2 cannot be serialized according to the constraints RW1–RW3. This can happen because of any of the following conditions:

RWv1. one of the sequential executions has a read rd(x,v) where v is never written and $v \neq \top$ (this would make it impossible to construct a sequence satisfying RW1 and RW2),

RWv2. one of the sequential executions has two writes w_1 and w_2 such that w_1 appears before w_2, but either the same execution or the other execution first reads the value written by w_2 and then reads the value written by w_1 (this would make it impossible to construct a sequence satisfying RW1–RW3), or,

RWv3. one of the sequential executions contains a write w_2 such that either itself or the other execution reads the value written by w_2 followed by the initial value $\top$ (this would make it impossible to construct a sequence satisfying RW1 and RW2).

To see that these are the only possible violations of the constraints, assume that the execution does not have any of the above violations. Then the following procedure can be used to construct a partial order $\subset_1$ such that any total order, $<_1$ consistent with $\subset_1$ can be used as a witness for the sequence satisfying the constraints RW1–RW3 for P_1. Then the roles of P_1 and P_2 can be reversed to construct a different partial order $\subset_2$ such that any total order, $<_2$, consistent with $\subset_2$ can be used as a witness for the sequence satisfying the constraints for P_2. The procedure starts with a partial order satisfying only RW3 and refines it so that it satisfies RW2 for one read instruction at a time until all read instructions are exhausted. The read instructions are considered in the order they appear in P_1. After a read instruction is considered by the procedure, the partial order can be used to construct a total order satisfying RW1–RW3 *up to* that read instruction. Hence when the procedure terminates, the resulting partial order can be used to define a total order $<_1$ satisfying the RW1–RW3 for *all* instructions.

1: Let R be the sequence of instructions obtained by projecting read instructions in P_1, W1 be the sequence of instructions obtained by projecting write instructions in P_1, W2 be the sequence of instructions obtained by projecting write instructions in P_2, and $\subset_1$ be the partial order that constrains two events e_1 and e_2 such that $e_1 \subset_1 e_2$ exactly when e_1 appears before e_2 in R, W1, or W2. Let n be the number of events in R.

2: For i in $1 \ldots n$ perform steps 2.1–2.3.

2.1: Let r be the i th instruction from R. Since R contains only rd instructions and the execution has only one address x, r must have the form rd(x, v) for some v.

2.2: Case 1, $v = \top$: Since the execution is assumed not to violate RWv3, there no r'=rd(x,v_1) in R such that $v_1 \neq \top$ and r' appears before r in R. Hence, by construction, there is no wr instruction w such that $w \subset_1 r$. $\subset_1$ is not modified in this step. $\subset_1$ now satisfies the constraints RW1–RW3 up to the ith instruction in R.

2.3: Case 2, $v \neq \top$: Since the execution is assumed not to violate RWv1, there is a w =wr(x,v) in W1 or W2. In addition, since the execution is unambiguous, there is exactly one such w. Assume that w is in W1 (the case when it is in W2 is similar). Since the execution does not violate RWv2, there is no rd instruction in R that appears *before* r (according to $\sqsubset_1$) but returns a value written by a wr instruction that appears *after* w (according to $\sqsubset_1$) in W1. Hence, by construction, $r \not\sqsubset_1 w$. In other words, $w \sqsubset_1 r$ can be added to $\sqsubset_1$ without introducing a cycle into the partial order $\sqsubset_1$. In this step, $\sqsubset_1$ is modified to include $w \sqsubset_1 r$. After this modification, $\sqsubset_1$ satisfies the constraints RW1–RW3 up to the ith instruction in R. □

At the end of this procedure, $\sqsubset_1$ can be used to construct a total order $<_1$ satisfying the constraints RW1–RW3. By reversing the roles of P_1 and P_2 in the above process a different partial order $<_2$ can be constructed as a witness for the total order satisfying RW1–RW3 for P_2. Hence all violations of (CMP, RO, WOS) are captured by RWv1–RWv3. A rd instruction returning a value that is neither $\top$ nor written by a wr instruction (violation RWv1) is trivial; hence the automatons in the rest of the chapter do not explicitly check for this possibility (the automatons, however, can be easily extended to check for this violation too).

To see that the automaton in Figure 5.7 is indeed a complete one address test for (CMP, RO, WOS) for two processes, without loss of generality, assume that the write instructions that lead to the violation of RWv2 or RWv3 belong to P_2. In other words, there are two read instructions r_1 and r_2 occurring in one of the sequential executions such that: (a) P_2 contains two writes w_1 followed by w_2, r_1 returns the value written by w_2 or a later instruction, and r_2 returns the value written by w_1 or an earlier instruction, or (b) P_2 contains a write instruction w_2, r_1 returns the value written by w_2 or a later instruction, and r_2 returns the initial value $\top$. It is easy to see that the following abstraction of the execution preserves the error.

- $\top$ is abstracted to 0,
- all data written by P_1 are abstracted to 2,
- all data written by P_2 until (but not including) w_2 are abstracted to 0, and
- all data written by P_2 from w_2 until the end are abstracted to 1.

This abstracted execution is covered by the test automata in Figure 5.7: x is A, P_1 is P, P_2 is Q, if r_1 and r_2 belong to P, then E1 is reached, and if r_1 and r_2 belong to Q, then E2 is reached.

Note that, in Figure 5.7, it is *not* an error for Q to be able to do rd(A,1) even before it does wr(A,1). This is because (CMP, RO, WOS) allows a rd to be reordered freely with respect to a wr even if they involve the same address. (Hence this formal memory model is not very interesting for practical systems and provided here only to help with the explanation of the tests for (CMP, POS) below.)

If the model consists of m addresses, then all the verification must be repeated with the address A being a different address in each verification. If the addresses are symmetric, of course, one verification is sufficient. Finally, to extend the test for n components, $n-1$ copies of P must be used with one copy of Q. If the components are not symmetric, then n verification runs must be conducted—in each run a different component acts as Q and other processes act as P.

5.7.1.2 Two address test for (CMP, RO, WOS)

Figure 5.8(a) shows a *complete* test for (CMP, RO, WOS) using two addresses when M has two components. The initial value of A and B is 0. In this figure $\Sigma(i,j)$ is used as an abbreviation for every sequence of instructions that writes i into A, writes j into B, and reads *any* value from A and B (i.e., the return value of the read instructions is immaterial), as shown in Figure 5.8(b).

As with the one address test of (CMP, RO, WOS), a violation occurs only when a component generates two writes and they appear to have happened in the opposite order to another component or itself. By a reasoning similar to the one used in Section 5.7.1.1, one can see that there are three ways in which this violation can be revealed:

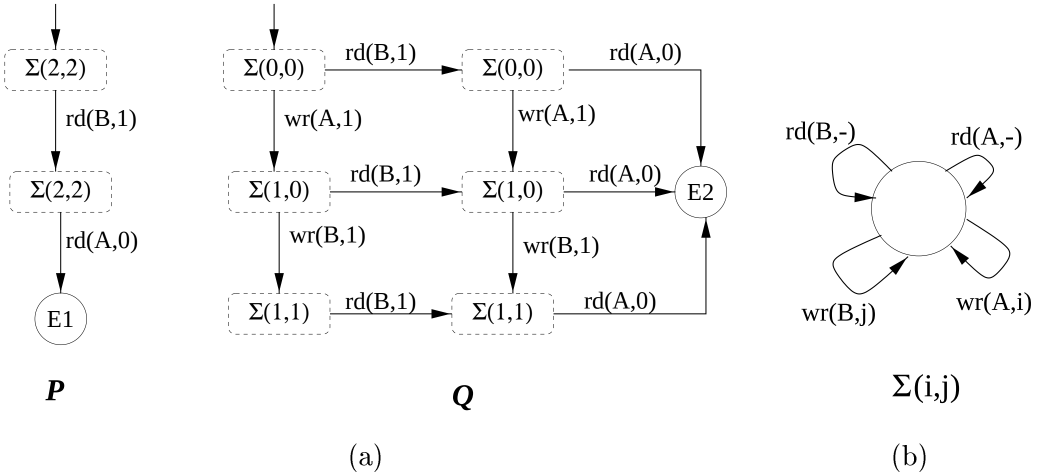


Figure 5.8. Complete test for (CMP, RO, WOS) using two addresses

Cv1. P or Q can do rd(A,1) followed by rd(A, 0). This is a violation of (CMP, RO, WOS) involving A only,

Cv2. P or Q can do rd(B,1) followed by rd(B, 0). This is a violation of (CMP, RO, WOS) involving B only,

Cv3. P or Q can do rd(B,1) followed by rd(A, 0). This is a violation of (CMP, RO, WOS) involving both A and B.

As can be seen, the first two violations, Cv1 and Cv2, involve only one address. These are not shown in Figure 5.8 to keep the figure simple. When the third violation occurs, P reaches E1 or Q reaches E2. If the model is first verified using the one address (CMP, RO, WOS) test in Figure 5.7, then it is not necessary to check for Cv1 and Cv2 above. Such simplifications, in general, reduce the state graph size.

By using an abstraction similar to the one used in Section 5.7.1.1, it can be shown that if there is an error in the model involving both A and B, then E1 or E2 is reached. As with the one address (CMP, RO, WOS) test, if the model has m addresses, the model must be verified for every possible pair of addresses, requiring a total of $m \times (m-1)/2$ runs. If the addresses are symmetric, of course, one verification is sufficient. To extend the test for n components, $n-1$ copies of P and one copy of Q can be used. If the components are not symmetric, then n verification runs must be conducted.

5.7.2 Verifying (CMP, POS)

From the definition of PRAM (Section 4.6), it is easy to see that PRAM and (CMP, POS) are identical. From Theorem 5.2, if a model M is verified to be (CMP, POS) for all 1-address and 2-address executions, then it is (CMP, POS) for all executions.

Figure 5.9 shows the complete one address test for (CMP, POS) when the model contains two components. The initial value of A is 0. If the model does not implement (CMP, POS), then E1 or E2 will be reached. This figure can be understood easily by observing that the difference between (CMP, RO, WOS) and (CMP, POS) is that the former allows a rd instruction to be reordered in every possible way with a wr instruction whereas the later does not. Thus the test can be obtained simply by adding more error transitions to (CMP, RO, WOS) test. Alternatively, one can also demonstrate that it is a complete test by using an abstraction similar to the one used in Section 5.7.1.1. Figure 5.10 shows the complete test using two addresses when the model contains two

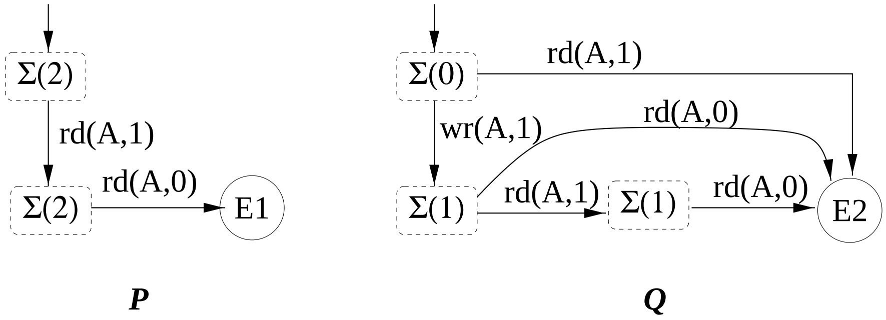


Figure 5.9. Complete test for (CMP, POS) using one address

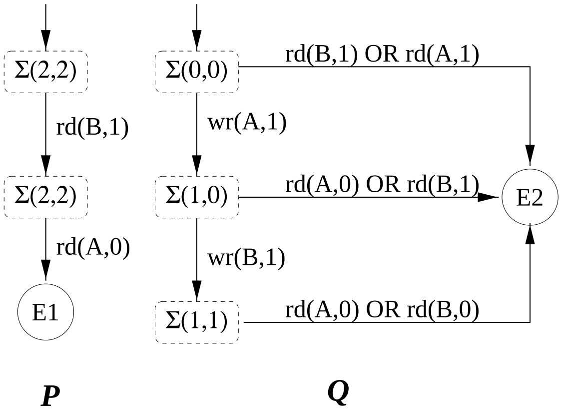


Figure 5.10. Complete test for (CMP, POS) using one addresses

components. Both A and B are initially 0.

As with the (CMP, RO, WOS) tests, if the model has m addresses, then the test model must be verified for every possible pair of addresses, requiring a total of m runs in the case of one address (CMP, POS) test and $m \times (m-1)/2$ runs in the case two address (CMP, POS) test. When the model has n components, $n-1$ copies of P can be used with 1 copy of Q. If the components are not symmetric, then n verification runs must be conducted.

5.7.3 Verifying (CMP, POS, WA)

From the definition of SC (Section 4.4), it is easy to see that SC is same as (CMP, POS, WA). From Theorem 5.3, if a model M has N components and is verified to be (CMP, POS, WA) for all $n \leq N$ address executions, then it is (CMP, POS, WA) for all programs. Figure 5.11 shows a complete test for (CMP, POS, WA) using one address. The initial value of A is 0.

Component P writes 0, 1, and 2 into A in that order. Component Q writes 3, 4, and

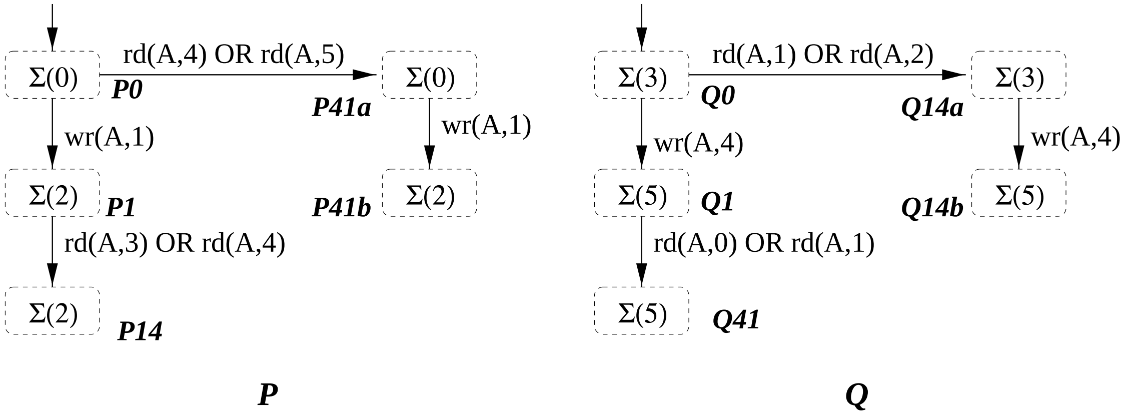


Figure 5.11. Complete test for (CMP, POS, WA) using one address

5 in that order into A. The initial value of A is 0. In addition, P writes 1 into A, and Q writes 4 into A exactly once. There is a WA related violation in the model if and only if one component sees the order of these writes as 1 followed by 4 while the other sees the order as 4 followed by 1. (If a component reads a value of 1 followed by 4 from A, then obviously, it saw 1 followed by 4. However, it may also see 1 followed by 4 indirectly in the following manner: if a read instruction for A returns 2—indicating that P has finished writing 1—and a later read instruction for A returns a value of 3 or 4. In this case also, the effect of writing of 1 into A is seen before the effect of writing of 4. Similarly, if Q reads a 1 for A before it writes 4 into A, then it saw 1 before 4. Obviously, a similar explanation holds when the component sees a 4 followed by 1.) More specifically, there is a (CMP, POS, WA) violation if and only if one of the following conditions is true:

SCv1. P or Q sees a value of 1 followed by 0, or 2 followed by 0 or 1 for A (this is a violation of (CMP, POS)),

SCv2. P or Q sees a value of 4 followed by 3, or 5 followed by 3 or 4 for A (this is a violation of (CMP, POS)),

SCv3. P sees a value of 1 or 2 before it writes the value (but not an error to see 0 before it writes 0, as 0 is the initial value of A). (this is a violation of (CMP, POS)),

SCv4. Q sees a value of 3, 4, or 5 before it writes the value (this is a violation of (CMP, POS)),

SCv5. P sees a 1 followed by 4 (i.e., it reaches the state P14) and Q sees 4 followed by 1 (i.e., it reaches Q41) (this is a violation of (CMP, POS, WA)), or

SCv6. P sees a 4 followed by 1 (i.e., it reaches P41a or P41b) and Q sees a 1 followed by 4 (i.e., it reaches Q14a or Q14b) (this is a violation of (CMP, POS, WA)).

As can be seen, the first four error conditions, SCv1–SCv4, do not involve WA. These four errors are not shown in Figure 5.11 to keep the figure simple. If the model is already verified to satisfy (CMP, POS)—for example by the automata in Figure 5.10—then it is not necessary to check for SCv1–SCv4.

The transitions of the automata and the last two error conditions are explained below. After writing 1 into A, P moves from P41a to P41b or from P0 to P1. In the first case, for P to reach P41a, it must make a transition from P0 to P41a—which occurs only if it sees a value of 4 or 5 before P writes a 1. Recall that Q writes 5 only after writing 4; hence P reading a value of 5 implies that it has also seen a value of 4 for P. Hence P moves from P0 to P41a only when it sees a value of 4 before it writes a value of 1. Similarly it moves from P0 to P1 and P1 to P14 if and only if it writes a value of 1 before seeing a value of 4. By a symmetric argument, Q reaches Q41 if and only if it writes a value of 4 before seeing a value of 1 and reaches Q14a or Q14b if and only if it sees a value of 1 before writing a value of 4. If one of the processes sees a value of 1 followed by 4 while the other sees a value of 4 followed by 1, then that execution shows a violation of (CMP, POS, WA), as summarized by the last two error conditions above.

The following argument essentially mirrors the argument in Section 5.7.1.1 to show that Figure 5.11 is indeed a complete test for (CMP, POS, WA). (CMP, POS, WA) is violated by a 1-address execution in a two component model if and only if one of the following conditions is true.

(a) there is a write instruction w_1 such that a read instruction returns the value written by w_1 and a later read instruction returns the value $\top$ (this is a violation of (CMP, POS)),

(b) there are two write instructions w_1 and w_2, two read instructions r_1 and r_2 such that w_1 and w_2 belong to the same component and occur in that order, r_1 and r_2 belong to the same component and occur in that order, r_1 returns the value written by w_1 or an earlier write instruction, and r_2 returns the value written by w_2 or a later instruction (this is a violation of (CMP, POS)), or

(c) there are two write instructions w_1 and w_2 in two different components, four read instructions r_1, r_2, r_3, and r_4 such that r_1 and r_2 belong to the same component and occur in that order, r_3 and r_4 belong to the same component and occur in that order, r_1 returns the value written by w_1 or an earlier write instruction (as given by POS), r_2 returns the value written by w_2 or a later write instruction (as given by POS), r_3 returns the value written by w_2 or an earlier write instruction (as given by POS), and r_4 returns a value written by w_1 or an earlier write instruction (as given by POS) (this is a violation of (CMP, POS, WA)).

As already mentioned, the figure shows only SCv5 and SCv6, which correspond to (c) above. The following abstraction shows how (c) can be converted into the violation SCv5 or SCv6.

- $\top$ is abstracted to 0,
- data written by all instructions occurring before w_1 in the program order are abstracted to 0,
- datum written by w_1 is abstracted to 1,
- data written by all instructions occurring after w_1 in the program order are abstracted to 2,
- data written by all instructions occurring before w_2 in the program order are abstracted to 3,
- datum written by w_2 is abstracted to 4,
- data written by all instructions occurring after w_2 in the program order are abstracted to 5.

Hence, the test in Figure 5.11 is a complete one address test for (CMP, POS, WA).

This test can be extended in a straightforward way to create a two address test, as shown in Figure 5.12. In this figure, both P and Q write the same value into A and B in each state. The initial value of A and B is 0. P writes 0 into both variables (in state P0) 0 or more times before writing 1 into A. Then it writes 2 into both variables. Similarly Q writes 3 into both variables (in state Q0) 0 or more times before writing 4 into B. Then Q writes 5 into both variables. (No process writes 1 into B or 4 into A, but rd(B,1) and

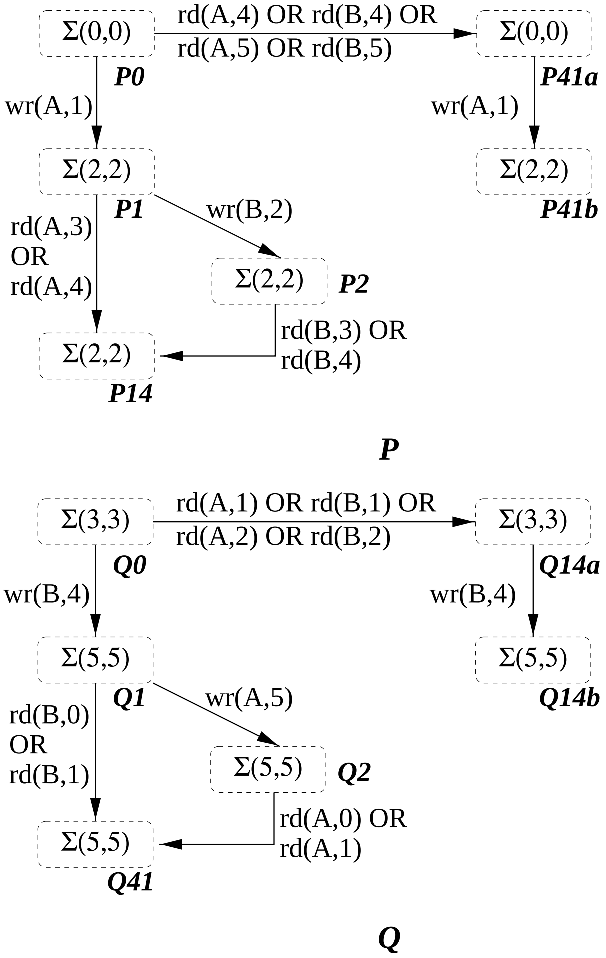


Figure 5.12. Complete test for (CMP, POS, WA) using two addresses

rd(A,4) appear in the figure to keep it symmetric.) If there is a (CMP, POS, WA) error involving both variables, then (a) P will reach P14 and Q will reach Q41 or (b) P will reach P14a or P14b and Q will reach Q14a or Q14b. By a reasoning similar to the one used in one address (CMP, POS, WA) test, it can be seen that (a) P reaches P14 if and only if it sees the effect of wr(A,1) before wr(B,4), (b) P reaches P41a or P41b if and only if it sees the effect of wr(B,4) before wr(A,1), (c) Q reaches Q14a or Q14b if and only if it sees the effect of wr(A,1) before wr(B,4), and (d) Q reaches Q41 if and only if it sees the effect of wr(B,4) before wr(A,1). Hence it is an error for P to reach P14 and Q to reach Q41, or for P to reach P41a or P41b and Q to reach Q14a or Q41b.

5.7.4 Application of complete tests to the Runway model

The tests above are applied to the Promela model of HP/Runway system. The results are summarized in Table 5.5.

Rows ROWO-1, ROWO-2, PO-1, PO-2, SC-1 are the tests corresponding to the automata shown in Figures 5.7, 5.8, 5.9, 5.10, and 5.11 respectively. Rows SC-2a and SC-2b together are equivalent to the automaton shown in Figure 5.12. The automaton corresponding to the row SC-2a tests for the possibility that P reaches P14 and Q reaches Q41; i.e., this test contains the states P0, P1, P2, P14, Q0, Q1, Q2, and Q41, but does not contain the states P41a, P41b, Q14a, and Q14b. The automaton corresponding to the row SC-2b tests for the possibility that P reaches P41a or P41b, and Q reaches Q14a or Q14b; i.e., this test contains states P0, P41a, P41b, Q0, Q14a, and Q14b, but does not contain the states P1, P2, P14, Q1, Q2, and Q41. On PO-2, SC-2a, and SC-2b, Spin did not finish verification in 512MB memory; it aborted the search after generating a large number of states, as shown in the table. On these tests, hence, the time taken for

Table 5.5. Application of complete tests to Runway memory model

Test	Spin		PV	
	states	runtime (sec)	states	runtime (sec)
PO-1	56449	4	2412	0.4
PO-2	>6e+06	—	2.85e+06	745
SC-1	499424	76	7880	12
SC-2a	>6e+06	—	5.97e+06	720
SC-2b	>4e+06	—	574,293	177

verification is shown as “—.”

5.8 Concluding Remarks

The chapter presented the basic ideas behind test model checking and how the technique can be made complete if the model under consideration is *projectable* and *data independent*. Appendix A presents a modeling language where all models expressible in the language satisfy these two conditions. Appendix B proves Theorems 5.1–5.3.

CHAPTER 6

CONCLUDING REMARKS AND FUTURE DIRECTIONS

6.1 Contributions

The dissertation showed that specializing formal methods for a particular domain leads to *efficient* verification techniques applicable to the designs arising in the domain, as well as increases the *applicability* of formal methods by making the following contributions applicable in the area of shared memory system design.

- A new partial order reduction called *two phase* is presented. This algorithm typically generates far fewer states than comparable algorithms and is especially effective in memory protocols. The algorithm is shown to preserve stutter free linear temporal logic formulae and has been implemented in an explicit state enumeration based model checker called Protocol Verifier (PV). The algorithm also supports selective caching—something not supported by other tools implementing partial order reductions.

- A design derivation algorithm, targeted to the design of distributed shared memory protocols is presented. This algorithm accepts a high-level specification and produces an equivalent detailed implementation. Since the high-level specification and the detailed implementation are equivalent, it is sufficient to verify the high-level specification, which can be orders of magnitude more efficient than verifying the implementation. The algorithm is shown to be correct using an automated theorem prover.

- A technique called test model checking for verifying a shared memory systems conformance to a formal memory model is presented. This technique is an adaptation of testing methods into the realm of model checking. By combining the two techniques, test model checking effectively eliminates the disadvantages of both techniques. On one hand, testing methods suffer from the fact that not all scheduling of events are executed, but model checking eliminates this problem. On the other hand, the logic supported by model checkers is not powerful enough to express that a model conforms to, say, sequential consistency, but testing approach eliminates this problem.

6.2 Extensions

The techniques presented in this dissertation can be extended in a number of ways to further improve their efficiency.

6.2.1 Partial order reductions

The two phase algorithm follows a heuristic that typically, but not always, generates smaller graph than the proviso based partial order reduction algorithms. The algorithm can be modified in the following way such that it never generates a graph larger than the graph generated by the proviso based algorithms. The proviso based algorithms behave as follows: if t is an enabled transition in s, $t(s)$ is in stack when s is expanded, and the set of transitions chosen by the algorithm includes t, then the algorithms expand s fully. If the algorithms are modified to expand $t(s)$ fully instead, the resulting graph would be smaller. However, it is not clear how such an algorithm can be combined with selective caching and on-the-fly model checking algorithms. This may require formulating different selective caching algorithms and on-the-fly model checking algorithms.

The two phase shows that more efficient heuristics can be found for a given domain. By examining other domains, heuristics that perform better in those domains may be found.

6.2.2 Protocol synthesis

All communication actions in the protocol generated by the synthesis algorithm are between home and a remote node. In other words, two remote nodes never communicate. This restriction can be relaxed to further improve the efficiency of the protocol. However, this must be done carefully so as to avoid unintentionally modifying the memory model provided by the protocol.

Another possible extension is to relax the syntactic constraints placed on the structure of the remote node. Of course, one must keep in mind that allowing a fewer syntactic restrictions lead to inefficient algorithms; hence care must be taken to relax the constraints only when the loss of efficiency is small.

One could also compare the performance of a hand-coded protocol with that of the protocol generated by the synthesis algorithm. We attempted to perform such comparisons, but due to practical limitations, this work was never completed.

6.2.3 Memory model verification

There is a growing interest in relaxed formal memory models such as total store ordering (TSO) and partial store ordering (PSO) [88]. The test model checking approach can be extended to handle such models by defining new ordering and atomicity rules [31], as well as to other domains such as databases and compilers.

The test program to verify if a model implements sequential consistency uses n addresses, where n is the number of components in the model. If the model under consideration is scalable, i.e., n is not a constant, but rather a parameter, then the test program can help debugging the model for one value of n at a time, but cannot provide absolute guaranty for an unbounded n. In restricted domains, one may can find symmetry arguments as in [29] to relax this limitation.

6.3 The Future of Formal Verification

The size and complexity of the concurrent systems are growing much faster than the size of the systems that formal methods can handle. If the verification tools do not address this growing imbalance, soon it would be impossible to formally verify most, if not all, state-of-the-art concurrent systems in any sufficient detail. There are two natural (and orthogonal) approaches to addressing the problem: more efficient algorithms and refinement.

6.3.1 More efficient algorithms

Given that the model checking problem is P-SPACE complete in the size of the description, our best hope is a set of heuristics that work well in a limited domain. The results in this dissertation demonstrate that formulating domain specific formal verification techniques can lead to efficient algorithms, which can help verifying larger state spaces. The disadvantage of such domain specific algorithms (limited applicability) is, as the results in this dissertation indicate, is an acceptable price to pay for the improved efficiency of the resulting algorithms.

6.3.2 Refinement

Refinement reduces the complexity of the verification because a high-level model of the concurrent system is first verified. Then the individual components in the system can be transformed into an implementation either by algorithms that are known to preserve the semantics or by hand. If the components are transformed by hand, then one may also verify that the implementation component indeed truly represents the corresponding component in the high-level model (in this sense, refinement is same as the more familiar divide and conquer approach).

Both approaches have their own strengths and weaknesses. Domain specific algorithms can improve the effectiveness of the verification tools by increasing the size of the problem they can handle. However, as mentioned before, model checking problem is P-SPACE complete; hence its effectiveness would always be limited. On the other hand, refinement, when applicable, can handle much larger problems. The disadvantages of the refinement are inefficient implementations (when the refinement is carried out by automatic methods) and intense labor (when the refinement is carried out by hand, and hence need to be proved).

Despite the pessimism echoed above, there is reason to be optimistic about the future of formal verification. As the size and complexity of the concurrent systems are increasing, traditional simulation methods are increasingly unable to cover the design to provide sufficient confidence. As a result, the computer industry is increasingly spending larger amount of resources on simulation. Although formal verification may not be able to handle all the entire design, one can apply it in strategic locations. Examples include ordering constraints (test model checking of Chapter 5), arithmetic circuits (which can be handled well by BDDs), and high-level coherency protocols (using partial order reductions and possibly refinement).

APPENDIX A

FORMAL MODEL OF A SHARED MEMORY SYSTEM

As discussed in Section 5.5.1, realistic memory systems do not compare data for control purposes, and address comparison is done only in caches and memories. These notions, called *data independence* and *projection* respectively, are formalized with the help of the following formal model of a component.

The appendix organized as follows. Section A.1 formalizes the notion of memory system of a uniprocessor, called *component.* Section A.2 formalizes the notion of memory system of a multiprocessor, called *shared memory system.* Section A.3 formalizes the notions of concurrent program, and execution. Sections A.4 and A.5 formalize the architectural rules introduced in Chapter 5, namely rule of computation (CMP), read order (RO), write order by storage (WOS), program order by storage (WOS), and write atomicity (WA). Using these notions, Sections A.6 and A.7 show that all shared memory systems as defined in Section A.2 are projectable and data independent. Finally, Section A.8 concludes the appendix.

A.1 Component

Intuitively, a component implements either main memory or memory subsystem of one processor. Two or more components can be connected using a communication network to form a multiprocessor. Structure of a component (without interconnection network) is shown in Figure A.1. As shown, a component contains a protocol, working information, cache state, and data information. This protocol is *parameterized* on both address and data value; i.e., the protocol may not compare the address or data value *directly.* In other words, the transitions of the protocol, for example, may not contain expressions of the form *"data>d"* and *"addr==a"* where a is address and d is a datum. `addr` and `data` in the working info in the figure show the current address and the data that the protocol is processing at any given time. The component also contains a table, shown as "cache

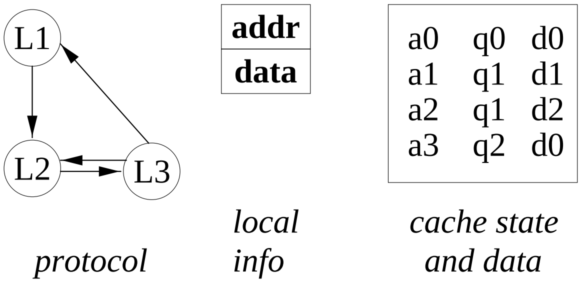


Figure A.1. Structure of a component

state and data." This table contains the cache state for each address and data value for each address.[1] (The table is read as address a_0 has a state of q_0 and a data of d_0, a_1 has a state of q_1 and a data of d_1, etc.) The protocol can use cache state of a given address to make decisions. In other words, the transitions of the protocol may contain, for example, the expression *"cache[addr]*$\neq q_0$*,"* where *cache* is the "cache state and data" table shown in the figure. One or more components can be interconnected together by channels (not shown in the figure).

Formally, a component P is a tuple ($\mathcal{C}$, $\mathcal{Q}$, $\mathcal{A}' = \mathcal{A} \cup \{\bot\}$, $\mathcal{D}' = \mathcal{D} \cup \{\top\}$, T, $\mathcal{M}$, c_0, c_1, $\ldots$ c_k, c_{k+1}, z_i, q_i), where

- $\mathcal{C}$ is a finite set of states of the protocol ("control states"),
- $\mathcal{Q}$ is a finite set of states ("states of a given cache-line") (see the description of local state below),
- $\mathcal{A}'$ is an infinite set of addresses with a special symbol $\bot$,
- $\mathcal{D}'$ is an infinite set of data values with a special symbol $\top$,
- T is a finite set of transitions of the protocol (described below),
- $\mathcal{M}$ is a finite set of message types with two special symbols, rd and wr,
- c_0 is an input channel through which P can respond to external events such as read and write commands. Each entry in this channel is of the form (rd, a, $\bot$) or (wr, a, d), where $a \in \mathcal{A}$, and $d \in \mathcal{D}$. If the component is main memory, in realistic

---

[1] Main memory, typically, does not maintain any cache state. Such a situation can be trivially treated, for example, by assuming that cache state of all addresses is a fixed known value.

descriptions, then it would not interpret or consume any entries from this channel. Otherwise, it would interpret the entry (rd, a, $\perp$) indicates an the instruction rd(a) and interpret the entry (wr, a, d) as the instruction of wr(a,d).

- c_{k+1} is an output channel that shows the value returned as a result of rd. Each entry in this channel is of the form (rd, a, d), where $a \in \mathcal{A}$ and $d \in \mathcal{D}^{'}$, indicating that rd(a) has returned the data value d. If the component is main memory, in realistic descriptions, then c_{k+1} would remain empty,

- $c_1 \ldots c_k$ are a set of channels that P can use to communicate with other components. Entries in these channels are of the form $(m, a, d) \in \mathcal{M} \times \mathcal{A} \times \mathcal{D}^{'}$,

- $z_i \in \mathcal{C}$ is the initial control state of the system, and

- $(q_i, \top) \in \mathcal{Q} \times \mathcal{D}^{'}$ is the initial state of the cache for each address $a \in \mathcal{A}$.

Note that it is not mentioned whether the channels c_i have finite or infinite capacity. The theorems proved in this appendix hold in all such cases.

A.1.1 Local state

The local state of a component consists of the state that is not visible to other component, i.e., it excludes the channel contents. It has the form $(l, F, a, d) \in \mathcal{C} \times (\mathcal{A} \rightarrow \mathcal{Q} \times \mathcal{D}) \times \mathcal{A}^{'} \times \mathcal{D}^{'}$. Intuitively, the first element, $l \in \mathcal{C}$, is the control state of the protocol. The second element, $F \in (\mathcal{A} \rightarrow (\mathcal{Q} \times \mathcal{D}))$, is the contents of its cache. The third element, $a \in \mathcal{A}^{'}$, is the address on which the component is operating (if it is not operating on any address, then this would be $\perp$). The fourth element, $d \in \mathcal{D}^{'}$, is the data that the component is currently operating on (if not operating on any data, this would be $\top$). The initial local state of the component, of course, is $(z_i, F_i, \perp, \top)$, where $F_i(a) = (q_i, \top)$ for each $a \in \mathcal{A}$.

A.1.2 Transitions

Transitions in T are divided into three classes: (a) *send* transitions, (b) *recv* transitions, and (c) *internal* transitions.

A.1.2.1 Send transitions

Intuitively, a component can send a message of the form $(m,\ a,\ d)$ on a channel only if the component is currently processing the address $a \neq \perp$ with data d. However, a send transition is parameterized by the address and data in the sense that the component *cannot* make a data comparison or an address comparison to decide whether to send a message or not—it is solely determined by the control point and the state of the cache line.

More formally, a send transition has the form $i : (l_1, q) \rightarrow (l_2, m)$, where $l_1, l_2 \in \mathcal{C}$, $m \in \mathcal{M}$, $q \in \mathcal{Q}$, and $i \in \{1 \ldots k+1\}$.[2] Such a transition is enabled when the local state of the component matches the pattern $(l_1,\ F,\ a,\ d)$, where $F(a) = (q, d^{'})$ for some $d^{'}$. From this state, the component can send a message $(m,\ a,\ d)$ on c_i and then move to state (l_2, F, a, d). The contents of c_i are appropriately modified to indicate the new message. If the message is sent on the output channel c_{k+1}, them m is rd.

A.1.2.2 Receive transitions

Intuitively, a component can receive a message for an address a only when it is either currently processing that address (i.e., local state has the form (l_1, F, a, d_1)) or it is currently not processing any address (i.e., local state has the form $(l_1, F, \mathit{bottom}, d_1)$). When the message is received, the data component and the program counter component of the local state may be modified. As with the send transitions, receive transitions are also parameterized.

Formally, a receive transition has the form $i : (l_1, q, m) \rightarrow l_2$, where $i \in \{0 \ldots k\}$, $l_1, l_2 \in \mathcal{C}$, $q \in \mathcal{Q}$, and $m \in \mathcal{M}$. Such a transition is enabled if the current local state of the component matches the pattern (l_1, F, a, d_1) or $(l_1, F, \perp, d_1)$ where $F(a) = (q, d^{'})$ for some $d^{'}$ and the head of c_i contains a message that matches pattern (m, a, d). From this state, the component can execute the transition by removing the message from c_i and going to state (l_2, F, a, d_2) where d_2 is d if $d \neq \top$, d_1 otherwise; i.e., when the component receives a message $(m, a, \top)$, it does not overwrite the data component in the local state.

---

[2] The idea behind leaving out the address and data components in the transition is to make it *generic*; i.e., the transition is applicable irrespective of the exact value of the address or the data.

A.1.2.3 Internal transitions

There are two internal transitions: *reset* and *copy*. Intuitively, a reset transition clears the address and data elements of the local state and resets the state back to the initial state. *reset* transitions are also parameterized.

Formally, a reset transition has the form $l_1 \to z_i$ (recall that z_i is the initial control state). Such a transition is enabled if the current state of the component matches the pattern (l_1, F, a, d) where $a \neq \perp$. Executing the reset transition from this state results in the local state $(z_i, F, \perp, \top)$.

A *copy* transition can be used to read/write the cache state and the data of a given line. *copy* transitions are also parameterized.

Formally, a copy transition has the form $(l_1,\ q_1) \to (l_2, q_2, i)$, where $l_1, l_2 \in \mathcal{C}$, $q_1, q_2 \in \mathcal{Q}$, and $i \in \{0,\ 1,\ 2\}$. If i is 0, then the data in the cache for a are updated with the data component of the local state. If i is 1, then the data in the cache for a are copied into the data component of the local state. If i is 2, then neither the data in the cache for a nor the data component of the local state are changed. In all three cases, the cache state for address a becomes q_2. In other words, the transition is enabled in a local state that matches the pattern (l_1, F, a, d_1) where $a \neq \perp$ and $F(a) = (q_1, d^{'})$ for some $d^{'}$. Executing the transition from this state results in $(l_2, F^{'}, a, d_2)$ where (a) if $i = 0$, $F^{'}$ is exactly like F except $F^{'}(a) = (q_2, d_1)$ and $d_2 = d_1$, (b) if $i = 1$, $F^{'}$ is exactly like F except $F^{'}(a) = (q_2, d^{'})$ and $d_2 = d^{'}$, and (c) if $i = 2$, $F^{'}$ is exactly like F except $F^{'}(a) = (q_2, d^{'})$ and $d_2 = d_1$.

A.2 Shared Memory System

A shared memory system M connects a set of components using communication channels such that each component has a dedicated input channel and a dedicated output channel (i.e., input and output are not shared). Formally, a shared memory system consists of a set of components $P_1 \ldots P_N$, a set of inter-connection channels $c_1 \ldots c_n$, a set of input channels $i_1 \ldots i_N$, and a set of output channels $o_1 \ldots o_N$. Input and output channels of a component P_j are i_j and o_j respectively. All other channels of the components are in the set $c_1 \ldots c_n$. The set of transitions of M consists of transitions of all components, and the state of M consists of local states of each component as well as the state of channels.

A.3 Program and Execution

A component is *open* in the sense that it can accept any sequence of rd and wr messages on its input channel.

A.3.1 Sequential program

A sequential program is a finite sequence of *instructions* where each element of the sequence is of them (rd, a, $\perp$) or (wr, a, d) where $a \in A$ and $d \in D$.

A.3.2 Concurrent program

A concurrent program I is an ordered set of sequential programs $\{I_1 \dots I_N\}$ that is intended to run on a shared memory system with N components.

A.3.3 Closed model

If M is a shared memory system with components $P_1 \dots P_N$ and $I_1 \dots I_N$ are sequential programs, then the open system M can be *closed* by feeding I_j as input to the P_j.

A.3.4 Execution

Let M be a shared memory system with N components and $I = \{I_1 \dots I_N\}$ be a concurrent program. The ordered set $O = \{O_1 \dots O_N\}$ is an execution of M on I if each sequence O_j can be observed at the channel o_j when M is closed with I.

A.4 Events

Each instruction $\mathcal{I}$ in I is represented by one event if it is a read instruction or N events if it is a write instruction. The intuitive meaning of an event is that if $\mathcal{I}$ is a read instruction, the event represents the time at which the data are read. If $\mathcal{I}$ is a write instruction, each event represents the time at which the effect of the instruction is visible to each component.

Let $\mathcal{I}$ be the kth instruction in the sequential program I_j. If $\mathcal{I}$ is a write instruction (wr, a, d), it generates N events of the form $e_i = \mathrm{wr}_k^j(i,\ a,\ d)$ for $1 \le i \le N$. e_i is said to be observed by the ith component. As mentioned before, the event e_i indicates the "time" at which the ith component observed the write instruction.

If $\mathcal{I}$ is a read instruction (rd, $a, \top$) then it generates the event $e = \mathrm{rd}_k^j(j,\ a,\ d)$ where d represents the value returned by the read instruction.[3] e is said to be *observed* by

---

[3] If $\mathcal{I}$ is the kth read instruction in I_j then d is found by looking for the kth message in O_j that is of

jth component (the same component to which the read instruction belongs) and is said to return d. Intuitively, the event e indicates the "time" at which the read action is performed.

In the rest of section, $S = \{s_1, \dots, s_n\}$ is assumed to be set of all events generated by the execution M on I. The ordering rules and the atomicity rule can be defined with the help of these events as follows.

A.5 Graph Theory

An event graph (or simply graph) is $G{=}(V, <, =)$ where V is a set of events, $< \subseteq V \times V$ is an ordering on V, and $=$ is an equivalence relation on V. It is customary to represent $(a, b) \in <$ as $a < b$ and $(a, b) \in =$ as $a = b$. G is said to contain a circuit if and only if there is a set of events $\{e_1 \dots e_n\}$ in V such that for each $1 \leq i \leq n$, $e_i < e_{i \bmod n+1}$ or $e_i = e_{i \bmod n+1}$, and there is at least one $1 \leq j \leq n$ such that $e_j < e_{j \bmod n+1}$. Note that requiring at least one $<$ in the loop eliminates spurious cycles of the form $e_1 = e_2 = e_1$. G is said to be closed under transitivity if and only if for each e_1, e_2, e_3 in V (a) if $e_1 < e_2$ and $e_2 < e_3$ then $e_1 < e_3$, (b) if $e_1 < e_2$ and $e_2 = e_3$ then $e_1 < e_3$, and (c) if $e_1 < e_2$ and $e_1 = e_3$ then $e_3 < e_2$.

The intuitive idea behind a graph is that it represents restrictions on the events. For example, if an architectural rule, say RO (defined below), requires that e_1 must happen before e_2, then graph defined by the rule, $<_{ro}$ will require that $e_1 <_{ro} e_2$.

Let $G_1 = (V_1, <_1, =_1)$ and $G_2 = (V_2, <_2, =_2)$ are two graphs, the composition of the two graphs $G = G_1 * G_2$ is defined as $(V, E) = (V_1 \cup V_2, <_1 \cup <_2, =_1 \cup =_2)$. In other words, every edge in G is either an edge in G_1 or an edge in G_2. If $G^1 = \{G_1 \dots G_n\}$ and $H^1 = \{H_1, \dots, H_m\}$ be two sets of graphs. The definition of composition $*$ operator is extended to graph sets $G^1 * H^1$ as the set $M = \{M_{11}, M_{12}, \dots M_{1m}, M_{21} \dots M_{2m} \dots M_{n1} \dots M_{nm}\}$ where M_{ij} is $G_i * H_j$. Note that $G_1 * G_2 = G_2 * G_1$ and $G_1 * G_1 = G_1$.

Intuitively, $*$ operator represents a union of conditions. For example, if G_r represents the constraints imposed under the rule RO and G_w represents the constraints imposed under WOS, then $G_r * G_w$ represents all the constraints of both RO and WOS.

---

the form (rd, a, d)–d represents the value returned by $\mathcal{I}$.

A.5.1 Linearization of events

If a graph $G = (E, <, =)$ does not contain any circuits, then all events in it can be arranged in a sequence S such that if e_1 occurs before e_2 for two events in S, then $\neg(e_2 <^+ e_1)$ where $<^+$ represents the transitive closure of $<$ under the equivalence relation $=$. The sequence S is said to be consistent with G.

A.5.2 Circuits in a graph set

Each ordering rule, CMP, RO, WOS, and POS defined below, generates a graph or a graph set on the events in the execution. As explained before, the graph set generated by a rule r defines the set of possible explanations for that rule r. For each rule r, let $A(r)$ represent the set of graphs generated by the rule r. If $R = \{r_1 \ldots r_n\}$ is a set of rules, then define $A(R)$ to be $A(r_1) * \ldots A(r_n)$. If every element of $A(R)$ contains a circuit, then R is said to be violated. Intuitively presence of such circuits mean that at least one of the rules in $r_1 \ldots r_n$ is not implemented by the memory system M.

A.5.3 Rule of computation (CMP)

The rule of computation or CMP requires that all rd and wr events observed by a given component for a given address can be linearized such that value returned by each rd event is same as the value written by the most recent wr event. If there is no such wr event, then the value returned by the rd event is $\top$. An example of a sequence satisfying CMP for address a and component 1 is shown in Figure A.2(a).

Formally, let $S = \{s(1), \ldots, s(n)\}$ be a set of events by an execution. $G = (P, <^{a,i}_{cmp}, \phi)$ is a graph induced by CMP on S for address a and ith component if

- $P \subseteq S = \{p(1), \ldots, p(m)\}$ contains all events of S observed by i for the address a,
- for each $1 \leq j < m$, $p(j) <^{a,i}_{cmp} p(j+1)$,
- for each $1 \leq j \leq m$ if $p(j)$ is a rd event of the form $p(j) = \mathrm{rd}^i_k(i,\ a,\ \perp)$, then there is no wr event in $p(1) \ldots p(j-1)$, and
- for each $1 \leq j \leq m$ if $p(j)$ is a rd event of the form $p(j) = \mathrm{rd}^i_k(i,\ a,\ d \neq \perp)$, then the last wr event in $p(1) \ldots p(j-1)$ is $\mathrm{wr}^t_l(i,\ a,\ d)$ for some l and t.

Note that $<^{a,i}_{cmp}$ may define more than one graph G. For example, in Figure A.2(a) shows the graph L1 $<^{ai}_{cmp}$L2 $<^{ai}_{cmp}$L3 $<^{ai}_{cmp}$L4 $<^{ai}_{cmp}$L5. However, the graph L1 $<^{ai}_{cmp}$L2

$<^{ai}_{cmp}$L3 $<^{ai}_{cmp}$L5 $<^{ai}_{cmp}$L4 also satisfies the definition. Theorem B.2 in Appendix B shows how $<^{ai}_{cmp}$can be effectively treated as though it defines a single graph.

Note that the above definition of CMP is one address and one component only. This definition can be trivially extended to all addresses and components as composition of all graphs generated for one address and one component at a time; i.e., $<_{cmp} = *_a *_i <^{a,i}_{cmp}$.

A.5.4 Read order (RO)

The read order states that if two read instructions $\mathcal{I}_i$ and $\mathcal{I}_j$ appear in that order in some sequential program, then the event corresponding to $\mathcal{I}_i$ must appear before the event corresponding to $\mathcal{I}_j$ in a linearization. An example of RO is shown in Figure A.2(b) when the observer is component 1. Note that the subscripts for rd, which indicates the position at which the instruction occurred, must increase monotonically in the sequence.

Formally, let $S = \{s(1), \ldots, s(n)\}$ be a set of events generated by an execution. $G = (P, <^{i}_{ro}, \phi)$ is a graph induced by RO on S for ith component if

- $P \subseteq S = \{p(1), \ldots, p(m)\}$ contains all rd events observed by i,
- for each $1 \leq j < k \leq m$, $p(j) <^{i}_{ro} p(k)$ and the instruction to which $p(j)$ belongs to occurs before the instruction to which $p(k)$ belongs to; i.e., $p(j) = \mathrm{rd}^{i}_{l1}(i, a, d_1)$ and $p(j) = \mathrm{rd}^{i}_{l2}(i, b, d_2)$ for some a, b, d_1, and d_2 such that $l1 < l2$.

Unlike $<^{ai}_{cmp}$ which defines a set of graphs, $<^{i}_{ro}$ defines a unique graph. As with $<_{cmp}$, $<_{ro}$ is defined as composition of all $<^{i}_{ro}$ graphs; i.e., $<_{ro} = *_i <^{i}_{ro}$.

A.5.5 Write order by storage(WOS)

The write order by storage states that if two write instructions $\mathcal{I}_i$ and $\mathcal{I}_j$ appear in that order in some sequential program, then the no component may see the event corresponding to $\mathcal{I}_j$ before it sees the event corresponding to $\mathcal{I}_i$. An example of WOS is shown in Figure A.2(c) when the observer is component 1 and writes are done by component 2. Note that the subscripts for wr, which indicate the position at which the corresponding instruction occurred in the component 2, must increase monotonically when they are generated by the same component.

Formally, let $S = \{s(1), \ldots, s(n)\}$ be a set of events generated by an execution. $G = (P, <^{i,t}_{wos}, \phi)$ is a graph induced by WOS on S with the observer i and generator t if

L1: $\mathtt{rd}_3^1(1,\ a,\ \bot)$;
L2: $\mathtt{wr}_7^2(1,\ a,\ 1)$;
L3: $\mathtt{wr}_4^2(1,\ a,\ 2)$;
L4: $\mathtt{rd}_1^1(1,\ a,\ 2)$;
L5: $\mathtt{rd}_2^1(1,\ a,\ 2)$;

(a) CMP

L1: $\mathtt{rd}_3^1(1,\ a,\ 1)$;
L2: $\mathtt{rd}_5^1(1,\ b,\ \bot)$;
L3: $\mathtt{rd}_6^1(1,\ a,\ \bot)$;

(b) RO

L1: $\mathtt{wr}_3^2(1,\ a,\ 1)$;
L2: $\mathtt{wr}_4^2(1,\ b,\ 7)$;
L2: $\mathtt{wr}_6^2(1,\ a,\ 1)$;

(c) WOS

Figure A.2. Example executions of CMP, RO, and WOS

- $P \subseteq S = \{p(1), \ldots, p(m)\}$ contains all wr events observed by ith component and generated by tth component, and
- for each $1 \leq j < k \leq m$, $p(j) <_{wos}^{i,t} p(k)$ and the instruction for which $p(j)$ belongs to occurs before $p(k)$; i.e., $p(j) = \mathrm{wr}_{l1}^t(i,\ a,\ d_1)$, and $p(k) = \mathrm{wr}_{l2}^t(i,\ b,\ d_2)$ for some a, b, d_1, and d_2 such that $l1 < l2$.

As with $<_{ro}^i$, $<_{wos}^{i,t}$ defines a unique graph. $<_{wos}$ is defined as composition of all $<_{wos}^{i,t}$ graphs in the system; i.e., $<_{wos} = *_i *_t <_{wos}^{i,t}$.

A.5.6 Program order by storage (POS)

The program order by storage states that if two instructions $\mathcal{I}_i$ and $\mathcal{I}_j$ appear in that order in some sequential program, then the no component may see the event corresponding to $\mathcal{I}_j$ before it sees the event corresponding to $\mathcal{I}_i$ in a linearization. Note that if either of the two instructions is a read, then only one component would see the event corresponding to that instruction.

Formally, let $S = \{s(1), \ldots, s(n)\}$ be a set of events generated by an execution. $G = (P, <_{pos}^{i,t}, \phi)$ is a graph induced by POS on S with the observer i, and generator t if

- $P \subseteq S = \{p(1), \ldots, p(m)\}$ contains all events observed by ith component and generated by tth component, and
- for each $1 \leq j < k \leq m$, $p(j) <_{pos}^{i,t} p(k)$ and the instruction for which $p(j)$ belongs to occurs before $p(k)$; i.e., $p(j) = \mathrm{rd/wr}_{l1}^t(i,\ a,\ d_1)$, and $p(j) = \mathrm{rd/wr}_{l2}^t(i,\ b,\ d_2)$ for some a, b, d_1, and d_2 such that $l1 < l2$, where each rd/wr indicates that event is either rd or wr.

As with $<_{ro}^i$ and $<_{wos}^{i,t}$, the graph defined by $<_{pos}^{i,t}$ is unique. $<_{pos}$ is defined as composition of all $<_{pos}^{i,t}$ graphs in the system; i.e., $<_{pos} = *_i *_t <_{pos}^{i,t}$.

Note A.1 $<_{pos}$ implies both $<_{ro}$ and $<_{wos}$, but not vice versa. In particular, $<_{pos}$ constrains a read and a write occurring in a given execution, whereas the ($<_{ro}$, $<_{wos}$) combination does not.

A.5.7 Write atomicity (WA)

The write atomicity rule states that if $\mathcal{I}$ is a write instruction, then all the events generated by $\mathcal{I}$ occur simultaneously.

Formally, let $\mathcal{I}$ =(wr, a, d) be the kth instruction of I_j and the events generated by $\mathcal{I}$ be $E=\{e_1 \ldots e_N\}$, where each e_i is $\mathrm{wr}_k^j(i,\, a,\, d)$. Write atomicity for this instruction is $(E, \phi, =_{wa})$ where $e_i =_{wa} e_j$ for $1 \leq i, j < N$.

A.5.8 Architectural rule violation

The rules defined above, namely CMP, RO, WOS, POS, and WA are useful only when combined together to form a new composite rule. For example, sequential consistency is defined as (CMP, POS, WA).

Assume that M is intended to obey some subset of the architectural rules, say $R = \{r_1 \ldots r_M\}$ (R is said to be a composite architectural rule). Then to determine whether M obeys this set of rules, conceptually, one must collect all possible executions of all possible concurrent programs on M and determine whether the events in the each execution can be arranged such that every one of the rules in R can be *simultaneously* satisfied. To determine whether the execution reveals the violation R, one must construct the graph set $G^1 = \mathrm{A}(r_1) * \mathrm{A}(r_2) \ldots \mathrm{A}(r_n)$ and see if at least one graph in G^1 is circuit free.

Note A.2 If R is a composite architectural rule and if R does not contain CMP, then, $A(R)$ does not contain any circuits. The reason for this is that RO, WOS, POS, and WA *do not* impose any well-founded requirements on a sequence of events. For example a sequence of the form "wr(a, 1); rd(a, 0)" (where the observers and generators of the events are suppressed for readability) could be compatible with all RO, WOS, POS, and WA. However, CMP disallows such a sequence by requiring that the rd event must return the most recently written value.

A.6 Projection Theorem

Theorem A.1 shows that every shared memory system is projectable. With the help of this theorem (and the Theorem A.2), Appendix B shows that one can test whether a given memory model conforms to (CMP, RO, WOS) or (CMP, POS) using at most two addresses and whether it conforms to (CMP, POS, WA) using at most n addresses, where n is the number of components in the model.

THEOREM A.1 (PROJECTION) Let

- M be a shared memory system with N components,
- $I = \{I_1 \dots I_N\}$ be a concurrent program,
- $O = \{O_1 \dots O_N\}$ be an execution of M on I,
- $\mathcal{A}_1$ be a subset of $\mathcal{A}$,
- $I_j^{'}$ be the projection of I_j onto addresses in $\mathcal{A}_1$ for $1 \leq j \leq N$,
- $O_j^{'}$ be the projection of O_j onto addresses in $\mathcal{A}_1$ for $1 \leq j \leq N$.

Then $O^{'} = \{O_1^{'} \dots O_N^{'}\}$ is an execution of M on $I^{'} = \{I_1^{'} \dots I_N^{'}\}$.

Proof: Let $t = t(1)t(2)\dots t(k)$ be the sequence of transitions that shows that O is an execution of M on I. Let the initial state of the system be Q_0 and the state sequence generated by t be $Q_0 Q_1 \dots Q_k$; i.e., executing $t(i)$ from Q_{i-1} results in Q_i.

The following algorithm constructs a sequence of transitions $r = r(1)\dots r(k)$ and a sequence of states $S_0 S_1 \dots S_k$ such that

1. $S_0 = Q_0$,
2. local state of each component as well as channel contents match in S_i and Q_i ($1 \leq i \leq k$) *when projected onto the addresses in* $\mathcal{A}_1$,
3. executing $r(i)$ from S_{i-1} results in S_i (just as executing $t(i)$ from Q_{i-1} results in Q_i),
4. r shows that $O^{'}$ is an output of M on $I^{'}$.

Let $t(j)$ is a transition of P_t, the local state of P_t in $Q(j-1)$ be (l, F, a, d), and the local state of P_t in $Q(j)$ be $(l^{'}, F^{'}, a^{'}, d^{'})$. There are 5 cases to consider:

Case (a): $a = a^{'} \neq \perp$. If $a \in \mathcal{A}_1$, let $r(j)$=$t(j)$; otherwise let $r(j) = \tau$.

Case (b): $a = \perp$ and $a^{'} \neq \perp$. Then $t(j)$ is not a send transition (a send transition is enabled only when $a \neq \perp$), not a copy transition (a copy transitions is enabled only if

$a \neq \perp$), not a reset transition (a reset transition results in $a' = \perp$). In other words, $t(j)$ is a receive operation. Define $r(j)$ to be $t(j)$ if a' is in $\mathcal{A}_1$, otherwise τ.

Case (c): $a \neq \perp$ and $a' = \perp$. By a case analysis similar to the one in Case (b) above, $t(j)$ is a reset transition. Define $r(j)$ to be $t(j)$ if a is in $\mathcal{A}_1$, otherwise τ.

Case (d): $a = a' = \perp$. By a case analysis similar to the one in Case (b), this case is impossible.

Case (e): $a \neq \perp$, $a' \neq \perp$, $a \neq a'$. By a case analysis similar to the one in Case (b), this case is impossible.

By construction, $Q(j-1)$ and $S(j-1)$ are identical when projected onto addresses in $\mathcal{A}_1$ and $t(j)$ is enabled in $Q(j-1)$. Hence $r(j)$ as defined above is enabled in $S(j-1)$. S_j is the state that results in after executing S_{j-1}. Obviously, S_j and $Q(j)$ are identical when projected onto the addresses in $\mathcal{A}_1$.

At the end of the construction, S_k and Q_k are identical when projected onto the addresses in $\mathcal{A}_1$. This implies that contents of the output channels of Q_k and S_k are identical when restricted to $\mathcal{A}_1$. The theorem holds with O' as the contents of the output channels in state S_k. □

A.7 Data Independence Theorem

Theorem A.2 shows that every shared memory system is projectable. With the help of this theorem (and the Theorem A.1), Appendix B shows that one can test whether a given memory model conforms to (CMP, RO, WOS) or (CMP, POS) using at most two addresses and whether it conforms to (CMP, POS, WA) using at most n addresses, where n is the number of components in the model.

THEOREM A.2 (DATA-INDEPENDENCE) Let

- M be a shared memory system with N components,
- $I = \{I_1 \ldots I_N\}$ be a concurrent program,
- $O = \{O_1 \ldots O_N\}$ be an execution of M on I, for $1 \leq j \leq N$,
- f be an arbitrary function from the data domain $\mathcal{D}$ to $\mathcal{D}$,
- a an arbitrary address,
- I' be the program obtained by transforming every (wr, a, d) instruction in I to (wr, a, $f(d)$), and

- O' be the execution obtained by transforming every $(rd,\ a,\ d)$ into $(rd,\ a,\ f(d))$ (where $f(\top)$ is taken as $\top$).

Then, O' is an execution of M on I'.

Proof: Follows directly from the observation that the model M cannot do any data comparisons. Hence if O is an execution of M on I, then O' is an execution of M on I'. □

A.8 Concluding Remarks

Any model expressed in the modeling language presented in this appendix satisfies the *projectable* and *data independence* conditions, as shown by Theorems A.1 and A.2. One can extend the language in a number of ways while still guaranteeing these properties. For example, all channels in the modeling language presented are reliable and preserve the order of the messages. A new channel type can be introduced where both these conditions can be relaxed without affecting the correctness of the theorems.

APPENDIX B

COMPLETE TESTS FOR MEMORY MODEL VERIFICATION

Theorems B.3, B.4, and B.5 below show that when a concurrent program reveals the violation of (CMP, RO, WOS), (CMP, POS), or (CMP, POS, WA) in some shared memory system, then there is another concurrent program that reveals the same violation using a "only a few" addresses. The chain of reasoning is as follows: Let $O = \{O_1 \dots O_N\}$ be an execution of M on $I = \{I_1 \dots I_N\}$, and M is intended to obey the rules $R = \{r_1, \dots r_M\}$. If O shows a violation of R, then the following conditions hold.

- There is an unambiguous concurrent program $I^{'}$ that [1] that produces output $O^{'} = \{O^{'}_1 \dots O^{'}_N\}$ that also reveals the violation of R (Theorem B.1).
- If R is any composite rule that includes RO then one can treat the set of graphs generated by R, $A(R)$ as a singleton (Theorem B.2).

Using these two results, following results are established.

- If R is (CMP, RO, WOS) and I is unambiguous, then there is a concurrent program $I^{'}$ with at most two addresses and with an execution $O^{'}$ that also reveals the violation of R (Theorem B.3).
- If R is (CMP, POS) and I is unambiguous, then there is a concurrent program $I^{'}$ with at most two addresses with an execution $O^{'}$ that also reveals the violation of R (Theorem B.4).
- If R is (CMP, POS, WA) and I is unambiguous, then there is a concurrent program $I^{'}$ with at most N addresses with an execution $O^{'}$ that also reveals the violation of R (Theorem B.5).

---

[1] As explained in Chapter 4, a concurrent program is said to be unambiguous if and only if no write instruction, wr(a, d), appears more than once in the concurrent program.

B.1 Unambiguous Execution

All the transitions in a shared memory system are parameterized by the data and address in the sense that they only move data without ever inspecting them and they never mix the data corresponding to one address with the data corresponding to another address. From this observation it follows that if a O is an output of M on I generated by a sequence of transitions T and every instruction $\mathrm{wr}(a,\, d)$ in I is replaced by $\mathrm{wr}(a,\, f(d))$ where f is an arbitrary function from $\mathcal{D}$ to $\mathcal{D}$, then by executing the sequence T yields a new output $O^{'}$ where O and $O^{'}$ are identical except that whenever O contains a message $(rd,\, a,\, d)$, $O^{'}$ contains $(rd,\, a,\, f(d))$. This observation is used in the proof of Theorem B.1 below.

Theorem B.1 Let

- $I = \{I_1 \ldots I_N\}$ be an unambiguous concurrent program, i.e., no write statement is repeated in I,
- $O = \{O_1 \ldots O_N\}$ be an execution of I on M,
- $R = \{r_1 \ldots r_M\}$ be an arbitrary composite architectural rule,
- f be an arbitrary function from the data domain $\mathcal{D}$ to $\mathcal{D}$,
- a an arbitrary address,
- $I^{'}$ be the program obtained by transforming every $(\mathrm{wr},\, a,\, d)$ instruction in I to $(\mathrm{wr},\, a,\, f(d))$, and
- $O^{'}$ be the execution obtained by transforming every $(rd,\, a,\, d)$ into $(rd,\, a,\, f(d))$ (where $f(\top)$ is taken as $\top$).

If O does not violate R then neither does $O^{'}$.

Note that theorem only states that if O does not violate R, $O^{'}$ does not violate R. The converse is not true in general. For example, if f maps all data values to a single value, say d, then even if O reveals a violation of R, $O^{'}$ may not reveal the violation.

Proof: The following argument shows that $O^{'}$ is an output of M on $I^{'}$ and that $O^{'}$ satisfies R. Let the sequence of events $E = e_1 \ldots e_n$ generated by the execution show that O satisfies R. For each e_i define $e_i^{'}$ as follows. (a) if e_i is an event corresponding to an address $b \neq a$, then $e_i^{'}$ is e_i, (b) if e_i is an event corresponding to a, then $e_i^{'}$ is e_i with the data component of e_i transformed with f. It is easy to see that $E^{'} = e^{'}_1 \ldots e^{'}_n$ is a sequence that shows that $O^{'}$ satisfies R. □

A direct consequence of the theorem is that if every execution O of every unambiguous concurrent program I satisfies R, then M satisfies R on all concurrent program (this statement is proved more rigorously in Lemma B.1 below).

LEMMA B.1 (Unique-Data) If a shared memory system M satisfies a composite rule R on all unambiguous concurrent programs, then it satisfies R on all concurrent programs (even if the concurrent program is ambiguous).

Proof: The proof is by contradiction. Let O be an execution of concurrent program I that reveals violation of R. Assume that (wr,a,d) is one of the repeated instructions in I. Let d_1 is a value that is not present in I at all. Construct I_1 from I where one of the write instructions is replaced by (wr, a, d_1). Obviously, I can be obtained from I_1 by applying the data transformation function "f(x)=if (x=d_1) then d else x" to address a. By applying the contrapositive of Theorem B.1, I_1 has an execution O_1 that shows violation of R.

This construction can be repeated until no write instruction is repeated in I to obtain I_n which has an execution O_n that shows that R is violated. In other words, I_n is unambiguous, has an execution O_n that violates R. This contradicts the hypothesis that M satisfies R on all unambiguous concurrent programs. Hence the assumption that there is an execution O of I that reveals the violation of R is incorrect, which establishes the lemma. □

B.2 Unique Graph with CMP

To determine whether a given shared memory system M satisfies a composite architecture R or not, conceptually one needs to analyze all outputs of all possible concurrent programs. However, Lemma B.1 allows one to restrict the attention to only unambiguous concurrent programs. Hence, in the rest of the chapter it is implicitly assumed that I is unambiguous.

In Section A.5.3, it is pointed out that CMP does not define a unique graph on an execution, but rather defines a set of possible graphs. (All other architectural rules define a single graph.) An execution is said to be consistent with CMP if and only if a sequence of events can be found that is consistent with at least one of these graphs. However, as Theorem B.2 below shows, if the concurrent program is unambiguous, one can effectively assume that CMP defines a unique graph. This property is later used in the proofs of Theorems B.3–B.5.

The important result in the section is that for every composite rule R, $A(R)$ can be treated as a single graph. The rest of the section is complicated and may be skipped except for Section B.2.1 on the first reading.

Let E be the set of events generated by an unambiguous concurrent program I. Let $A(R) = G_R = (E, <_R, =_R)$ be an event graph *without any circuits* for some architectural rules R that *includes RO.*[2] Using $A(R)$, the algorithm shown in Figure B.1 generates a graph $G_c = (E, <_c, \phi)$ such that $G_R * G_c$ contains a circuit if and only if every graph in $G_R * A(CMP)$ contains a circuit. Intuitively, G_c acts as a proxy for all graphs in the graph set $A(CMP)$ in presence of G_R. The basic idea is that G_c is constructed as an approximation for $A(CMP) * A(R_1)$ where $A(R_1)$ is same as G_R but it relates two read events (via RO) only if they are for the same address and are observed by the same component

This algorithm is shown in Figure B.1 and explained by the example in Figure B.2. Figure B.2(a) shows an execution of two sequential programs W and R (writer and reader respectively) that violates (CMP, RO). Intuitively, this is a violation of (CMP, RO) since the write event of W1 becomes visible to R2, then the write event of W2 becomes visible to R3, and then again the write event of W1 becomes visible to R4. In other words, the wr event of W1 becomes visible twice, revealing that the model does not correctly implement (CMP, RO).

Step Subdivide constructs the sets $S_1 \ldots S_4$ as shown in Figure B.2(b) (where the event W1 (in the set S2) really means wr_1^W(R, A, 1) indicating that it is generated by the first instruction of W, the event is visible to R, address is A, and the data component is 1. Similar comments apply for all other events). Step Initial-Read imposes edges from S_1 to S_2, S_3 and S_4. This is because all rd events in S_1—the events that must return $\top$—must be completed before any wr event for A takes place. Step Impose-$<$ introduces the arc from S_2 to S_3 (since $R2 <_{ro} R3$) and an arc from S_3 to S_2 (since $R3 <_{ro} R4$). Hence the graph constructed by the Read-Graph step contains a number of circuits, for

---

[2] One of the reasons $A(CMP)$ contains multiple graphs is that when two read events from the same process return the same value, $A(CMP)$ does not impose any ordering between the two events. Hence, it is necessary to force an ordering between such reads to make $A(R)$ to have a single graph. Of the other four architectural rules—RO, WOS, POS, and WA—RO is ideally suited for this, as WOS and WA do not impose such an order, and POS is stronger than RO. Hence, R is required to include at least RO.

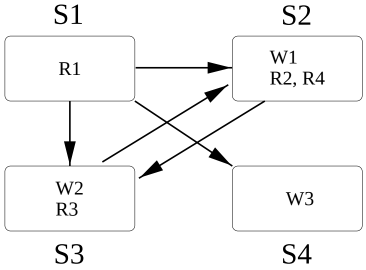
Input: An event graph $G_R = (E, <_R, =_R)$ without any circuits and closed under transitivity. G_R contains at least contains the arcs from RO.
Output: An event graph $G_c = (E, <_c, \phi)$ such that $G_R * A(CMP)$ contains a circuit if and only if every graph in $G_R * G_c$ does.

```
Construct-Graph:
 for each address $a$ and component $i$ do
    Select: $E^{a,i}$={$e_1$, ... , $e_m$} be the set of events
        for address $a$ visible to component $i$;
    Subdivide: Divide $E^{a,i}$ into sets $S_1^{a,i}$, ... , $S_n^{a,i}$ such that
        all elements in $S_i^{a,i}$ have the same data component.
        Without loss of generality, assume that the elements in
        $S_1^{a,i}$ have $\top$ as the data.
    Initial-Read: Draw an arc from $S_1^{a,i}$ to all other sets
        $S_2^{a,i}$ ... $S_m^{a,i}$
    Impose-<: For each $S_j^{a,i}$ and $S_k^{a,i}$ where $j \neq k$ draw an arc
        from $S_j^{a,i}$ to $S_k^{a,i}$ if and only if $S_j^{a,i}$ contains a
        $e$ and $S_k^{a,i}$ contains $e'$ such that $e <_R e'$.
    Trivial-Circuits: From each $S_j^{a,i}$ draw an arc to itself iff
        it contains two events $e$ and $e'$ such that $e <_R e'$ and
        $e$ is a rd and $e'$ is a wr.  These edges represent violations
        because CMP requires an edge from $e'$ to $e$ while $G_R$
        requires an edge from $e'$ to $e$.
 end for each

Read-Graph:
    The graph $G_c = (E, <_c, \phi)$ is simply read from the graph
    constructed above:
    (a) if $e_1$, $e_2$ are two events in the same set and
        $e_1$ is a wr and $e_2$ is rd,
        then $e_1 <_c e_2$
    (b) if $e_1$, $e_2$ are two events in the same set and
$e_1 <_{ro} e_2$, then $e_1 <_c e_2$
    (c) if $e_1$ is in set $S_j^{a,i}$, and $e_2$ is in $S_k^{a,i}$ and Impose-< or Trivial-Circuits
        drew an arc from $S_j^{a,i}$ to $S_k^{a,i}$ then $e_1 <_c e_2$.
```

Figure B.1. An algorithm to compute a unique graph that is equivalent to CMP under the ordering relation given by G_R

W	R
W1: wr(A, 1);	R1: rd(A, $\top$);
W2: wr(A, 2);	R2: rd(A, 1);
W3: wr(A,3);	R3: rd(A, 2);
	R4: rd(A, 1)

(a)



(b)

Figure B.2. Example construction of G_c

example, W1 $<_c$ W2 $<_c$ W1, which shows that (CMP, RO) is violated.

B.2.1 Properties of G_c

The graph constructed by the algorithm in Figure B.1 has a number of properties that are summarized below and used the proofs later on.

P1. If $e_1 <_c e_2$ for some events, then e_1 and e_2 involve the same address and are observed by the same component. This is because the events chosen by algorithm in step Select always have the same address and observer.

P2. A(CMP) defines a graph set, whereas G_c is a single graph *given* G_R.

P3. If w_1 and w_2 are two wr events and $w_1 <_c w_2$, then (a) $w_1 <_R w_2$, or (b) R is (RO, WOS) and there are two rd events such that $w_1 <_c r_1 <_c w_2 <_c r_2$, or (c) R is not (RO, WOS) and there is a rd event r such that $w_1 <_c r <_R w_2$, or $w_1 <_R r$ and $w_2 <_c r$.

P4. If e_1 and e_2 are two events observed by the same component for the same address and $e_1 <_R e_2$, then $e_1 <_c e_2$ (from step Impose-<).

P5. If w_1 and w_2 are two wr events, r_1 is a rd event, and r_1 and w_1 are in the same set, then $w_1 <_c w_2$ iff $r_1 <_c w_2$ (from step Read-Graph (b)).

P6. From step Read-Graph, if r is a rd event that returned the data value $d \neq \top$, then there is a wr event w such that $w <_c r$ and $w <_{cmp} r$ (from step Initial-Read).

P7. If two sets S_j^{ai} and S_k^{ai} contain rd events, then every event of S_j^{ai} is related to every event of S_k^{ai} by $<_c$. This is because R is assumed to contain at least RO, which implies that every rd in S_j^{ai} is related to every rd in S_k^{ai}; hence Impose-$<$ step would add an arc between the two sets. Read-Graph (b) then would an arc between the elements of the two sets. This also implies that, if all sets contain a rd event, $G_R * A(CMP)$ is exactly the same as $G_R * G_c$.

B.2.2 Equivalence of G_c and CMP

THEOREM B.2 Let

- I be an unambiguous concurrent program,
- M be a shared memory system,
- O be an execution of I on M,
- E be the set of events in this execution,
- G_R be an arbitrary event graph with no cycles that contains at least RO, and
- G_c be the graphs generated by Figure B.1.

The graph $G_R * G_c$ is circuit free iff $G_R * A(CMP)$ is partially circuit free.

Proof: Since I is unambiguous, every set $S_j^{a,i}$ constructed by Step Subdivide contains at most one wr event. In addition, due to data independence, every set contains at least wr event, with the exception of $S_1^{a,i}$ which contains no wr events at all. In other words, $S_1^{a,i}$ contains no wr events and $S_2^{a,i}$ ... $S_n^{a,i}$ contains exactly one write event for every a and i. The following case analysis shows that $G_R * G_c$ is equivalent to $G_R * A(CMP)$.

Case 1: $G_R * G_c$ is circuit free. This means that all events in the execution can be linearized into a sequence S that is consistent with the graph $G_R * G_c$. In other words, if e_1 occurs before e_2 in S then $\neg(e_2 <^+ e_1)$ where $<^+$ represents transitive closure of $G_R * G_c$. There are two cases to consider.

Case 1.1: Every set S_j^{ai} contains a rd event. From the property P7 (Section B.2.1), $G_R * G_c$ is same as $G_R * A(CMP)$. Hence $G_R * A(CMP)$ is also circuit free.

Case 1.2: Some of the sets S_j^{ai} do not contain rd events. Since S is constructed so as to satisfy $G_R * G_c$, it also satisfies G_R. Hence, if S satisfies $A(CMP)$, then it also satisfies $G_R * A(CMP)$, and the lemma holds. If S does not satisfy $A(CMP)$ this is because a rd event did not return the most recently written data. Let r_1 be such a rd. Without loss of generality, let r_1 be observed by component 1, and address a. If r_1 returns a value other than the initial value $\top$, Case 1.2.1 below shows how to construct a new sequence S' that does not contain the violation. If, on the other hand, r_1 returns $\top$, Case 1.2.2 shows that it leads to a contradiction. These two cases together complete the proof of Case 1.

Case 1.2.1: r_1 returns data $d \neq \top$. Let w_1 be the wr event observed by component 1 that writes d into a. By step Subdivide w_1 and r_1 belong to the same set. By step Read-Graph, $w_1 <_c r_1$. The CMP violation must be because S contains a sequence $w_1 \ldots w_2 \ldots r_1$, such that CMP requires that r_1 to return the value written by w_2, while r_1 actually returns the value written by w_1. From property P3 and the fact that G_R contains at least RO, it is easy to see that such a sequence S is in $G_R * G_c$ only when the set to which w_2 belongs to does not contain any rd events, i.e., w_2 is the lone element of some set.

If $w_1 <_c w_2$ or $w_1 <_R w_2$, then from step Read-Graph $r_1 <_c w_2$. This contradicts the assumption that S contains $w_2 \ldots r_1$, and is consistent with $G_R{*}G_c$. From the observation that S satisfies $G_R * G_c$ and it contains the sequence $w_1 \ldots w_2$, one can conclude that neither $w_2 <_c w_1$ nor $w_2 <_R w_1$. In other words, w_1 and w_2 are not related under $G_R * G_c$. Hence they can be reordered to obtain a new sequence $S^{'}$ that removes the CMP violation, and is allowed under $G_R * G_c$. Continuing this way, all CMP violations can be removed to result in a new sequence that is allowed under both $G_R * G_c$ and $G_R * A(CMP)$. Hence $G_R * A(CMP)$ is partially circuit free.

Case 1.2.2: r_1 returns $d = \top$. The CMP violation must be because S contains the sequence $w_1 \ldots r_1$, CMP requires r_1 to return the value written by w_1, and r_1 returned $\top$. This case is not possible because the step Initial-Read would have added an edge from the set containing r_1 to the set containing w_1. Hence the subsequence $w_1 \ldots r_1$ of S is not consistent with $G_R * G_c$ contradicting the assumption that S is consistent with $G_R * G_c$.

Case 2: $G_R * G_c$ contains a cycle, say, $C = \{e_1 \ldots e_n\}$. The following case analysis shows that either (a) each (e_i, e_{i+1}) pair is also in $G_R * A(CMP)$ (where e_{n+1} is considered as e_1), or (b) there is a trivial cycle $C^{'}$ in $G_R * A(CMP)$.

Case 2.1: $e_i <_R e_{i+1}$ or $e_i =_R e_{i+1}$. This edge is in all graphs of $G_R * A(CMP)$.

Case 2.2: $e_i <_c e_{i+1}$. This can happen in one of the following three ways.

Case 2.2.1: e_i and e_{i+1} belong to two distinct sets S^{ai}_j, S^{ai}_k respectively and Impose-$<$ added an arc from $S^{ai}_j$to S^{ai}_k. This edge can also be inferred from G_R and $A(CMP)$ (i.e., there is a sequence of events $s_1 \ldots s_n$ such that $e_i <_x s_1 <_x \ldots <_x s_n \ldots e_{i+1}$ where each $<_x$ is either $<_{cmp}$ or $<_R$).

Case 2.2.2: Both e_i and e_{i+1} belong to the same set $S_j^{ai}$and Trivial-Circuits step added an arc from $S_j^{ai}$to itself. There is a trivial circuit from one of the events in S_j^{ai} to another event in S_j^{ai}(which need not involve e_i and e_{i+1}).

Case 2.2.3: Both e_i and e_{i+1} belong to the same set S_j^{ai} and Read-Graph step added the edge from e_i to e_{i+1}. In this case, $e_i <_{cmp} e_{i+1}$ or $e_i <_{ro} e_{i+1}$. Since G_R is assumed to contain $<_{ro}$, the second case can be rewritten as $e_i <_R e_{i+1}$.

Proof continued: From Cases 2.1 and 2.2, if there is a circuit in $G_R * G_c$ then there is a circuit in every graph in $G_R * A(CMP)$. From Cases 1 and 2 above $G_R * G_c$ contains a circuit exactly when $G_R * A(CMP)$ does. □

In the rest of the appendix, when the graph for CMP, $<_{cmp}$, is considered an alias for the graph $<_c$, and for all other architectural rules, R, $A(R)$ is considered to be G_R as constructed above. Hence for any architectural rule R, the graph set $A(R, CMP)$ is treated as the graph $G_R * G_c$.

B.3 Notation

If an execution O of I on M reveals the violation of some architectural rules R then by a direct application of Theorem B.1 there is another execution $O^{'}$ of $I^{'}$ that also reveals the violation of R and $I^{'}$ is unambiguous. If R contains RO, applying Theorem B.2 on $O^{'}$, one can conclude that there is a single graph $G_R * G_c$ that reveals the violation. The rest of the theorems implicitly apply these two theorems; i.e., whenever O is said to reveal the violation of R for some R containing RO, it is assumed that (a) I is unambiguous, (b) the graph set $A(R)$ has a single graph, and (c) $A(R)$ has a cycle.

B.4 Complete Tests for Formal Memory Models

Theorems B.3, B.4, and B.5 to establish that when a concurrent program reveals the violation of (CMP, RO, WOS), (CMP, POS), or (CMP, POS, WA) in some shared memory system, then there is another concurrent program that reveals the same violation using a "only a few" addresses.

B.4.1 Complete test for CMP, RO, WOS

Theorem B.3 (CMP, RO, WOS) Let M be a shared memory system with N components, $I = \{I_1 \ldots I_N\}$ a concurrent program, and $O = \{O_1 \ldots O_N\}$ be an execution of M on I. If O shows that the composite rule (CMP, RO, WOS) is violated, then there is a concurrent program $X = \{X_1 \ldots X_N\}$ with no more than two addresses that also reveals the violation.

Proof: Since O shows that the composite rule (CMP, RO, WOS) is violated, there is a cycle in this set of events. From Theorem B.2, the graph set $A(CMP, RO, WOS)$ can be considered equivalent to the graph $G = G_R * G_c$, where $G_R = A(RO, WOS)$. Let $E = e_1 \ldots e_n$ be a circuit in G that shows the violation. From the definitions of RO, WOS, and P1 (from Section B.2.1) it is clear that all these events are observed by the same component. Hence any cycle should involve only the events observed by a single component.

To show that E can be converted into a circuit containing no more than two addresses, it is useful to define the following operations.

If C is a cycle in $G_R * G_c$, then it can be subjected to the following steps to obtain a new cycle $C^{'}$ also in $G_R * G_c$.

Introduce-RO-Edge: If r_1 and r_2 are two rd events in C that are not directly connected; i.e., C contains the sequence $r_1 x_1 x_2 \ldots x_n r_2$, then a new $C^{'}$ is obtained from C as follows. If $r_1 <_{ro} r_2$, then $C^{'}$ is obtained from C by removing $x_1 \ldots x_n$. If $r_2 <_{ro} r_1$, then $C^{'}$ is $r_1 x_1 \ldots x_n r_2 <_{ro} r_1$. This step brings the two rd events "closer" in the cycle.

Introduce-WOS-Edge: Similarly, if w_1 and w_2 are two wr events in C *generated by the same component* that are not directly connected and $w_1 <_{wos} w_2$, then a new cycle $C^{'}$ is constructed from C by removing the events between w_1 and w_2.

Introduce-Read: If w_1 and w_2 are two wr events in C and $w_1 <_c w_2$, then from P3 either $w_1 <_{wos} w_2$ or there are two rd events r_1 and r_2 such that $w_1 <_c r_1 <_c w_2 <_c r_2$. If $w_1 <_{wos} w_2$ is not true then $C^{'}$ is constructed from C by replacing $w_1 <_c w_2$ with $w_1 <_c r_1 <_c w_2$.

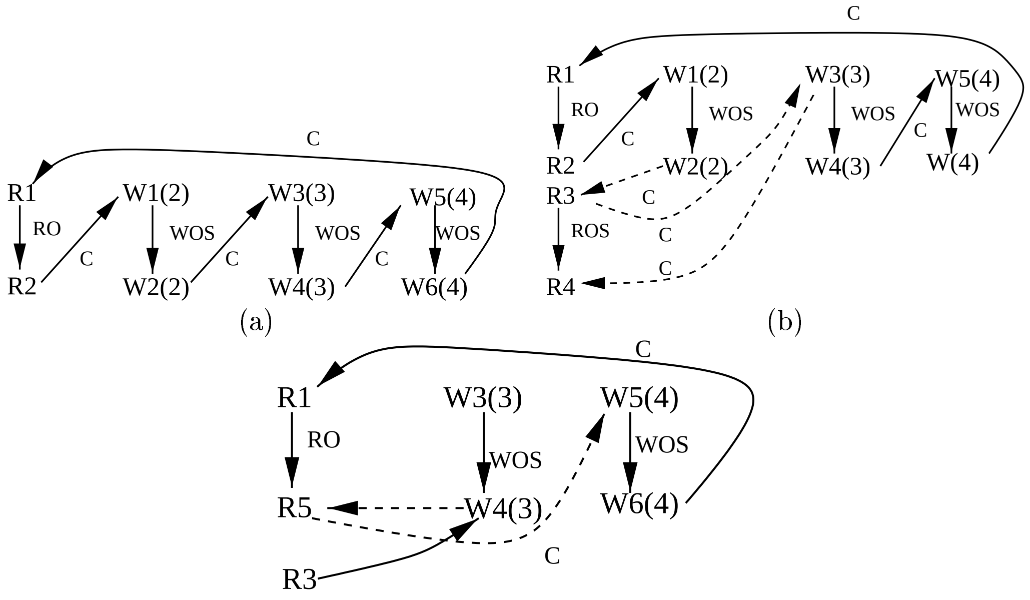
From steps Introduce-RO-Edge and Introduce-WOS-Edge, it is clear that whenever a circuit E contains more than two rd events, or more than two wr events *per component*, it can be reduced to a circuit with *exactly* two rd events and two wr events per component. Similarly if E contains a $<_c$ edge between two wr events, from the step Introduce-Read, a rd event can be introduced between the two events. Hence, without loss of generality, assume that (a) E contains two rd events, (b) E contains at most two wr events generated by any given component such that the rd events are connected by $<_{ro}$, (c) if a component has generated two wr events in E, they are connected by $<_{wos}$, (d) the cycle in E is formed by introducing $<_c$ edges between these events, and (e) all events in E are observed by the first component. Intuitively, since the circuit contains only two rd events, then a test for (CMP, RO, WOS) with only one address (if the two read events involve the same address) or only two addresses (if the read events involve different addresses). The rest of the section formalizes the steps hidden in this intuition.

Any such circuit can be represented in the form shown in Figure B.3(a). In this figure, the ith column shows the events generated by the ith component (but, of course, observed by the first component). In this figure, the circuit contains no wr events observed by the first component; however, the argument below can be trivially to the case where the circuit contains wr events.

In the figure, rd events R1 and R2 are generated by component 1, wr events W1 and W2 are generated by component 2, W3 and W4 are generated by component 3, and W5 and W6 are generated by component 4. All the events are observed by component 1. As shown the figure, they are connected by $<_{ro}$, $<_{wos}$, and $<_c$ edges (shown as RO, WOS, and C respectively). This loop potentially can contain 4 different addresses as R1, R2, W2, and W4 can contain different addresses (W1 must have the same address as R2 as they are related by $<_c$ and similar comments apply to W3, W5, and W6).

From W2 $<_c$ W3, and P3, there are two rd events R3 and R4 observed by component 1 such that W2 $<_c$ R3 $<_c$ W3 $<_c$ R4. R3 and R4 can be ordered in any way with R1 and R2. One of these situations is shown in Figure B.3(b) where R3 and R4 occur after R2. The newly introduced edges are shown as dotted lines and the $<_c$ edge from W2 to W3 is removed. In this figure, there is a new cycle, R1, R2, R3, W3, W4, W5, W6, R1. This cycle can then be shorted by eliminating R2 by applying the Introduce-RO-Edge operation. Then applying the Introduce-Read step on the W4–W5 edge yields a new event R5 such that W4 $<_c$ R5 $<_c$ W5. Figure B.3(c) shows the resulting graph if R5 assumed to be before R3. In this graph, the cycle R1–R5–W5–W6–R1 contains only two addresses.

Hence any cycle E can be converted to a cycle containing no more than two addresses, say a and b. From the Theorem A.1, when the concurrent program I is projected onto those two addresses, the resulting program X has an execution that reveals the violation of (CMP, RO, WOS). □



(a) (b)

(c)

Figure B.3. An example circuit violating CMP, RO, WOS, and its transformation

B.4.2 Complete test for CMP, POS

THEOREM B.4 (CMP, POS) Let M be a shared memory system with N components, $I = \{I_1 \ldots I_N\}$ be a set of sequential programs, and $O = \{O_1 \ldots O_N\}$ be an execution of M on I. If O shows that the composite rule (CMP, POS) is violated, then there is a set of sequential programs $X = X_1 \ldots X_N$ with no more than two addresses that reveals the violation.

Proof: Let E be a cycle that involves the least number of events that reveals that M violates (CMP, POS) when it produces O when the input is I. (Note that POS also includes RO.) By an argument similar to the one used in the proof of Theorem B.3, one can conclude that there are at most N edges labeled $<_{pos}$. In addition, there are at most two rd events. By an argument identical to the one used in Theorem B.3, one can conclude that the circuit can be shortened until there are only two addresses. $\square$

B.4.3 Complete test for CMP, POS, WA

As discussed in Section A.5, (CMP, POS, WA) is sequential consistency. Theorem B.5 below shows that if M is a shared memory system with N components, then M violates (CMP, POS, WA) if and only if it violates (CMP, POS, WA) on a concurrent program with more than N addresses.

The following two facts are used to simplify the proof of the theorem.

NOTE B.1 From the definition of $<_{pos}$, it is clear that if $e_1 <_{pos} e_2$ and $e_2 <_{pos} e_3$ for any e_1, e_2, e_3, then $e_1 <_{pos} e_3$. Also, if e_4 and e_5 are two events generated by the same component, say P_g and observed by the same component, say P_a, then either $e_4 <_{pos} e_5$ or $e_5 <_{pos} e_4$.

NOTE B.2 If C is a circuit involving $<_c$, $<_{pos}$, and $=_{wa}$ edges, with at most k edges with label $<_{pos}$ for some k, then C contains at most k addresses. This follows directly from the observation that $<_c$ and $=_{wa}$ can connect two events only if the two events involve the same address.

THEOREM B.5 (CMP, POS, WA) Let M be a shared memory system with N components, $I = \{I_1 \ldots I_N\}$ be a set of sequential programs, and $O = \{O_1 \ldots O_N\}$ be an execution of M on I. If O shows that the composite rule (CMP, POS, WA) is violated, then there is a set of sequential programs $X = X_1 \ldots X_N$ with no more than N addresses that reveals the violation.

Proof: The proof is by showing that there is a circuit C involving at most N addresses and using Theorem A.1 to project I onto the addresses in C to obtain X.

Let C be a circuit with the fewest number of events that shows that O reveals a violation of (CMP, POS, WA) of M. The following argument shows that C has at most N addresses.

Case 1: C contains N or fewer edges labeled $<_{pos}$. From Note B.2, C contains at most N addresses. The proof continues at **Proof Continued** below.

Case 2: C contains more than N edges labeled $<_{pos}$. As M contains only N components, at least two $<_{pos}$ edges are generated by the same component, say P_g and observed by P_a and P_b. (The argument so far does not require P_a, P_b, and P_g to be distinct.) If P_a and P_b are the same, then from Note B.1 above, at least one event can be removed from C to create a new circuit, which contradicts the assumption that C has fewest number of events. Hence, in the ensuing discussion, P_a and P_b are assumed to be distinct.

C can be depicted as shown in Figure B.4: a_0 and a_1 are two events observed by P_a, b_2 and b_3 are two events observed by P_b, and all four events are generated by P_g. a_1 and b_2 are related by path p_1 involving $<_{pos}$, $<_c$, and $=_{wa}$ such that $a_1 < b_2$. Similarly, b_3 and a_0 are related by path p_2 involving $<_{pos}$, $<_c$ and $=_{wa}$ such that $b_3 < a_0$.

Without loss of generality, assume that P_g is not P_a. If P_g is indeed P_a, then P_a and P_b can be renamed such that P_g is P_b, but not P_a. Since a_0 is generated by P_g and observed

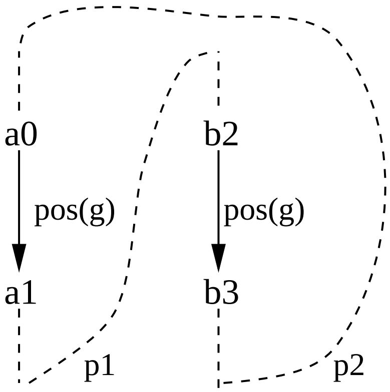


Figure B.4. A CMP, POS, WA violation

by P_a, a_0 must be a wr event (recall that rd events observed only by the generator). Similarly, a_1 is also a wr event.

Let a_0, a_1, b_2, and b_3 be generated by instructions i_0, i_1, i_2, and i_3 respectively (since P_g generated these events, these instructions belong to P_g). Since a_0 and a_1 are two distinct events, i_0 and i_1 are distinct. Similarly, since b_2 and b_3 are distinct, i_2 and i_3 are distinct. There are five cases to consider depending on the relative order of i_0 with respect to i_2 and i_3 as given by the program order of P_g.[3]

1. $i_0 < i_2 < i_3$,
2. $i_0 = i_2 < i_3$,
3. $i_2 < i_0 < i_3$,
4. $i_2 < i_0 = i_3$, and
5. $i_2 < i_3 < i_0$

($i_2 < i_3$ is fixed by the fact that $b_2 <_{pos} b_3$.)

In all the cases, let b_0 and b_1 be the events observed by P_b for i_0 and i_1 (as a_0 and a_1 are wr events, b_0 and b_1 exist). The case analysis below shows that all the above cases lead to a contradiction with the assumption that C has fewest events. In each case, we construct a new circuit C' such that it contains at least one $<_{pos}$ edge[4] and has fewer

---

[3] In the following discussion, $<$ is used to indicate the program order of P_g.

events than C, thus leading to a contradiction.

Case 2.1: $i_0 < i_2 < i_3$. From Note B.1, $b_0 <_{pos} b_2$, and $b_2 <_{pos} b_3$, a new circuit $C^{'}$ can be obtained by deleting b_2, inserting b_0, deleting the edge from a_0 to a_1, and deleting the path p_1, as shown in Figure B.5. $C^{'}$ contains fewer events than C, leading to a contradiction with the assumption that C contains fewest events. (Proof continues at “Case 2 (Contd).”)

Case 2.2: $i_0 = i_2 < i_3$. A circuit $C^{'}$ can be obtained by removing the edge from a_0 to a_1, removing the path p_1, and inserting an edge from a_0 to $b_2 = b_0$ (see Figure B.6). This circuit has fewer events than C, leading to a contradiction with the assumption that C contains fewest events. (Proof continues at “Case 2 (Contd).”)

Case 2.3: $i_2 < i_0 < i_3$. A circuit $C^{'}$ can be obtained by inserting an edge from a_0 to b_0, from b_0 to b_3, deleting p_1, deleting the edge from a_0 to a_1, and deleting edge from b_2 to b_3, as shown in Figure B.7. This circuit has fewer events than C, leading to a contradiction with the assumption that C contains fewest events. (Proof continues at “Case 2 (Contd).”)

Case 2.4: $i_2 < i_0 = i_3$. This case differs from the above three cases in that, $C^{'}$ is constructed by retaining p_1 but deleting p_2. Combining $i_0 < i_1$ with $i_2 < i_0 = i_3$, we can conclude that $i_2 < i_1$. Hence, $b_2 <_{pos} b_1$. A circuit $C^{'}$ can be obtained by inserting a $=_{wa}$ edge from b_1 to a_1, deleting the path p_2, deleting the edge from a_0 to a_1, deleting the

---

[4] Recall that a circuit must contain at least one $<_{cmp}$, $<_{ro}$, $<_{wos}$, or $<_{pos}$ edge as defined in Section A.5. Hence the construction ensures that $C^{'}$ has at least one $<_{pos}$.

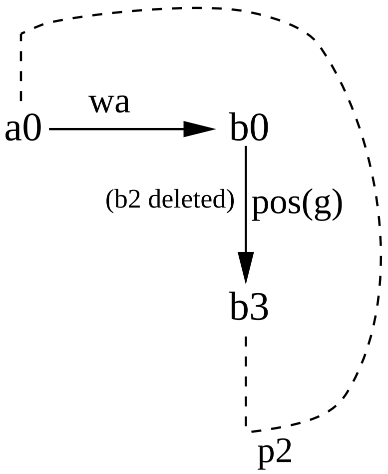


Figure B.5. Case 1: $i_0 < i_2 < i_3$

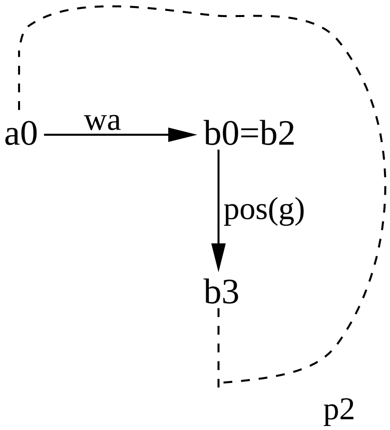


Figure B.6. Case 2: $i_0 = i_2 < i_3$

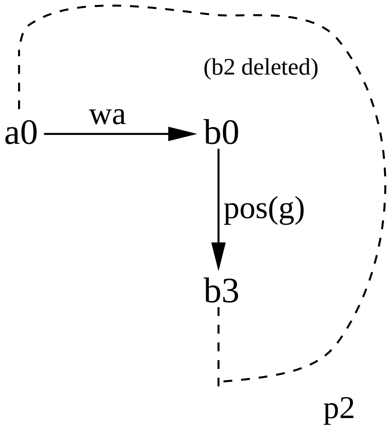


Figure B.7. Case 3: $i_2 < i_0 < i_3$

edge from b_2 to b_3, and inserting a $<_{pos}$ edge from b_2 to b_1 as shown in Figure B.7. This circuit has fewer events than C, leading to a contradiction with the assumption that C contains fewest events. (Proof continues at "Case 2 (Contd).")

Case 2.5: $i_2 < i_3 < i_0$. In this case also C' is constructed by retaining p_1 but deleting p_2. From $i_0 < i_1$, we can conclude that $i_2 < i_1$. A circuit C' can be obtained inserting a $=_{wa}$ edge from b_1 to a_1 and deleting b_3 and a_0, as shown in Figure B.9. This circuit has fewer events than C, leading to a contradiction with the assumption that C contains fewest events. (Proof continues at "Case 2 (Contd).")

Case 2 (Contd): Hence in all five cases above, a circuit C' can be constructed that contains fewer events than C, leading to a contradiction with the assumption that C contains fewest number of events. Hence Case 2 leads to contradiction and need not be considered further.

Proof Continued: From Case 1 above, the circuit C contains at most N addresses. Since

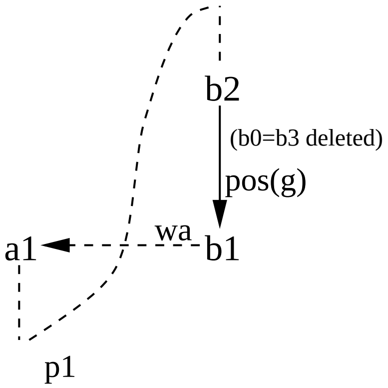


Figure B.8. Case 4: $i_2 < i_0 = i_3$

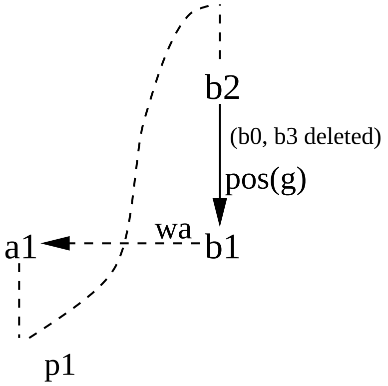


Figure B.9. Case 5: $i_2 < i_3 < i_0$

M is projectable, from Theorem A.1, I can be projected onto the addresses appearing only in C to obtain X such that X would reveal that M does not implement (CMP, POS, WA). $\square$

REFERENCES

[1] Adve, S. V., and Gharachorloo, K. Shared memory consistency models: A tutorial. *Computer 29*, 12 (Dec. 1996), 66–76.

[2] Ahamad, M., Bazzi, R. A., John, R., Kohli, P., and Neiger, G. The power of processor consistency (extended abstract). In *Proceedings of the 5th ACM Annual Symposium on Parallel Algorithms and Architectures (SPAA'93)* (June 1993), pp. 251–260.

[3] Alur, R., Brayton, R. K., Henzinger, T. A., and Qadeer, S. Partial-order reduction in symbolic state space exploration. *Lecture Notes in Computer Science 1254* (1997).

[4] Alur, R., McMillan, K., and Peled, D. Model-checking of correctness conditions for concurrent objects. In *11th Annual IEEE Symposium on Logic in Computer Science* (New Brunswick, New Jersey, July 1996), pp. 219–228.

[5] Boyer, R. S., and Moore, J. S. *A Computational Logic.* Academic Press, New York, 1979.

[6] Brayton, R. K., Hachtel, G. D., Sangiovanni-Vincentelli, A., Somenzi, F., Aziz, A., Cheng, S.-T., Edwards, S., Khatri, S., Kukimoto, Y., Pardo, A., Qadeer, S., Ranjan, R. K., Sarwary, S., Shiple, T. R., Swami, G., and Villa, T. VIS: A system for verification and synthesis. In *Computer Aided Verification* (July 1996), pp. 428–432.

[7] Brock, B., Kaufmann, M., and Moore, J. S. ACL2 theorems about commercial microprocessors. In *First international conference on formal methods in computer-aided design* (Palo Alto, CA, USA, Nov. 1996), M. Srivas and A. Camilleri, Eds., vol. 1166 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 275–293.

[8] Bryant, R. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers C-35*, 8 (Aug. 1986), 677–691.

[9] Bryant, R. E. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys 24*, 3 (1992), 293–318.

[10] Bryg, W. R., Chan, K. K., and S.Fiduccia, N. A high-performance, low-cost multiprocessor bus for workstations and midrange servers. *Hewlett-Packard Journal* (Feb. 1996), 18–24.

[11] Buckley, G. N., and Silberschatz, A. An effective implementation for the generalized input-output construct of CSP. *ACM Transactions on Programming Languages and Systems 5*, 2 (Apr. 1983), 223–235.

[12] Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L. H. Symbolic model checking:10^{20} states and beyond. In *5th Annual IEEE Symposium on Logic in Conputer Science* (Philadelphia, Pennsylvania, June 1990), IEEE Computer Society Press, pp. 428–439.

[13] Butler, R. W. An introduction to requirements capture using PVS: Specification of a simple autopilot. Tech. Rep. NASA Technical Memorandum 111025, NASA Langley Research Center, Hampton, Virginia, May 1996.

[14] Carter, J. B., Kuo, C.-C., and Kuramkote, R. A comparison of software and hardware synchronization mechanisms for distributed shared memory multiprocessors. Tech. Rep. UUCS-96-011, University of Utah, Salt Lake City, UT, USA, Sept. 1996.

[15] Chandra, S., Richards, B., and Larus, J. R. Teapot: Language support for writing memory coherency protocols. In *SIGPLAN Conference on Programming Language Design and Implementation* (May 1996).

[16] Clarke, E., Filkorn, T., and Jha, S. Exploiting symmetry in temporal logic model checking. In *Computer Aided Verification* (Elounda, Greece, June 1993), pp. 450–463.

[17] Clarke, E. M., Emerson, E. A., Jha, S., and Sistla, A. P. Symmetry reductions in model checking. In *Computer Aided Verification* (Vancouver, BC, Canada, June 1998), A. J. Hu and M. Y. Vardi, Eds., vol. 1427 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 147–158.

[18] Clarke, E. M., Emerson, E. A., and Sistla, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems 8*, 2 (1986), 244–263.

[19] Clarke, E. M., Emerson, E. A., and Sistla, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions in Programming Languages and Systems 8*, 2 (1986), 244–263.

[20] Collier, W. W. Multiprocessor diagnostics. http://www.infomall.org/diagnostics-/archtest.html.

[21] Collier, W. W. *Reasoning About Parallel Architectures.* Prentice-Hall, Englewood Cliffs, NJ, 1992.

[22] Corella, F. Verifying memory ordering model of I/O systems. In *Invited talk at Computer Hardware Description Languages 1997, Toledo* (Spain, Apr. 1997).

[23] Courcoubetis, C., Vardi, M., Wolper, P., and Yannakakis, M. Memory efficient algorithms for the verification of temporal properties. In *Computer Aided Verification* (June 1990), pp. 233–242.

[24] Cousot, P., and Cousot, R. Abstract intepretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of 4th POPL* (Los Angeles, CA, ACM Press, 1977), pp. 238–252.

[25] Cray Research, Inc. *CRAY T3D System Architecture Overview*, hr-04033 ed., September 1993.

[26] Dill, D. The Stanford Murphi verifier. In *Computer Aided Verification* (New Brunswick, New Jersey, July 1996), R. Alur and T. A. Henzinger, Eds., vol. 1102 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 390–393. Tool demo.

[27] Dill, D. L., Park, S., and Nowatzyk, A. Formal specification of abstract memory models. In *Research on Integrated Systems* (1993), G. Borriello and C. Ebeling, Eds., MIT Press, pp. 38–52.

[28] Eiriksson, A. T., and McMillan, K. Using formal verification/analysis methods on the critical path in system design: A case study. In *Computer Aided Verification* (1995), vol. 939 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 367–380.

[29] Emerson, E. A., and Namjoshi, K. S. Automatic verification of parameterized synchronous systems. In *Computer Aided Verification* (New Brunswick, NJ, USA, June 1998), R. Alur and T. A. Henzinger, Eds., vol. 1102 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 87–87. Extended abstract.

[30] Gerth, R. Introduction to sequential consistency and the lazy caching algorithm. *Distributed Computing* (1995). Also can be found in http://www.-research.digital.com/SRC/tla/papers.html.

[31] Ghughal, R. Test model-checking approach to verification of formal memory models. Master's thesis proposal. University of Utah, Salt Lake City, UT, USA, June 1998.

[32] Gibbons, P. B., and Korach, E. On testing cache-coherent shared memories. In *Proceedings of the 6th Annual Symposium on Parallel Algorithms and Architectures* (New York, NY, USA, June 1994), ACM Press, pp. 177–188.

[33] Gibbons, P. B., and Korach, E. Testing shared memories. *SIAM Journal on Computing 26*, 4 (Aug. 1997), 1208–1244.

[34] Godefroid, P. Using partial orders to improve automatic verification methods. In *Computer Aided Verification* (New Brunswick, NJ, USA, June 1990), R. Alur and T. A. Henzinger, Eds., vol. 1102 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 176–185.

[35] Godefroid, P. *Partial-Order Methods for the Verification of Concurrent Systems: An approach to the State-Explosion Problem.* PhD thesis, Univerite De Liege, 1994–95.

[36] Godefroid, P., and Pirottin, D. Refining dependencies improves partial-order verification methods. In *Computer Aided Verification* (Elounda, Greece, June 1993), pp. 438–450.

[37] Godefroid, P., and Wolper, P. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Computer Aided Verification* (Berlin, Germany, July 1992), K. G. Larsen and A. Skou, Eds., vol. 575 of *LNCS*, Springer-

Verlag, pp. 332–342.

[38] Goodman, J. R. Cache consistency and sequential consistency. Tech. Rep. 61, IEEE Scalable Coherence Interface Working Group, Mar. 1989.

[39] Gopalakrishnan, G., Ghughal, R., Hosabettu, R., Mokkedem, A., and Nalumasu, R. Formal modeling and validation applied to a commercial coherent bus: A case study. In *CHARME* (Montreal, Canada, 1997), H. F. Li and D. K. Probst, Eds.

[40] Gosling, J., Joy, B., and Steele, G. *The Java^{TM} Language Specification*, 1.0 ed. Sun Microsystems, Aug. 1996. appeared also as book with same title in Addison-Wesleys 'The Java Series'.

[41] Graf, S. Verification of a distributed cache memory by using abstractions. *Lecture Notes in Computer Science 818* (1994).

[42] Gribomont, E. P. From synchronous to asynchronous communication. In *Specification and Verification of Concurrent Systems*, C. Rattay, Ed. Springer-Verlag, University of Stirling, Scotland, 1990, pp. 368–383.

[43] Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Apporach.* Morgan Kaufmann, 1996. Second Edition, Appendix E.

[44] Herlihy, M. P., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12*, 3 (July 1990), 463–492.

[45] Hoare, C. A. R. Communicating sequential processes. *Communications of the ACM 21*, 8 (1978), 666–677.

[46] Hojati, R., Mueller-Thuns, R., Loewenstein, P., and Brayton, R. Automatic verification of memory systems which service their requests out of order. In *CHDL* (1995), pp. 623–639.

[47] Holzmann, G. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[48] Holzmann, G., Godefroid, P., and Pirottin, D. Coverage preserving reduction strategies for reachability analysis. In *International Symposium on Protocol Specification, Testing, and Verification* (Lake Buena Vista, Florida, USA, June 1992).

[49] Holzmann, G., and Peled, D. An improvement in formal verification. In *Proceedings of Formal Description Techniques* (Bern, Switzerland, Oct. 1994).

[50] Holzmann, G., Peled, D., and Yannakakis, M. On nested depth first search. In *The* Spin *Verification System* (1996), American Mathematical Society, pp. 23–32. Proceedings of the Second Spin Workshop.

[51] Holzmann, G. J. The model checker spin. *IEEE Transactions on Software Engineering 23*, 5 (May 1997), 279–295. Special issue on Formal Methods in Software

Practice.

[52] Holzmann, G. J., and Peled, D. The state of Spin. In *Computer Aided Verification* (New Brunswick, New Jersey, July 1996), R. Alur and T. A. Henzinger, Eds., vol. 1102 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 385–389. Tool demo.

[53] Ip, C. N., and Dill, D. L. Better verification through symmetry. In *International Conference on Computer Hardware Description Language* (1993).

[54] Kurshan, R. P. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press, 1994.

[55] Kurshan, R. P., Levin, V., Minea, M., Peled, D., and Yenigun, H. Verifying hardware in its software context. In *International Conference on Computer Aided Design* (San Jose, CA, USA, 1997).

[56] Kuskin, J., and et al., D. O. The Stanford FLASH multiprocessor. In *SIGARCH94* (May 1994), pp. 302–313.

[57] Ladkin, P., Lamport, L., Olivier, B., and Roegel, D. Lazy caching in tla. *Distributed Computing* (1997).

[58] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers 9*, 29 (1979), 690–691.

[59] Lamport, L. How to make a correct multiprocess program execute correctly on a multiprocessor. Tech. rep., Digital Equipment Corporation, Systems Research Center, Feb. 1993.

[60] Lamport, L. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16*, 3 (May 1994), 872–923. Also appeared as SRC Research Report 79.

[61] Lamport, L., and Schneider, F. B. Pretending atomicity. Tech. Rep. Research Report 44, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, May 1989.

[62] Laudon, J., and Lenoski, D. The SGI origin: A ccNUMA highly scalable server. In *International Symposium on Computer Architecture* (New York, June 1997), vol. 25,2 of *Computer Architecture News*, ACM Press, pp. 241–251.

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

[64] Lipton, R. J. Reduction: A method of proving properties of parallel programs. *Communications of the ACM 18*, 12 (Dec. 1975), 717–721.

[65] Lipton, R. J., and Sandberg, J. S. Pram: A scalable shared memory. Tech. Rep. CS-TR-180-88, Dept. of Computer Science, Princeton University, Sept. 1988.

[66] Manna, Z., and Pnueli, A. *The Temporal Logic of Reactive and Concurrency Systems.* Springer-Verlag, 1992.

[67] McMillan, K. L. *Symbolic Model Checking.* Kluwer Academic Press, 1993.

[68] Nalumasu, R. Formal verification techniques for shared memory systems. Tech. Rep. UUCS-98-XXX, University of Utah, Salt Lake City, UT, USA, Aug. 1998.

[69] Nalumasu, R., and Gopalakrishnan, G. Explicit-enumeration based verification made memory-efficient. In *Proceedings of the 1995 Conference on Computer Hardware Description Languages (CHDL'95), Chiba, Japan* (1995), pp. 617–622.

[70] Nalumasu, R., and Gopalakrishnan, G. A new partial order reduction algorithm for concurrent system verification. In *CHDL* (Toledo, Spain, Apr. 1997), Chapman Hall, ISBN 0 412 78810 1, pp. 305 – 314.

[71] Nalumasu, R., and Gopalakrishnan, G. PV: a model-checker for verifying ltl-x properties. In *Fourth NASA Langley Formal Methods Workshop* (1997), NASA Conference Publication 3356, pp. 153–161.

[72] Nalumasu, R., and Kurshan, R. P. Translation between S/R and Promela. Tech. Rep. ITD-95-27619V, Bell Labs, July 1995.

[73] Owre, S., Rajan, S., Rushby, J. M., Shankar, N., and Srivas, M. PVS: Combining specification, proof checking and model checking. In *Computer Aided Verification* (New Brunswick, NJ, USA, 1996), R. Alur and T. A. Henzinger, Eds., vol. 1102 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 411–414.

[74] Park, S., and Dill, D. L. Protocol verification by aggregation of distributed transactions. In *Computer Aided Verification* (New Brunswick, NJ, USA, July 1996), R. Alur and T. A. Henzinger, Eds., vol. 1102 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 300–309.

[75] Park, S., and Dill, D. L. Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In *SPAA* (Padua, Italy, June 1996), pp. 288–296.

[76] Paulson, L. C. Introduction to Isabelle. Tech. Rep. 280, University of Cambridge, Computer Laboratory, 1993.

[77] Peled, D. All from one, one for all: On model checking using representatives. In *Computer Aided Verification* (Elounda, Greece, June 1993), pp. 409–423.

[78] Peled, D. Combining partial order reductions with on-the-fly model-checking. *Journal of Formal Methods in Systems Design 8 (1)* (1996), 39–64. also in Computer Aided Verification, 1994.

[79] Pong, F., and Dubois, M. New approach for the verification of cache coherence protocols. *IEEE Transactions on Parallel and Distributed Systems 6*, 8 (Aug. 1995), 773–787.

[80] Pong, F., Nowatzyk, A., Aybay, G., and Dubois, M. Verifying distributed directory-based cache coherence protocols: S3.mp, a case study. In *Proceedings of the First International EURO-PAR Conference* (Stockholm, Sweden, Aug. 1995), S. Haridi, K. Ali, and P. Magnusson, Eds., Lecture Notes in Computer Science, Springer-Verlag, pp. 287–300.

[81] Saulsbury, A., Pong, F., and Nowatzyk, A. Missing the memory wall : The case for Processor/Memory integration. In *Proceedings of the 23rd Annual International Symposium on Computer Architecure* (New York, May 1996), ACM Press, pp. 90–103.

[82] Shen, X., and Arvind. Specification of memory models and design of provably correct cache coherence protocols. Tech. Rep. CSG-Memo-398, Massachusetts Institute of Technology, Cambridge, MA, June 1997.

[83] Tarjan, R. Depth-first search and linear graph algorithms. *SIAM Journal on Computing 1*, 2 (June 1972), 146–160.

[84] Valmari, A. A stubborn attack on state explosion. In *Computer Aided Verification* (New Brunswick, NJ, USA, June 1990), R. Alur and T. A. Henzinger, Eds., vol. 1102 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 156–165.

[85] Valmari, A. A stubborn attack on state explosion. *Journal of Formal Methods in Systems Design 1* (1992), 297–322. Also in Computer Aided Verification, 1990.

[86] Valmari, A. On-the-fly verification with stubborn sets. In *Computer Aided Verification* (Elounda, Greece, June 1993), pp. 397–408.

[87] van Leeuwen, J., Ed. *Handbook of Theoretical Computer Science, Volume B:Formal Models and Semantics.* Elsevier / MIT Press, 1990.

[88] Weaver, D. L., and Germond, T. *The SPARC Architecture Manual – Version 9.* P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1994.