TEST MODEL-CHECKING APPROACH TO VERIFICATION OF FORMAL MEMORY MODELS

by

Rajnish Ghughal

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

Master of Science

Department of Computer Science

The University of Utah

December 1999

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Rajnish Ghughal

This thesis 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
	Chris Myers

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Rajnish Ghughal 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

Shared memory multiprocessors are becoming increasingly important as high performance parallel computers and servers. A *formal memory model* is a contract between hardware and software that describes the behavior of memory system in response to various memory operations. Sequential consistency is the strongest memory model proposed for multiprocessors. Sequential consistency is an intuitive and easy to program memory model, but it prohibits the use of various architectural optimizations commonly employed in memory systems. Hence, many modern multiprocessors adapt weaker memory models to achieve higher performance.

We consider the problem of verifying shared memory multiprocessor memory systems for their conformance to a formal memory model. We propose a solution to this problem in the form of a new technique *Test model-checking* (collaborative work with Nalumasu), which can be used to verify memory systems for various formal memory models. We provide test model-checking solutions to verify memory systems for their conformance to sequential consistency and also various weaker memory models. We demonstrate the effectiveness of this technique by applying it to a number of memory systems, some of complexity comparable to industrial designs.

The main contributions of this thesis are : i) the *test model-checking* methodology which is applicable in domain of both sequential consistency and weaker memory models and ii) a verification methodology for weaker memory models and *test model-checking* solutions for these weaker memory models based on the new verification methodology.

To my parents

CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank my advisor, Prof. Ganesh Gopalakrishnan, without whose constant guidance, support and patience this work would not have been possible.

I would like to thank my committee members Prof. John Carter and Prof. Chris Myers for their valuable comments, time and suggestions.

I would like to thank W. W. Collier for his help in explaining his work, his very informative emails and for providing ARCHTEST under a no-fee, research-only license. I would like to thank Prof. Paliath Narendran for many fruitful discussions and encouraging comments. I would like to thank Prof. Al Davis and his Avalanche team for offering us the unique opportunity to work on state-of-the-art processors and busses.

I would like to thank Dr. Abdel Mokkedem for his constant encouragement, insightful comments, his hard-work during the collaborative part of this work and a most wonderful friendship that I will cherish. I would like to thank Dr. Ratan Nalumasu for all the heated and very thought provoking discussions, his insightful thoughts, his hard-work during the collaborative part of this work and his very helpful nature. I would like to thank my colleagues Ravi Hosabettu, Hans Jacobson, Mike Jones and Annette Bunker for being such great and helpful people to work with.

I would like to thank all of my friends at Utah. In particular, I would like to thank Nitin Kapur for being a constant source of help and support when I needed it most. I would like to thank Arcot Preetham, Nikhil Mehta, Sudeepto Sen, Kumar Bhairav and all of my friends who made my stay at Utah one of the most memorable time of my life.

Last and foremost, I would like to thank my parents and my family members for their constant support, encouragement and everything that they have given me.

CHAPTER 1

INTRODUCTION

1.1 Formal Memory Models

Shared memory multiprocessors are becoming increasingly important as efficient and high-performance parallel computers. A *memory model* is a contract between hardware designers of memory systems and programmers that describes how a memory system behaves in response to memory operations such as reads and writes. A memory model for a typical uniprocessor system is the classical von Neumann memory model that requires that all memory operations in a program complete in the order in which they appear in the program. Memory models for multiprocessors are usually much more involved because of complex interactions between memory operations on different processors. One of the first memory models proposed for multiprocessors is *sequential consistency* [41], which extends the uniprocessor memory model for multiprocessors in a natural and intuitive way. Sequential consistency is the strongest memory model proposed for multiprocessor memory systems. However, it restricts the use of many commonly used optimizations in the design of memory systems [28].[1] Hence, weaker memory models [1] have been proposed as an alternative to sequential consistency to achieve better performance.

With the growing gap between processor and memory system performance, memory system architects are employing increasingly aggressive optimization techniques. Such optimizations can often change the behavior of the memory system in subtle ways. As a result, it is becoming increasingly difficult for memory system designers to ensure that the optimization techniques used in their design do not alter the memory model. For example, designers typically like to know whether speculatively issued memory operations or bypass buffers used in a bus-based protocol will violate sequential consistency. We consider the problem of verifying whether a memory system provides the intended formal memory model. In spite of the central importance of this problem, all existing verification

[1] For example, read or write operations on different addresses cannot be reordered.

methods fail to provide an approach that could be directly used by designers of realistic multiprocessor systems to help them quickly detect bugs in their design level models of the memory system.

1.2 Test Model-checking

In this thesis, we propose 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 [12] 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 [14]. Nevertheless, its tests have been shown to be incisive in practice [12]. Most importantly, the formal theory of memory ordering rules developed by Collier in [14] 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, [50] 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 [21], a simple memory system, expressed in various temporal logics (by [31] in $\forall$CTL* [11] and [40] in TLA [44]) are highly complex. We do not believe that descriptions of this style scale up. On the other hand, the test model-checking method has not only been able to comfortably handle the memory system defined by the symmetric multiprocessor (SMP) bus called *Runway* [8,30] used by Hewlett-Packard in their high-end machines, but it also discovered many subtle bugs in our early Utah Runway Model (URM) that we created. Our URM includes a number of detail such as split transactions, out of order transaction completions, and even an element of speculative execution. The errors we made in capturing these detail 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 only grow in significance, regardless of whether the methods are complete or not.

Test model-checking has a number of other desirable features. It involves model-

checking a *fixed* set of safety properties for each formal memory model, that are *very nearly 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 does 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, and 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 nontrivial 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 bugs detected by test model-checking on realistic examples that are reported in Section 3.7.5 corroborate our intuition that test model-checking is indeed an effective debugging tool for memory models.

1.3 Weaker Memory Models

Sequential consistency is one of the first memory models proposed for multiprocessors. One of the main advantages of sequential consistency is that it provides programmers an intuitive, very easy to understand and very easy to program memory model. However, it is also a very strong formal memory model. Memory system providing sequential consistency cannot employ many common optimizations of uniprocessor memory system designs. For example, memory operations on different addresses cannot bypass each other. Hence, convenience of sequential consistency usually comes with a considerable lack in

performance.

As an alternative to sequential consistency, many researchers have proposed various weaker memory models. Weaker memory models relax the constraints of sequential consistency in one or more ways enabling various optimizations. Modern-day multiprocessor systems provide weaker memory models providing a considerable enhancement in performance. As a cost of weaker memory models, they are comparatively more difficult to program than when using sequential consistency. However, many existing applications can be programmed for weaker memory models without a great change. Often such weaker memory models provide mechanisms to selectively provide sequential consistency for critical memory operations. Also, there have been recent attempts at providing tools that translate a program written for sequential consistency to some weaker memory model.

The main contribution of this thesis is to provide a test model-checking methodology for verifying weaker memory models. As weaker memory models allow for various optimizations, it is likely that designers use more aggressive optimizations that could result in a violation of the intended memory model. Also, often weaker memory models differ from each other in subtle ways making it harder to detect the violation. Despite the emerging trend of weaker memory models, verification of memory systems for weaker memory systems has not been studied very much. I have developed test model-checking solutions for various weaker memory models identifying new rules, tests and test automata suitable for weaker memory models.

1.4 Related Work - Sequential Consistency

In [31], abstract interpretation [15] 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 [31]. 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, which results in a few pages of complex temporal logic formulas. These formulas are heavily dependent on the memory system being verified and need to be changed as the memory system model changes in the process of debugging.

A technique very similar to test model-checking was proposed in [46] under the section heading 'Sequential Consistency'. To give a historic perspective, our test model-checking idea originated in our attempt to answer the following two questions: (i) which memory ordering rule(s) is [46] really verifying? (ii) is this a general technique?, i.e., can other memory ordering rules be verified in the same fashion? We still have not found a satisfactory answer to the first question because the test in [46] uses only one location which, then could not make it a test for *sequential consistency*; it could plausibly be a test for coherence—which again does not correspond to what Collier formally proves in [14]. One of our contributions is that we answer these questions by elaborating on the theoretical as well as practical aspects of test model-checking.

Alur et. al [6] showed that the problem of finding if there is a sequentially consistent string σ in a regular expression r is undecidable. In the model they consider r is made up of instructions $read(p, x, v)$ and $write(p, x, v)$ where p is the process that issued the instruction, x is the shared variable, and v is the value the read instruction returns or the value to write. The problem is undecidable as the actions that follow a read instruction can depend on the value returned by a read instruction. This result is not applicable in our case, because the models we consider do not make decisions based on the value returned by a read instruction. This is detailed in Section 3.2.

In [50], 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 [34] as well as [16] 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 [26, 27], 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 proving that detailed models of memory systems are correct, while 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 pinpointing which ordering rules are violated (if any).

1.5 Related Work - Weaker Memory Models

Many researchers have proposed weaker memory models and many existing commercial systems provide weaker memory model architectures. However, in spite of the great interest in weaker memory models, hardly any effort has been targeted at developing formal verification methods for them. Most existing work either proposes new weaker memory models, presents a framework to represent various memory models in a formal and cohesive manner, or deals with how such weaker memory models can be programmed. Though most work presenting new weaker memory models suggests various hardware optimizations that could be implemented to improve the performance without violating the memory model, they do not address the problem of verifying a detailed implementation of the intended memory model.

Cache consistency [28] was one of the early weaker memory models proposed that relaxes many of sequential consistency requirements significantly. An intermediate weaker memory model *processor consistency* was presented as an attempt to combine ease of programmability with high performance implementations [28]. *Weak ordering* memory model was presented with an implementation for cache-based systems [2]. *Causal memory* [5] model was proposed, which is intermediate between sequential consistency and other earlier proposed weaker memory model *PRAM* [45]. A weaker memory model called *processor consistency* was proposed in [4]; however their definition differs from processor consistency definition of [29]. Various weaker memory models have been proposed that relax sequential consistency a great deal providing selective ordering among memory operations as necessary. *Release consistency* [24,25], *Lazy Release Consistency* [38], *Scope Consistency* [36] are some examples.

Many weaker memory models have been proposed by commercial architectures also. The SPARC V9 architecture [54] proposed *Total Store Ordering (TSO)* and *Partial Store Ordering (PSO)* weaker memory models, which can be selectively implemented. Alpha architecture [52] proposed a *Alpha Consistency* model that uses explicit fence instructions for ordering in between two instructions in a program. A survey of various weaker memory models and related issues was presented in [1, 48].

Among the various weaker memory models described above, cache consistency is the weakest. It has been shown that cache consistency is the weakest reasonable memory model [18] (under an intuitive notion of reasonable-ness).[2] A memory model weaker than

[2]It is referred as *Location Consistency* in [18].

cache consistency, named *Location Consistency* was proposed in [20].[3] However, it was shown that location consistency referred in [20] does not meet the reasonable-ness criteria of [18] and hence, in practice, we should adopt cache consistency as the memory model for a realistic implementation of [20]'s location consistency [18].

There have been many approaches proposing a formal framework to define and compare various weaker memory models [17,19,22,32,35,39,51]. Approaches such as [22] adopt a hardware-centric view: they define memory models in terms of how memory accesses can be executed by the system. Approaches such as [32, 39, 51] define memory models in terms of partial orders on memory access events. In contrast to the hardware-centric view, approaches such as [19, 35] adopt a programmer-centric view and define memory models in terms of the interface provided to the programmer by specifying possible results of an execution.[4]

As weaker memory models relax sequential consistency in various order, a parallel program that runs correctly on a sequentially consistent memory system may not execute correctly on memory systems obeying weaker memory models. For example, a typical critical section code that executes correctly under sequential consistency may not execute as expected under a weaker memory model. Most weaker memory models, e.g., [7], provide ways to program the proposed weaker memory model. How various weaker memory models such as TSO, PSO, etc. could be programmed is presented in [23]. How Lamport's Bakery algorithm and Peterson's algorithm for critical section can be programmed under the Alpha memory model is presented in [17]. Friendman [17] describes several techniques for turning programs written for sequential consistency into programs that work for the weaker memory model proposed. He also developed a general method that transforms any given noncooperative algorithm for the mutual exclusion problem based on sequential consistency into a correct algorithm based on Alpha consistency. We can see that although weaker memory models make it harder to program a multiprocessor machine in general, there have been various approaches to overcome this difficulty.

---

[3] Of course, location consistency referred in [20] is different from location consistency referred in [18]. This is just one more example of all too ubiquitous name-overloading in memory models literature.

[4] Also known as computation-centric view.

1.6 Our Contributions

The main contributions of this thesis are as follows.

- A new formal verification methodology *test model-checking* to verify multiprocessor memory system's conformance to intended memory model.
- We showed how sequential consistency memory model could be checked using test model-checking technique. We applied test model-checking technique to a number of examples to demonstrate its strengths and applicability.
- We proposed test model-checking solutions for various weaker memory models. We proposed various new rules to precisely capture the behavior of weaker memory models. We demonstrated how various commercial weaker memory models could be represented using these rules. We proposed various new tests which could be used to verify for conformance to weaker memory models. We proposed test model-checking solutions for each of such test.

1.7 Road-map

Chapter 2 describes formal memory models and the formal framework of ARCHTEST to specify them.[5] Chapter 3 describes test model-checking method developed for sequential consistency.[6] Chapter 4 describes how to extend test model-checking technique for various weaker memory models.[7] Chapter 5 gives conclusions and directions for future work.

[5] This chapter includes the work done by W.W.Collier. [12]

[6] The work presented in this chapter was done jointly by Ratan Nalumasu, Abdel Mokkedem and the author.

[7] The work presented in this chapter is a sole contribution of the author.

CHAPTER 2

FORMAL MEMORY MODELS AND ARCHTEST

This chapter introduces formal memory models and describes example formal memory models. Also, the formal framework behind ARCHTEST methodology is presented that can be used to specify and reason about formal memory models. The formal framework of ARCHTEST was developed by William W. Collier [12].

2.1 Formal Memory Model

A memory model is a description of the behavior of a multiprocessor memory system in response to various memory operations such as reads and writes. It is a contract between hardware and software that precisely defines all possible behaviors of hardware in response to requests from software. Conceptually, it could be visualized as shown in Figure 2.1. Processes P_1, P_2....P_n issues various memory operations to the memory system. Typically, these memory operations are reads and writes to various memory locations. A memory model defines the set of possible values that the memory system can generate in response to these operations from various processes. A *formal memory model* is a description of a memory model in a formal framework. Often, we use the term memory model and formal memory model interchangeably in our discussion.

Some memory models are *stronger* than other memory models, i.e., they impose stricter restrictions on the set of possible responses of the memory system in response to memory operations. Some memory models are *weaker* than other memory models, i.e., they impose fewer restrictions on the set of possible responses of the memory system in response to memory operations. *Sequential consistency* is one of the strongest memory models defined for multiprocessors. We discuss the sequential consistency memory model below and then we discuss a memory model that is weaker than sequential consistency.

2.1.1 Sequential Consistency

Sequential consistency extends the classical von Neumann uniprocessor memory model to multiprocessors in a natural and intuitive way. It was formally defined by Lamport [41] as follows.

> [A multiprocessor system is *sequential consistent* if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Operationally, sequentially consistency can be viewed as shown in Figure 2.2. As shown in Figure 2.2, multiple processors share a single ported memory. All memory operations must be done by each processor one at a time by accessing the memory via the single port. As a result, all memory operations by each processor are executed in program order and all writes become "instantly visible" to all processors.

2.1.2 SPARC V9 Total Sorted Order (TSO)

The SPARC V9 architecture proposes a weaker memory model Total Store Order (TSO) [54]. An operational model of TSO is shown in Figure 2.3. This architectural model is similar to that of sequential consistency in that it has a single port memory and only one processor accesses the memory at a time. In addition, each processor has a write buffer associated with it. All write operations are enqueued in the write buffer in program order. These write operations are completed by updating the memory when the processor can access the memory. Read operations are allowed to bypass a write operation to a different address. If a write operation for the same address as a read operation is enqueued in the queue then the read operation completes by reading the value associated with the write operation in the queue. Otherwise, the read operation bypasses the write buffer and completes by reading the value from the memory. Note that all reads complete in program order and no write is allowed to bypass a read.

2.2 ARCHTEST

We saw brief descriptions of two memory models in the previous section. Though these descriptions are very useful to understand these memory models, a mathematical *formal framework* is needed to *precisely* capture the definition of these memory models and to enable us to *reason* about these memory models effectively and formally. ARCHTEST has a formal framework in which memory models can be defined and could be reasoned

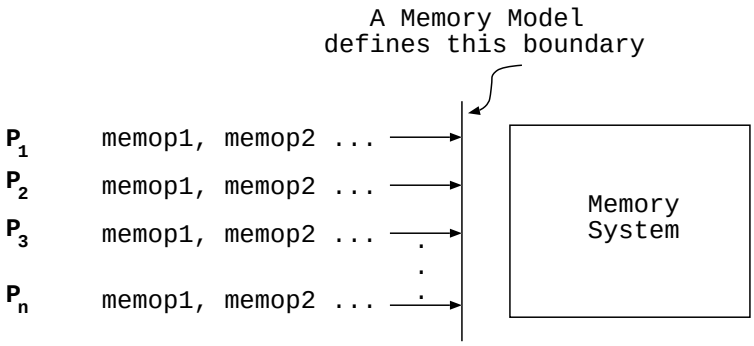

Figure 2.1. Memory model : a contract between hardware and software.

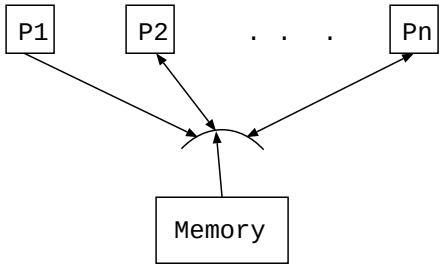

Figure 2.2. Operational view of sequential consistency.

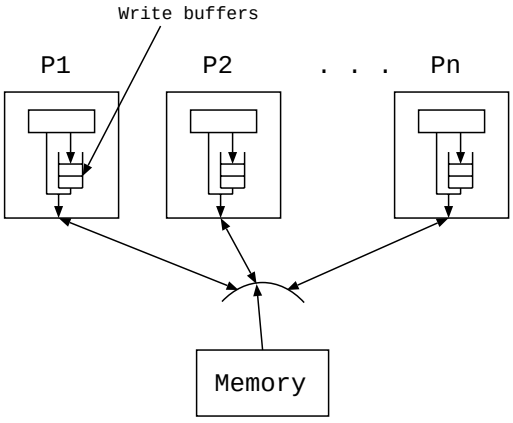

Figure 2.3. SPARC V9 total sorted order operational view.

about. We describe the formal framework of ARCHTEST below. It is presented in [14] in complete detail.

ARCHTEST formally defines a memory model using *architecture rules* that characterize the behavior of the memory system of multiprocessors. Each such architecture rule represents a particular facet of the memory system behavior. A combination of various such rules captures different aspects of the memory system behavior acting together. The memory model is defined as a conjunction of various such rules. Now, we see how ARCHTEST formally defines these rules and the framework needed to do so.

A multiprocessing system consists of a set of processes. All processes share a *global operand* space and each process also has its own *local operands*. The memory operations are either *read* or *write* performed by a process on a global or a local operand. [1] When a write is performed by a process on a global operand, different processes may observe the operand assume the new value of the write at different times. To capture this distinction, we associate one *store* per *each process*. Each process P_i has its own store S_i. Conceptually, each process P_i has a local copy of **all** *global operands* in its own store S_i. A write operation on a global operand A by a process P_i becoming visible to a process P_j is modeled by the write operation updating the value of the operand A in the store S_j.

2.2.1 Execution

Now, we define the notion of an *execution*. An execution represents an instance of a set of programs executing on a computer. One such execution is shown in Figure 2.4. For each operand, there is an initial value of the operand in each store. For each operand, there is a final value of the operand in each store.[2] Each process has a sequence of labeled assignment statements.[3]

---

[1] The formal framework of ARCHTEST considers only reads and write memory operations. Other memory operations such as barriers, etc. are not discussed in the framework definition in [14]. However, we can extend it to include other memory operations, as will be discussed later.

[2] When the initial or the final value of the operand is the same in all stores, the store is not explicitly specified. Also, local operands are typically named U, V, X, Y, etc. and global operands are typically named A, B, C, D, etc.

[3] Control statements are not considered in this formal framework. As per ARCHTEST, *statements that govern the flow of control are not essential* [14, 32].

Initially, $A = B = X = Y = 0$

P_1	P_2
$L_1 : X := B;$	$L_1 : Y := A;$
$L_2 : A := 1;$	$L_2 : B := 1;$

Finally, $A = B = 1, X = Y = 0$

Figure 2.4. An execution.

2.2.2 Event

We introduce the notion of an event that is intended to represent the activity of each process of accessing various store for the purpose of reading or writing operands. For each statement in an execution there is one read event per source operand and one write event per store. For example, consider the statement $X := B$ labeled L_1 of process P_1 of Figure 2.4. There is one read event with the source operand B and store S_1 associated with this statement - represented as $(P_1, L_1, R, 0, B, S_1)$. This read event represented that a read from operand B was executed by the instruction labeled L_1 in process P_1 which read value 0 from store S_1 There are also two write events with the sink operand X, each with store S_1 and S_2 respectively - represented as $(P_1, L_1, W, 0, X, S_1)$ and $(P_1, L_1, W, 0, X, S_2)$. [4] The two write events for the two stores represent the write becoming visible to each processor.

In general, an event is a tuple of the form (P, L, A, V, O, S) where,

- P is the name of the process in which the statement occurs,
- L is the label of the statement,
- A is either "R" for a read event or "W" for a write event,
- V is the value of the operand associated with the event: in case of read it is the value the read returns and in case of write it is the value being written,
- O is either a source operand (for read event) or a sink operand (for a write event) and

[4] Note that in the event of a local operand such as X the write events associated with other stores are not of any significance. They are defined however. Also, note that the write events for global operands such as A for various different stores are of significance and each such write event represents when this write to a global operand becomes visible to other processes.

- S is for a read event, the name of the store for the process in which the statement occurs, or, for a write event, the name of any one of the stores in the execution.

All the events associated with the execution in Figure 2.4 are shown below.

$$\begin{array}{ll} (P_1, L_1, R, 0, B, S_1) & (P_2, L_1, R, 0, A, S_2) \\ (P_1, L_1, W, 0, X, S_1) & (P_2, L_1, W, 0, Y, S_1) \\ (P_1, L_1, W, 0, X, S_2) & (P_2, L_1, W, 0, Y, S_2) \\ (P_1, L_2, R, 1, 1, S_1) & (P_2, L_2, R, 1, 1, S_2) \\ (P_1, L_2, W, 1, A, S_1) & (P_2, L_2, W, 1, B, S_1) \\ (P_1, L_2, W, 1, A, S_2) & (P_2, L_2, W, 1, B, S_2) \end{array}$$

ARCHTEST formal framework is built considering the notion of an execution and events associated with it as building blocks. Reasoning regarding the behavior of a multiprocessor can be done in terms of the executions it exhibits and the relationship between the events associated with the executions it allows and disallows.

2.2.3 Graph Set

Now, we introduce the notion of a graph set. We define graph and graph set and then establish its relevance and significance in the context of formal framework for defining memory models.

A graph G is a tuple (S, R) where S is a finite set of nodes and R is a finite set of edges or arcs. We consider directed graphs only, i.e., each edge is directed from the source node to destination node. Each edge has a label associated with it. The sum of two graphs $G_1 = (S, R_1)$ and $G_2 = (S, R_2)$ with the same set of nodes is defined as a graph $G = (S, R_1 \cup R_2)$. The sum of two graphs represent a graph that has edges belonging to both the graphs. A *circuit* or *cycle* is a sequence of arcs in a graph whose starting and ending node are same.

A *graph set* GS is a set of graphs such that all of the graphs in the set have the same set of nodes. A graph set is *circuit-full* if every graph in the graph set contains a circuit. A graph set is *partly circuit-free* if at least one graph in the graph set contains no circuit. If GS_1 and GS_2 are graph sets with the same set of nodes, then the graph set product $GS_1 * GS_2$ is a graph set consisting of exactly those graph sums G_1 and G_2 such that G_1 is in GS_1 and G_2 is in GS_2. A graph set product of two graph sets represents a graph set that includes all pairwise sums of graphs in the two graph sets.

2.2.4 Architecture

Now, we see how we can use graph set to formally define architecture and architecture rules. An architecture consists of a set of architecture rules. $A(R_1, R_2, R_3, ..., R_n)$ denotes the architecture consisting of architecture rules $R_1, R_2, ..., R_n$. Each architecture rule represents a facet of the memory model that imposes a restriction on the events of an execution in terms of which event occurs before/after another event. An architecture $A(R_1)$ consisting of a single rule R_1 maps the events of an execution into a graph set. The mapping is different for each different rule. For a given execution, the graphs in the graph set have as nodes the events of an execution. Within each graph of the graph set, an arc from event E_1 to an event E_2 represents E_1 occurring earlier in time than E_2. The intent of having an architecture rule define a graph set is to represent concretely and explicitly via the graph set all of the possible orderings of events permitted by the architecture rule. For some rules the graph set may contain only one graph whereas for some rules the graph set may contain a large number of graphs. The graph set of an execution under architecture $A(R_1, R_2)$ is defined to be the product of the graph sets of the execution under architecture $A(R_1)$ and $A(R_2)$. Each member of the graph set of an execution under architecture $A(R_1, R_2)$ has the arcs caused by both rules R_1 and R_2. The individual pairwise sums of graphs in the graph set under $A(R_1)$ and $A(R_2)$ ensures that the graph set under $A(R_1, R_2)$ includes all possible "interactions" between rules R_1 and R_2. We often do not distinguish between an architecture and the graph set it induces for the sake of convenience. So, we can denote the above definition by $A(R_1, R_2) = A(R_1) * A(R_2)$. Note that even if $A(R_1)$ is circuit-free and $A(R_2)$ is circuit-free, it is possible that $A(R_1, R_2)$ can have a graph in it that has a circuit that represents conflicting restrictions of two architecture rules.

An execution E is said to obey an architecture if the graph set induced by the architecture on the events of E is partly circuit-free. If the graph set is partly circuit-free, then the events in the circuit-free graph of the graph set can be linearized, and this linear ordering can be viewed as being the order in time in which the events occurred on a machine obeying the architecture. Such a linear ordering represents an explanation of how this machine can exhibit the execution. Architecture A_1 implies architecture A_2, written $A_1 \Rightarrow A_2$, if whenever an execution obeys A_1, it also obeys A_2. An execution E does not obey an architecture if the graph set induced by the architecture on the events of E is not partly circuit-free, i.e., each member of the graph set has a circuit in it.

Intuitively, this represents that there is no explanation of how this machine can exhibit the execution while still obeying all the architecture rules, i.e., if the machine does exhibit the execution that one of the architecture rules is violated on this machine.

Now, we formally define a number of architecture rules defined by ARCHTEST. Most rules could be categorized into two segments : 1) rules related to *ordering* between events and, 2) rules related to *atomicity* of events. We use the definitions of these architecture rules to define memory models later. For each rule R being defined, we describe how the graph set for an execution under $A(R)$ is defined. For each rule R begin defined, we also specify transition templates associated with each rule. Each such transition template identifies the information that is preserved when moving from one event to another event along an arc in a graph.

2.2.5 Rule of Read Order - *RO*

The rule of read order RO requires that two read events from the same process occur in the program order. To form the single graph in the graph set for an execution E under $A(RO)$, draw an arc, labeled $<_{RO}$, from E_1 to E_2 if

1. event E_1 belongs to a statement which occurs before the statement of event E_2, denoted by $E_1 <_o E_2$ in the program and,
2. event E_1 and E_2 are both read events.[5]

The transition template for RO is

$$(P, L, R, V, O, S) <_{RO} (=, -, R, -, -, =)$$

This transition template indicates that an $<_{RO}$ arc is between two events which belong to the same process, which are read events and which have the same store components. The fact that the second event must belong to a statement that occurs after the statement corresponding to the first event is implicit. There is also a subrule of rule of RO, denoted ROO (read order on same operand). Rule of ROO requires that both the read events have the same operand component.

[5]Sometimes, we denote the label just as RO when the context is clear enough to avoid any confusion.

2.2.6 Rule of Write Order - WO

The rule of write order WO requires that two write events from the same process occur in the program order. To form the single graph in the graph set for an execution E under $A(WO)$, draw an arc, labeled $<_{WO}$, from E_1 to E_2 if

1. $E_1 <_o E_2$ and,
2. event E_1 and E_2 are both write events.

The transition template for WO is

$$(P, L, W, V, O, S) <_{WO} (=, -, W, -, -, -)$$

An arc labeled $<_{WO}$ occurs between write events in the same process, which may or may not have same store components or operands. There is also a subrule of rule of WO, denoted WOS (write order on same store). Rule of WOS requires that both the write events have the same store component.

2.2.7 Rule of Program Order - PO

The rule of program order PO requires that both all events, i.e., reads or writes, from the same process occur in the program order. To form the single graph in the graph set for an execution E under $A(PO)$, draw an arc, labeled $<_{PO}$, from E_1 to E_2 if

1. $E_1 <_o E_2$.

The transition template for PO is

$$(P, L, A, V, O, S) <_{PO} (=, -, -, -, -, -)$$

An arc labeled $<_{PO}$ occurs between any two events in the same process: each event could be either read or write. It could be seen that RO and WO are *subrules* of PO. Similarly we can define RW and WR subrules which respectively imposes ordering between a read-write pair and write-read pair of events.

2.2.8 Rule of Uniprocessor Order - UPO

The rule of uniprocessor order UPO consists of three subrules : UWR, URW and UWW. Each of these rules imposes constraints on the events with the same process,

operand and store components to occur in the program order. UWR, URW and UWW represent the *write after read*, *read after write* and *write after write* hazards in a uniprocessor system. The operation type of each event is dependent upon the subrule.

2.2.8.1 UWR

To form the single graph in the graph set for an execution E under $A(UWR)$, draw an arc, labeled $<_{UWR}$, from write event W_1 to read event R_1 if

1. $W_1 <_o R_1$ and,
2. W_1 and R_1 are in the same process and have the same operand and store components.

The transition template for UWR is

$$(P, L, W, V, O, S) <_{UWR} (=, -, R, -, =, =)$$

2.2.8.2 URW

To form the single graph in the graph set for an execution E under $A(URW)$, draw an arc, labeled $<_{URW}$, from read event R_1 to write event W_1 if

1. $R_1 <_o W_1$ and,
2. R_1 and W_1 are in the same process and have the same operand and store components.

The transition template for URW is

$$(P, L, R, V, O, S) <_{URW} (=, -, W, -, =, =)$$

2.2.8.3 UWW

To form the single graph in the graph set for an execution E under $A(UWW)$, draw an arc, labeled $<_{UWW}$, from write event W_1 to write event W_2 if

1. $W_1 <_o W_2$ and,
2. W_1 and W_2 are in the same process and have the same operand and store components.

The transition template for UWW is

$$(P, L, W, V, O, S) <_{UWW} (=, -, W, -, =, =)$$

The rule of uniprocessor order UPO is defined as a conjunction of all the subrules: $A(UPO) = A(UWR, URW, UWW)$.

The transition template for UPO is

$$(P, L, A, V, O, S) <_{UPO} (=, -, -, -, =, =)$$

The implicit information in the transition template is the fact that both the events are not read events.

2.2.9 Rule of Write Atomicity - WA

The rule of write atomicity WA captures the atomicity of a write operation, i.e., the write becomes visible to all the processes instantaneously. The atomicity is modeled by requiring that all the write events for one write operation form an atomic set $W_1, W_2, ..., W_n$. Any event that occurs before (or after) some event W_i must also occur before (and respectively. after) all other W_j events. In other words, if $W_1, W_2, ..., W_n$ form an atomic set then for any event X not in this atomic set, the following is true.

1. Either $X <_{WA} W_1$ or $X >_{WA} W_1$, i.e., each event X either occurs before the events in the atomic set or it occurs after the events in the atomic set.

2. If $X <_{WA} W_i$ for some i then $X <_{WA} W_j : \forall j : 1 \leq j \leq n$. Similarly, if $W_i <_{WA} X$ for some i then $W_j <_{WA} X : \forall j : 1 \leq j \leq n$.

The transition template for WA is

$$(P, L, A, V, O, S) <_{WA} (-, -, -, -, -, -)$$

Consider an execution that obeys an architecture that is write atomic. Let W_1 and W_2 be write events for one write operation in different stores and let E be any event. If it is the case that $E < W_1$ then it must also be that $E <_{WA} W_1$ and so $E <_{WA} W_2$. We often abbreviate this as $E < W_1 =_{WA} W_2$ for the sake of convenience. The transition template for $=_{WA}$ can be viewed as :

$$(P, L, W, V, O, S) =_{WA} (=, =, W, =, =, -)$$

It is important to notice that $=_{WA}$ is a matter of notational convenience and is not an architecture rule. We often deal with atomicity using $=_{WA}$ arc only because it is much easier to reason about using its transition template.

2.2.10 Rule of Consistency - CON

The rule of consistency CON requires that writes to an operand are visible to all the processes in the same order. It requires that if statement L_1 writes to an operand O in store S_a before statement L_2 (not necessarily from the same process) writes to operand O in store S_a, then for any other store S_b, statement L_1 writes to operand O in store S_b before statement L_2 writes into operand O in store S_b.

To form the graph set for an execution E under $A(CON)$, the following steps are performed.

1. For every pair of statements, L_1 and L_2 in E, with the same sink operand, form two graphs. In the first graph draw an arc labeled $<_{CON}$ from each write event in L_1 to the write event in L_2 with the same store component. In the second graph draw an arc labeled $<_{CON}$ from each write event in L_2 to the write event in L_1 with the same store component.

2. Form the product of the pairs of the graphs constructed in Step 1.

3. Delete from the product graph set all graphs containing a circuit. This diminished graph set constitutes the graph set for E under $A(CON)$.

The transition template for CON is

$$(P, L, W, V, O, S) <_{CON} (-, -, W, -, =, =)$$

Consider the following execution :

$$\textit{Initially,} \quad A = B = 0$$

$$\begin{array}{cc} P_1 & P_2 \\ L_1 : A := 1; & L_1 : B := 1; \\ L_2 : B := 2; & L_2 : A := 2; \end{array}$$

$$\textit{Finally,} \quad A = B = 2$$

We illustrate how the graph set for above execution under $A(CON)$ is constructed.

1. In the first step, we create two graph sets GS_1 and GS_2. Each graph in the two graph sets represents the order in which write events for A and B are seen by each process. Each graph set has two graphs representing the order in which writes are seen. GS_1 and GS_2 are shown in Figure 2.5.

2. In the second step, we form the product of the pairs of the graphs resulting in the graph set with four graphs.

3. Deleting from the product graph set all graphs containing a circuit results in the graph set with four graphs. This is the graph set for the above executing under $A(CON)$.

It could be shown that the rule of atomicity WA is strictly stronger than the rule of consistency CON.[6]

2.2.11 Rule of Computation - CMP

The rule of computation CMP is the most basic and fundamental rule of any of the rules. It describes how the value of an operand is calculated from other operands and how the value of an operand is passed from one process to another. It consists of four subrules : SRW, CWR, CRW and CWW.

---

[6] A rule R is said to be stronger than a rule R' if an execution obeys $A(R_1, R_2, ..., R_n, R)$ then it also obeys $A(R_1, R_2, ..., R_n, R')$. Intuitively, if R is stronger than R' then whenever the memory system obeys rule R, it also obeys rule R'. A rule R is said to be strictly stronger than a rule R' if R is stronger than R' and there exists at least one execution which obeys an architecture $A(R_1, R_2, ..., R_n, R')$ but does not obey $A(R_1, R_2, ..., R_n, R)$.

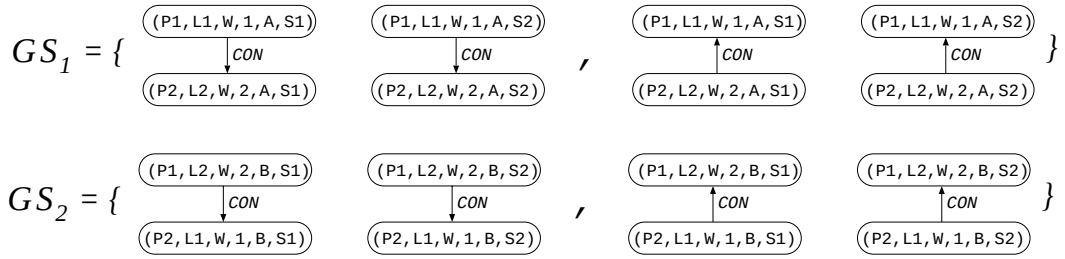

Figure 2.5. Graph set for $A(CON)$: GS_1 and GS_2 in the first step of graph set creation.

2.2.11.1 *SRW*

The subrule *SRW* requires that a read event in a statement must happen before the write event of the same statement. An arc, labeled $<_{SRW}$, leads from a read event R_1 in a statement to a write event, say W_1 if R_1 and W_1 events belong to the same statement.

The transition template for *SRW* is

$$(P, L, R, V, O, S) <_{SRW} (=, =, W, -, -, -)$$

2.2.11.2 *CWR*

The subrule *CWR* requires that if a read event reads the value written by another write event then the write event must occur before the read event. An arc, labeled $<_{CWR}$, leads from a write event W_1 to a read event R_1 where the two events have the same operand and store components. The two events and the arc connecting them represent the case in which a write operation stores a value into an operand in a store and a subsequent read operation fetches the value from the same operand in the same store.

The transition template for *CWR* is

$$(P, L, W, V, O, S) <_{CWR} (-, -, R, =, =, =)$$

2.2.11.3 *CRW*

The subrule *CRW* requires that if a read event reads the value from a store before a write event updates the store then the read event must occur before the write event. An arc, labeled $<_{CRW}$, leads from a read event R_1 to a write event W_1 where the two events have the same operand and store components. The two events and the arc connecting them represent the case in which a read operation fetches the value of an operand in a store prior to the time a subsequent write operation stores a (possibly different) value in the same operand and same store.

The transition template for *CRW* is

$$(P, L, R, V, O, S) <_{CRW} (-, -, W, -, =, =)$$

2.2.11.4 *CWW*

The subrule *CWW* requires that if a write event W_1 writes a value in a store before another write event W_2 updates the value in the store then the W_1 must occur before W_2. An arc, labeled $<_{CWW}$, leads from a write event W_1 to a write event W_2 where the

two events have the same operand and store components. The two events and the arc connecting them represent the case in which the first write operation writes the value of an operand in a store prior to the time a second write operation writes a (possibly different) value in the same operand and same store.

The transition template for CWW is

$$(P, L, W, V, O, S) <_{CWW} (-, -, W, -, =, =)$$

The process of creating a graph set incorporating all the aspects of the four subrules is intricate and the interested reader is referred to [14] for the detail. The rule of CMP is an architecture rule that captures the most basic and fundamental restrictions that the memory model can impose. In the absence of such a rule, it would be almost impossible to meaningfully argue about the behavior of the machine as read operations can return random values and write operations can write random values. Hence, all memory models in our discussion are implicitly assumed to obey the rule of CMP.

Now, we are ready to formally define a memory model in terms of the framework developed so far. A memory model is simply an architecture that consists of a set of architecture rules. In this framework, Sequential consistency can be defined as the architecture $A(CMP, PO, WA)$. The conventional von Neumann memory model for the uniprocessor memory system can be defined as the architecture $A(CMP, UPO)$.

2.2.12 An Execution Violating $A(CMP, RO, WO)$

Now, we shall see how we can employ the formal framework developed so far to effectively reason about the behavior of machines. Consider the following execution.

$$\begin{array}{ll} \textit{Initially,} & A = X = Y = 0 \\ P_1 & P_2 \\ L_1 : A := 1 & L_1 : X := A \\ L_2 : A := 2 & L_2 : Y := A \\ \textit{Finally,} & A = 2, X = 2, Y = 1 \end{array}$$

Intuitively, if the machine obeys architecture read and write order (and computation too), then it is not possible for X to have value 2 and Y to have value 1. The only values possible for X and Y are $(0,0), (0,1), (0,2)(1,1), (1,2)$ or $(2,2)$. Hence, if the machine indeed shows that X and Y have values $(2,1)$ then it must not be obeying either the read

or write order. [7]

Formally, we can show that the above execution does not obey the architecture $A(CMP, RO, WO)$. To show that, consider the graph set for this execution under the architecture $A(CMP, RO, WO)$. We shall show that this graph set is not partially circuit-free, i.e., each member of this graph set involves a cycle.

Consider any member of the graph set for this execution under $A(CMP, RO, WO)$. The events involved in this execution are :

$$\begin{array}{ll} (P_1, L_1, R, 1, 1, S_1) & (P_2, L_1, R, 2, A, S_2) \\ (P_1, L_1, W, 1, A, S_1) & (P_2, L_1, W, 2, X, S_2) \\ (P_1, L_1, W, 1, A, S_2) & (P_2, L_1, W, 2, X, S_1) \\ (P_1, L_2, R, 2, 2, S_1) & (P_2, L_2, R, 1, A, S_2) \\ (P_1, L_2, W, 2, A, S_1) & (P_2, L_2, W, 1, Y, S_2) \\ (P_1, L_2, W, 2, A, S_2) & (P_2, L_2, W, 1, Y, S_1) \end{array}$$

Each member of the graph set involves the events and the arcs shown in Figure 2.6. There are other events and other arcs in this graph we are considering, however we have shown an interesting subgraph of this graph only. The WO and RO arcs follow from the definition of these architecture rules. The two CWR arcs represent how the read events of P_2 to operand A obtain values 2 and 1. The interesting arc is the CRW arc between $(P_2, L_2, R, 1, A, S_2)$ and $(P_2, L_2, W, 2, A, S_2)$. This arc is induced because either $(P_1, L_2, W, 2, A, S_2)$ event occurs before $(P_2, L_2, R, 1, A, S_2)$ or $(P_2, L_2, R, 1, A, S_2)$ event occurs before $(P_1, L_2, W, 2, A, S_2)$. If $(P_1, L_2, W, 2, A, S_2)$ event occurs before $(P_2, L_2, R, 1, A, S_2)$ then the value of the event $(P_2, L_2, R, 1, A, S_2)$ must be 2 because $(P_2, L_2, W, 2, A, S_2)$ occurs after $(P_2, L_2, W, 1, A, S_2)$ due to WO.[8] We see that each such member of the graph set involves the following cycle.

$$\begin{array}{lll} (P_2, L_2, R, 2, A, S_2) & <_{RO} & (P_2, L_2, R, 1, A, S_2) \\ & <_{CRW} & (P_1, L_2, W, 2, A, S_2) \\ & <_{CWR} & (P_2, L_1, R, 2, A, S_2) \end{array}$$

This cycle is shown by bold edges in Figure 2.6.

We defined the formal framework of ARCHTEST in detail and saw how it can be used to define memory models and how it is useful in reasoning formally about architecture rules that memory systems are supposed to obey. ARCHTEST provides tests for some

[7] Remember that we always assume that the machine obeys rule of computation implicitly. It is really hard to imagine a machine that does not obey the rule of computation.

[8] We shall see such reasoning for CRW arcs multiple times in our discussions.

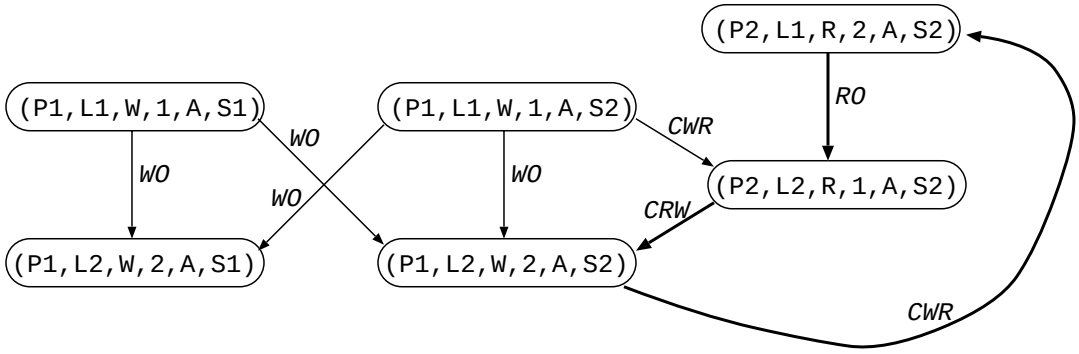

Figure 2.6. A cycle corresponding to violation of $A(CMP, RO, WO)$.

combination of architecture rules which can be run on machines to see if the machine exhibit a violation of such architecture rules. Each such test checks for an architecture say $A(R_1, R_2, ..., R_n)$. There is a condition with each test on the outcomes of these tests that is checked to see if the architecture is obeyed. If the condition is violated then the machine does not obey architecture $A(R_1, R_2, , , , , R_n)$, i.e., the machine violates at least one of the architecture rules $R_1, R_2, ..., R_n$. Now, we discuss some of the ARCHTEST tests.

2.2.13 $Test_{\text{ROWO}}$: ARCHTEST Test for $A(CMP, RO, WO)$

The test of ARCHTEST for architecture $A(CMP, RO, WO)$, is shown in Figure 2.7. Process P_1 executes a sequence of write instructions (intended to check for WO), and P_2 executes a sequence of read instruction (intended to check for RO). If the memory system correctly realizes $A(CMP, RO, WO)$, then Condition 1 produces a positive outcome:

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

If MONOTONIC condition is violated then at least one of the CMP, RO and WO rules is violated. Now, we provide a proof for the correctness of this test.

$$
\begin{array}{cc}
\textit{Initially,} & A = 0 \\
P_1 & P_2 \\
L_1 : A := 1; & X[1] := A; \\
L_2 : A := 2; & X[2] := A; \\
L_3 : A := 3; & X[3] := A; \\
\cdots & \cdots \\
L_k : A := k & X[k] := A;
\end{array}
$$

Figure 2.7. $Test_{\text{ROWO}}$: ARCHTEST test for $A(CMP, RO, WO)$.

THEOREM 2.1 If an execution violates the condition 1 then the execution does not obey $A(CMP, RO, WO)$.

Proof : Consider a violation of condition 1 during an execution of $Test_{\text{ROWO}}$.

$$\begin{array}{lll} & \exists i: & X[i] > X[i+1] \\ \iff & \exists i, \alpha, \beta: & X[i] = \alpha \wedge X[i+1] = \beta \wedge \alpha > \beta \end{array}$$

The statements of $Test_{\text{ROWO}}$ associated with the values involved in the condition violation shown above are :

P_1	P_2
$L_\beta : A := \beta;$	$\ldots$
$\ldots$	$\ldots$
$L_\alpha : A := \alpha;$	$\ldots$
$\ldots$	$\ldots$
	$L_i : X[i] := A;$
	$L_{i+1} : X[i+1] := A;$
$\ldots$	$\ldots$

Every member of the graph set for this execution under $A(CMP, RO, WO)$ involves a cycle shown in Figure 2.8. This cycle is very similar to that of the execution described earlier. Hence, the execution does not obey $A(CMP, RO, WO)$.

□

2.2.14 $Test_{\text{WA}}$: ARCHTEST Test for $A(CMP, RO, WO, WA)$

$Test_{\text{WA}}$, shown in Figure 2.9 tests for $A(CMP, RO, WO, WA)$, with the conditions checked being: (i) the MONOTONIC condition (suitably modified for arrays U, V, X, Y), and (ii) ATOMIC, which is:

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 became visible in different orders to P_2 and P_3. Now, we provide a proof for the correctness of this test.

THEOREM 2.2 If an execution violates the condition 2 then the execution does not obey $A(CMP, RO, WO, WA)$.

Proof : Consider a violation of condition 2 during an execution of $Test_{\text{WA}}$.

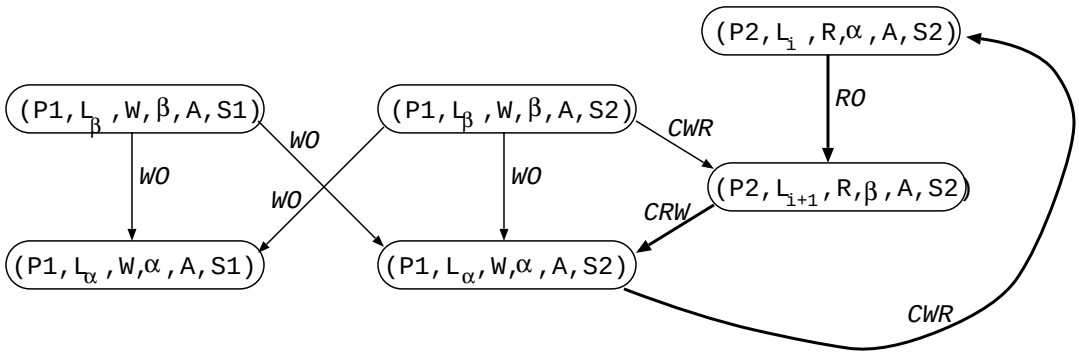

Figure 2.8. A cycle showing violation of $A(CMP, RO, WO)$ corresponding to a violation of condition 1 of $Test_{\text{ROWO}}$.

Initially, $A = B = 0$

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

Figure 2.9. $Test_{\text{WA}}$: ARCHTEST test for $A(CMP, RO, WO, WA)$.

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

The statements of $Test_{\text{WA}}$ associated with the values involved in the condition violation shown above are :

P_1	P_2	P_3	P_4
$L_\alpha : A := \alpha;$	$L_{A_i} : U[i] := A;$	$L_{B_j} : X[j] := B;$	$L_\beta : B := \beta;$
$L_{\alpha'} : A := \alpha';$	$L_{B_i} : V[i] := B;$	$L_{A_j} : Y[j] := A;$	$L_{\beta'} : B := \beta';$

Every member of the graph set for this execution under $A(CMP, RO, WO, WA)$ involves a cycle shown in Figure 2.10. Note that write atomicity is indicated by representing the atomic set of all write events with a single event. This single event is shown with the store component as $S*$. We can view this single event as a representation of all the write events with $=_{WA}$ arc between them. Hence, the execution does not obey $A(CMP, RO, WO, WA)$.

□

2.2.15 $Test_{\text{PO}}$: ARCHTEST Test for $A(CMP, PO)$

$Test_{\text{PO}}$, shown in Figure 2.11 tests for $A(CMP, PO)$, with the conditions checked being: (i) the MONOTONIC condition (suitably modified for arrays X, Y), and (ii) PO_CROSS, which is:

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

Now, we provide a proof for the correctness of this test.

THEOREM 2.3 If an execution violates the condition 3 then the execution does not obey $A(CMP, PO)$.

Proof : Consider a violation of condition 3 during an execution of $Test_{\text{PO}}$.

$$\begin{array}{ll} \exists i,j: & (X[i] < j \wedge Y[j] < i) \vee \\ & (X[i] > j \wedge Y[j] > i) \end{array}$$

There are two cases in which the condition violation can occur. We consider each case separately.

Case I: Consider the first case of the condition violation.

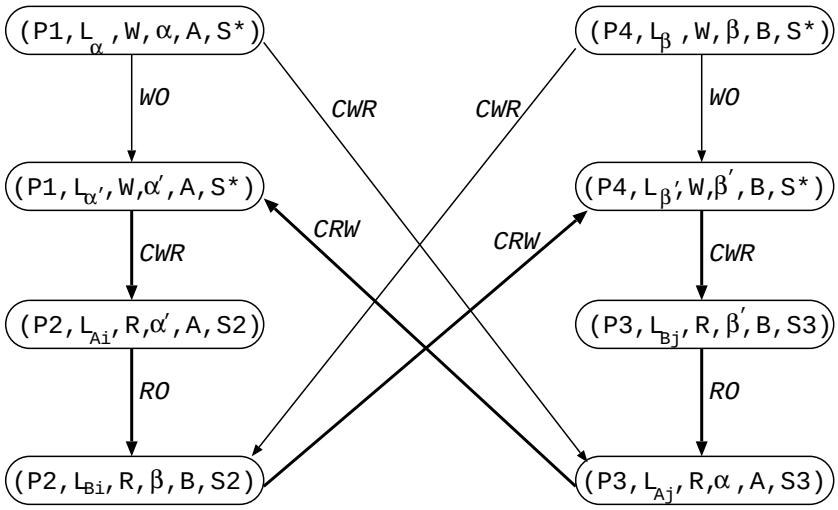

Figure 2.10. A cycle showing violation of $A(CMP, RO, WO, WA)$ corresponding to a violation of condition 2 of $Test_{\text{WA}}$: WA is indicated by representing the atomic set of write events with a single event which has $S*$ as the store component.

$$\textit{Initially, } A = B = 0$$

P_1	P_2
$L_{11} : A := 1;$	$L_{11} : B := 1;$
$L_{12} : Y[1] := B;$	$L_{12} : X[1] := A;$
$L_{21} : A := 2;$	$L_{21} : B := 2;$
$L_{22} : Y[2] := B;$	$L_{22} : X[2] := A;$
$\ldots$	$\ldots$
$L_{k1} : A := k;$	$L_{k1} : B := k;$
$L_{k2} : Y[k] := B;$	$L_{k2} : X[k] := A;$

Figure 2.11. $Test_{\text{PO}}$: ARCHTEST test for $A(CMP, PO)$.

$$\exists i, j : X[i] < j \wedge Y[j] < i \iff \exists i, j, \alpha, \beta : X[i] = \alpha \wedge \alpha < j \wedge Y[j] = \beta \wedge \beta < i$$

The statements of $Test_{\text{PO}}$ associated with the values involved in the condition violation shown above are :

P_1	P_2
$L_{\alpha 1} : A := \alpha;$	$L_{\beta 1} : B := \beta;$
$\ldots$	$\ldots$
$L_{j1} : A := j;$	$L_{i1} : B := i;$
$L_{j2} : Y[j] := B;$	$L_{i2} : X[i] := A;$
$\ldots$	$\ldots$

Every member of the graph set for this execution under $A(CMP, PO)$ involves a cycle shown in Figure 2.12. Hence, in this case, the execution does not obey $A(CMP, PO)$.

Case II: Consider the second case of the condition violation.

$$\exists i, j : X[i] > j \wedge Y[j] > i \iff \exists i, j, \alpha, \beta : X[i] = \alpha \wedge \alpha > j \wedge Y[j] = \beta \wedge \beta > i$$

The statements of $Test_{\text{PO}}$ associated with the values involved in the condition violation shown above are :

P_1	P_2
$\ldots$	$\ldots$
$L_{j1} : A := j;$	$L_{i1} : B := i;$
$L_{j2} : Y[j] := B;$	$L_{i2} : X[i] := A;$
$\ldots$	$\ldots$
$L_{\alpha 1} : A := \alpha;$	$L_{\beta 1} : B := \beta;$
$\ldots$	$\ldots$

Every member of the graph set for this execution under $A(CMP, PO)$ involves a cycle shown in Figure 2.13. Hence, in this case, the execution does not obey $A(CMP, PO)$.

□

All ARCHTEST test programs such as $Test_{\text{WA}}$, $Test_{\text{PO}}$, etc. are meant to be run on real machines and there cannot be any real guarantees that the particular interleavings that reveal violations (such as for memory ordering rule WA watched by condition ATOMIC in $Test_{\text{WA}}$) 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 $k = \infty$. Test model-checking achieves this by transforming each ARCHTEST test into a test automata that exploits nondeterminism to effectively check for $k = \infty$. Also, the model-checking framework guarantees that we explore all possible interleavings rather than only particular interleavings.

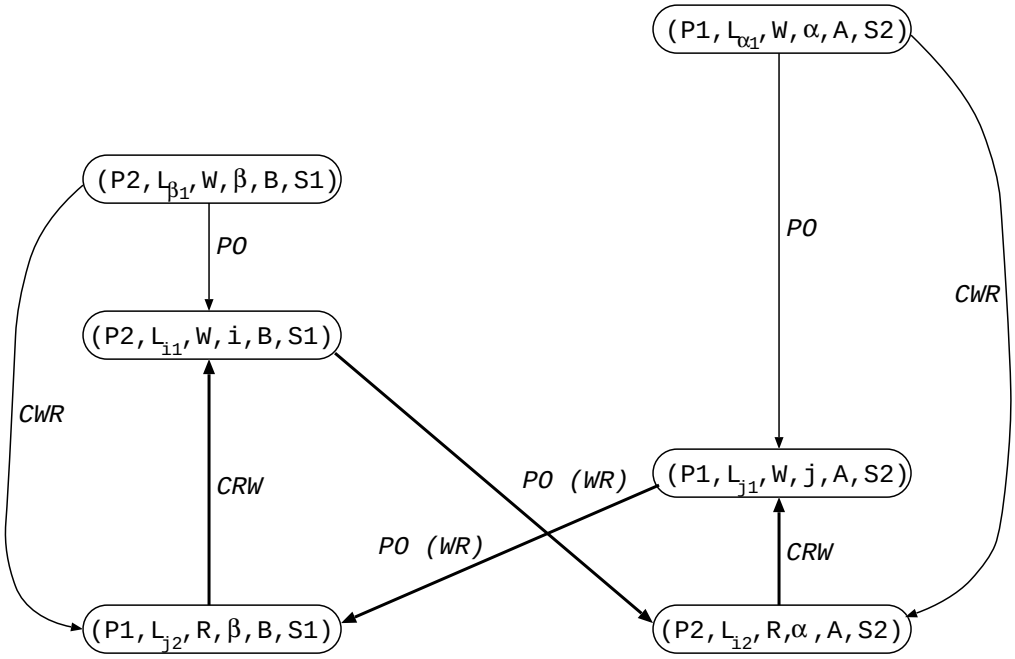

Figure 2.12. A cycle showing violation of $A(CMP, PO)$ corresponding to a violation of condition 3 of $Test_{\mathrm{PO}}$: *Case I*.

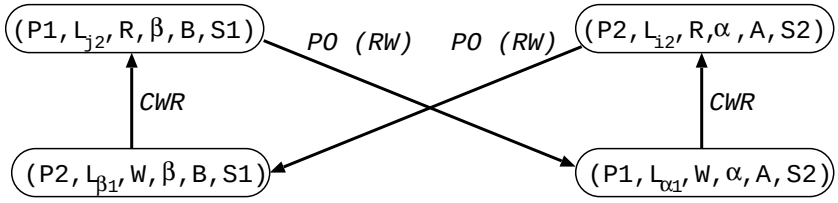

Figure 2.13. A cycle showing violation of $A(CMP, PO)$ corresponding to a violation of condition 3 of $Test_{\mathrm{PO}}$: *Case II*.

CHAPTER 3

TEST MODEL-CHECKING

In this chapter, a technique called *test model-checking* is presented to verify memory systems for their conformance to the formal memory model sequential consistency. Experimental results on example memory systems are reported. The work presented in this chapter was done jointly by Ratan Nalumasu, Abdel Mokkedem and the author.

3.1 Introduction

Test model-checking converts the tests of ARCHTEST to corresponding *memory rule test automata* ("test automata") that drive the model of the memory system being examined. In our experiments, we use the Verilog language supported by VIS [53] to capture the memory system models as well as the test automata. Precisely, the automata are modeled as (Verilog) processes which run in parallel with the memory system and simply enqueue test automata instructions in the instruction stream of the processors. The CONDITIONs corresponding to each compound memory rule being tested are turned into corresponding *memory rule safety properties* that are checked by the VIS tool. The reader may take a peek at Section 3.7.1 to know which compound rules define sequential consistency [42]. 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.

3.2 Memory Systems Assumptions

Memory systems realized in hardware as well as finite-state models thereof are assumed to be *data independent*, i.e., the control logic of the system moves data around, and does not base its control-point settings on the data values themselves. We also assume that the system is address *semi dependent* [33], i.e., the control logic can at most compare two addresses for equality or inequality and base its actions on the outcome of this test.

These assumptions are standard, and form the basis for defining test automata as well as memory rule safety properties.

3.3 Creation of Test Automata

As illustrated in Figure 3.1, we obtain test automata for various memory models by finitely abstracting the data used in ARCHTEST tests, using nondeterminism to justify the abstraction. For example, we abstract the specific activities of process P_1 of Figure 2.7 into that of (nondeterministically) writing *all possible* ascending values over {0,1}, as shown in P_1 of Figure 3.1. 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 same memory system model, i.e., a violation of a condition occurs in a test of ARCHTEST for $k = \infty$ iff the same violation occurs in model-checking the corresponding memory rule safety property when 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 2.7 can be viewed as repeating the initialization and then repeating the instruction at label L_1 of P_1 of Figure 2.7. Our proof sketches are illustrated on the three tests presented in Section 2.

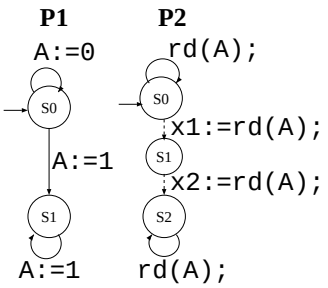

Figure 3.1. *Test* $_{\text{ROWO}}$ test automata : test automata for $A(CMP, RO, WO)$.

test described in this section.

3.4 Abstracting $Test_{\text{ROWO}}$

We show that if the test program in $Test_{\text{ROWO}}$ shows that MONOTONIC is violated, then the test automaton also reveals the error. Since MONOTONIC is violated,

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

Since, the last formula compares $X[i]$ and $X[i+1]$ only to α, we can rewrite the test program as shown in Figure 3.2(a) *assuming data independence*, and rewrite the last formula as

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

Note that in Figure 3.2(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 3.2(b). Figure 3.2(c) is obtained by simplifying Figure 3.2(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 3.1. 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 3.2(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 $Test_{\text{ROWO}}$ 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.

3.5 Abstracting $Test_{\text{WA}}$

$Test_{\text{WA}}$ test automata is shown in Figure 3.3. In this automaton P_1 and P_4 write all possible ascending sequences of $\{0,\ 1\}$ in A and B respectively. Each processor

Initially, $A = 0$

P_1	P_2
$L_1 : A := 1;$	$X[1] := (A > \alpha);$
$L_2 : A := 2;$	$X[2] := (A > \alpha);$
$L_3 : A := 3;$	$X[3] := (A > \alpha);$
$\ldots$	$\ldots$
$L_k : A := k$	$X[k] := (A > \alpha);$

(a)

Initially, $A = (0 > \alpha)$

P_1	P_2
$L_1 : A := (1 > \alpha);$	$X[1] := A;$
$L_2 : A := (2 > \alpha);$	$X[2] := A;$
$L_3 : A := (3 > \alpha);$	$X[3] := A;$
$\ldots$	$\ldots$
$L_k : A := (k > \alpha)$	$X[k] := A;$

(b)

Initially, $A = 0$

P_1	P_2
$L_1 : A := 0;$	$X[1] := A;$
$\ldots$	$\ldots$
$L_\alpha : A := 0$	$X[\alpha] := A$
$L_{\alpha+1} : A := 1$	$X[\alpha + 1] := A$
$L_{\alpha+2} : A := 1$	$X[\alpha + 2] := A$
$\ldots$	$\ldots$
$L_k : A := 1$	$X[k] := A;$

(c)

Figure 3.2. Abstraction of $Test_{\text{ROWO}}$: (a) 1 bit captures the ordering information. (b) 1 bit is written by P_1. (c) writing values just 0 and 1.

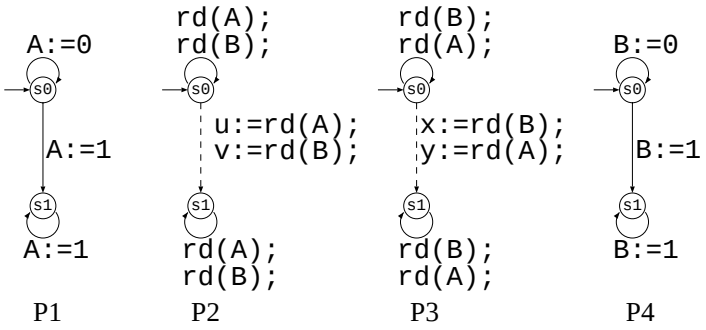

Figure 3.3. $Test_{\text{WA}}$ test automata : test Automata for $A(CMP, RO, WO, WA)$.

independently and *nondeterministically* decides to switch from writing 0 to writing 1. Modifications similar to those in $Test_{\text{ROWO}}$ 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 3.3 for $Test_{\text{ROWO}}$, our abstraction avoids having to remember the entire extent of the arrays U, V, X, and Y. (In $Test_{\text{WA}}$, one has to check for MONOTONIC also; this is done similarly to that in $Test_{\text{ROWO}}$.)

To show that the abstraction preserves ATOMIC, let ATOMIC be violated in $Test_{\text{WA}}$ 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 $Test_{\text{ROWO}}$, assuming *data independence*, we have an execution of the test automaton (Figure 3.3) 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 $Test_{\text{WA}}$ for $k = \infty$ would. A violation of ATOMIC happens exactly when $u = 1 \wedge v = 0 \wedge x = 1 \wedge y = 0$.

3.6 Abstracting $Test_{\text{PO}}$

We now discuss a test for the elemental ordering rule Program Order (PO), which is somewhat more complex than the previous two tests. PO requires that two events of the same process occur in the order specified by the program. ARCHTEST provides the test for the compound rule $A(CMP, PO)$ shown in Figure 3.4. Violation of $A(CMP, PO)$ is detected if Condition 3 fails: We obtain the test automaton and the memory rule safety property for $Test_{\text{PO}}$ of Figure 2.11 as illustrated in Figure 3.4. 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 instruction 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 3.4 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* $\Rightarrow (x \geq j \vee y \geq i) \wedge (x \leq j \vee y \leq i)$.

To show that this abstraction preserves PO_CROSS, let PO_CROSS be violated in ARCHTEST test $Test_{\text{PO}}$.

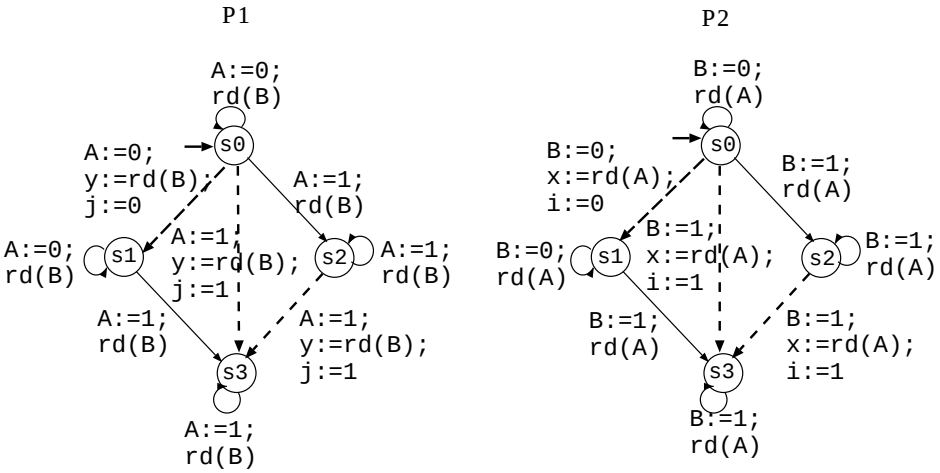

Figure 3.4. $Test_{\text{PO}}$ test automata: test automata for $A(CMP, PO)$.

$$\begin{array}{rll} & \exists i,j: & (X_i < j \wedge Y_j < i) \vee (X_i > j \wedge Y_j > i) \\ \Longleftrightarrow & \exists, i, j, \alpha, \beta: & ((X_i = \alpha) \wedge (j > \alpha) \wedge (Y_j = \beta) \wedge (i > \beta)) \\ & & \vee((X_i > \alpha) \wedge (j = \alpha) \wedge (Y_j > \beta) \wedge (i = \beta)) \end{array}$$

Similar to the case of $Test_{\text{WA}}$, if $\exists i, j : X[i] < j \wedge Y[j] < i$, then we can get a case in the test automata where $x = 0 \wedge j = 1 \wedge y = 0 \wedge i = 1$. Similarly, if $\exists i, j : X[i] > 0 \wedge Y[j] > i$, then we can get a case in the test automata where $x = 1 \wedge j = 0 \wedge y = 1 \wedge i = 0$. Hence, the memory rule safety property corresponding to PO_Cross is violated in test automata if and only if PO_Cross is violated in Archtest test $Test_{\text{PO}}$ for $k = \infty$.

3.7 Case Studies

To demonstrate the effectiveness of our approach, we verified three different memory systems, namely serial memory, lazy caching, and URM, all using a symbolic model-checker VIS (See [53] for more information). These three memory systems are described in some detail below, along with some of the subtle bugs that we could detect using test model-checking. Detail of all our experiments can be obtained from the Web [47] or by contacting the authors.

3.7.1 How Do We Check for Sequential Consistency?

A sequentially consistent memory system [43] requires that there be a single self-consistent trace t of memory operations that when projected onto the memory operations of each individual processor P_i ($R_i(a, d)$ and $W_i(a, d)$ for processor i) is according to

program order for P_i. As suggested in [14], one's intuition about sequential consistency matches the behavior described by $A(CMP, PO, WA)$.

As [14] does not list a single compound test to check for $A(CMP, PO, WA)$, we can use the following two tests that are available: $Test_{\text{WA}}$ which tests for $A(CMP, RO, WO, WA)$ and $Test_{\text{PO}}$ which tests for $A(CMP, PO)$. This combination is exactly equivalent to testing for sequential consistency because PO implies RO and WO (as formally defined in [14]). While sequential consistency matches the behavior described by $A(CMP, PO, WA)$, successful test model-checking outcomes are only necessary but not sufficient conditions to ensure sequential consistency. As part of our future work, we are exploring the ways to arrive at sufficient conditions for such tests. For every memory system we consider, these two tests are model-checked separately.

3.7.2 Serial Memory and Lazy Caching

The **serial memory** protocol for n processors and a memory is shown in Table 3.1. Serial memories are often used to define SC operationally. The **lazy caching** protocol [3,21], shown in Table 3.2, also implements sequential consistency, and is geared towards a bus based architecture. 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 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 cache 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

Table 3.1. Serial memory transaction rules

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

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 update requests are marked (with a $\star$). Figure 3.5 shows the structure of the Verilog model we created for the memory model verification which we shall discuss in section 3.7.5.

3.7.3 Runway-PA8000 Memory System

Figure 3.6 shows a simplified view of 2 HP PA8000 CPUs and a memory controller (HOST) interconnected by HP Runway Bus [8, 9, 37].[1] We describe the Runway-PA8000 system in some detail to facilitate a clear description of some of the subtle bugs in URM unearthed by the test model-checking technique. Runway is a synchronous, split-transaction bus which is responsible for providing a coherent view of shared memory to the processors (*clients*) while still allowing the clients to maintain private copies of memory lines in their caches. Cache Coherency is maintained by a snoopy coherency protocol described below.

3.7.3.1 Snoopy Coherency Protocol

Each cache line in a client can be in one of the four states: invalid, shared, private-clean or dirty.[2] If a client suffers a read miss in cache, it generates a *rsp* (read shared or private) transaction; if it suffers a write miss, it generates a *rp* (read private) transaction. The transaction is broadcast on the Runway when it wins the bus mastership. All clients snoop the transaction into their CCC (cache coherency check) queues and process the

[1] We have purposefully avoided arbitration lines and other detail for the sake of clarity. The actual Runway allows up to four CPUs and one I/O processor and also many more transactions including coherent, noncoherent and I/O transactions than we describe here. We provide a simplified view of its operation which captures the essential complexity of its behavior.

[2] There are also transient states that the cache line may assume when it is changing from one of these clean states to another.

Table 3.2. Gerth's version of the lazy caching algorithm

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

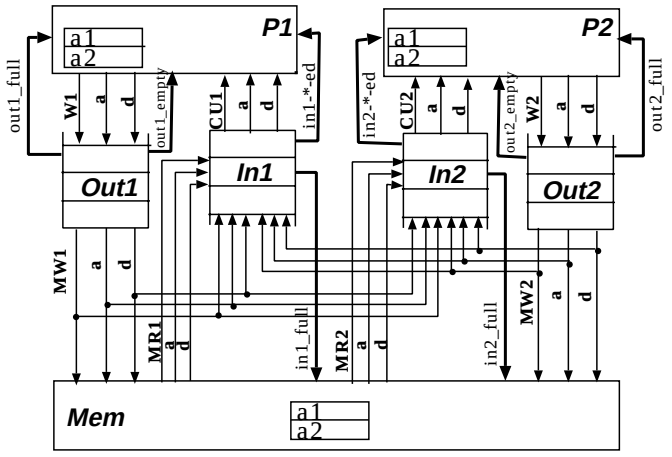

Figure 3.5. Verilog architecture of two processors lazy caching parallel machine.

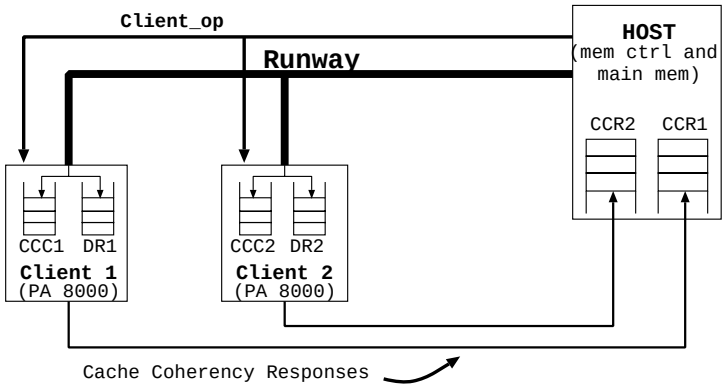

Figure 3.6. Simplified view of Runway-PA8000 memory system.

entries in CCC queue at their own speed. When a transaction gets to the head of CCC of client C_i, it sends a *ccr* (cache coherency response) to HOST according to Table 3.3, and also changes its state to reflect the transaction; for example, if the transaction is *rp* generated by C_i, it would assume "invalid–private-clean" transient state. If a client generates a *coh_copyout* as *ccr*, it would later issue a *c2cw* (cache to cache write) to supply the data. HOST enters the *ccr*'s into its CCR queue, and after all clients have responded to a transaction, the HOST determines if the data would be supplied by another client. If no client is going to supply the data, the HOST would generate a *hdr* (host data return) transaction on the Runway to supply the data to the requester. It would also drive Client_op lines to indicate whether the data must be shared (i.e., at least one of the *ccr*s is *coh_shared*). When a client notices a data return (a *hdr* or *c2cw*) targeted towards it, it enters the information into data return (DR) queue. Note that a client might receive a data return before it generates the corresponding *ccr*. In this case, the client keeps the

Table 3.3. Cache coherency response *ccr* generated when a transaction gets to the head of CCC queue

Transaction	Generated by	State	*ccr*
–	self	–	*coh_ok*
–	other	invalid	*coh_ok*
rsp	other	private-clean	*coh_shared*
rsp	other	shared	*coh_shared*
rp	other	shared	*coh_ok*
rp	other	private-clean	*coh_ok*
–	other	dirty	*coh_copyout*

data in data return queue until the *ccr* is sent out.

3.7.3.2 Delay in *ccr* Generation

If a client has a *c2cw* transaction for a line yet to go on Runway, then it delays generating any more *ccr*'s for that line. To see why this is necessary, consider the following. Suppose a client C1 has a dirty line. Client C2 requests this line by issuing *rsp* transaction on bus. C1 generates *coh_copyout* in response to C2's request, invalidate its own line, and create a *c2cw* transaction for C2. Note that the most recent data for this line is with C1 and not HOST. Now, a client C3 requests the same line by issuing *rsp*. C2 and C3 generate respectively *coh_shared* and *coh_ok ccr*s in response to C3's request. C1's *ccr* will be *coh_ok* in response to C3's request. If C1 sends *coh_ok* to HOST before its *c2cw* goes on the bus then HOST can provide stale data to C3 by its *hdr* transaction. To avoid this, C1 delays generating *ccr* until the *c2cw* goes on the bus.

3.7.3.3 Arbitration

Runway follows a complex pipelined arbitration algorithm to determine the bus master. Here, we only present an approximation of the algorithm. Every bus user (client or HOST) must become the bus master before it can drive the bus. Bus mastership at cycle N+2 is acquired by initiating the arbitration in cycle N by driving the request through dedicated arbitration lines (not shown in the figure). During cycle N+1, every potential bus user evaluates the others' drives and, in conjunction with round-robin pointers for arbitration priorities, determines who wins bus-mastership for cycle N+2. Those who do not win bus mastership keep off the bus. Bus arbitration proceeds in a pipelined manner concurrently with transaction processing.

3.7.3.4 PA8000 Runway Interface

In addition to the Runway specifics described above, PA8000 Runway interface also adheres to the following constraints in order to ensure Program Order and Write Atomicity [8, 37].

Runway interface allows a client to initiate Runway transactions for various cache misses; it is possible that these transactions complete out of order. However, all instructions strictly *complete* in program order. Runway interface guarantees that the client stalls the coherency response for any cache line which it has an outstanding miss for (i.e., it has initiated a Runway transaction, has assumed the ownership but is still waiting for

the data). The coherency response is generated only after the client has received the data and has used it to make forward progress at least one instruction. Runway interface guarantees that if a client receives data for its Runway transaction before it assumed the ownership, then it does not modify or use the data until it processes its own transaction (and thus assumes ownership). Runway interface guarantees that if a client has a *c2cw* transaction enabled to go to Runway then it gets the highest priority to go to the Runway.

3.7.4 URM in VIS Verilog

We constructed a Verilog model of the Runway-PA8000 system, Utah Runway Model (URM), and the two abstractions of $Test_{\mathrm{PO}}$ and $Test_{\mathrm{WA}}$ to verify that its memory model is sequential consistent. The complexity of the system stems from a number of sources: (a) multiple outstanding transactions for each processor, (b) out-of-order completion of the Runway transactions, but in-order completion of instructions, (c) eager assumption of ownership without receiving the corresponding data, (d) "equivalent" states introduced by decoupled execution due to coherency queues, (e) speculative execution features of the processor to ensure performance in spite of in-order completion of the instructions, (f) an involved distributed pipelined arbitration algorithm. We did not try to model each of these features in their full glory, but we *did* include a modicum of these aggressive features into our URM, which in fact occupies more than 2,000 lines of VIS Verilog code (see [47]). For instance all essential features of (a), (b), (c), and (e) are included, (f) is abstracted by using nondeterminism. (d) is abstracted as explained below.

3.7.4.1 Abstraction of Queues

Additional abstraction effort was necessary to make our URM digestible by VIS. This essentially consists in getting rid of the CCC, CCR, and DR queues, which are the main cause of state explosion, but retain HDR queue in the HOST and C2CW queues in the HOST and clients.

In Runway, most of the conflicts are detected and resolved by the HOST. There is one situation where a client detects conflict: the client has a pending *c2cw* transaction. The client resolves this by delaying its coherency response; the net result of this delay is that the HOST would not generate *hdr* transactions until the *c2cw* goes on the Runway. Since we abstracted away the CCR queues, in our URM the clients send the coherency response for a coherent transaction immediately after its occurrence on the bus. Hence, in

our URM the clients cannot resolves conflicts by *delaying* the coherency response; instead the HOST *computes* if the coherency response needs to be delayed, and if so, delays the *hdr*s appropriately. This is achieved as follows. A counter is associated with each HDR queue entry. If the counter is nonzero, then it is waiting for some *c2cw* transactions for that line from the clients, hence the *hdr* needs to be delayed. After all the pending *c2cw* transactions for that line go on the bus, the counter becomes zero, and hence the *hdr* transaction can go on the bus. In our URM, we used a two-bit counter, which allows up to four processors.

In Runway, all clients save the data returns (*hdr* and *c2cw* transactions) in DR queue until the corresponding request appears at the head of its CCC queue. This is necessary to enforce in-order completion of instructions. We abstract away the CCC queues and the data return queues by associating one bit of information with each cache line in each client. This bit is set for an address a whenever a data return happens for a, but a preceding instruction is not yet completed. After all preceding instructions are completed, the data is used, and the bit is reset indicating the completion of the instruction.

3.7.5 Verification Results

Table 3.4 shows execution time for model-checking our Serial memory, Lazy caching and URM models for tests of $A(CMP, PO)$ and $A(CMP, RO, WO, WA)$ (recall that $A(CMP, PO, WA)$ implies SC). The three models running separately the two tests $Test_{\text{WA}}$ and $Test_{\text{PO}}$ are model-checked for the following conditions: (Figure 3.4 does not show some of these states)

$Test_{\text{WA}}$: MONOTONIC:
$$\begin{aligned}
&\wedge\ (P_2.inS_2) \implies (P_2.U_1 \leq P_2.U_2)\\
&\wedge\ (P_2.inS_2) \implies (P_2.V_1 \leq P_2.V_2)\\
&\wedge\ (P_3.inS_2) \implies (P_3.X_1 \leq P_3.X_2)\\
&\wedge\ (P_3.inS_2) \implies (P_3.Y_1 \leq P_3.Y_2)
\end{aligned}$$

ATOMIC:
$$(P_2.inS_1 \wedge P_3.inS_1) \implies (P_2.V \geq P_3.X \vee P_3.Y \geq P_2.U)$$

$Test_{\text{PO}}$: PO_CROSS:
$$\begin{aligned}
&(P_1.inS_3 \wedge P_2.inS_3) \implies\\
&(P_1.Y \geq P_2.I \vee P_2.X \geq P_1.J) \wedge\\
&(P_1.Y \leq P_2.I \vee P_2.X \leq P_1.J)
\end{aligned}$$

As can be seen, all these conditions are safety properties, and independent of the model itself, which is a distinct advantage over other methods.

The size of the state space and number of nodes in BDDs are also reported. Note that lazy caching has more states than URM due to the queues present in the model.

Table 3.4. Verification results using VIS on a SPARC ULTRA-1 with 512 MB memory.

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

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

Vacuity: Antecedent of $\Longrightarrow$ is not always false

However, the complexity of the URM protocol is much higher, which results in large BDD size and higher run time. 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 [14], using which designers can probe their model.

Our Verilog models captures quite faithfully the cache coherence protocol and the ordering rules of the three memory systems.

After an extensive debugging using test model-checking driven by $Test_{\,\mathrm{PO}}$ and $Test_{\,\mathrm{WA}}$, we have a high degree of confidence that the memory systems built based on the Lazy caching model or URM would be sequentially consistent.

3.7.5.1 Description of a Bug Found in a Preliminary Model of Lazy Caching

The following bug in our model of Lazy Caching was caught by a violation of PO_Cross in $Test_{\,\mathrm{PO}}$. The bug is in the queues used by Lazy Caching, which is 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 bug it is possible that In_i queue is not $\star$-ed when it should be, and consequently reads in P_i may bypass writes. This results in a violation of PO. This is a difficult bug to catch because its detection involves understanding the complex feedback from all components of the protocol to each other (queues, memory, and caches). Moreover, this bug is interesting because it violates PO but does not violate WA. This is so because only write-read (WR) order is affected by this bug. Our technique effectively caught this bug: the PO_Cross condition does not pass when we model-checked the model for $Test_{\mathrm{PO}}$. However, $Test_{\mathrm{WA}}$ (note that it does not involve PO) *passes!* This shows the futility of *ad hoc* testing methods: one could apply subjective criteria to consider a test similar to $Test_{\mathrm{WA}}$ to be sufficiently incisive, when in fact it fails to account for a crucial ordering relation such as PO.

3.7.5.2 Description of a Bug Found in Preliminary URM

Similarly, another corner-case bug was caught by *test model-checking* in our URM by a violation of PO_Cross condition using $Test_{\mathrm{PO}}$. This bug generated a long counter-example trace, due to the depth of the sequential logic of the model. The trace revealed the following situation:

(1) $client_i$ has removed its own read transaction from the bus, then

(2) $client_i$ sends *coh_ok* in response to a subsequent coherent transaction for the same line before getting the data for its transaction (by *hdr* or *c2cw*).

This problem is fixed using the counter in the HOST's HDR entries to record the pending *c2cw*s and the one-bit information in the client's cache lines to record whether the data is supplied, as explained in Section 3.7.4.1. After fixing the bug, the PO condition passed.

CHAPTER 4

WEAKER FORMAL MEMORY MODELS

We saw how ARCHTEST tests for $Test_{\text{ROWO}}$, $Test_{\text{PO}}$, and $Test_{\text{WA}}$ can be adapted into test model-checking automata. Sequential consistency can be viewed as $A(CMP, PO, WA)$ and can be verified using test automata for $Test_{\text{PO}}$ and $Test_{\text{WA}}$. The rule of Program Order (PO) can be viewed as conjunction of four subrules : rule of Read order (RO), rule of Write Order (WO), rule of Read-Write (RW) and rule of Write-Read (WR). Weaker memory models typically relax one or more of these four ordering subrules to allow for more optimizations. Weaker memory models relax the atomicity requirement in various degrees. The rule of Write Atomicity (WA) requires that all processors see all writes in one global order. Many weaker memory models relax WA in various degrees.

Test automata for $Test_{\text{PO}}$ or $Test_{\text{WA}}$ cannot be used to test memory systems for weaker memory models. For example, violation of memory rule safety property for $Test_{\text{PO}}$ test automata implies a violation of PO but cannot distinguish between violations of different subrules of PO. A violation of PO could occur as a result of violation of one or more of its subrules. Similarly, violation by $Test_{\text{WA}}$ test automata for memory systems supposed to obey a typical weaker memory model cannot detect whether a weaker memory model is violated or not.

The rules presented in ARCHTEST [14] are not adequate for capturing all weaker memory models. Many weaker memory models in prevalent architectures differ from each other in subtle ways that can often not be captured using ARCHTEST rules. For example, the difference between how memory model provided by IBM-370 and SPARC V9 TSO relax WA cannot be captured using existing ARCHTEST rules.

4.1 Chapter Overview

In this chapter, we study how the existing test model-checking framework can be extended to verify memory systems for weaker memory models. We propose new archi-

tectural rules in the framework of ARCHTEST that can faithfully capture many weaker memory models. We propose a representation for various weaker memory models in terms of the architectural rules. We propose new architectural tests similar to ARCHTEST tests for such weaker memory models. We provide formal proofs that these architectural tests indeed capture violation of the intended architectural rules. We propose test automata corresponding to architectural tests for weaker memory models. We have applied these test automata on operational model of TSO weaker memory system and obtained experimental results.

4.1.1 The Process of Creating Tests and Corresponding Test Automata

The process of creating new tests and test automata follows a very similar pattern. First, an execution is examined which depict the behavior violation we are interested in. We extend the pattern of this execution in an iterative structure along with a condition which captures the expected behavior. This process could be nontrivial sometimes - in some cases, we need to use existential qualifier to capture the behavior we want. We provide a formal proof of the test to prove that the condition violation indeed captures the behavior violation we are interested in. We propose test model-checking automata corresponding to the new test. This may involve using complex and innovative data abstraction techniques. We provide a formal proof for the soundness of such data abstraction.

4.2 SPARC V9 Total Sorted Order (TSO)

To exemplify the points made above, let us consider a weaker memory model proposed in SPARC V9 architecture TSO [1, 54] described in Section 2.1.2.

TSO relaxes both PO and WA. It relaxes PO by allowing reads to bypass writes. So, WR is relaxed; however RO,WO and RW are all obeyed. For example, the execution showing in Figure 4.1 is possible with TSO but not with sequential consistency because of the WR relaxation. Under sequential consistency, at least X or Y would be 1 at the end. However, since TSO allows reads to bypass writes it is possible that both X and Y are 0 under TSO.

Though processors in this TSO operational model do not have cache and there is a single port memory, TSO relaxes WA in a subtle way. Since reads of a processor are allowed to read value of the same processor's write from write buffers (before they

$$\textit{Initially,}\ \ A = B = X = Y = 0$$

$$\begin{array}{cc} P_1 & P_2 \\ L_1 : A := 1; & L_1 : B := 1; \\ L_2 : X := B & L_2 : Y := A; \end{array}$$

$$\textit{Finally,}\ \ A = B = 1; X = Y = 0$$

Figure 4.1. An execution valid under TSO but not SC - PO relaxation.

are updated in memory), WA is violated. A processor's read is atomic with respect to all other processors. But, since it reads its own write early, a write operation is not completely atomic. Consider the execution shown in Figure 4.2. This execution is not possible under sequential consistency. If sequential consistency is relaxed by allowing WR violation but not WA then also this execution is not possible under such a weaker memory model. However, this execution is possible under TSO as explained below.

- Both P_1 and P_2 enqueue writes for A, C and B, D into write buffers respectively.
- P_1 and P_2 read C and D from write buffers respectively.
- P_1 and P_2 read B and D from memory respectively (value read being 0).

We can also show that if we relax only PO (by allowing a read to bypass a write) and do not relax WA then this execution is not obtained. If WA is not relaxed then before P_1 or P_2 can read C or D respectively, the writes for A, C and B, D must be pushed out from the write buffer to the memory. The subsequent reads of B and A obtain 1 instead of 0 in this case.

We can formally show that each member of the graph set of this execution under

$$\textit{Initially,}\ \ A = B = C = D = U = V = X = Y = 0$$

$$\begin{array}{cc} P_1 & P_2 \\ L_1 : A := 1; & L_1 : B := 1; \\ L_2 : C := 1; & L_2 : D := 1; \\ L_3 : U := C; & L_3 : V := D; \\ L_4 : X := B; & L_4 : Y := A; \end{array}$$

$$\textit{Finally,}\ \ A = B = C = D = U = V = 1; X = Y = 0$$

Figure 4.2. An execution valid under TSO but not SC - WA relaxation.

$A(CMP, UPO, WO, RO, RW, WA)$ must contain the cycle shown in Figure 4.3. Hence, this execution does not obey $A(CMP, UPO, WO, RO, RW, WA)$.

It is not possible to capture the behavior of weaker memory model such as TSO by rules of the ARCHTEST method. Also, tests for $Test_{\text{PO}}$ and $Test_{\text{WA}}$ are not enough to check for such weaker memory model. To test for weaker memory model such as TSO, we propose new rules, new tests and corresponding test automata.

4.3 Weaker Memory Models : Categories

As described earlier, sequential consistency can be specified in the formal framework of ARCHTEST as $A(CMP, PO, WA)$. Various weaker memory models proposed in literature relax either program order(PO) or atomicity(WA) in various degrees. Most proposed weaker memory models can be roughly divided into two categories:

4.3.1 Partial PO-relaxation

These weaker memory models relax the program order (PO) rules in various orders. All weaker memory models that relax PO relax WR. Intuitively, the relaxation of WR helps to decrease read latency by making it possible to complete reads without completing earlier writes (as the completion of writes could be a costly operation). Assuming that

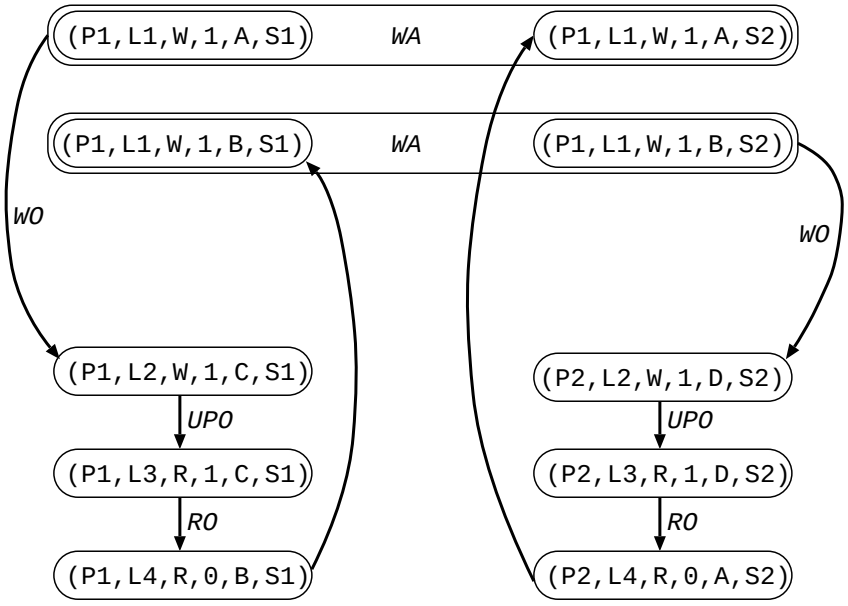

Figure 4.3. Cycle corresponding to an execution valid under TSO but not $A(CMP, UPO, WO, RO, RW, WA)$.

there are more load operations than store operations in the instruction mix, this single relaxation can greatly improve overall performance. Formally, we could see why all weaker memory models that relax PO relax WR by examining the relations between various subrules of PO (as explained later in Section 4.4. These weaker memory models also may relax other subrules of PO (e.g., SPARC V9 PSO).

Further, these weaker memory models may relax atomicity requirement WA in various degrees. Examples of weaker memory models in this category are SPARC V9 TSO, PSO, IBM-370, Processor consistency, Pentium Pro memory model, etc.

4.3.2 Complete PO-relaxation

Weaker memory models in this category relax PO completely. The only ordering requirement between memory operations is UPO. Most weaker memory models in this class obey WA or a very minor relaxation of it. Examples of memory models in this class are: Alpha, Release Consistency, PowerPC, SPARC V9 RMO, etc.

Weaker memory models in either class provide mechanisms variously known as *membar*, *fence* or *barrier* to enforce additional PO-constraints not provided by the memory model but designed by the programmer.[1] Membar operations often provide a way to specify exactly which subrules of PO are to be enforced.

Weaker memory models in either class are expected to obey $A(CMP, UPO)$ because it is incredibly difficult to argue about any program in the absence of $A(CMP, UPO)$. However, there seems to be a surprising exception to it in the case of RMO while considering application of these architectural rules on global vs. local operands. The execution shown in Figure 4.4 is valid under RMO (this execution is mentioned in [49]). We can see that this execution does not obey $A(CMP, UPO)$ as explained below.

In above execution, R0, R1 and R2 are registers and A and C are global memory operands. This execution would not be possible if R0, R1 and R2 were global operands. However, CMP and UPO as defined in ARCHTEST's formal framework are enforced equally for both global operands and registers.[2]

In above execution, *membar LoadStore* in P_2 enforces that read of C in P_2 must occur before write of A. There is one common operand between every successive statements in

[1] In this report, we refer to them by membar.

[2] ARCHTEST considers both local and global operands while defining various architectural rules. The basic memory operations defined are *read* and *write*. Keeping in line with ARCHTEST's definition, we consider both local and global operands in this report.

Initially A = C = R0 = R1 = R2 = 0 = 0

P_1	P_2
$L_1 : R1 := A;$	$L_1 : R0 := C;$
$L_2 : A := 1;$	$L_2 : membarLoadStore;$
$L_3 : R2 := A;$	$L_3 : A := 2;$
$L_4 : C := R2;$	

Finally A = C = R0 = R2 = 1; R1 = 2

Figure 4.4. An execution valid under SPARC V9 RMO, which does not obey $A(CMP, UPO)$.

P_1. Hence, if CMP and UPO are obeyed, it is not possible for R1 to read 2 and R2 to read 1. We can show an explicit cycle between all events with just CMP and UPO arcs.

4.4 Subrules of PO

In this section, we see how various subrules of PO are interrelated. Figure 4.5 shows how WR, WO, RO and RW can be compared with each other in the presence of CMP. An arrow from a rule R1 to R2 in this figure indicates that $A(CMP, R1) \Rightarrow A(CMP, R2)$. This can be easily shown by considering arcs between various events of two statements in the same process.

WR is the strongest of all four subrules and implies the rest of them. Hence, WR can be considered equivalent to PO in presence of CMP.

THEOREM 4.1 $A(CMP, WR) \Rightarrow A(CMP, PO)$. $A(CMP, WO) \Rightarrow A(CMP, RW)$. $A(CMP, RO) \Rightarrow A(CMP, RW)$.

Proof : Consider events associated with any two statements in a process P_1 as shown in Figure 4.6(a). There are two read events: one for operand B and one for operand D. There are n multiple write events for each write operation: one associated with each store. There are various SRW (a subrule of CMP) arcs from the read event to the write events of the same statement. Consider imposing WR arcs from write events of A to read event for D. These WR arcs order two read events hence RO is implied. Also, they order all write events hence WO is implied. Similarly, they order read event of B with respect to write events for C, hence RW is implied. Hence, $A(CMP, WR) \Rightarrow A(CMP, PO)$.

Similarly consider the events and the arcs shown in Figure 4.6(b). Just by imposing SRW and WO arcs we can order the read event for B with respect to write events for C.

Hence, $A(CMP, WO) \Rightarrow A(CMP, RW)$. Also, by imposing SRW and RO arcs we can achieve the same ordering. Hence, $A(CMP, RO) \Rightarrow A(CMP, RW)$.

□

This inter-relationship between various subrules of PO could be used to show nonexistence of effective tests for some set of rules as explained later. This also explains why weaker memory models which relax PO always relax WR.

4.5 How to *Specify* Weaker Memory Models in ARCHTEST's Framework ?

The ARCHTEST formal framework of representing behavior of memory systems described in section 2.2.12 needs to be extended to specify weaker memory models for following reasons:

- Some of the relaxed behavior of weaker memory models cannot be captured with the existing rules of the ARCHTEST framework, e.g., atomicity behavior of SPARC V9 TSO.
- The ARCHTEST framework considers only two kinds of memory operations : read and write. We need to also consider membar operations and define rules corresponding to it.

Though the ARCHTEST framework does not already support this, it is general and flexible enough to allow us to easily extend it as necessary.

4.5.1 Rule of Membar (MB)

For each membar instruction in the program, there is an event corresponding to it. Consider the following statement in process P.

L : membar storestore

Corresponding to this statement, there is a membar event (P,L,MB-WW,-,-,-), where MB-WW denotes the memory operation membar between two write operations. The operand, value and store component of the events are nonsignificant. Similarly, different membar events for RW, WR, RR can be defined.

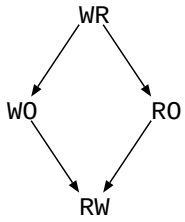

Figure 4.5. Subrules of PO : an arrow from R_1 to R_2 means $A(CMP, R_1) \Rightarrow A(CMP, R_2)$.

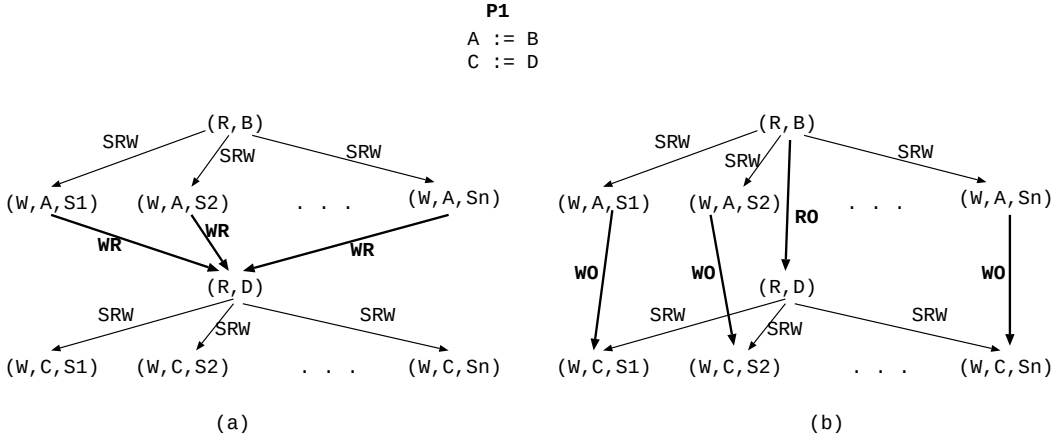

Figure 4.6. PO subrules WR, WO, RO and RW relationships : (a) $A(CMP, WR) \Rightarrow A(CMP, WO) \wedge A(CMP, RO) \wedge A(CMP, RW)$, (b) $A(CMP, WO) \Rightarrow A(CMP, RW), A(CMP, RO) \Rightarrow A(CMP, RW)$.

The rule of MB requires that memory operations separated by MB statement are ordered as per restriction imposed by MB. The programmer must explicitly use membar instruction to enforce the restriction imposed by MB. The transition template for MB is

$$\text{(P,L,X,V,O,S)} <_{MB} \text{(=,-,MB-XY,-,-,-)}$$
$$\text{(P,L,MB-XY,-,-,-)} <_{MB} \text{(=,-,Y,-,-,-)}$$

where X or Y could be either R or W.

The graph set for an execution E under $A(MB)$ could be constructed as follows. Draw an arc, labeled $<_{MB}$, from event X to an event M if

1. X $<_o$ M
2. X and M are in the same process and M's template is (P,L,MB-XY,-,-,-).

Also, draw an arc, labeled $<_{MB}$, from an event M to event Y if

1. M $<_o$ Y
2. M and Y are in the same process and M's template is (P,L,MB-XY,-,-,-).

.

A membar event M acts as an intermediate event between every two events X and Y where X is an event occurring before M in program order, Y is an event occurring after M in program order and membar event is for the operation MB-XY. Similar to various subrules of PO, we can define various subrules of MB such as MB–RR, MB–RW, MB–WR and MB–WW. The definitions of these subrules could be obtained straightforwardly by restricting X and Y to respective operations. The transition templates of these subrules are shown below. The graph set for these subrules could be created similarly to that for MB.

$MB-RR$:

$$\text{(P,L,R,V,O,S)} <_{MB} \text{(=,-,MB-RR,-,-,-)}$$
$$\text{(P,L,MB-RR,-,-,-)} <_{MB} \text{(=,-,R,-,-,-)}$$

$MB-RW$:

$$\text{(P,L,R,V,O,S)} <_{MB} \text{(=,-,MB-RW,-,-,-)}$$

$$(\text{P,L,MB-RW,-,-,-}) <_{MB} (=,\text{-,W,-,-,-})$$

$MB-WR$:

$$(\text{P,L,W,V,O,S}) <_{MB} (=,\text{-,MB-WR,-,-,-})$$
$$(\text{P,L,MB-WR,-,-,-}) <_{MB} (=,\text{-,R,-,-,-})$$

$MB-WW$:

$$(\text{P,L,W,V,O,S}) <_{MB} (=,\text{-,MB-WW,-,-,-})$$
$$(\text{P,L,MB-WW,-,-,-}) <_{MB} (=,\text{-,W,-,-,-})$$

Now, we discuss how various example weaker memory models could be specified in ARCHTEST framework.

4.5.2 SPARC V9 Total Sorted Order (TSO)

An operational model of TSO is shown in Figure 4.7. This architectural model is similar to that of sequential consistency in that it has a single port memory and only one processor accesses the memory at a time. Each processor has a write buffer associated with it. All write operations are enqueued in the write buffer in program order. These write operations are completed by updating the memory when the processor can access the memory. Read operations are allowed to bypass a write operation to a different address. If a write operation for the same address as a read operation is enqueued in the queue then the read operation completes by reading the value associated with the write operation in the queue. Otherwise, the read operation bypasses the write buffer

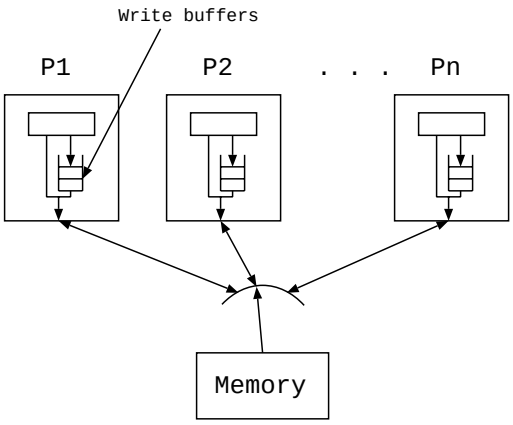

Figure 4.7. SPARC V9 total sorted order operational view.

and completes by reading the value from the memory. Note that all reads complete in program order and no write is allowed to bypass a read.

TSO relaxes both PO and WA. It relaxes PO by allowing reads to bypass writes. So, WR is relaxed; however RO,WO and RW are all obeyed. Since a process can read its own write before a previous write is visible to all other processes, it may seem that TSO obeys a weaker version of WO (such as WOS). However, the operational model of TSO does provide WO because there is no way for a processor to detect if a read operation is executed in another processor. The unique order in which all writes are seen by the memory prohibits cycles showing violations of WO. A theorem proving this is given at the end of this section.

Though processors in the TSO operational model do not have cache and there is a single port memory, TSO relaxes WA in a subtle way. Since reads of a processor are allowed to read value of the same processor's write from write buffers (before they are updated in memory), WA is violated. A processor's write is atomic with respect to all other processors. But, since it reads its own write early, a write operation is not completely atomic. It is not possible to capture the behavior of weaker memory model such as TSO by rules of the ARCHTEST method. Rule of WA or WS is too strong to capture this behavior, whereas the rule of CON is too weak. We define a new rule that capture the atomicity behavior of this nature.

4.5.2.1 Rule of $WA-S$

The rule of $WA-S$ consists of the following three subrules :

- $=_{WA\text{–}S}$: A write operation becomes visible to all processes atomically *except for the process executing the write*:

  The write operation becomes visible first to the process which is executing the write. The atomic sets are the sets of write events for each statement such that the store component is different from the process component in each event. We would refer to this aspect of $WA-S$ as $=_{WA\text{–}S}$.

  The transition template for $=_{WA\text{–}S}$ is

$$(P, L, W, V, O, \neq P) =_{WA\text{–}S} (=, =, W, =, =, \neq P)$$

- $WA-S_{intra}$: A write operation becomes visible to the process executing the write before it becomes visible to other processes.

  We would refer to this subrule of $WA-S$ as $WA-S_{intra}$.[3]

  The transition template for $WA-S_{intra}$ is

$$(P, L, W, V, O, = P) <_{WA\text{–}S_{intra}} (=, =, W, =, =, \neq P)$$

- CON: We require that $WA-S$ is at least as strong as CON, i.e., all processors see the same sequence of values for *each* operand. [4]

Figure 4.8 shows the difference between atomic sets of WA and $WA-S$ rules. The atomic set of events for WA rule contains all the n write events for operand A. The atomic set of events for $WA-S$ rule contains only n-1 write events shown in bold. It excludes the write event whose store component matches with processor component.

The graph set for an execution E under $A(WA-S)$ could be constructed similarly to WA by considering appropriate atomic set of write events and then imposing additional constraints required for CON. Note that rule of $WA-S$ is strictly weaker than WA as shown by an execution in Figure 4.2 which obeys $WA-S$ but does not obey WA. Also it could be shown that rule of $WA-S$ is *strictly* stronger than CON. This is easy to show because rule of $WA-S$ can restrict the ordering between two write events to different operands, where as rule of CON cannot.

Hence, in the ARCHTEST framework, TSO could be specified as $A(CMP, UPO, RO, WO, RW, WA-S, MB-WR)$.

---

[3]This subrule is named $WA\text{–}S_{intra}$ because it involves restriction between two write events associated with the same write operation.

[4]This requirement is necessary and is not implied by the atomic set for $WA\text{–}S$. Consider the following execution:

Initially,	$A = X = Y = 0$
P_1	P_2
$A := 1$	$A := 2$
$X := A$	$Y := A$
Finally,	$X = 2, Y = 1$

In this execution, CON is violated (as the two processors see difference sequence of values for A). However, this execution obeys $A(CMP, =_{WA\text{–}S}, WA-S_{intra})$, i.e., the other two aspects of $WA-S$ are obeyed.

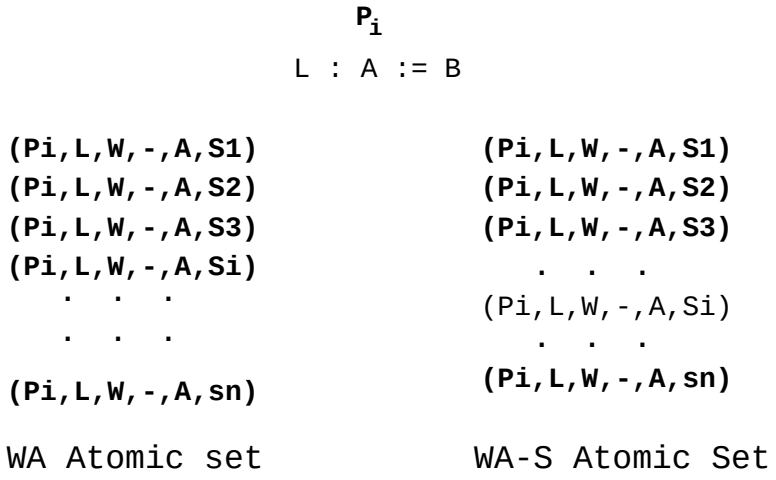

Figure 4.8. WA and $WA-S$ atomic sets of events.

- Rules of RO, WO and RW capture the relaxation that read is allowed to bypass previous write.
- Rule of $WA-S$ capture that a write operation becomes instantly visible to all *other* processes.
- Rule of $MB-WR$ capture that *membar* instruction enforces ordering between a write and a subsequent read operation.

Now, we present a theorem which proves that TSO does indeed provide WO and not a subrule of WO (such as WOS) as discussed earlier.

THEOREM 4.2 If an execution obeys $A(CMP, WOS, WA-S)$ then it also obeys $A(CMP, WO, WA-S)$.

Proof : *Intuitive Idea.* Intuitively, there is a unique order in which all memory updates to one address are seen by all processors (in spite of being able to read from the write-buffers). If a cycle involving a WO arc is given then we can use this unique order to construct a cycle involving WOS arcs in the case of TSO operational model. The detail of the proof follow.

The proof is by contradiction. Consider an execution E which obeys $A(CMP, WOS, WA-S)$ but does not obey $A(CMP, WO, WA-S)$. Consider a cycle corresponding to this violation. This cycle must contain a WO arc between two write

events with different stores as it must be a violation of WO but not WOS. Since we assume that $WA-S$ is obeyed, this arc cannot be of the following form.

$$(P_i, L_1, W, -, A, S_i) <_{WO} (P_i, L_2, W, -, B, S_j)$$

This is so because if the arc is of the form shown above, we can show a cycle with a $WA-S_{intra}$ arc between $(P_i, L_1, W, -, A, S_i)$ and $(P_i, L_1, W, -, A, S_j)$ and a WOS arc between $(P_i, L_1, W, -, A, S_j)$ and $(P_i, L_2, W, -, B, S_j)$. Hence, this arc must be between two events of the form :

$$(P_i, L_1, W, -, A, S_j) <_{WO} (P_i, L_2, W, -, B, S_i)$$

Consider such a cycle shown in Figure 4.9. Consider the latest event $(P_i, L_2, W, -, B, S_i)$ in the program order $<_o$ which could be part of such a cycle. (If such an event occurs later in the program order, then we would consider that event as our $(P_i, L_2, W, -, B, S_i)$ event in our argument.)

The rest of the proof follows by case-analysis of the arc following the event $(P_i, L_2, W, -, B, S_i)$. The purpose of the analysis is to show an existence of a write event W' from some other processor P_k to the S_i store with operand B in the cycle. Once we have such an arc W' then we can show that an alternative cycle exists which involves only WOS arc.

Now, consider all possible cases for the arc following the event $(P_i, L_2, W, -, B, S_i)$.

- The arc cannot be SRW or CRW because the transition templates do not match.
- The arc cannot be WOS, because of our assumption that $(P_i, L_2, W, -, B, S_i)$ is the latest such event from P_i that can be involved in the cycle.
- If the arc is CWR then consider the read event must be of the form $(P_i, -, R, -, B, S_i)$. Let us consider the arc following this read event in the cycle. This arc must be either a SRW or CRW arc. In either case the event following this read event in the cycle must be of the form $(-, -, W, -, B, S_i)$. If this write event is from process P_i then it contradicts our assumptions above. Hence, this write event must be from some other process P_k. Hence, this write event $(P_k, -, W, -, B, S_i)$ is our desired W' event.
- If the arc is CWW then the write event following this CWW arc must be of the form $(-, -, W, -, B, S_i)$. Again, this write event must be the desired W' write event

or our assumption is contradicted.

- If the arc is $WA{-}S$ then this arc must be $WA{-}S_{intra}$ arc. In this case, we can show that existence of a cycle which does not include the WO arc but includes WOS.

Using the W' event of the form $(P_k, -, W, -, B, S_i)$, we show a cycle involving WOS arc instead of WO arc. There exists a $WA{-}S$ arc between W' and $W'' = (P_k, -, W, -, B, S_k)$. Also, there exists $WA - S$ arc from $(P_i, L_2, W, -, B, S_i)$ to $(P_i, L_2, W, -, B, S_k)$ and to $(P_i, L_2, W, -, B, S_j)$. We also have arcs from $(P_i, L_2, W, -, B, S_k)$ and $(P_i, L_2, W, -, B, S_j)$ to W'' because of the ordering between $(P_i, L_2, W, -, B, S_i)$ and $(P_k, -, W, -, B, S_i)$. Now, there is a WOS arc between $(P_i, L_1, W, -, A, S_j)$ and $(P_i, L_2, W, -, B, S_j)$. Hence, we can get a cycle which involves WOS and not WO by considering the path $(P_i, L_1, W, -, A, S_j) <_{WOS} (P_i, L_2, W, -, B, S_j) <_{WA{-}S} (P_k, -, W, -, B, S_k) <_{WA{-}S} (P_k, -, W, -, B, S_i)$. This is contradictory to our assumption that the execution E obeys $A(CMP, WOS, WA-S)$. Intuitively, Since P_i observed W' after observing $(P_i, L_2, W, -, B, S_i)$, the write operation corresponding to L_2 must have been updated to the main memory before the write operation corresponding to W'. Hence, proved. □

4.5.3 SPARC V9 Partial Sorted Order (PSO)

The operational model for PSO [1,54] is shown in Figure 4.10. It is similar to that of TSO except that write operations in the buffer are allowed to be reordered for different operands. Hence, WO is violated. PSO still required RO and RW as all reads are blocking. PSO also provides the same kind of atomicity relaxation as TSO. Hence, in ARCHTEST framework, PSO could be specified as $A(CMP, UPO, RO, RW, WA{-}S, MB{-}WR, MB-WW)$.

4.5.4 IBM 370

The IBM 370 [1, 10] shared memory model is similar to TSO except that it does not relax atomicity requirement. Hence, in ARCHTEST framework, IBM 370 weaker memory model can be specified as $A(CMP, UPO, RO, WO, RW, WA, MB-WR)$.

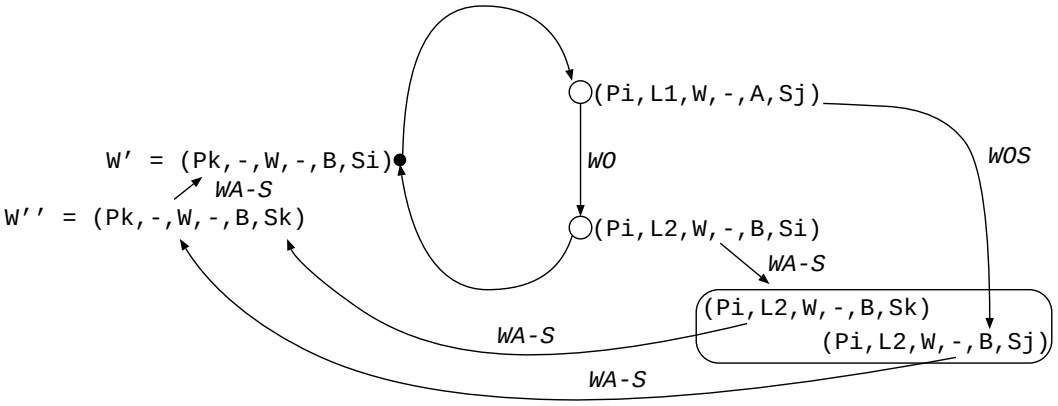

Figure 4.9. A cycle corresponding to violation of $A(CMP, WO, WA-S)$.

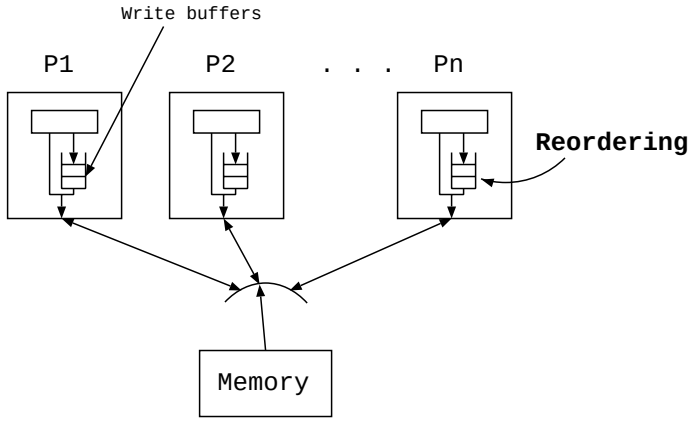

Figure 4.10. SPARC V9 partial sorted order (PSO) operational view.

4.6 Partial PO-relaxation Weaker Memory Models

In this section, we discuss how we can check for weaker memory models in the category of partial PO-relaxation. We can check for the weaker memory model by picking up the tests corresponding to the weaker memory model specification in the ARCHTEST framework.

4.6.1 Pure Tests

Ideally we would like to get *pure tests* for various subrules of PO. Pure test for a rule R is a test for checking $A(CMP, R)$ or if such a test is not possible then a test for $A(CMP, UPO, R)$. The motivation behind such a test is that most memory systems are expected to obey CMP, UPO; hence a violation of such a test gives high degree of confidence that Rule R is violated (unlike a violation of test of the form $A(CMP, UPO, R_1, R_2)$: from such a violation nothing can be deduced about violation of R1 or R2). For example, $Test_{\text{ROWO}}$ could not be used to check for weaker memory model PSO as PSO relaxes WO. If we want to check whether a PSO implementation provides RO, we need to have a test for RO without WO as a part of the test. We could use a pure test for RO in this case.

As a methodology, we can use the appropriate pure tests developed for various subrules to verify the weaker memory model as applicable. Pure tests facilitate applying such a methodology.

4.6.2 Pure Test for WR

As shown in section 4.4, $A(CMP, WR) \Rightarrow A(CMP, PO)$. Hence a test for $A(CMP, PO)$ could be considered as a pure test for $A(CMP, WR)$. Any violation of any subrule of PO could be considered a violation of WR. Specifically, we consider a part of the condition 3 mentioned in Section 2.2.15 for the $Test_{\text{PO}}$ to obtain the condition for $Test_{\text{WR}}$.

4.6.2.1 $Test_{\text{WR}}$

$Test_{\text{WR}}$, shown in Figure 4.11 tests for $A(CMP, WR)$, with the condition checked being:

CONDITION 4 (WR_CROSS) $\forall i, j : 1 \leq i, j \leq k : X[i] \geq j \vee Y[j] \geq i$

$$
\begin{array}{ll}
\multicolumn{2}{c}{\textit{Initially,}\ \ A = B = 0} \\
L_{11} : A := 1; & L_{11} : B := 1; \\
L_{12} : Y[1] := B; & L_{12} : X[1] := A; \\
L_{21} : A := 2; & L_{21} : B := 2; \\
L_{22} : Y[2] := B; & L_{22} : X[2] := A; \\
\ldots & \ldots \\
L_{k1} : A := k; & L_{k1} : B := k; \\
L_{k2} : Y[k] := B; & L_{k2} : X[k] := A;
\end{array}
$$

Figure 4.11. $Test_{\text{WR}}$: ARCHTEST test for $A(CMP, WR)$ or $A(CMP, RW)$.

Consider a violation of condition 4. Hence,

$$\exists i, j, \alpha, \beta : X[i] = \alpha \wedge \alpha < j \wedge Y[j] = \beta \wedge \beta < i.$$

We could show that under $A(CMP, WR)$, every member of the graph set for the execution corresponding to condition violation contains the following cycle.

$$
\begin{array}{lll}
(P_1, L_{j2}, R, \beta, B, S_1) & <_{CRW} & (P_2, L_{i1}, W, i, B, S_1) \\
 & <_{WR} & (P_2, L_{i2}, R, \alpha, A, S_2) \\
 & <_{CRW} & (P_1, L_{j1}, W, j, A, S_2) \\
 & <_{WR} & (P_1, L_{j2}, R, \beta, B, S_1)
\end{array}
$$

The test model-checking automata corresponding to $Test_{\text{WR}}$ is as shown in Figure 3.4. The memory rule safety property corresponding to condition WR_CROSS is: P_1 *and* P_2 *in their final states* $\Rightarrow (x \geq j \vee y \geq i)$. We could provide an argument similar to that for $Test_{\text{PO}}$ test automata to justify the abstraction.

4.6.3 Pure Test for RW

A Pure test for RW can be obtained by just considering a part of the condition 3 of the $Test_{\text{PO}}$ of ARCHTEST mentioned in Section 2.2.15.

4.6.3.1 $Test_{\text{RW}}$

$Test_{\text{RW}}$, shown in Figure 4.11 tests for $A(CMP, RW)$, with the condition checked being:

CONDITION 5 (RW_CROSS) $\forall i, j : 1 \leq i, j \leq k : X[i] \leq j \vee Y[j] \leq i$

Consider a violation of condition 5. Hence,

$$\exists i, j, \alpha, \beta : X[i] = \alpha \wedge \alpha > j \wedge Y[j] = \beta \wedge \beta > i.$$

We could show that under $A(CMP, RW)$, every member of the graph set for the execution corresponding to condition violation contains the following cycle.

$$\begin{array}{lll} (P_1, L_{\alpha 1}, W, \alpha, A, S_2) & <_{CWR} & (P_2, L_{i2}, R, \alpha, A, S_2) \\ & <_{RW} & (P_2, L_{\beta 1}, W, \beta, B, S_1) \\ & <_{CWR} & (P_1, L_{j2}, R, \beta, A, S_1) \\ & <_{RW} & (P_1, L_{\alpha 1}, W, \alpha, A, S_2) \end{array}$$

The test model-checking automata corresponding to $Test_{\mathrm{RW}}$ is as shown in Figure 3.4. The memory rule safety property corresponding to condition RW_CROSS is: P_1 *and* P_2 *in their final states* $\Rightarrow (x \leq j \vee y \leq i)$. We could provide an argument similar to that for $Test_{\mathrm{PO}}$ test automata to justify the abstraction.

4.6.4 Pure Test for RO

A pure test for RO could not be obtained by a direct extension of any of the existing tests of ARCHTEST. We propose a new test for $A(CMP, RO)$ described below. The test is based on the following execution which does not obey $A(CMP, RO)$.

$$\begin{array}{cc} \textit{Initially,} & A = X = Y = Z = 0 \\ P_1 & P_2 \\ L_1 : A := 1; & L_1 : X := A; \\ L_2 : A := 2; & L_2 : Y := A; \\ & L_3 : Z := A; \\ \textit{Finally,} & A = 2; X = Z = 1; Y = 2 \end{array}$$

Intuitively, assuming CMP, the only explanation for this execution is to consider that the second read of A in P_2 must have occurred after the third read of A in P_3. Formally, we could show a cycle in every member of the graph set for the execution under $A(CMP, RO)$.

We could construct a linear ordering of events of the above execution which shows that the execution obeys $A(CMP)$ and $A(RO)$. Under $A(CMP, RO)$, every member of the graph set for the execution contains the following cycle :

$$\begin{array}{lll} (P_1, L_2, W, 2, A, S_2) & <_{CWR} & (P_2, L_2, R, 2, A, S_2) \\ & <_{RO} & (P_2, L_3, R, 1, A, S_2) \\ & <_{CRW} & (P_1, L_2, W, 2, A, S_2) \end{array}$$

A graphical view of this cycle is shown in Figure 4.12. Most of the arcs are self-evident except for the CRW arc. The CRW arc between events $(P_2, L_3, R, 1, A, S_2)$ and $(P_1, L_2, W, 2, A, S_2)$ follow from the following two reasons:

- there is one CMP arc between these two events : either CRW from R to W event or CWR from W to R event.
- if the arc between these two events was CWR from W to R then the value returned by the R event would have been 2 as that is the latest value seen by that R event.

The cycle in the graph is highlighted by the bold edges.

4.6.4.1 $Test_{\text{RO}}$

A pure test for $A(CMP, RO)$ is proposed in Figure 4.13. The test is similar to $Test_{\text{ROWO}}$ in structure. If the memory system correctly realizes $A(CMP, RO)$, then Condition 6 is satisfied.

CONDITION 6 (EQUALITY) The sequence of X values satisfy the following property:
$\forall p, q, r : 1 \leq p < q < r \leq k : X[p] = X[r] \Rightarrow X[p] = X[q] = X[r]$

Intuitively, if two X array values are same then all the values in X array "between" these two values must be same also. Otherwise, one of the three read must have occurred out of order hence violating RO. Formally, we could show that a violation of condition 6 implies that architecture $A(CMP, RO)$ is violated.

Consider a violation of condition 6

$$\begin{array}{ll} & \exists p, q, r : 1 \leq p < q < r \leq k : X[p] = X[r] \wedge X[p] \neq X[q] \\ \iff & \exists p, q, r, \alpha, \beta : 1 \leq p < q < r \leq k : X[p] = X[r] = \alpha \wedge X[q] = \beta \wedge \alpha \neq \beta \end{array}$$

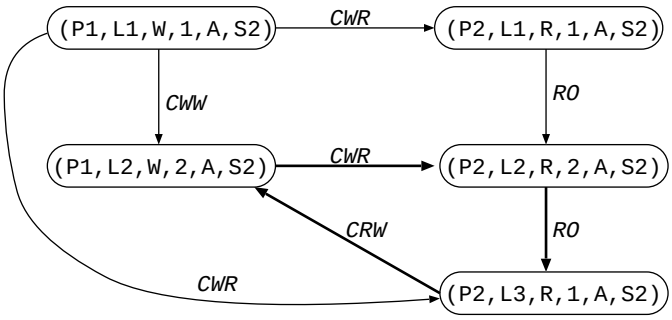

Figure 4.12. A cycle corresponding to violation of $A(CMP, RO)$.

$$
\begin{array}{cc}
\multicolumn{2}{c}{\textit{Initially,}\ \ A = 0} \\
P_1 & P_2 \\
L_1 : A := 1; & L_1 : X[1] := A; \\
L_2 : A := 2; & L_2 : X[2] := A; \\
L_3 : A := 3; & L_3 : X[3] := A; \\
\ldots & \ldots \\
L_k : A := k & L_k : X[k] := A;
\end{array}
$$

Figure 4.13. $Test_{\ \mathrm{RO}}$: a new test for $A(CMP, RO)$.

Consider an execution corresponding to a violation of the condition. We can provide a linear ordering of all the events of the execution under $A(CMP)$ showing that the execution obeys $A(CMP)$. To show that the execution does not obey $A(CMP, RO)$, we show a cycle in any member of the graph set for the execution under $A(CMP, RO)$. The exact cycle and the events involved in the cycle depends on the order in which α and β values of operand A become visible to P_2. This is captured by the direction of CWW arc between two write events. We need to consider two cases.

Case I: Consider the case when value α of A becomes visible to P_2 before value β does. The events and the arcs involving these events are shown in Figure 4.14(a). In this case, the analysis is similar to that of the example execution described earlier. The following cycle exists in the graph.

$$
\begin{array}{lll}
(P_1, L_\beta, W, \beta, A, S_2) & <_{CWR} & (P_2, L_q, R, \beta, A, S_2) \\
 & <_{RO} & (P_2, L_r, R, \alpha, A, S_2) \\
 & <_{CRW} & (P_1, L_\beta, W, \beta, A, S_2)
\end{array}
$$

The cycle is indicated by bold edges in Figure 4.14(a).

Case II: Consider the case when value α of A becomes visible to P_2 after value β does. The events and the arcs involving these events are shown in Figure 4.14(b). The CRW arc between events $(P_2, L_q, R, \beta, A, S_2)$ and $(P_1, L_\alpha, W, \alpha, A, S_2)$ follow from the following two reasons:

- there is one CMP arc between these two events : either CRW from R to W event or CWR from W to R event.
- if the arc between these two events was CWR from W to R then the value returned by the R event would have been α as that is the latest value seen by that R event.

Thus, the following cycle exists in the graph.

$$\begin{array}{lll}(P_1, L_\beta, W, \beta, A, S_2) & <_{CWR} & (P_2, L_q, R, \beta, A, S_2) \\ & <_{RO} & (P_2, L_r, R, \alpha, A, S_2) \\ & <_{CRW} & (P_1, L_\beta, W, \beta, A, S_2)\end{array}$$

The cycle is indicated by bold edges in Figure 4.14(b).

4.6.4.2 Test Automata for $Test_{\text{RO}}$

We can obtain a test automata for $Test_{\text{RO}}$ as shown in Figure 4.15. The process of obtaining a test automata is similar to that for $Test_{\text{ROWO}}$, $Test_{\text{PO}}$, etc. We again use nondeterminism and data abstraction to arrive at the test automata. Process P_1 writes value 0 into A for some number of times and nondeterministically switches to another state, writes value 1 into A and continues writing 0 into A. Process P_2 reads A and stores 3 values at nondeterministic points in $x1$, $x2$ and $x3$. The memory rule safety property to be model-checked is : P_2 *in final state* $\Rightarrow (x1 = x3 = 1 \Rightarrow x1 = x2 = x3)$

Now, we show that if the test program in $Test_{\text{RO}}$ shows that condition 6 is violated, then the test automata also reveals the error. Since condition 6 is violated,

$$\begin{array}{ll} & \exists p, q, r : 1 \leq p < q < r \leq k : X[p] = X[r] \wedge X[p] \neq X[q] \\ \iff & \exists p, q, r, \alpha, \beta : 1 \leq p < q < r \leq k : X[p] = X[r] = \alpha \wedge X[q] = \beta \wedge \alpha \neq \beta \end{array}$$

We need to consider two cases.

Case I : $\alpha > \beta$. In this case, the violation of condition 6 could be captured by a nondeterministic run of test automata for $Test_{\text{RO}}$ when P_1 switched to s1 after $\alpha - 1$ iterations on s0 . Variables $x1$, $x2$ and $x3$ get the value which would have been stored in $X[p]$, $X[q]$ and $X[r]$ respectively. In this case, $x2$ gets the value of the write event executed in s1 state. The memory rule safety property of test automata would be violated with $x1$, $x2$ and $x3$ values being 1, 0 and 1.

Case II : $\alpha < \beta$. In this case, the violation of condition 6 could be captured by a nondeterministic run of test automata for $Test_{\text{RO}}$ when P_1 switched to s1 after $\alpha - 1$ iterations on s0. Variables $x1$, $x2$ and $x3$ store the value which would have been stored in $X[p]$, $X[q]$ and $X[r]$ respectively. In this case, $x2$ gets the value of the write event executed in s0 state. The memory rule safety property of test automata would be violated with $x1$, $x2$ and $x3$ values being 1, 0 and 1. [5]

[5]Note that in both cases, the corresponding cycle could be of either type as shown in Figure 4.14.

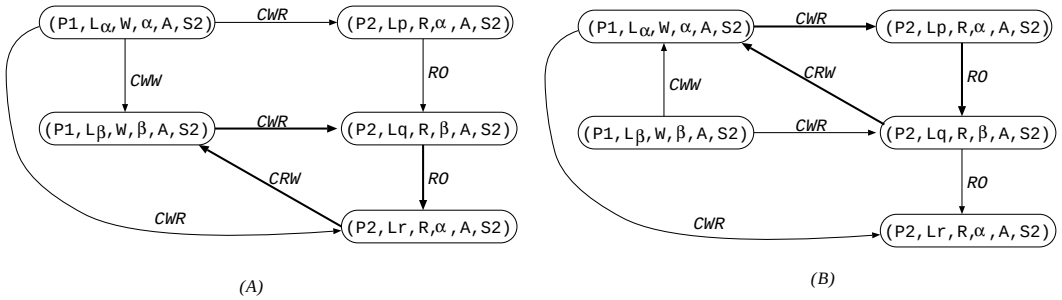

Figure 4.14. A cycle corresponding to violation of $Test_{\text{ROWO}}$ condition : (a) α becomes visible before β (b) β becomes visible before α.

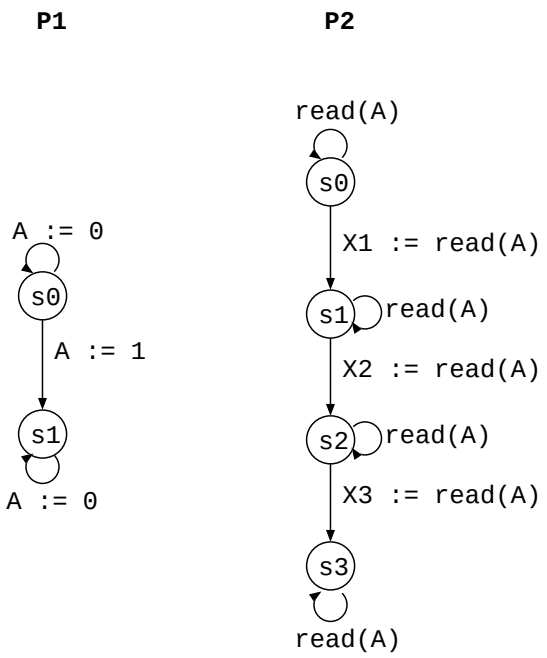

Figure 4.15. Test automata for $Test_{\text{RO}}$.

Hence, in either case, a corresponding violation would occur in test automata for $Test_{\text{RO}}$. Thus, the test automata memory rule safety property corresponding to condition 6 would be found violated whenever the condition is found violated in $Test_{\text{RO}}$.

4.6.5 Pure Test for RO

The pure test for RO described in previous section actually tests for a violation of read order when both reads involve the same operand. Hence, the test above would not be able to detect a violation of RO when the two read events involve different addresses. However, we would like to detect if the memory system orders two reads with the same operand but does not order two reads with different operands. We propose a test for $A(CMP, RO)$ which can detect such violations of two read events involving different operands.

The test is based upon the following execution.

$$\begin{array}{ccc} \textit{Initially,}\ A = B = C = X = Y = U = 0 & & \\ P_1 & P_2 & P_3 \\ L_1 : B := 1; & L_1 : X := A; & L_1 : U := C; \\ & L_2 : Y := B; & L_2 : A := U; \\ & L_3 : C := Y; & \\ \textit{Finally,}\ A = B = C = X = Y = U = 1 & & \end{array}$$

All variable are 0 initially and 1 at the end. A value 1 is written byP_1 in operand B. P_2 reads this value written by P_1 into Y and in turn writes into C. P_3 reads the value from C and writes into A, which in turn is read by P_2 and assigned to X. If P_2 reads A before B then it is not possible for final value of X to be 1. Hence, read for operands A and B must have happened out of order.

Let us consider all the events associated with above execution. We can construct a linear ordering of events under $A(CMP)$ and $A(RO)$ which shows that the above execution obeys $A(CMP)$ and $A(RO)$. Formally, let us consider the graph set for this execution under $A(CMP, RO)$. Every member of this graph set must contain the following cycle.[6]

$$\begin{array}{lll} (P_2, L_2, R, 1, B, S_2) & <_{SRW} & (P_2, L_2, W, 1, Y, S_2) \\ & <_{CWR} & (P_2, L_3, R, 1, Y, S_2) \end{array}$$

[6]We prefer to show a cycle involving only one RO arc and not show a cycle involving RO arc between B and $Y[j]$ read events or C and $U[k]$ read events. This is because we prefer not to include RO arcs involving local operands such as X and Y array as they may represent unshared memory or registers and we are typically interested in ordering between shared memory operands.

$$
\begin{array}{ll}
<_{SRW} & (P_2, L_3, W, 1, C, S_3) \\
<_{CWR} & (P_3, L_1, R, 1, C, S_3) \\
<_{SRW} & (P_3, L_1, W, 1, U, S_3) \\
<_{CWR} & (P_3, L_2, R, 1, U, S_3) \\
<_{SRW} & (P_3, L_2, W, 1, A, S_2) \\
<_{CWR} & (P_2, L_1, R, 1, A, S_2) \\
<_{RO} & (P_2, L_2, R, 1, B, S_2)
\end{array}
$$

A graphical view of this cycle is shown in Figure 4.16. The edges involved in the cycle are shown in bold. Most of the edges in the cycle are self-evident and represents the flow in which value 1 was propagated to all the variable.

4.6.5.1 $Test_{\text{RO}}$

A test for $A(CMP, RO)$ which is based on the execution shown above is described in Figure 4.17. The test is an extension of the above execution in its form.

If the memory system correctly realizes $A(CMP, RO)$ then Condition 7 is satisfied.[7]

CONDITION 7 (EXISTENCE) The values in array X and Y satisfy the property : $\forall i : 1 \leq i \leq k : \exists j : 0 \leq j < i : X[i] = Y[j]$

The condition essentially requires that every value of $X[i]$ must be equal to some value of $Y[j]$ where $j < i$, because otherwise $X[i]$ has a value same as some Y value occurring later in the program order. Hence, the read events for A and B corresponding to this X and Y values must have happened out of order, similarly to the case in the execution described above. The form of this condition is slightly different from all the previous conditions as it involves an existential qualifier which is necessary to indicate that the value of each $X[i]$ should be equal to some Y value occurring before it in program order. Now, we prove that the violation of condition 7 implies that $A(CMP, RO)$ is violated by the memory system.

First we show that if an execution of the $Test_{\text{RO}}$ obeys $A(CMP)$ then, every value in array X also occur in array Y (occurring earlier or possibly later in the program order). Intuitively, each value of *read(A)* must be result of some value of B passed on from P_2 to P_3 via C operand.

[7]The assignment $Y[0] := 0$; in the test is shown only for the sake of simplicity of the form of the condition. It eliminates need of a special case (in which X[i] value is 0), which could result in a slightly complicated form of the condition.

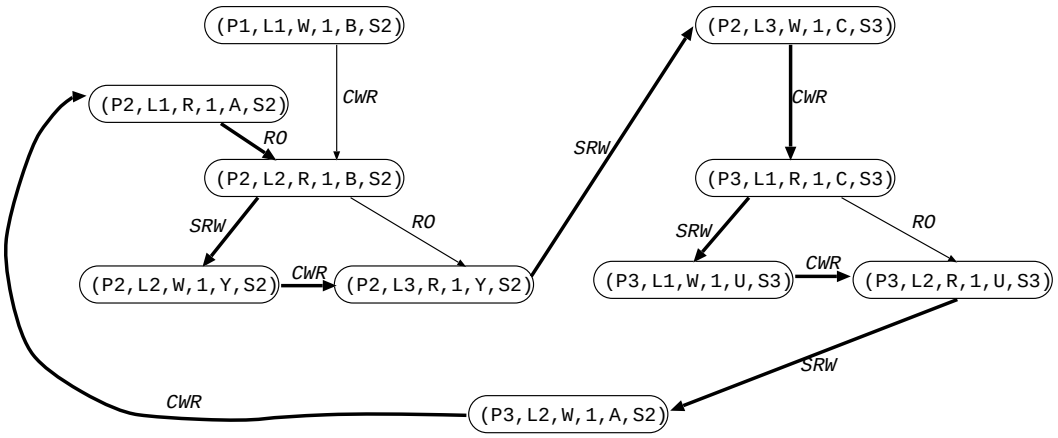

Figure 4.16. A cycle showing violation of $A(CMP, RO)$, which involves two read operations with different operands.

Initially, $A = B = 0$

P_1	P_2	P_3
	$L_{02} : Y[0] := 0;$	
$L_1 : B := 1;$	$L_{11} : X[1] := A;$	$L_{11} : U[1] := C;$
$L_2 : B := 2;$	$L_{12} : Y[1] := B;$	$L_{12} : A := U[1];$
$L_3 : B := 3;$	$L_{13} : C := Y[i];$	$L_{21} : U[2] := C;$
	$L_{21} : X[2] := A;$	$L_{22} : A := U[2]$
...	$L_{22} : Y[2] := B;$	...
	$L_{23} : C := Y[2];$	
...	...	...
$L_k : B := k$	$L_{k1} : X[k] := A;$	$L_{k1} : U[k] := C;$
	$L_{k2} : Y[k] := B;$	$L_{k2} : A := U[k];$
	$L_{k3} : C := Y[k];$	

Figure 4.17. $Test_{\text{RO}}$: a test for $A(CMP, RO)$, which stresses two different operand read events.

LEMMA 4.1 If an execution of $Test_{\text{RO}}$ obeys $A(CMP)$ then the final values in array X and Y are such that : $\forall i : 1 \leq i \leq k : \exists j : 0 \leq j \leq k : X[i] = Y[j]$.

Proof : Consider an array element, $X[i]$. If $X[i] = 0$ then, j = 0. If $X[i] \neq 0$ then, consider such a nonzero value say, $X[i] = \alpha$. Now, consider the path via which this nonzero value α is read. This value must have written by P_3 which in turn must have been read by some *read(C)*, which in turn must have read some value written by P_2 from some $Y[j]$. Formally, the following sequence must exist.

$$\begin{array}{lll} (P_2, L_{j2}, W, \alpha, Y[j], S_2) & <_{CWR} & (P_2, L_{j3}, R, \alpha, Y[j], S_2) \\ & <_{SRW} & (P_2, L_{j3}, W, \alpha, C, S_3) \\ & <_{CWR} & (P_3, L_{k1}, R, \alpha, C, S_3) \\ & <_{SRW} & (P_3, L_{k1}, W, \alpha, U[j], S_3) \\ & <_{CWR} & (P_3, L_{k2}, R, \alpha, U[j], S_3) \\ & <_{SRW} & (P_3, L_{k2}, R, \alpha, U[j], S_3) \\ & <_{CWR} & (P_2, L_{i1}, R, \alpha, A, S_2) \\ & <_{SRW} & (P_2, L_{i1}, W, \alpha, X[i], S_2) \end{array}$$

Hence, there must exist some $Y[j]$ which shares the same value as $X[i]$.[8]

□

Now, we show that the violation of condition 7 implies that $A(CMP, RO)$ is not obeyed.

THEOREM 4.3 If condition 7 is violated during an execution of $Test_{\text{RO}}$ then the execution does not obey $A(CMP, RO)$.

Proof : Consider a violation of condition 7 during an execution of $Test_{\text{RO}}$. Hence,

$$\exists i : 1 \leq i \leq k : \forall j : 0 \leq i : X[i] \neq Y[j]$$

Now, using the lemma 4.1, we know that there must be some $Y[j]$ corresponding to value $X[i]$. Hence,

$$\begin{array}{ll} & \exists i : 1 \leq i \leq k : \exists j : i < j \leq k : X[i] = Y[j] \\ \Longleftrightarrow & \exists i : 1 \leq i \leq k : \exists j : i < j \leq k : \exists \alpha : X[i] = Y[j] = \alpha \end{array}$$

Intuitively, there must be a $Y[j]$ value corresponding to $X[i]$ value later in the program order. Now, every member of the graph set for this execution under $A(CMP, RO)$ involves a cycle shown in Figure 4.18. This cycle is very similar to that of the execution

[8]Note that this argument does not imply that $Y[j]$ occurs before or after $X[i]$ in program order.

described earlier. This cycle involves the events corresponding to the following statements of $Test_{\text{RO}}$.

P_1	P_2	P_3
...	...	...
$L_l : B := \alpha;$	...	...
...	$L_{i1} : X[i] := A;$	...
...	...	...
...	...	...
	$L_{j2} : Y[j] := B;$	
	$L_{j3} : C := Y[j];$	
...	...	...
	...	$L_{m1} : U[m] := C;$
	...	$L_{m2} : A := U[m];$
...	...	...

Hence a violation of condition 7 implies a violation of $A(CMP, RO)$.

□

4.6.5.2 Test Automata for $Test_{\text{RO}}$

Test automata for $Test_{\text{RO}}$ is slightly involved then previous test automata. This is mainly due to the nature of the condition as it involves an existential qualifier. Intuitively, we need to catch any value of *read(A)* which has not already happened as a result of a previous *read(B)*. Such a value of *read(A)* shows a violation of read order between *read(A)* and some subsequent *read(B)* which returns the same value. We could capture this using one-bit by looking for a *read(A)* value to return 1 before any of the *read(B)* value returns 1.

A test automata for $Test_{\text{RO}}$ is shown in Figure 4.19. P_1 nondeterministically executes writing values 0 and 1 into B. P_3 reads value from C and writes it into A as in the test. P_2 reads a value from A, then reads a value from B and assigns it to C and so on. When P_2 is in state s_0, variable y keeps track whether any value of B read so far is 1. This is done by keeping a cumulative disjunction of all B values read so far into y. P_2 switches from state s_0 to s_1 nondeterministically and records the value of *read(A)* into x. P_2 continues to read A, read B and assign value of B to C in state s_1. However, it no longer updates y in state s_1.

The memory rule safety property for this test automata is :

$$P_2 \ in \ \ s_1 \Rightarrow (y = 0 \Rightarrow x = 0).$$

Intuitively, if final value of y is equal to 0 then x must be equal to 0 as well. Because, if $y = 0$ but $x = 1$ then during the execution, *read(A)* must have returned 1 while all the previous *read(B)*s returned 0, which is a violation similar to that of the test execution described above. Formally, we prove that whenever a condition violation occurs for $Test_{\text{RO}}$, test automata also reveals the error.

THEOREM 4.4 If condition 7 is violated during an execution, then there exists a test automata run in which the memory rule safety property is violated.

Proof : Consider a condition 7 violation. As argued previously,

$$\exists i : 1 \leq i \leq k : \exists j : i < j \leq k : \exists \alpha : X[i] = Y[j] = \alpha$$

Now, consider a nondeterministic run of test automata during which,

- P_1 executes $B := 0$ $\alpha - 1$ times, executes and $B := 1$ and continues executing $B := 0$ forever after, and
- P_2 loops on the first state s_0 for $i - 1$ times, and then switches to s_1.

Assuming data independence, this particular run results in $y = 0$ and $x = 1$. This could be seen by observing that x corresponds to value $X[i]$ and all the value of B prior to $X[i]$ are different from α and hence 0 in test automata.[9]

□

4.6.6 Pure Test for WO

ARCHTEST provides some tests of the form $A(CMP, UPO, WO)$ which are pure tests for WO. However, all these tests effectively check for $A(CMP, UPO, WOS)$ as the arc between two write events in the cycle corresponding to the violation is such that both write events have the same store. We would like to have tests that stress *all aspects* of the rule rather than a particular subrule only.

Consider the small program segment shown in Figure 4.20. Considering two processors in the system, there are four write events corresponding to two write operations. ARCHTEST tests check only for the violation of the WOS arcs, shown in dotted line. We

[9] An implicit and minor detail of the proof is to consider the case of smallest i for which $X[i] = \alpha$.

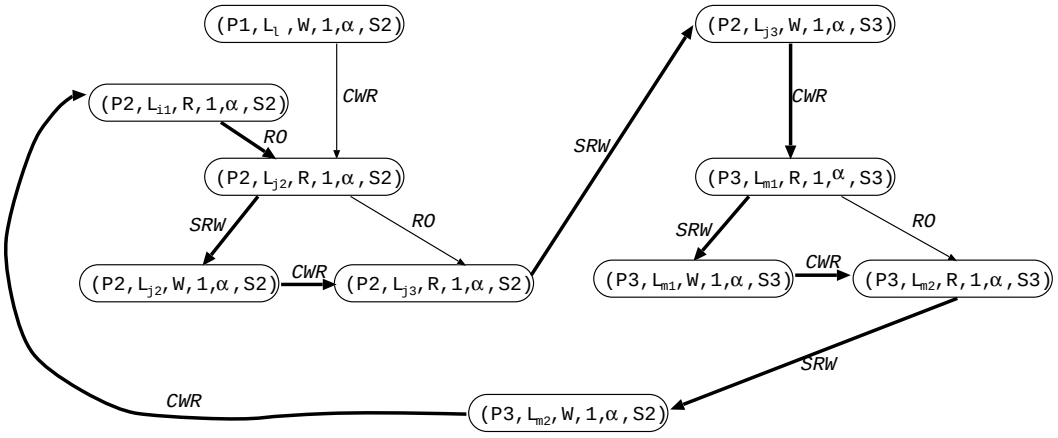

Figure 4.18. A cycle showing violation of $A(CMP, RO)$ corresponding to a violation of condition 7 of $Test_{\mathrm{RO}}$.

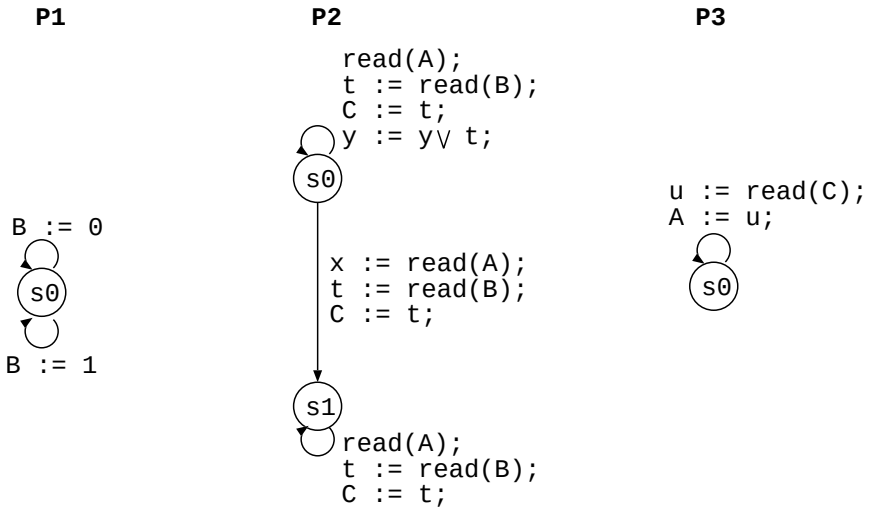

Figure 4.19. Test automata for $Test_{\mathrm{RO}}$.

would like to get a test which could check if the WO ordering involving events to *different* stores is violated or not.

As a first step towards such a test, consider the following execution. This execution obeys $A(CMP, UPO, WOS)$ but does not obey $A(CMP, UPO, WO)$.

$$\begin{array}{cc} \textit{Initially,} & A = B = U = V = 0 \\ P_1 & P_2 \\ L_1 : A := 1; & L_1 : B := 1; \\ L_2 : B := 2; & L_2 : A := 2; \\ L_3 : U := B; & L_3 : V := A; \\ \textit{Finally,} & U = V = 1 \end{array}$$

Consider the segment of a member of the graph set of the execution under $A(CMP, UPO, WO)$ involving only events pertaining to A and B operands. This is graphically shown in Figure 4.21. There is no cycle among these events if we consider only the arcs involving CMP, UPO and WOS. However, there is a cycle under $A(CMP, UPO, WO)$ shown by the bold edges.

Under $A(CMP, UPO, WO)$, every member of the graph set for the execution contains the following cycle.[10]

$$\begin{array}{lll} (P_1, L_1, W, 1, A, S_2) & <_{WO} & (P_1, L_2, W, 2, B, S_1) \\ & <_{CWW} & (P_2, L_1, W, 1, B, S_1) \\ & <_{WO} & (P_2, L_2, W, 2, A, S_2) \\ & <_{CWW} & (P_1, L_1, W, 1, A, S_2) \end{array}$$

The $_{CWW}$ arc between $(P_1, L_2, W, 2, B, S_1)$ and $(P_2, L_1, W, 1, B, S_1$ is induced between these two events because of the value of the read events $<_{CWR}$ arc) and $<_{UPO}$ arc. Similarly, the other $_{CWW}$ arc is induced.

4.6.6.1 *Test* WO

A Test for $A(CMP, UPO, WO)$ is proposed in Figure 4.22 P_1 is writing odd values to A and even values to B while reading values from B into an array U. Similarly, P_2 is writing odd values to B and even values to A while reading values from A into an array X. If the memory system obeys $A(CMP, UPO, WO)$, the following condition would be satisfied.

---

[10]Note that even though no arc labeled UPO is involved in the cycle, UPO is required for the cycle to exist.

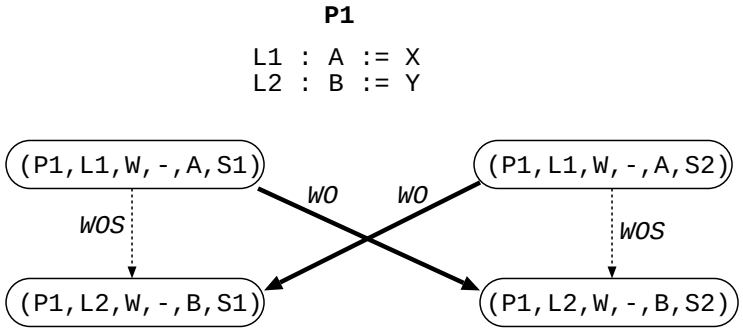

Figure 4.20. WO and WOS arcs.

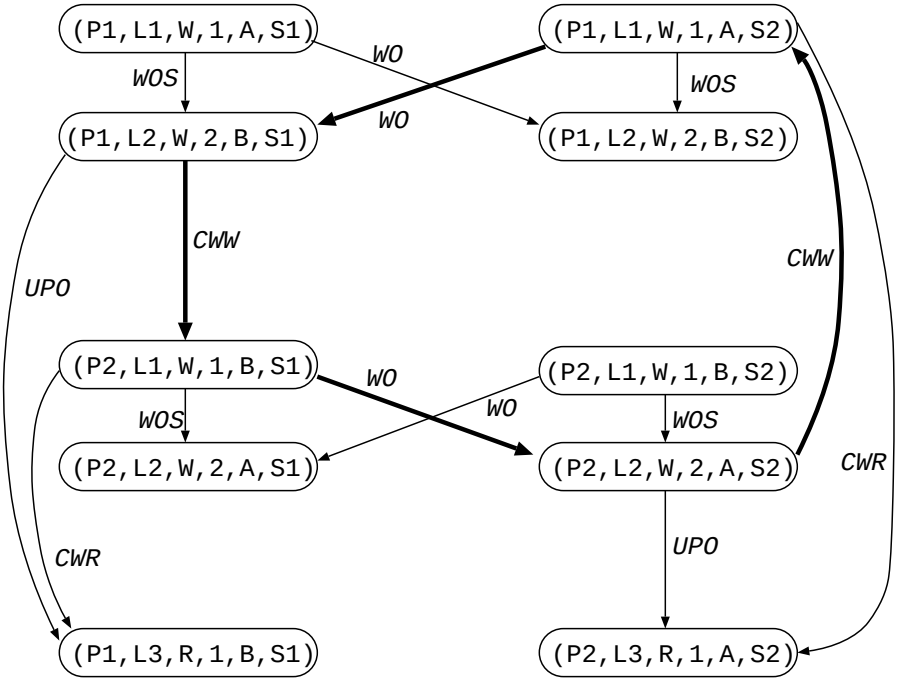

Figure 4.21. A cycle under $A(CMP, UPO, WO)$ but not under $A(CMP, UPO, WOS)$.

$$
\begin{array}{cc}
\multicolumn{2}{c}{\textit{Initially,}\ \ A = B = 0} \\
P_1 & P_2 \\
L_{a1} : A := 1; & L_{b1} : B := 1; \\
L_{b1} : B := 2; & L_{a1} : A := 2; \\
L_{u1} : U[1] := B; & L_{v1} : V[1] := A; \\
L_{a2} : A := 3; & L_{b2} : B := 3; \\
L_{b2} : B := 4; & L_{a2} : A := 4; \\
L_{u2} : U[2] := B; & L_{v2} : V[2] := A; \\
\cdots & \cdots \\
\cdots & \cdots \\
L_{ak} : A := 2k-1; & L_{bk} : B := 2k-1; \\
L_{bk} : B := 2k; & L_{ak} : A := 2k; \\
L_{uk} : U[k] := B; & L_{vk} : V[k] := A;
\end{array}
$$

Figure 4.22. $Test_{\text{WO}}$: a new test for $A(CMP, WO)$.

CONDITION 8 (WO_CROSS) $\forall i, j : 1 \leq i, j \leq k$:

$$
\begin{array}{c}
U[i]\ \textit{is even}\ \vee U[i] \geq 2j \vee \\
V[j]\ \textit{is even}\ \vee V[j] \geq 2i
\end{array}
$$

Now, we show that the violation of condition 8 implies that the memory system does not obey $A(CMP, UPO, WO)$. The execution and the cycle involved in proving this is similar to that of the execution shown above.

THEOREM 4.5 If condition 8 is violated during an execution of $Test_{\text{WO}}$ then the execution does not obey $A(CMP, UPO, WO)$.

Proof : Consider a violation of condition 8 during an execution of $Test_{\text{WO}}$.

$$
\begin{array}{lll}
 & \exists i, j : & U[i]\ \textit{is odd} \wedge U[i] < 2j \wedge \\
 & & V[j]\ \textit{is odd} \wedge V[j] < 2i \\
\Longleftrightarrow & \exists i, j, \alpha, \beta : & U[i] = \alpha \wedge V[j] = \beta \wedge \\
 & & \alpha\ \textit{is odd}\ \ \wedge \alpha < 2j \wedge \\
 & & \beta\ \textit{is odd}\ \ \wedge \beta <\ 2i
\end{array}
$$

Since α and β are odd, these values must have been originated from a write statement in P_2 and P_1 respectively, which are read by P_1 and P_2 respectively. Let $\alpha' = (\alpha + 1)/2$ and $\beta' = (\beta + 1)/2$. The statements of $Test_{\text{WO}}$ associated with the values involved in the condition violation shown above are :

P_1	P_2
$L_{a\beta'} : A := \beta;$	$L_{b\alpha'} : B := \alpha;$
$\ldots$	$\ldots$
$L_{ai} : A := 2i - 1;$	
$L_{bi} : B := 2i;$	
$L_{ui} : U[i] := B;$	
$\ldots$	$\ldots$
	$L_{bj} : B := 2j - 1;$
	$L_{aj} : A := 2j;$
	$L_{vj} : V[j] := A;$

Note that, since β is odd, write statement $A := \beta$ must occur in P_1 and since $\beta < 2i$, it must occur before $B := 2i$ in the program order. Similarly, $B := \alpha$ must occur in P_2 and it must occur before $A := 2j$ in program order.

Every member of the graph set for this execution under $A(CMP, UPO, WO)$ involves a cycle shown in Figure 4.23. This cycle is very similar to that of the execution described earlier. Hence, the execution does not obey $A(CMP, UPO, WO)$.

□

4.6.6.2 Test Automata for $Test_{\text{WO}}$

Test automata for $Test_{\text{WO}}$ is shown in Figure 4.24. Unlike previous test automata, we need two bits of data instead of one bit to be able to capture the difference between odd and even data values. As in $Test_{\text{WO}}$, P_1 writes odd values to A and even values to B and P_2 writes even odd values to B and even values to A. Data abstraction with two bits of data and nondeterminism are used to arrive at test automata by a process similar to earlier test automata.

The memory rule safety property for test automata for $Test_{\text{WO}}$ is :

$$P_1 \textit{ and } P_2 \textit{ in their final states } \Rightarrow$$
$$u \textit{ is even } \vee u = 11 \vee$$
$$v \textit{ is even } \vee v = 11$$

The memory rule safety property correlates directly with the test automata condition. When the safety property is violated, u and v has value 01, which is written by a write statement in other process occurring earlier in program order than the read statement. Hence, we observe a violation of $A(CMP, UPO, WO)$ just as in condition violation for $Test_{\text{WO}}$. Also, every condition violation of $Test_{\text{WO}}$ would be captured using test automata as proved below.

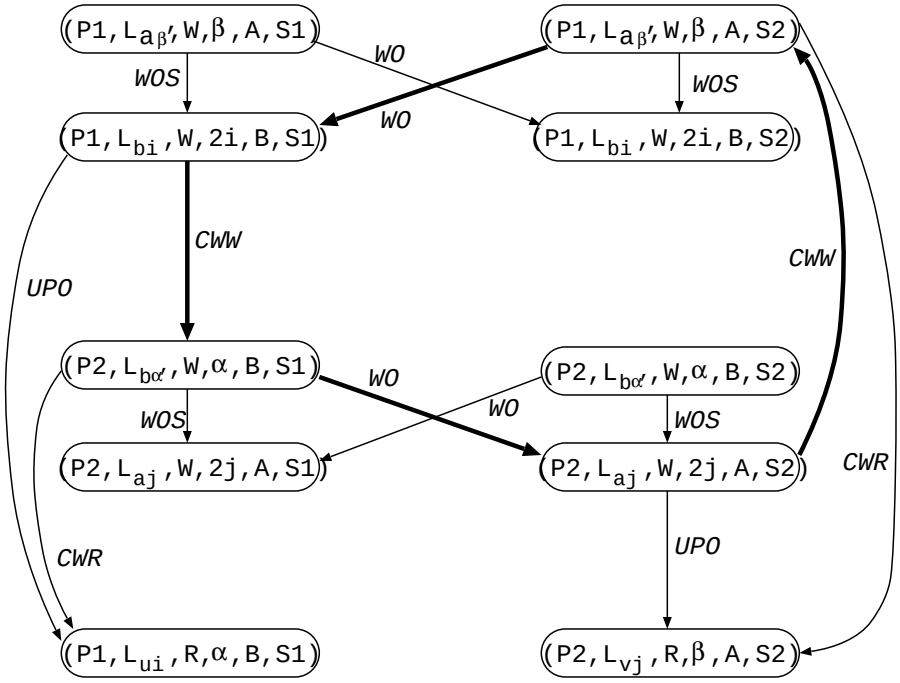

Figure 4.23. A cycle showing violation of $A(CMP, UPO, WO)$ corresponding to a violation of condition 8 of $Test_{\text{WO}}$.

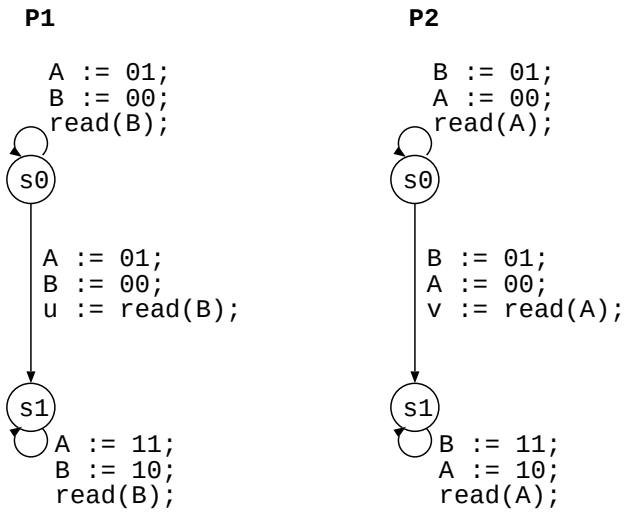

Figure 4.24. Test automata for $Test_{\text{WO}}$.

THEOREM 4.6 If condition 8 is violated during an execution of $Test_{\text{WO}}$, then there exists a test automata run in which the memory rule safety property is violated.

Proof : Consider a violation of condition 8 in an execution of $Test_{\text{WO}}$.

$$\begin{array}{lll} & \exists i,j: & U[i] \ is \ odd \wedge U[i] < 2j \wedge \\ & & V[j] \ is \ odd \wedge V[j] < 2i \\ \Longleftrightarrow & \exists i,j,\alpha,\beta: & U[i] = \alpha \wedge \ V[j] = \beta \wedge \\ & & \alpha \ is \ odd \ \ \wedge \alpha < 2j \wedge \\ & & \beta \ is \ odd \ \ \wedge \beta < 2i \end{array}$$

The statements of $Test_{\text{WO}}$ associated with the values involved in the condition violation shown above are :

P_1	P_2
$L_{a\beta'} : A := \beta;$	$L_{b\alpha'} : B := \alpha;$
$\ldots$	$\ldots$
$L_{ai} : A := 2i - 1;$	
$L_{bi} : B := 2i;$	
$L_{ui} : U[i] := B;$	
$\ldots$	$\ldots$
	$L_{bj} : B := 2j - 1;$
	$L_{aj} : A := 2j;$
	$L_{vj} : V[j] := A;$

Now, consider a nondeterministic run of test automata in which P_1 loops $i-1$ times on state s_0 and P_2 loops $j-1$ times on state s_0 before switching to s_1. Assuming data independence, u will have value 01 and v will have value 01 as both α and β are odd and $\alpha < 2j$ and $\beta < 2i$. Hence, memory rule safety property would be violated.

□

4.6.7 Testing for Atomicity in Weaker Memory Models

As we saw earlier, weaker memory models can relax atomicity in various orders. Some weaker memory models do not relax WA at all (e.g., IBM-370), while others obey a weaker version of WA such as WA–S (e.g., TSO). Some weaker memory models in PO-relaxation category may even relax atomicity to obey only CON. In this section, we see how we can check for various atomicity relaxations.

Let us consider the tests provided by ARCHTEST for checking for atomicity constraints. There are 3 tests provided by ARCHTEST for checking for WA described below. One of the tests have been described earlier and is briefly discussed below. Two other tests have not been previously described and hence we consider them in more detail.

4.6.7.1 $Test_{\text{WA1}}$: ARCHTEST Test for $A(CMP, RO, WO, WA)$

This test was described earlier, however it is reproduced here for the sake of convenience. $Test_{\text{WA1}}$, shown in Figure 4.25 tests for $A(CMP, RO, WO, WA)$, with the following condition :

CONDITION 9 (ATOMIC) $\forall i, j : 1 \leq i, j \leq k : V[i] \geq X[j] \vee Y[j] \geq U[i]$.

Let us examine the violation of the condition 9 and the relevant cycle of events. An atomicity violation is detected by this test when processes P_2 and P_3 observe writes done by P_1 and P_4 in different orders. The rule of $WA-S$ also guarantees that the processes P_2 and P_3 observe writes done by P_1 and P_4 in different orders. This can be seen by observing that stores S_2 and S_3 are in $WA-S$ atomic sets of write events of both P_1 and P_2. Hence, this test could also be used to detect violation of $A(CMP, RO, WO, WA-S)$. Note that this test stresses only one aspect of $WA-S$: writes done by two processes are observed in one unique order by all other processes. We can also use the test automata developed for $A(CMP, RO, WO, WA)$ to check for $A(CMP, RO, WO, WA-S)$.

4.6.7.2 $Test_{\text{WA2}}$: ARCHTEST Test for $A(CMP, RO, WO, WA)$.

We examine another ARCHTEST test for $A(CMP, RO, WO, WA)$ involving 3 processes. $Test_{\text{WA2}}$, shown in Figure 4.26 tests for $A(CMP, RO, WO, WA)$ with the condition checking being:

CONDITION 10 (ATOMIC2) $\forall i : 1 \leq i \leq k : X[i] \leq Y[i]$.

P_1 writes increasing values to A. P_2 reads from A and writes these values to B. P_3 reads from B and A repeatedly. A condition violation occurs when P_3 observes a larger

Initially, $A = B = 0$

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

Figure 4.25. $Test_{\text{WA1}}$: ARCHTEST test for $A(CMP, RO, WO, WA)$.

Initially, $A = B = 0$

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

Figure 4.26. *Test* $_{\text{WA2}}$: ARCHTEST test for $A(CMP, RO, WO, WA)$.

value of B followed by a smaller value of A. Intuitively a write from P_1 must not have been atomic if a larger value of A is visible to P_2 (which in turn gets passed to P_3 with write of B), but not to P_3 (which is inferred from a smaller value of A).

Formally, if condition 10 is violated in an execution of *Test* $_{\text{WA2}}$ then we show that $A(CMP, RO, WO, WA)$ is violated. Consider a violation of condition 10.

$$\begin{array}{ll} \exists i : 1 \leq i \leq k : & X[i] > Y[i] \\ \exists i : 1 \leq i \leq k, \alpha, \beta & X[i] = \beta \wedge Y[i] = \alpha \wedge \beta > \alpha \end{array}$$

Consider the statements involved corresponding to the values in the condition violation.

P_1	P_2	P_3
$\ldots$	$\ldots$	$\ldots$
$L_\alpha : A := \alpha;$	$\ldots$	$\ldots$
$\ldots$	$\ldots$	$\ldots$
$L_\beta : A := \beta;$	$\ldots$	$\ldots$
$\ldots$	$\ldots$	$\ldots$
$\ldots$	$L_{A_j} : U[j] := A;$	$\ldots$
$\ldots$	$L_{B_j} : B := U[j];$	$\ldots$
$\ldots$	$\ldots$	$\ldots$
$\ldots$	$\ldots$	$L_{B_i} : X[i] := B;$
$\ldots$	$\ldots$	$L_{A_i} : Y[i] := A;$

In each member of the graph set of this execution under $A(CMP, RO, WO, WA)$, the cycle shown in the Figure 4.27 exists. The three write events from the same write statement of P_1 are grouped to indicate the atomic set of events for WA. The cycle involved is shown by bold edges. Only arcs related to the cycle are shown for clarity and readability.

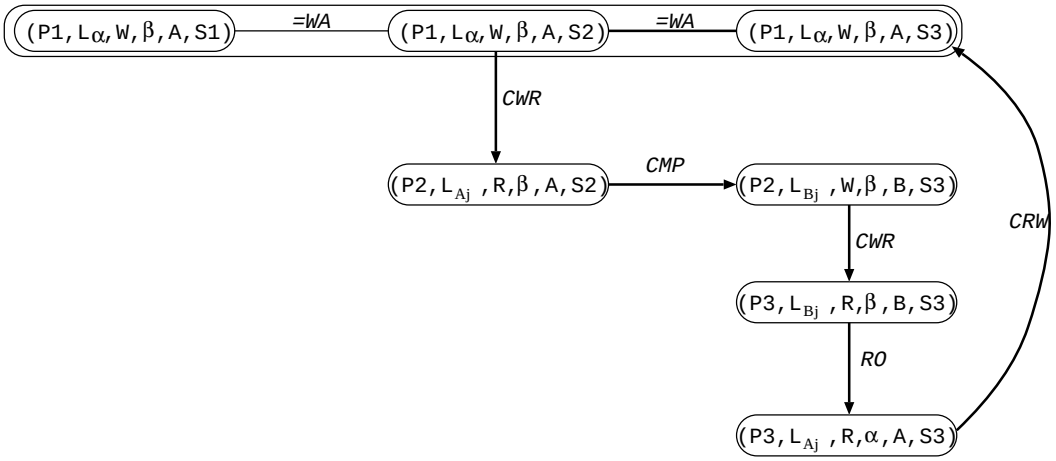

Figure 4.27. Cycle corresponding to the condition 10 violation of $Test_{\text{WA2}}$ involving WA.

We can show that each member of the graph set of this execution under $A(CMP, RO, WO, WA-S)$ also includes the same cycle as shown in Figure 4.27. Note that the $=_{WA}$ arc involved in the cycle is between two write events of P_1 in the stores of P_2 and P_3 (i.e., stores of *other* processors). This arc is present even if we consider only the atomic set of events for $WA-S$. So, each member of the graph set of this execution under $A(CMP, RO, WO, WA-S)$ has a cycle shown in Figure 4.28 Hence, $Test_{\text{WA2}}$ can also be used to check for $A(CMP, RO, WO, WA-S)$.

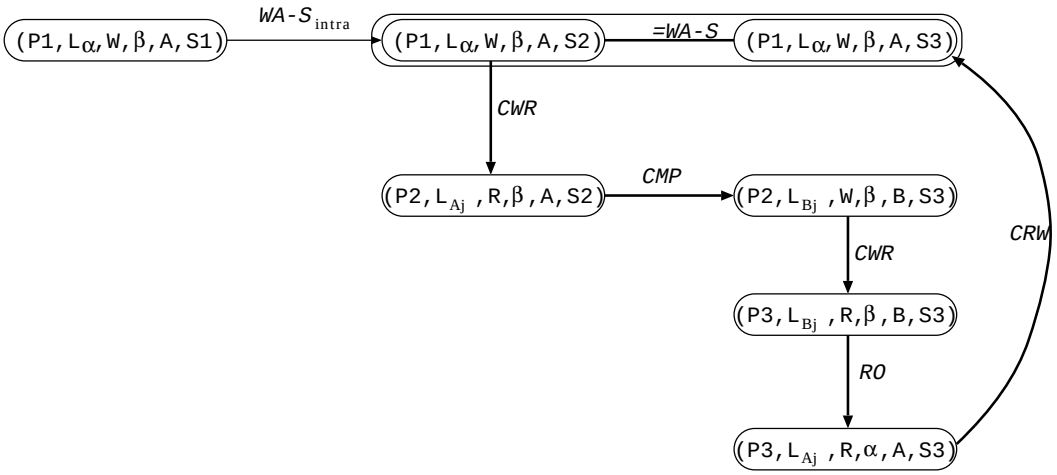

Figure 4.28. Cycle corresponding to the condition 10 violation of $Test_{\text{WA2}}$ involving $WA-S$.

4.6.7.3 Test Automata for $Test_{\text{WA2}}$

Test automata for $Test_{\text{WA2}}$ is presented in Figure 4.29.

The construction of this test automata from the ARCHTEST test is similar to most other tests we have considered earlier. Data abstraction with 1 bit of value is sufficient to arrive at the test automata. The memory rule safety property is :

$$P_3 \textit{ in its final state} \Rightarrow x < y$$

We now prove that every condition violation of $Test_{\text{WA2}}$ is captured by the test automata with a memory rule safety property violation.

THEOREM 4.7 If condition 10 is violated during an execution of $Test_{\text{WA2}}$, then there exists a test automata run in which the memory rule safety property is violated.

Proof: Consider a violation of condition 10 in an execution of $Test_{\text{WA2}}$.

$$\begin{array}{ll} \exists i : 1 \leq i \leq k : & X[i] > Y[i] \\ \exists i : 1 \leq i \leq k, \alpha, \beta : & X[i] = \beta \wedge Y[i] = \alpha \wedge \beta > \alpha \end{array}$$

Now, we construct a nondeterministic run of test automata correlating the above condition violation which violates the memory rule safety property. Consider a run of test automata in which P_1 loops α times on state s_0 before switching to state s_1. P_3 loops on state s_0 for $i-1$ iterations before switching to state s_1. Invoking data independence and address semi independence, x will have value 1 and y will have value 0. Hence, memory rule safety property would be violated.

□

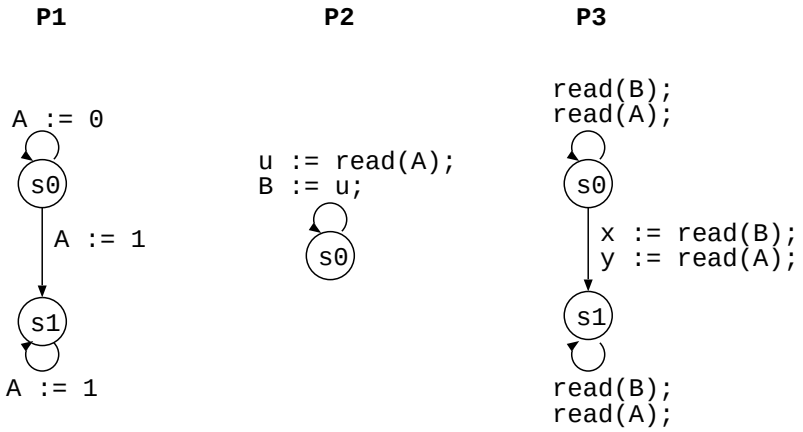

Figure 4.29. Test automata for $Test_{\text{WA2}}$.

4.6.7.4 $Test_{\text{WA3}}$: ARCHTEST Test for $A(CMP, UPO, RO, WO, WA)$

ARCHTEST provides another test for $A(CMP, UPO, RO, WO, WA)$ with just two processes. $Test_{\text{WA3}}$ is shown in Figure 4.30.

This test is a collapsed version of $Test_{\text{WA1}}$. P_1 and P_4 of $Test_{\text{WA1}}$ are merged into P_2 and P_3 to interleave writes with reads. The condition 9 for $Test_{\text{WA1}}$ can also be applied to $Test_{\text{WA3}}$ to check for $A(CMP, RO, WO, WA)$. In addition, the following condition is checked which detects a violation of $A(CMP, UPO, RO, WO, WA)$.

CONDITION 11 (ATOMIC3) $\forall i, j : 0 \leq i, j \leq k : V[i] \geq j \vee Y[j] \geq i$

We show how this condition violation detects a violation of $A(CMP, UPO, RO, WO, WA)$. Consider a violation of condition 11.

$$\begin{array}{ll} \exists i, j : 0 \leq i, j \leq k : & V[i] < j \wedge Y[j] < i \\ \exists \alpha, \beta, j, i : 0 \leq i, j \leq k : & V[j] = \beta \wedge Y[j] = \alpha \wedge \beta < j \wedge \alpha < i \end{array}$$

Consider the statements involved corresponding to the values in the condition violation.

$$\begin{array}{ll} P_1 & P_2 \\ \ldots & \ldots \\ L_{\alpha 1} : A := \alpha; & \ldots \\ \ldots & L_{\beta 1} : B := \beta; \\ \ldots & \ldots \\ \ldots & \ldots \end{array}$$

$$\begin{array}{ll} \multicolumn{2}{c}{\textit{Initially,}\ \ A = B = 0} \\ L_{11} : A := 1; & L_{11} : B := 1; \\ L_{12} : U[1] := A; & L_{12} : X[1] := B; \\ L_{13} : V[1] := B; & L_{13} : Y[1] := A; \\ L_{21} : A := 2; & L_{21} : B := 2; \\ L_{22} : U[2] := A; & L_{22} : X[2] := B; \\ L_{23} : V[2] := B; & L_{23} : Y[2] := A; \\ \ldots & \ldots \\ L_{k1} : A := k; & L_{k1} : B := k; \\ L_{k2} : U[k] := A; & L_{k2} : X[k] := B; \\ L_{k3} : V[k] := B; & L_{k3} : Y[k] := A; \end{array}$$

Figure 4.30. $Test_{\text{WA3}}$: ARCHTEST test for $A(CMP, UPO, RO, WO, WA)$.

$L_{i1} : A := i;$ ...
$L_{i2} : U[i] := A;$ ...
$L_{i3} : V[i] := B;$ ...
... ...
... ...
... $L_{j1} : B := j;$
... $L_{j2} : X[j] := B;$
... $L_{j3} : Y[j] := A;$

In each member of the graph set of this execution under $A(CMP, UPO, RO, WO, WA)$, the cycle shown in the Figure 4.31 exists. Only the arcs involved in the cycle are shown for clarity and readability. The cycle involved is shown by bold edges.

Now, consider the cycle involved when condition 9 is violated for $Test_{\text{WA3}}$. The condition 9 violation $V[i] < X[j] \wedge Y[j] < U[i]$ is essentially same as $V[i] < j \wedge Y[j] < U[i]$ This is so because $X[j] = j \wedge U[i] = i$ due to UPO. Hence, the cycle corresponding to

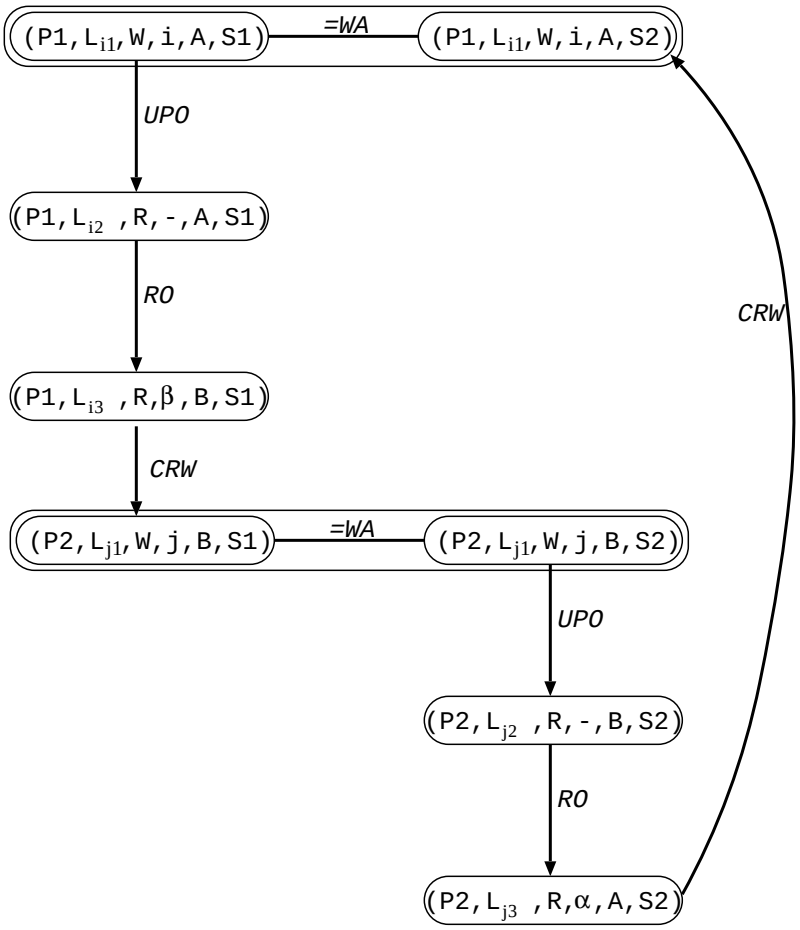

Figure 4.31. Cycle corresponding to the condition 11 violation of $Test_{\text{WA3}}$ involving WA.

the violation of condition 9 for $Test_{\text{WA3}}$ is essentially of the same nature as the one describe above.

Unlike previous two tests, $Test_{\text{WA3}}$ actually checks for violations of WA which are not violations of $WA-S$. Intuitively, this is so because there are only two processes involved and hence the atomic set of events for $WA-S$ contains only one event, i.e., no atomicity (as opposed to atomic set of events for WA which contains both the write events for each write operation - forcing both the processes to observe a write instantly). So, $WA-S$ induces only one arc in this case : the arc indicating that a write occurs in the writing process's store before it becomes visible to other process. If we induce this arc in addition with CMP, UPO, RO and WO arcs , the cycle does not exist as shown in Figure 4.32. Hence, $Test_{\text{WA3}}$ is strictly stronger than $Test_{\text{WA1}}$ and $Test_{\text{WA2}}$ and is a very useful test for WA as it can check memory systems which obey $WA-S$ but not WA. [11]

4.6.7.5 Test Automata for $Test_{\text{WA3}}$

$Test_{\text{WA3}}$ has two conditions 9 and 11 that can be used to check for WA. These two conditions differ in the way that 9 compares the array values U, V, X, Y whereas 11 compares the array values V, Y with indices i, j. However, these two conditions are very similar because of UPO which enforces that $\forall i : U[i] = i \wedge X[j] = j$. We have presented the test automata corresponding to each of these conditions and provided a proof of the completeness of the first one below. The proof of the other test automata is very similar in nature to the first one.

Test automata for $Test_{\text{WA3}}$ corresponding to condition 9 is shown in Figure 4.33. The memory rule safety property for this test automata is :

$$P_1 \textit{ and } P_2 \textit{ in their final states} \Rightarrow v \geq x \vee y \geq u$$

We show that each violation of 9 of $Test_{\text{WA3}}$ would be captured by the test automata memory rule safety property violation.

---

[11] This may be an explanation to the observation mentioned in the research problems on ARCHTEST's web page [13] which states that "Tests T5, T6, and T7 all test for a relaxation of write atomicity. Test T7 shows that numerous machines relax WA. No machine to date has failed T5 or T6. Why do the machines that fail test T7, never fail the logically equivalent tests T5 or T6?" (here T5, T6 and T7 correspond to $Test_{\text{WA2}}$, $Test_{\text{WA1}}$ and $Test_{\text{WA3}}$).

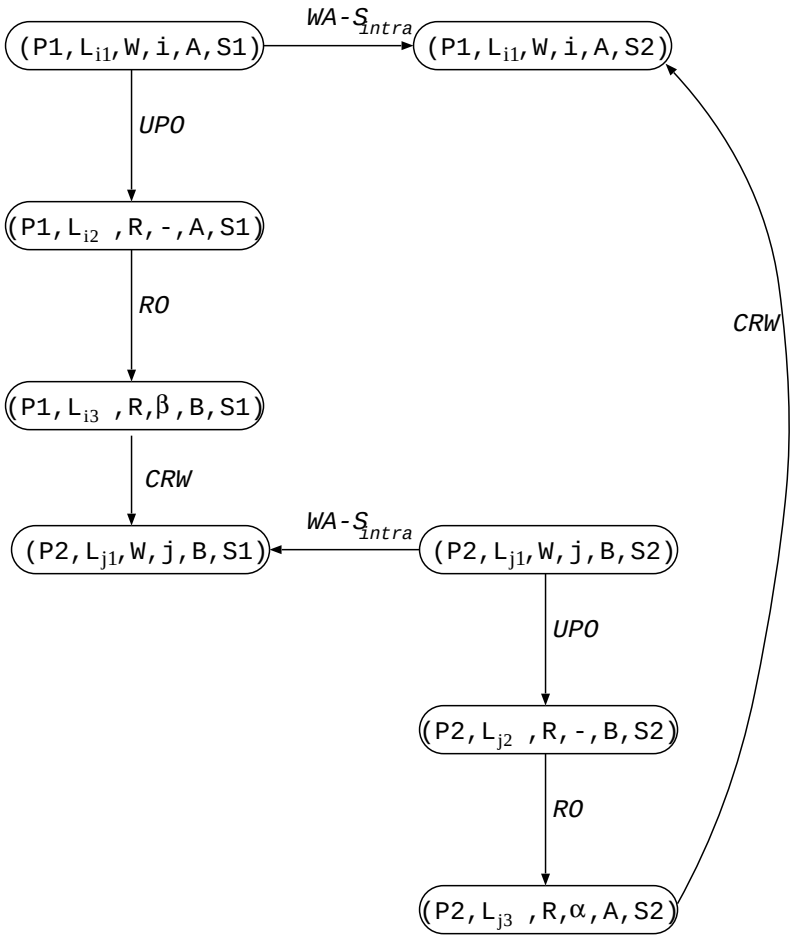

Figure 4.32. No cycle corresponding to the violation of condition 11 of $Test_{\text{WA3}}$ involving $WA-S$.

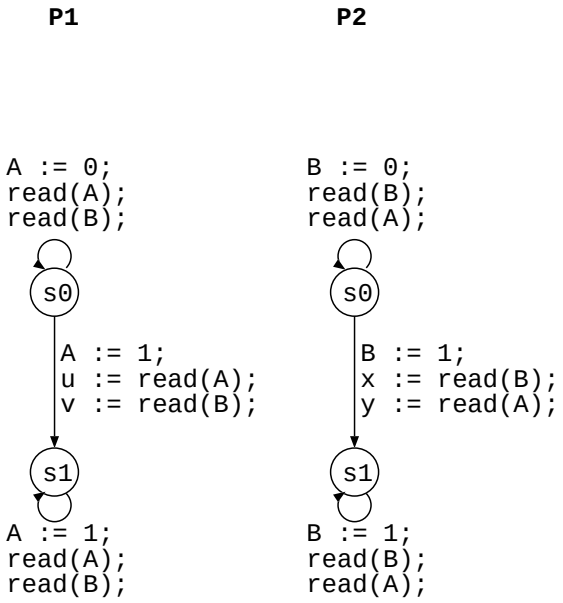

Figure 4.33. Test automata for $Test_{\text{WA3}}$.

THEOREM 4.8 If condition 9 is violated during an execution of $Test_{\text{WA1}}$, then there exists a test automata run in which the memory rule safety property is violated.

Proof: Consider a violation of condition 9 in an execution of $Test_{\text{WA3}}$.

$$\exists i,j : 1 \leq i,j \leq k : \quad V[i] < X[j] \wedge Y[j] < U[i]$$
$$\exists i,j,\alpha,\beta : 1 \leq i,j \leq k : \quad V[i] = \beta \wedge X[j] = j \wedge Y[j] = \alpha \wedge U[i] = i \wedge \beta < j \wedge \alpha < i$$

The fact that $X[j] = j$ and $U[i] = i$ follows from the implicit assumption that $A(CMP, UPO)$ is obeyed.

Now, we construct a nondeterministic run of test automata correlating the above condition violation which violates the memory rule safety property. Consider a run of test automata in which P_1 loops α times on state s_0 before switching to state s_1. P_2 loops β times on state s_0 before switching to state s_1. Invoking data independence and address semi dependence, $u = x = 1 \wedge v = y = 0$. Hence, memory rule safety property would be violated.

□

Test automata for $Test_{\text{WA3}}$ corresponding to condition 11 is shown in Figure 4.34. The memory rule safety property for this test automata is :

$$P_1 \textit{ and } P_2 \textit{ in their final states } \Rightarrow v \vee y$$

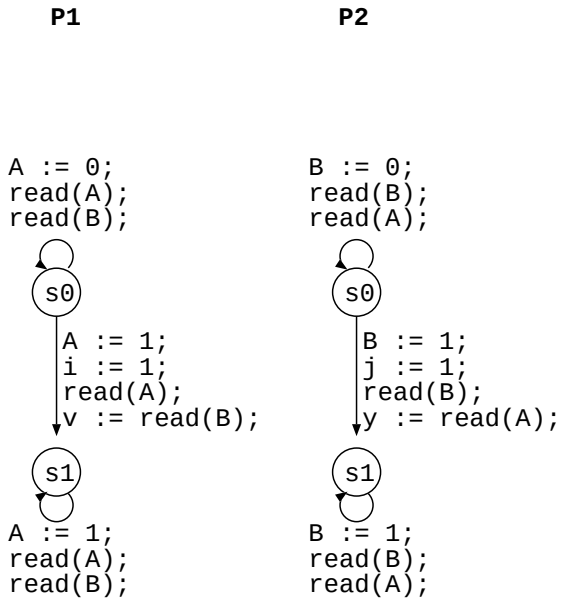

Figure 4.34. Test automata 2 for $Test_{\text{WA3}}$.

The proof of the completeness of this test automata is similar to the above proof.

$Test_{\text{WA1}}$ and $Test_{\text{WA2}}$ can and should be used for checking $WA-S$. However, they do not stress all the aspects of $WA-S$ that should be checked for. One of the aspect of $WA-S$ we would like to stress and which is not stressed in $Test_{\text{WA1}}$ and $Test_{\text{WA2}}$ is the arc enforcing that a process's write becomes visible to its store before it becomes visible to other process's stores. [12] This is exactly what the $WA-S_{intra}$ arc represents. Now, we consider how we can observe a violation of $WA-S_{intra}$ arc.

Consider the following execution which obeys $A(CMP, UPO)$ but does not obey $A(CMP, UPO, WA-S_{intra})$.

$$\begin{array}{cc} \textit{Initially,}\ \ A = B = U = X = 0 & \\ P_1 & P_2 \\ L_1 : U := A; & L_1 : X := B; \\ L_2 : B := U; & L_2 : A := X; \\ & L_3 : A := 1; \\ \textit{Finally,}\ \ U = X = 1 & \end{array}$$

[12] This may not be as trivially guaranteed as it sounds. Consider, for example, a multi-threaded CPU which executes two streams of instructions simultaneously on a single chip. For various reasons, it is often the case that such designs involve a single shared store buffer. In such cases it is possible that a process's write (stored in the common store buffer) may become visible to other process before it becomes visible to itself(if the shared store buffer is not implemented carefully).

Intuitively this execution could be explained as follows. Consider the new value 1 of A, which is written by L_3 of P_2. This value becomes visible to P_1 before it becomes visible to P_2, which in turn is written into B and read into X, following which the writes of A into store S_2 happens. Hence, $A(CMP, UPO)$ is obeyed but $A(CMP, UPO, WA{-}S_{intra})$ is violated.

The cycle corresponding to this violation is shown in Figure 4.35. Note that without the $WA{-}S_{intra}$ arc the cycle would not be possible. Hence, the execution does not obey $A(CMP, UPO, WA{-}S_{intra})$, however the execution does obey $A(CMP, UPO)$ as we can construct a linear ordering of events of the execution under $A(CMP, UPO)$, which shows that the execution does obey $A(CMP, UPO)$.

Now, let us consider the case when memory system obeys CON. Imposing CON arc for this particular execution leads to a cycle without any $WA{-}S_{intra}$ arc in it as shown in Figure 4.36. Hence, the execution does not obey the $A(CMP, UPO, CON)$. So, if a memory system obeys $A(CMP, UPO, CON)$ then the above execution would not be very helpful in detecting whether $WA{-}S_{intra}$ is violated by the memory system or not. Since, even most relaxed practical memory systems obey CON, we would like to find an execution which obeys $A(CMP, UPO, CON)$ but does not obey $A(CMP, UPO, CON, WA{-}S_{intra})$. However, we show below that this is not possible. In other words, if the memory system obeys $A(CMP, UPO, CON)$ then it also obeys $A(CMP, UPO, CON, WA - S_{intra})$.

Intuitively, this could be explained as follows. Suppose the memory system obeys $A(CMP, UPO, CON)$ but violated $WA - S_{intra}$. To violate $WA - S_{intra}$, there must be a write to some operand, say A by some process, say P_j which was seen by some other process, say P_i before it was observed by P_j. The only way to "catch" that P_j observed this value before P_i did is by "somehow" writing another value into store S_j which derives its value from the write of P_j and which is read before P_j's write. If this write into store S_j is to the operand A then CON is violated, as there are some two processes which observe these two values differently. If this write into store S_j is to an operand other than A (as it was in the execution shown above), there is a consequential violation of CON (as it was in the execution shown above). The formal proof is given below.

THEOREM 4.9 If an execution obeys $A(CMP, UPO, CON)$ then it also obeys $A(CMP, UPO, CON, WA - S_{intra})$.

Proof: Consider an execution which obeys $A(CMP, UPO, CON)$ but does not obey $A(CMP, UPO, CON, WA - S_{intra})$. Consider the graph set of this execution under

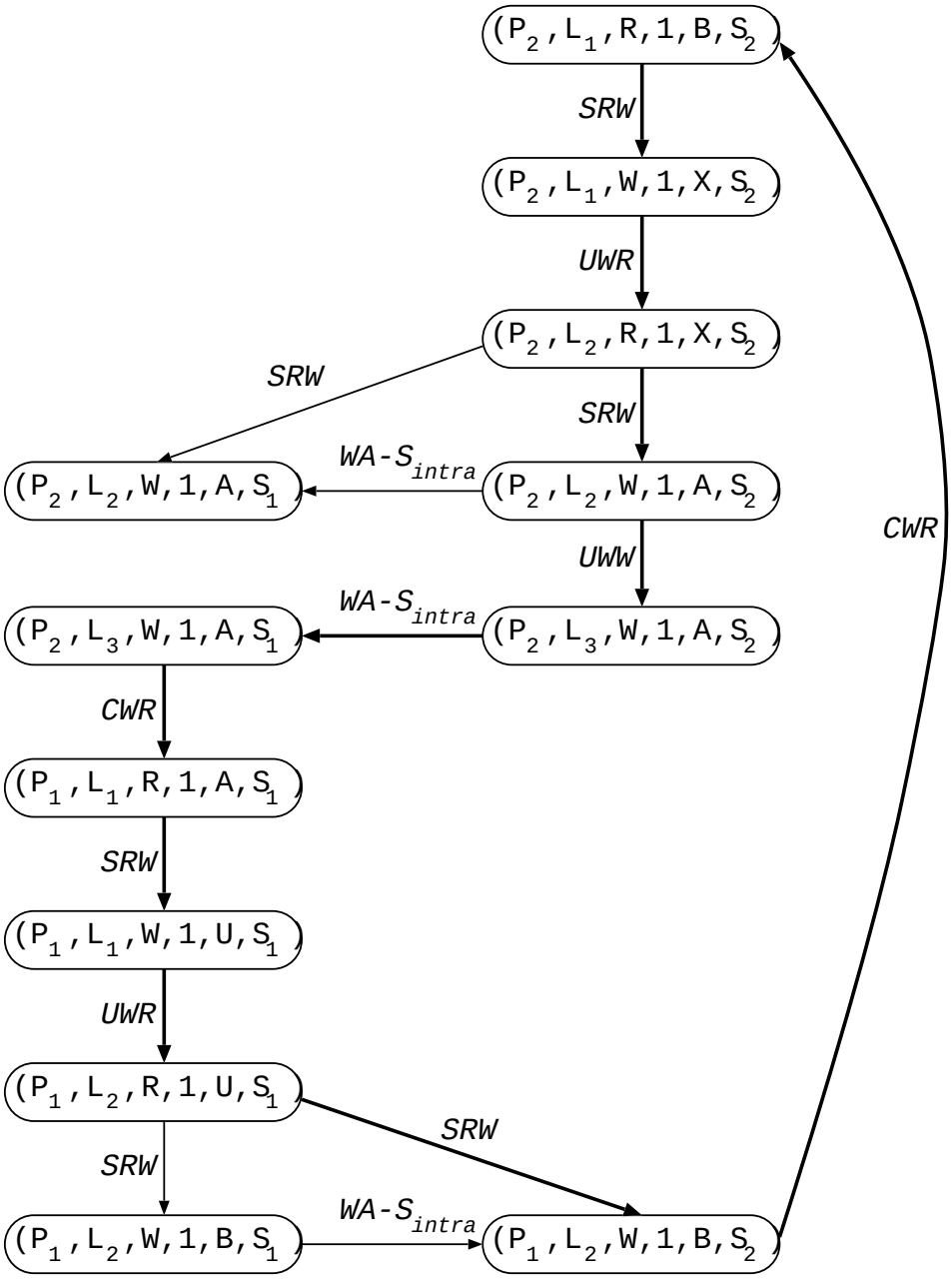

Figure 4.35. The cycle showing violation of $A(CMP, UPO, WA-S_{intra})$.

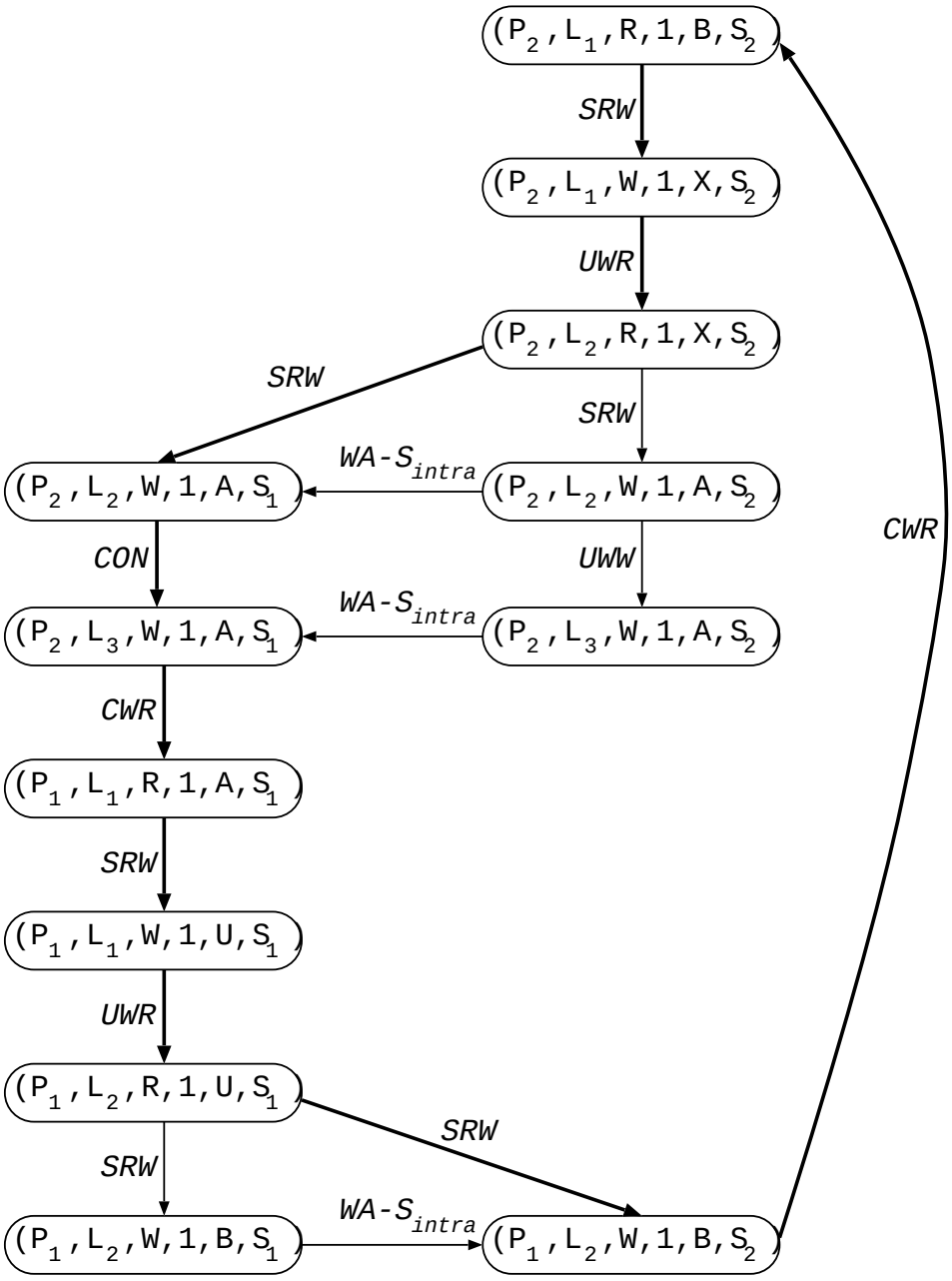

Figure 4.36. The cycle involving only CMP,UPO and CON arcs.

$A(CMP, UPO, CON)$. This graph set has at least one member which is cycle-free. Imposing WA–S_{intra} arc on this graph set yields a graph set for $A(CMP, UPO, CON, WA$–$S_{intra})$. Hence, each member of this graph set involves a cycle which involves at least one $WA-S_{intra}$ arc in it.

Let us consider such a cycle. Consider such a cycle *of minimum length*. Let the $WA-S_{intra}$ arc be from write event (P_j, L_p, W, v, A, S_j) to (P_j, L_p, W, v, A, S_i) in this cycle as shown in Figure 4.37.

We would show that such a cycle is not possible by showing a contradiction. We would demonstrate that we can obtain another cycle of *smaller length* of the same type, contradicting our assumption above that this cycle is of minimum length.

Let us consider two different cases depending upon whether there exists a write event from P_j into store S_j for the operand A which occurs in program order before (P_j, L_p, W, v, A, S_j).

- **Case I :** The cycle involves a write event from P_j into store S_j for the operand A which comes before (P_j, L_p, W, v, A, S_j) in the program order. Such a write event would be of the form $(P_j, L_q, W, -, A, S_j)$ where statement labeled L_q comes before L_p in program order. These write events and the cycle involved are shown in Figure 4.38. The CON arc between $(P_j, L_q, W, -, A, S_i)$ and (P_j, L_p, W, v, A, S_i) follow because of UWW ordering and the definition of CON. As shown in the figure, cycle 1 and cycle 2 both are of the same length and both involve the $WA-S_{intra}$ arc. So, we can consider cycle 2 instead of cycle 1 in our discussion without loss of generality.

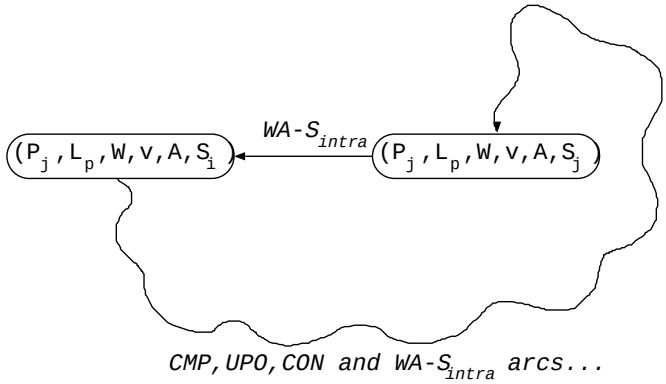

Figure 4.37. Cycle involving $WA-S_{intra}$ arc.

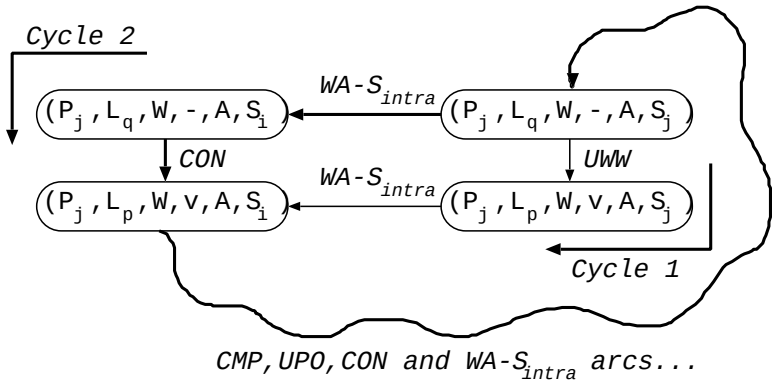

Figure 4.38. Two write events from the same process to the same operand and store in the cycle involving $WA-S_{intra}$ arc : We can consider cycle 2 instead of cycle 1.

- **Case II:** In this case, the cycle does not involve any write event from P_j into store S_j for the operand A which comes before (P_j, L_p, W, v, A, S_j).

So, in either case we have a cycle involving $WA-S_{intra}$ arc that involves a write event (P_j, L_p, W, v, A, S_j) such that,

1. the cycle involves WA–S_{intra} arc between (P_j, L_p, W, v, A, S_j) and (P_j, L_p, W, v, A, S_i), and
2. the cycle does not involve any write event from P_j into store S_j for the operand A which comes before (P_j, L_p, w, v, A, S_i) in the program order, and
3. the cycle is of minimum length.

Now, we do a case analysis depending on the type of the arc of the cycle right before the event (P_j, L_p, W, v, A, S_j). Consider such an arc e_1 as shown in Figure 4.39. e_1 must be one of $CWR, CRW, CWW, SRW, UWW, UWR, URW, CON$ or $WA-S_{intra}$.

- **Case I :** e_1 is CWR, UWR or $WA-S_{intra}$.

  This case is not possible as the transition template for these rules do not match (P_j, L_p, W, v, A, S_j). e_1 cannot be CWR or UWR because (P_j, L_p, W, v, A, S_j) is not a read event. e_1 cannot be $WA-S_{intra}$ because (P_j, L_p, W, v, A, S_j) has the process and the store component same.

- **Case II:** e_1 is UWW.

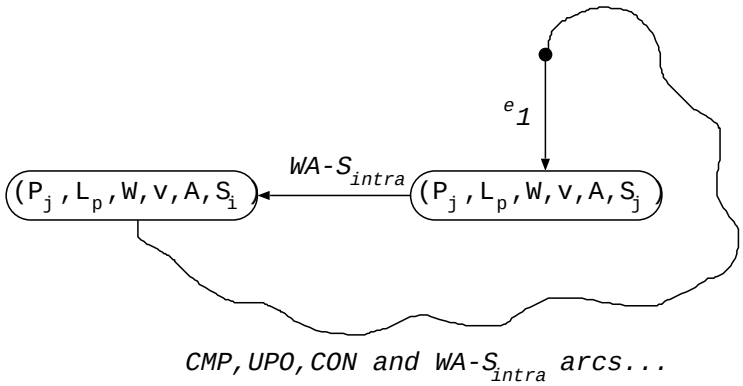

Figure 4.39. Arc e_1 of the cycle just before the event (P_j, L_p, W, v, A, S_j).

This case is not possible because it contradicts our assumption above that the cycle does not involve any write event from the same process to the same store and the same operand occurring earlier in the program order.

- **Case III:** e_1 is SRW.

  This case is shown in Figure 4.40. Consider the read event r preceding (P_j, L_p, W, v, A, S_j) in the cycle. There is also a SRW arc between event r and the event (P_j, L_p, W, v, A, S_j). This arc gives us a cycle of smaller length than the original cycle, which contradicts our assumption. Hence, this case is not possible.

- **Case IV:** e_1 is CRW or URW.

  This case is shown in Figure 4.41. In this case, the event preceding arc e_1 in the cycle must be a read event r from P_j to the operand A. Consider the arc e_2 just

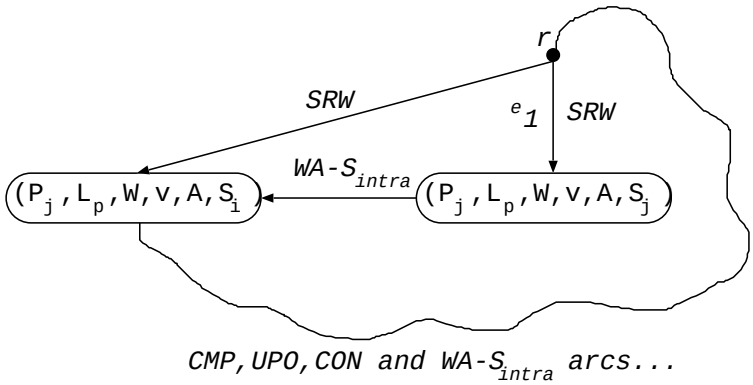

Figure 4.40. Case $e_1 = SRW$: in this case, we get a smaller cycle by the direct SRW arc.

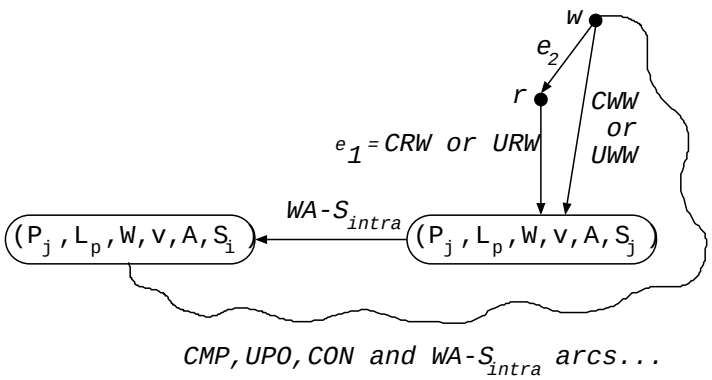

Figure 4.41. Case $e_1 = CRW \ or \ URW$: in this case, we get a smaller cycle by the direct CWW or UWW arc.

before the even r in the cycle. e_2 could be either CWR or UWR (because these are the only two cases when the possible for a read event). In either case the event w just before the arc e_2 must be a write event to the store S_j. Hence, there must be a CWW or UWW arc between w and (P_j, L_p, W, v, A, S_j). This arc gives us a cycle of smaller length than the original cycle, which contradicts our assumption. Hence, this case is not possible.

- **Case V:** e_1 is CWW or CON.

  This case is shown in Figure 4.42. In this case, the event preceding arc e_1 in the cycle must be a write event w from some process P_m into store S_j to the operand A, say $(P_k, -, W, -, A, S_j)$. Note that there is also a CON arc from $(P_k, -, W, -, A, S_i)$ to (P_j, L_p, W, v, A, S_i) induced because of the ordering between the write events in store S_j.

  Now, consider the arc e_2 in the cycle preceding the write event $(P_k, -, W, -, A, S_i)$. We do a case analysis on the type of event e_2.

  - **Case I:** e_2 is CWR, UWR, URW or UWW.

    e_2 cannot be CWR or UWR as $(P_k, -, W, -, A, S_j)$ is a write event. e_2 cannot be URW or UWW as $(P_k, -, W, -, A, S_j)$ does not have the same process and store components (which is necessary for URW and UWW arcs). Hence, this case is not possible.

  - **Case II:** e_2 is CWW or CON.

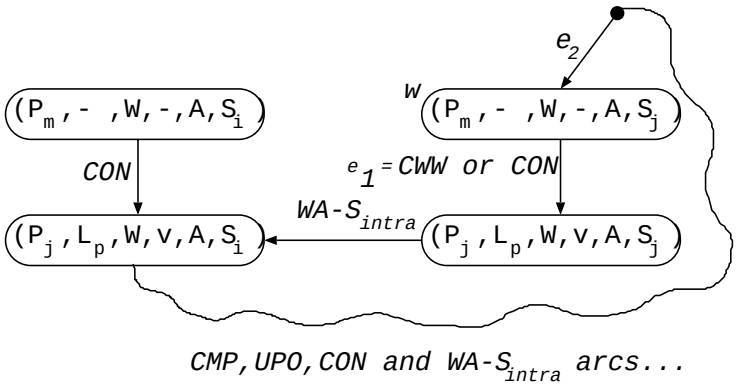

Figure 4.42. Case $e_1 = CWW \ \ or \ \ CON$: in this case, further case analysis is needed.

This case is shown in Figure 4.43. Consider the event preceding arc e_2 in the cycle. This event must be a write event, say w' from some process to operand A into store S_j (this is evident from the transition template of CWW and CON). There also must exist a CWW or CON event from w' to (P_j, L_p, W, v, A, S_j). This gives a cycle of a smaller length, which is a contradiction. Hence, this case is not possible.

– **Case III:** e_2 is SRW.

This case is shown in Figure 4.44. Consider the event preceding arc e_2 in the cycle. This event must be a read event from P_m into store S_m on operand A. Consider this event $(P_m, -, R, -, A, S_m)$. We have a SRW arc from this event $(P_m, -, R, -, A, S_m)$ to $(P_m, -, W, -, A, S_i)$. Hence, we can consider *Cycle 2* instead of a *Cycle 1* and obtain a cycle of smaller length. Hence, this case is not possible.

– **Case IV:** e_2 is $WA{-}S_{intra}$.

This case is shown in Figure 4.45. Consider the event preceding arc e_2 in the cycle. This event must be either a write event from P_m into store S_m on operand A. Consider this event $(P_m, -, W, -, A, S_m)$. We have a $WA{-}S_{intra}$ arc from this event $(P_m, -, W, -, A, S_m)$ to $(P_m, -, W, -, A, S_i)$. Hence, we can consider *Cycle 2* instead of a *Cycle 1* and obtain a cycle of smaller length. The only exception to the above scenario is when $m = i$ in which case $(P_m, -, W, -, A, S_m) = (P_m, -, W, -, A, S_i)$ and we have a smaller cycle in this case as well using the CON arc between $(P_m, -, W, -, A, S_m)$ and (P_j, L_p, W, v, A, S_i) events. Hence, this case is not possible.

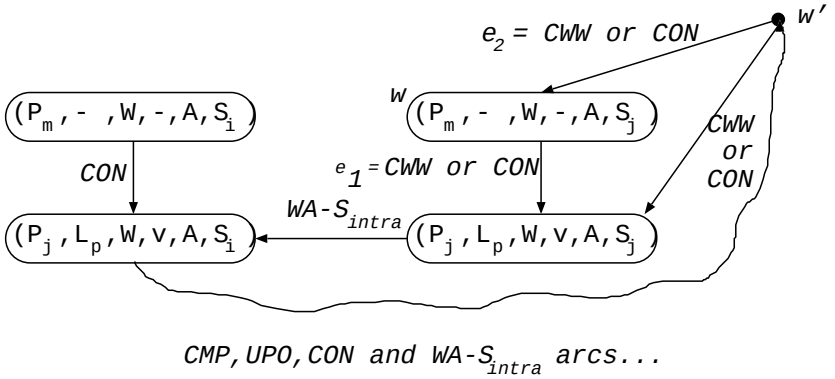

Figure 4.43. Case $e_1 = CWW \ or \ CON$ and $e_2 = CWW \ or \ CON$: we have a cycle of smaller length.

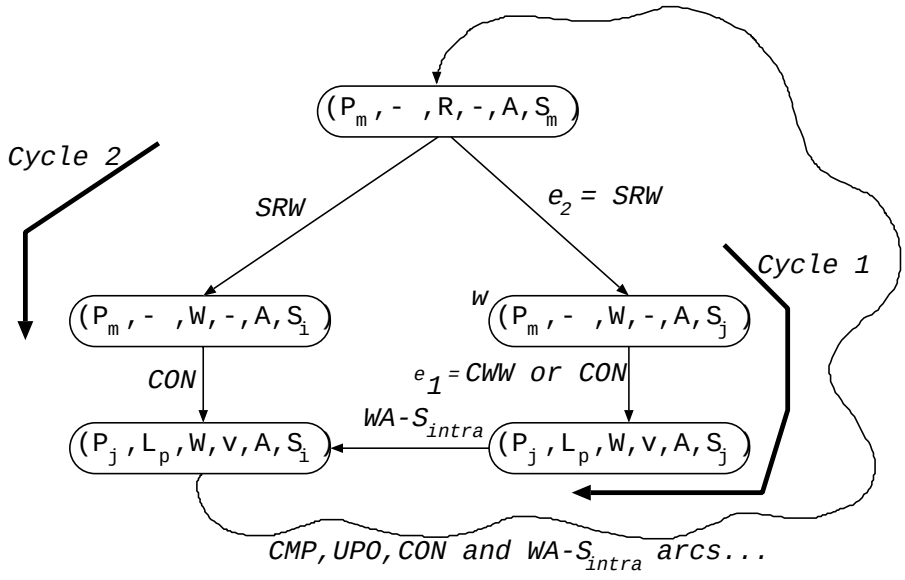

Figure 4.44. Case $e_1 = CWW \ or \ CON$ and $e_2 = SRW$: we have a cycle of smaller length in this case too.

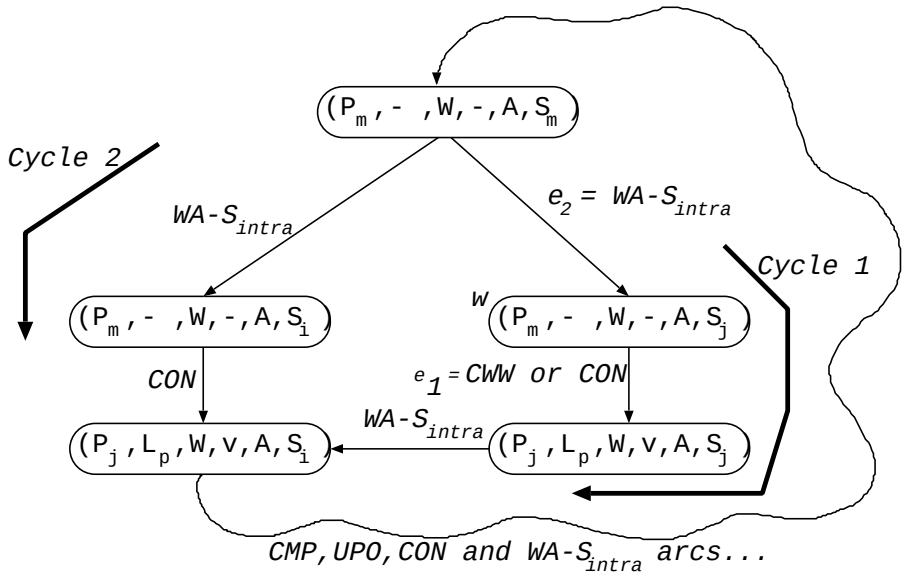

Figure 4.45. Case $e_1 = CWW \ or \ CON$ and $e_2 = WA-S_{intra}$: we have a cycle of smaller length in this case too.

– **Case V:** e_2 is CRW.

This case is shown in Figure 4.46. Consider the event preceding arc e_2 in the cycle. This event must be a read event, say r from process P_j to operand A into store S_j (this is evident from the transition template of CRW). There also must exist a CRW or URW event from r to (P_j, L_p, W, v, A, S_j). This gives a cycle of a smaller length, which is a contradiction. Hence, this case is not possible.

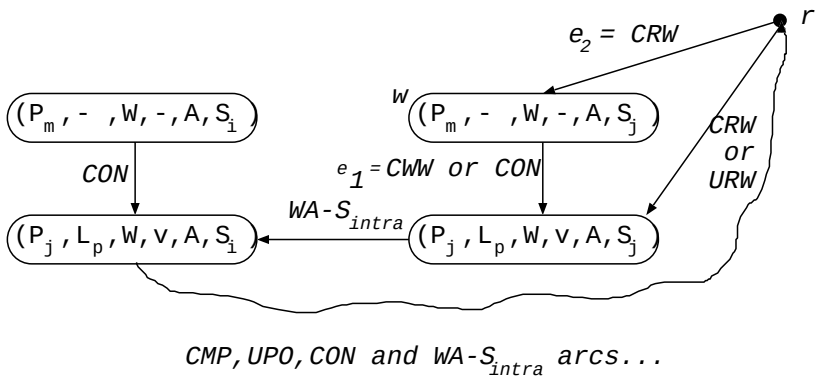

Figure 4.46. Case $e_1 = CWW \ or \ CON$ and $e_2 = CRW$: we have a cycle of smaller length.

Thus, by case analysis on the e_2 arc, this case is not possible.

In each of the above case analysis we reached to a contradiction and the case analysis is exhaustive. Hence, proved.

□

We showed that if an execution obeys $A(CMP, UPO, CON)$ then it also obeys $A(CMP, UPO, CON, WA-S_{intra})$. In other words, in the presence of rules CMP, UPO and CON only, it is not possible to detect a violation of $WA-S_{intra}$. Now, let us consider if it is possible to detect the violation of $WA-S_{intra}$ rule in the presence of CMP, UPO, CON and some other rule(s). [13]

First, we show that even in the presence of rules $CMP, UPO, CON, =_{WA\text{–}S}$, it is not possible to detect a violation of $WA-S_{intra}$.

THEOREM 4.10 If an execution obeys $A(CMP, UPO, CON, =_{WA\text{–}S})$ then it also obeys $A(CMP, UPO, CON, =_{WA\text{–}S}, WA-S_{intra})$.

Proof: For the sake of brevity, we would not provide a complete formal proof as in the previous case. The proof follows on a line similar to the earlier proof and involves one new case analysis.

Consider the case when $e1$ is CWW or CON and $e2$ is $=_{WA\text{–}S}$. This case is shown in Figure 4.47. In this case also, we can get a smaller cycle by considering the *Cycle 2* involving $=_{WA\text{–}S}$ and CON arc as shown in Figure 4.47. Hence, proved.

□

Since, WA is a strictly stronger rule than $WA-S$, we can obviously not detect a violation of $WA-S_{intra}$ in the presence of rules CMP, UPO, CON, WA We can also show that in the presence of rule CMP, UPO, CON, PO, it is not possible to detect a violation of $WA-S_{intra}$. The proof is similar to the above two proofs with various new case analysis, all of which can be shown to lead to cycles of smaller length. Rather than providing a formal proof of this result, we just sketch the outline of the proof below. The new cases involved are $e_1 = RW \vee e_1 = WO$. In either scenario, we could demonstrate a

[13] The property that if an execution obeys $A(CMP, UPO, CON)$ then it also obeys $A(CMP, UPO, CON, WA\text{–}S_{intra})$ is not sufficient to guarantee that $WA\text{–}S_{intra}$ violation is not detected in the presence of other rules. In other words, if $A(R_1, R_2, R_3) \Rightarrow A(R_1, R_2, R_3, R)$ then it is not necessary that $A(R_1, R_2, R_3, R_4) \Rightarrow A(R_1, R_2, R_3, R_4, R)$. This is so because it is possible that an execution obeys $A(R_1, R_2, R_3, R_4)$ and $A(R_1, R_2, R_3, R)$ but it does not obey $A(R_1, R_2, R_3, R_4, R)$. This can happen when both an arc of rule R_4 and an arc of R are necessary to create a cycle in the member of the graph set.

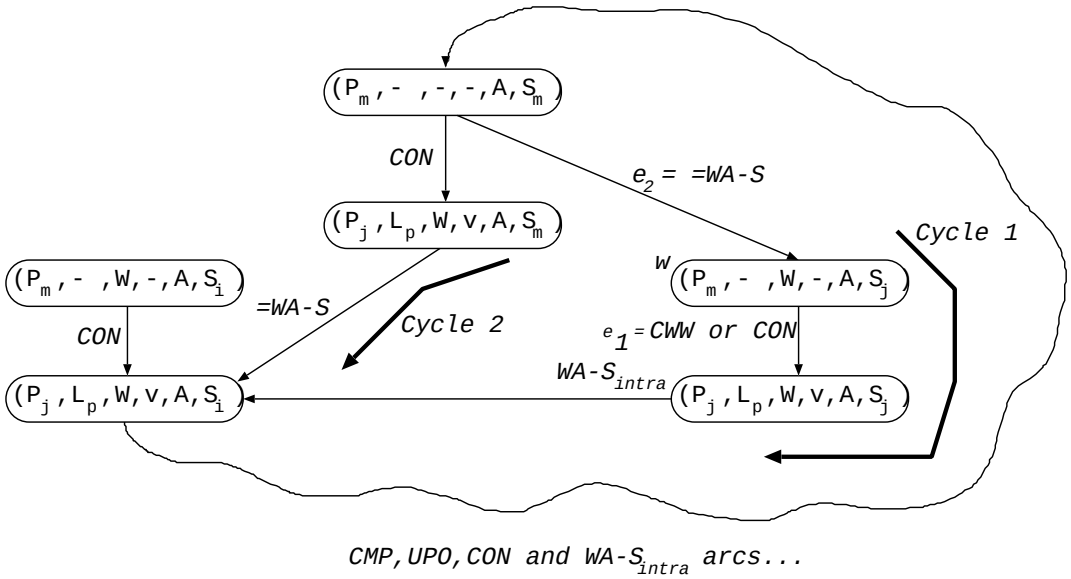

Figure 4.47. Case $e_1 = CWW\ or\ \ CON$ and $e_2 = =_{WA–S}$: we have a cycle of smaller length in this case too.

cycle of smaller length by a construction similar to the case of $e_1 = SRW$. Similar results could be obtained for either $RW, WW, RR\ or\ \ WR$ subrules of PO.

All practical memory systems obey $A(CMP, UPO)$ and most memory systems obey CON as well. Hence, for most practical memory systems, it is not possible to detect a violation of $WA-S_{intra}$ as demonstrated by the discussion above in this section.

4.6.7.6 $Test_{\text{CON}}$: Test for CON

Let us consider how we can test for the rule of CON. Rule of CON says that all writes *for the same operand* appear to all processes in the same order. Rule of WA says that all writes appear to all processes in the same order (regardless of operand). Rule of CON is essentially a special case of rule of WA and so we may be able to take the tests for WA and suitably modify them for one address to check for CON. Consider the two process test for WA $Test_{\text{WA3}}$ modified for one operand. This test is shown in Figure 4.48. [14] Operands A and B are substituted with operand A and two reads from the same operand are converted to one single read. Also, P_1 writes odd values into A and P_2 writes even values into A so that orders from different processes could be distinguished. This test checks for $A(CMP, UPO, CON)$ with the condition checked being :

[14] ARCHTEST also provides a test for $A(CMP, UPO, CON)$ which involves four processes and is similar to this test and it involves a complex condition due to four processes.

Initially, $A = 0$

$L_{11} : A := 1;$	$L_{11} : A := 2;$
$L_{12} : U[1] := A;$	$L_{12} : X[1] := A;$
$L_{21} : A := 3;$	$L_{21} : A := 4;$
$L_{22} : U[2] := A;$	$L_{22} : X[2] := A;$
$\ldots$	$\ldots$
$L_{k1} : A := 2k - 1;$	$L_{k1} : A := 2k - 2;$
$L_{k2} : U[k] := A;$	$L_{k2} : X[k] := A;$

Figure 4.48. *Test* $_{\text{CON}}$: test for $A(CMP, UPO, CON)$.

CONDITION 12 (CONSISTENCY) $\forall i, j : 1 \leq i, j \leq k : U[i] \geq 2j \vee U[i]$ *is odd* $\vee X[j] \geq 2i \vee X[j]$ *is even.*

Now, we show that if an execution violates condition 12 then $A(CMP, UPO, CON)$ is violated. Intuitively, when a condition is violated, P_1 and P_2 observe two values of A which gets stored in $U[i]$ and $X[j]$, differently.

Formally, consider a violation of condition 12.

$$\begin{array}{ll} \exists i, j : 1 \leq i, j \leq k : & U[i] < 2j \wedge U[i] \textit{ is even } \wedge \\ & X[j] < 2i \wedge X[j] \textit{ is odd} \\ \exists i, j, \alpha, \beta : 1 \leq i, j \leq k : & U[i] = \beta \wedge X[j] = \beta \wedge \\ & \beta < 2j \wedge \beta \textit{ is even } \wedge \\ & \alpha < 2i \wedge \alpha \textit{ is odd} \end{array}$$

The statements involved corresponding to the values in the condition violation above are :

P_1	P_2
$\ldots$	$\ldots$
$L_{\alpha 1} : A := \alpha;$	$\ldots$
$\ldots$	$L_{\beta 1} : A := \beta;$
$\ldots$	$\ldots$
$\ldots$	$\ldots$
$L_{i1} : A := 2i;$	$\ldots$
$L_{i2} : U[i] := A;$	$\ldots$
$\ldots$	$\ldots$
$\ldots$	$\ldots$
$\ldots$	$L_{j1} : A := 2j;$
$\ldots$	$L_{j2} : X[j] := A;$

Note that we could enforce that α and β writes are in P_1 and P_2 processes respectively because of the odd and even conditions. Now, P_1 observes values α and β in order α, β however, P_2 observes values β and α in order β, α. Hence, CON is violated. More

formally, every member of the graph set for above execution contains the cycle shown in Figure 4.49(a) or the cycle shown in Figure 4.49(b). These two cases are based upon whether the sequence of values seen by P_1 and P_2 is α, β (case (a)) or β, α (case (b)). Note that though UPO is not directly involved in the cycle, it is necessary to induce the CWW arc.

4.6.7.7 Test Automata for $Test_{\text{CON}}$

Test automata for $Test_{\text{CON}}$ is shown in Figure 4.50. We employ data abstraction using two bits to abstract even and odd values of A. [15] Test automata is derived from the above test in a manner very similar to other test automata. The memory rule safety property being checked is :

$$P_1 \textit{ and } P_2 \textit{ in their final states} \Rightarrow u = 10 \vee u \textit{ is odd } \vee x = 11 \vee x \textit{ is even}$$

We show that each violation of condition 12 of $Test_{\text{CON}}$ would be captured by the test automata memory rule safety property violation.

THEOREM 4.11 If condition 12 is violated during an execution of $Test_{\text{CON}}$, then there exists a test automata run in which the memory rule safety property is violated.

Proof: Consider a violation of condition 12 in an execution of $Test_{\text{CON}}$.

[15] If we generalize this test to more than two processes, then we need as many bits as there are processes for data abstraction.

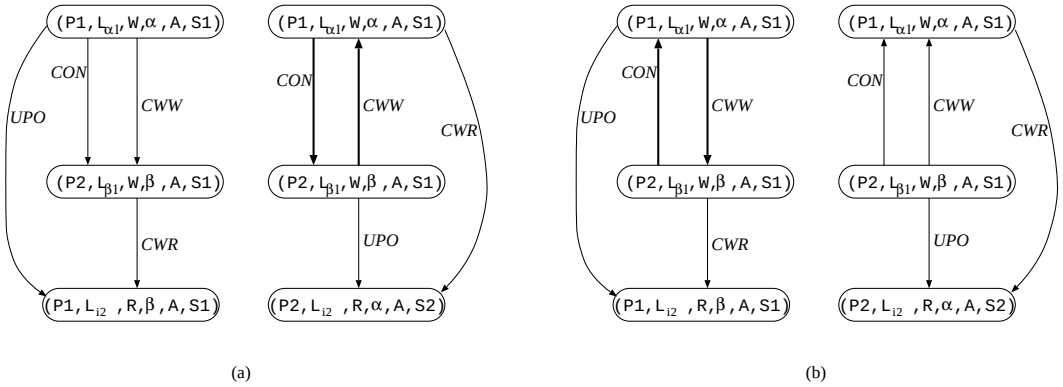

Figure 4.49. Cycle corresponding to the violation of $A(CMP, UPO, CON)$ in $Test_{\text{CON}}$.

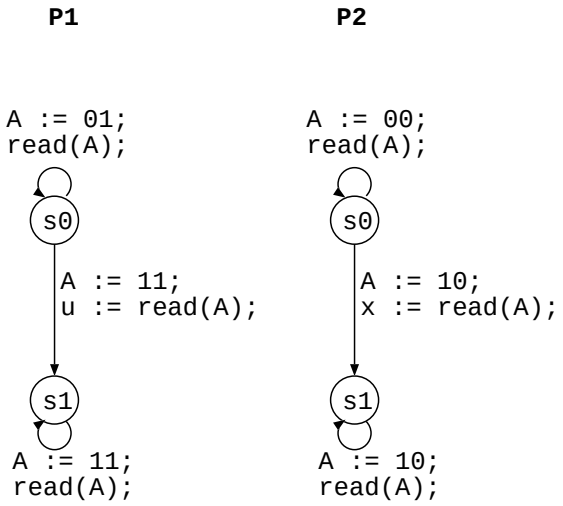

Figure 4.50. Test automata for $Test_{\text{CON}}$.

$$\begin{array}{ll} \exists i,j : 1 \leq i,j \leq k : & U[i] < j \wedge U[i] \textit{ is even } \wedge \\ & X[j] < i \wedge X[j] \textit{ is odd} \\ \exists i,j,\alpha,\beta : 1 \leq i,j \leq k : & U[i] = \beta \wedge X[j] = \beta \wedge \\ & \beta < j \wedge \beta \textit{ is even } \wedge \\ & \alpha < i \wedge \alpha \textit{ is odd} \end{array}$$

Now, we construct a nondeterministic run of test automata correlating the above condition violation that violates the memory rule safety property. Consider a run of test automata in which P_1 loops α times on state s_0 before switching to state s_1. P_2 loops β times on state s_0 before switching to state s_1. Invoking data independence, we would have $u = 00 \wedge x = 01$. Hence, memory rule safety property would be violated.

□

4.6.8 Test for MB

In this section, we would consider how to test for rule of MB. The rule of MB essentially functions as various subrules of PO except that it only provides ordering where explicitly indicated. Hence, we could extend the various test programs described above for subrules of PO to check for corresponding subrules of PO. We would illustrate this by considering the $MB-RR$ subrule of MB in detail. Similar treatment could be applied to the other subrules of MB.

In the various tests for subrules of PO described earlier, when the condition is violated, we employ the corresponding subrule of PO to the memory operations pertained to the variables involved in the condition violation to show that the subrule of PO is violated.

We can adopt the same approach for subrules of MB: instead of relying on subrule of PO to provide the ordering, we would like to explicitly provide the ordering by introducing a subrule of MB. However, since we do not know the exact memory operations that causes the condition violation, we cannot predetermine where to insert subrule of MB. We can insert subrule of MB *everywhere* necessary to ensure that when a violation would occur, it would always involve two memory operations ordered separated by a subrule of MB.

4.6.8.1 Test for $MB-RR$ $Test_{\text{MB-RR}}$

We illustrate the above methodology for subrule of $MB-RR$. A pure test for $A(CMP, MB-RR)$ is proposed in Figure 4.51. If the memory system correctly realizes $A(CMP, MB-RR)$, then Condition 13 is satisfied.

CONDITION 13 (EQUALITY) The sequence of X values satisfy the following property:
$\forall p, q, r : 1 \leq p < q < r \leq k : X[p] = X[r] \Rightarrow X[p] = X[q] = X[r]$

If the condition 13 is violated then we could show a cycle in each member of graph set of the execution under $A(CMP, MB-RR)$. Such a cycle would be similar to that for the case of $A(CMP, RO)$ except that the ordering between two read events would be enforced by $MB-RR$. Such cycles are shown in Figure 4.52.

4.6.8.2 Test Automata for $Test_{\text{MB-RR}}$

We can obtain a test automata for $Test_{\text{MB-RR}}$ using an abstraction similar to that for $Test_{\text{RO}}$. However, because of the nondeterminism nature of the test automata, we apriorily know that what read operations contribute the values used in evaluation of memory rule safety property. Hence, rather than schedule MB-RR operation between

Initially, $A = 0$

P_1	P_2
$L_1 : A := 1;$	$L_1 : X[1] := A;$
	$L_2 : MBRR;$
$L_2 : A := 2;$	$L_3 : X[2] := A;$
	$L_4 : MBRR;$
$L_3 : A := 3;$	$L_5 : X[3] := A;$
	$L_6 : MBRR;$
$\ldots$	$\ldots$
	$L_{2k-2} : MBRR;$
$L_k : A := k$	$L_{2k-1} : X[k] := A;$

Figure 4.51. $Test_{\text{MB-RR}}$: a new test for $A(CMP, MB-RR)$.

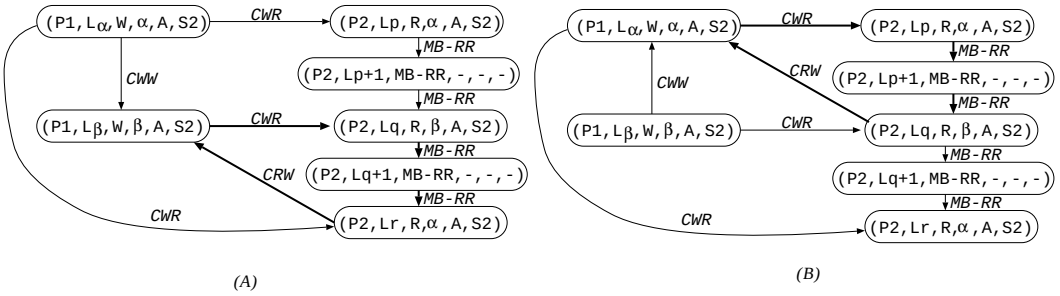

Figure 4.52. A cycle corresponding to violation of $Test_{\text{MB-RR}}$ condition.

every two instruction as we did for in the test for $Test_{\text{MB-RR}}$, we can schedule it only where necessarily. In this case, we just need to schedule MB-RR after each of the first two read operations. Such a test automata for $Test_{\text{MB-RR}}$ is shown in Figure 4.53. The memory rule safety property is same as that for $Test_{\text{RO}}$.

4.6.8.3 How Many MB-RRs Are Necessary ?

Clearly, a violation of memory rule safety property for the test automata shown in Figure 4.53 clearly indicates that the memory system does not obey $A(CMP, MB-RR)$. However, an important issue to consider is whether including more MB-RRs in test automata can catch some violation which otherwise would not have been caught ? We consider this issue here and identify assumptions (satisfied by most common memory subsystem designs) under which an argument could be presented that including more MB-RRs cannot catch any more violations.

Instead of the test automata for $Test_{\text{MB-RR}}$ shown in Figure 4.53, consider the test automata shown in Figure 4.54. Clearly, the test automata in Figure 4.53 is a special case of the test automata in Figure 4.54. Hence, any violation caught by the test automata in Figure 4.53 is also caught by the test automata in Figure 4.54. Also, it is possible that there may be some violation that would be captured only by the test automata in Figure 4.54. For example, consider a memory system that counts the number of MB-RRs and decides to change its behavior based upon that. However, such memory systems are not typical. Most memory systems do not count the number of MB-RRs, they process it only to guarantee an ordering between the respective read operations. If we assume that the only effect of MB-RRs is to ensure an ordering between the relevant read operations and in the absence of such MB-RRs memory system can execute two read operations in the program order then we can show that the two test automata in Figure 4.53 and 4.54

are equivalent.

THEOREM 4.12 Consider a memory system that obeys the rule of $MB-RR$. Suppose the following assumptions are valid for the memory system.

- *No side-effect :* The only effect of an MB-RR is to ensure an ordering between two relevant read operations.
- *Natural order :* If two read operations are not ordered by MB-RR then memory system can execute them in the program order.

For such a memory system, any violation of memory rule safety property for the test automata in Figure 4.54 would be caught by the test automata in Figure 4.53.

Proof : Consider a memory rule safety property violation for the test automata in Figure 4.54. Consider a similar nondeterministic run of the test automata in Figure 4.53 in which same number of self-loops executing *read(A)* are taken but only two *MB RR* are executed. Because of *No side-effect* assumption, the extra *MB RR*s executed in the test automata in Figure 4.54 does not have any effect but to order some of the read operations in the program order. Because of *Natural order* assumption, an execution of test automata in Figure 4.53 that executes all the unordered read operations in the same order as the other automata does. Hence, if memory rule safety property is violation for the test automata in Figure 4.54, such a violation would also manifest itself in one of the nondeterministic run of the test automata in Figure 4.53. Hence, proved. □

4.7 Complete PO-relaxation Models

4.7.1 How to *Specify* and *Verify* Complete PO-relaxation Weaker Memory Models ?

In this section, we consider how complete PO-relaxation weaker memory models could be specified and verified with the rules that we earlier defined and the test model-checking automata for various subrules that we have developed. We briefly describe the shared memory model of DEC-Alpha based multiprocessors and examine how they could be specified in the framework of ARCHTEST and verified using test automata.[16]

[16]We shall restrict ourselves with the description of memory instructions and memory barrier instructions. Issues related with instruction fetch ordering, translation buffer updates and conditional memory access instructions are not considered.

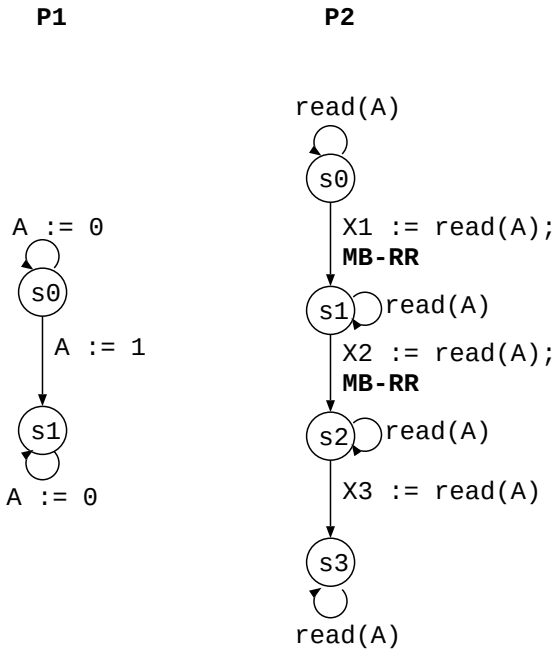

Figure 4.53. Test automata for $Test_{\text{MB-RR}}$.

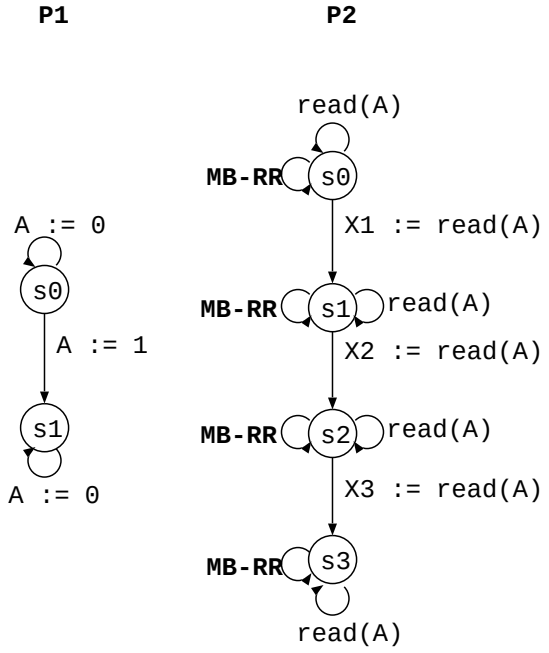

Figure 4.54. Another test automata for $Test_{\text{MB-RR}}$.

4.7.2 Alpha Shared Memory Model

The description presented in this section follows the specification in [52]. Complete detail and a formal definition are available in Alpha Architecture Manual [52]. Alpha shared memory memory model has four types of memory operation: read, write, membar and write membar. Alpha is a highly optimized architecture which allows a great degree of freedom for memory operations to be reordered. Each processor maintains the same order as the program order between memory operations with the same operand. Each process can reorder memory operations between different operands to enhance performance. Whenever the order between memory operations must be preserved, a membar instruction or a write-membar instruction must be used. It is guaranteed that all memory operations of a processor occurring before membar in program order are completed before the membar operation. Similarly, all memory operations of a process occurring after membar in program order are viewed by all processors after membar. Also, Alpha shared memory model permits a read to observe the write of the to the same operand (of the same processor) before it becomes visible to other processors. In this context, the atomicity behavior of alpha shared memory model is similar to TSO.

4.7.3 ARCHTEST Specification of Alpha Shared Memory Model

In the context of ARCHTEST framework, rule of PO is completely relaxed in Alpha shared memory model. Alpha shared memory model obeys rule of CMP and UPO. It also obeys rule of ROO as it requires that reads to the same operand of the same processor should occur in the program order. Beyond this, there are no ordering restrictions between events other than enforced by membar instructions. The MB instruction imposes ordering on both read and write operations: it can be viewed as imposing all $MB{-}RR$, $MB{-}RW$, $MB-WR$ and $MB-WW$ constraints. This membar behavior can be captured by rule of MB. The WMB instruction imposes ordering on only write operations: it can be viewed as imposing $MB-WW$ constraints. Hence, the write-membar restriction can be captured by rule of $MB-WW$. Since Alpha shared memory model allows a read to observe a write to the same operand before the write becomes visible to other processors, we can use the rule of $WA{-}S$ to capture its atomicity behavior. Hence, in the ARCHTEST framework Alpha shared memory model can be specified as $A(CMP, UPO, ROO, WA{-}S, MB, MB-WW)$.

Alpha architecture manual [52] describes a number of executions called *Litmus tests*

to illustrate which shared memory behavior is allowed and not allowed in alpha shared memory model. We shall consider some of the Litmus tests and show how specification within ARCHTEST framework describes behavior of each Litmus test. For each of the test, we can construct the graph set under $A(CMP, UPO, ROO, WA-S, MB, MB-WW)$ and show that it contains a cycle if the test behavior is termed impossible.

4.7.3.1 Litmus Test 1

This test describes that the following execution is not possible under Alpha shared memory model.

$$\begin{array}{ll} \multicolumn{2}{c}{\textit{Initially,}\quad A = X = Y = 0} \\ P_1 & P_2 \\ A := 1 & X := A \\ & Y := A \\ \multicolumn{2}{c}{\textit{Finally,}\quad X = 1, Y = 0} \end{array}$$

This execution shows that the two read operations of P_2 must be ordered with respect to each other. This is imposed by rule of ROO in ARCHTEST framework.

4.7.3.2 Litmus Test 2

This test describes that the following execution is not possible under Alpha shared memory model.

$$\begin{array}{ll} \multicolumn{2}{c}{\textit{Initially,}\quad A = X = Y = 0} \\ P_1 & P_2 \\ A := 1 & A := 2 \\ & X := A \\ & Y := A \\ \multicolumn{2}{c}{\textit{Finally,}\quad X = 1, Y = 2} \end{array}$$

This execution is not possible since CMP, UPO and ROO enforce that the second read of P_2 must also have value 1 and not 2.

4.7.3.3 Litmus Test 3

This test describes that the following execution is not possible under Alpha shared memory model.

$$\begin{array}{lll} \multicolumn{3}{c}{\textit{Initially,}\quad A = X = Y = Z = 0} \\ P_1 & P_2 & P_3 \\ A := 1 & A := 2 & Y := A \\ X := A & & Z := A \\ \multicolumn{3}{c}{\textit{Finally,}\quad X = 2, Y = 2, Z = 1} \end{array}$$

This execution shows that since P_1 sees write to A of P_2 with value 2 after its own write with value 1, P_3 must also see the two writes in the same sequence. This restriction is enforced by CON component of $WA-S$ in ARCHTEST framework.

4.7.3.4 Litmus Test 4

This test describes that the following execution is possible under Alpha shared memory model.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textit{Initially,}\quad A = B = X = Y = 0} \\
P_1 & P_2 \\
A := 1 & X := B \\
B := 1 & Y := A \\
\multicolumn{2}{l}{\textit{Finally,}\quad X = 1, Y = 0}
\end{array}
$$

This execution shows that the writes of P_1 and reads of P_2 to different operands could be reordered with respect to each other. This is modeled by absence of any component of PO with different operands in the ARCHTEST framework.

4.7.3.5 Litmus Test 5

This test describes that the following execution is possible under Alpha shared memory model.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textit{Initially,}\quad A = B = X = Y = 0} \\
P_1 & P_2 \\
A := 1 & X := B \\
 & MB \\
B := 1 & Y := A \\
\multicolumn{2}{l}{\textit{Finally,}\quad X = 1, Y = 0}
\end{array}
$$

This execution is the same as the one in Litmus test 4 except that the two reads in P_2 are explicitly ordered with respect to each other by inserting a memory barrier instruction between them. However, this still allows writes of P_1 to be reordered with respect to each other.

4.7.3.6 Litmus Test 6

This test describes that the following execution is possible under Alpha shared memory model.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textit{Initially,}\ \ A = B = X = Y = 0} \\
P_1 & P_2 \\
A := 1 & X := B \\
MB & \\
B := 1 & Y := A \\
\multicolumn{2}{l}{\textit{Finally,}\ \ X = 1, Y = 0}
\end{array}
$$

This execution is the same as the one in Litmus test 4 except that the two writes in P_1 are explicitly ordered with respect to each other by inserting a memory barrier instruction between them. However, this still allows reads of P_2 to be reordered with respect to each other.

4.7.3.7 Litmus Test 7

This test describes that the following execution is not possible under Alpha shared memory model.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textit{Initially,}\ \ A = B = X = Y = 0} \\
P_1 & P_2 \\
A := 1 & X := B \\
MB & MB \\
B := 1 & Y := A \\
\multicolumn{2}{l}{\textit{Finally,}\ \ X = 1, Y = 0}
\end{array}
$$

Inserting memory barrier instructions in both P_1 and P_2 ensures that both the reads and writes are executed in the program order. This enforces P_2 to see updates of A and B in the program order and also to execute reads in the program order. Note that this execution is still impossible if the memory barrier in P_1 is replaced with a write memory barrier. However, replacing the memory barrier in P_2 with write makes this execution possible as the reads of B and A are not forced to execute in program order by write memory barrier. The rules of MB And $MB - WW$ enforce all the necessary constraints to enforce the execution behavior in Litmus test 4,5,6 and 7.

4.7.3.8 Litmus Test 8

This test describes that the following execution is not possible under Alpha shared memory model.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textit{Initially,}\ \ A = B = X = Y = 0} \\
P_1 & P_2 \\
A := 1 & B := 1 \\
MB & MB \\
X := B & Y := A \\
\multicolumn{2}{l}{\textit{Finally,}\ \ X = 0, Y = 0}
\end{array}
$$

Again, rule of MB ensures that read is executed only after write in each processor. Hence, both processors reads getting value 0 is not possible.

4.7.3.9 Litmus Test 9

This test describes that the following execution is not possible under Alpha shared memory model.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textit{Initially,}\ \ A = X = Y = U = V = 0} \\
P_1 & P_2 \\
A := 1 & A := 2 \\
X := A & U := A \\
Y := A & V := A \\
\multicolumn{2}{l}{\textit{Finally,}\ \ X = 1, Y = 2, U = 2, V = 1}
\end{array}
$$

In this execution P_1 and P_2 observes the writes to A in different order that is disallowed by the rule of CON.

Various aspects of Alpha shared memory model specification $A(CMP, UPO, ROO, WA\text{–}S, MB, MB\text{–}WW)$ can be checked by using the test automata developed for these subrules. The test automata strategy described for MB could be used to check the behavior of MB and $MB - WW$. Also, the test automata developed for various subrules of PO could be used as applicable. It should be noted that pure tests and test automata for various subrules of PO and other subrules are much likelier to be applicable for verification of various weaker memory models as we can select the particular aspect of the PO or atomicity behavior that is applicable for each weaker memory model.

4.8 Experimental Results

We have developed an operational model of TSO in VIS Verilog on which we have tested the test automata developed in this chapter. We have specified some of the test automata developed in this section in this operational model of TSO. This section briefly describes the operational model of TSO in VIS and then discuss the results of our experiments.

4.8.1 The Operational Model of TSO in VIS

The operational model of TSO in VIS is shown in Figure 4.55. It is a direct adaption of the operational model of TSO shown in Figure 4.7. Each processor contains a write buffer of length 16 which stores all the write operations issued by the processor. Each processor executes write operation to an address by simply en-queuing the write in the queue if the queue is not full. Each processor executes read operation to an address by first looking it up in the queue or by accessing the memory if the address is not found in the queue. Each processor's read is blocking, i.e., it stops executing any further memory operations until the read is complete. Each processor access the memory by a bus which the memory arbitrarily assigns to either process. Memory executes each write operation by updating its copy of the address immediately. Memory executes each read operation by immediately supplying data corresponding to the requested address.

4.8.2 Brief Description of VIS Verilog TSO Operational Model

Our simples VIS Verilog model of TSO operational model includes a generic module for each processor and a module for memory. Memory module simply performs reads and writes one at a time from each processors. Each processor is arbitrarily selected to perform reads or writes to memory. Each processor includes a common part that checks

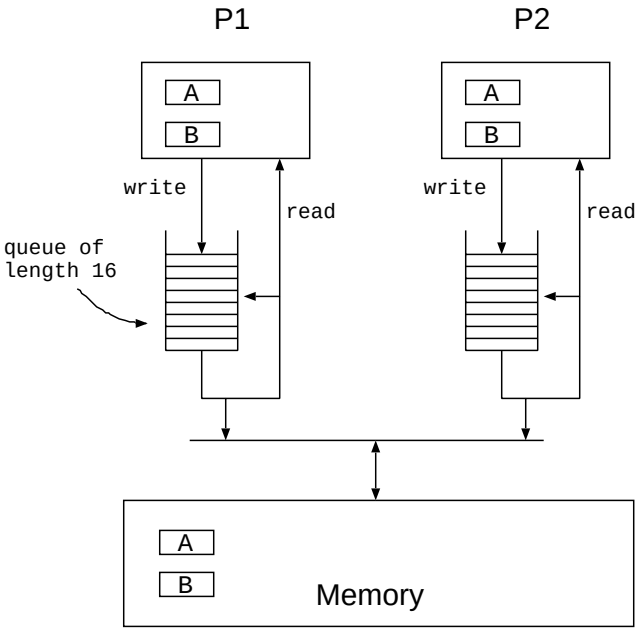

Figure 4.55. Operational model of TSO in VIS.

to see if there are any pending reads or writes and performs these reads and writes to memory as necessary and allowed. Each processor includes a part which implements the corresponding reads and write requests as per the test model-checking automata and also updates the variables used in memory rule safety property.

4.8.3 Results

We have specified test automata mentioned in Table 4.1 in the operational model of TSO in VIS. The results of the VIS model-checking run for model-checking the memory rule safety property for each test automata is shown in Table 4.1. As we can see from the table, the memory rule safety properties for $A(CMP, PO)$ and $A(CMP, WR)$ are violated as it is expected. The memory rule safety properties for $A(CMP, RO, WO)$, $A(CMP, RW)$ and $A(CMP, RO)$ are satisfied.

We can also see that the number of reachable states is significantly smaller for the $A(CMP, RO, WO)$ and $A(CMP, ROO)$ runs compared to $A(CMP, PO)$, $A(CMP, RO)$, and $A(CMP, RW)$. This shows that for the test model-checking automata for these tests are much more involved than the others as they involve two reads for different operands. Similarly, we see that the BDD nodes are significantly larger for the test model-checking automata involving more than one operand.

4.9 Summary

We provide a brief tabular summary of various different rules. We also provide a summary of various formal memory models and their specifications in terms of these rules.

Table 4.2 shows various architecture rules and their transition templates.

Table 4.3 shows the architecture rules in our discussion and the subrules each of them consists of.

Table 4.1. Verification results on an operational model of TSO using VIS

test automata	#states	#bdd nodes	runtime (mn:sec)	status
CMP, RO, WO	3819	4872	$<$ 1s	pass
CMP, PO	6.50875e+06	50051	2:38	fail
CMP, WR	6.50875e+06	50051	1:25	fail
CMP,RW	6.50875e+06	50051	3:02	pass
CMP, RO	10187	2463	0:37	pass

Table 4.2. Architecture rules and their transition templates

Architecture rule	Page	Transition template
SRW	22	$(P,L,R,V,O,S) <_{SRW} (=,=,W,-,-,-)$
CRW	22	$(P,L,R,V,O,S) <_{CRW} (-,-,W,-,=,=)$
CWR	22	$(P,L,W,V,O,S) <_{CWR} (-,-,R,=,=,=)$
CWW	22	$(P,L,W,V,O,S) <_{CWW} (-,-,W,-,=,=)$
URW	18	$(P,L,R,V,O,S) <_{URW} (=,-,W,-,=,=)$
UWR	18	$(P,L,W,V,O,S) <_{UWR} (=,-,R,-,=,=)$
UWW	18	$(P,L,W,V,O,S) <_{UWW} (=,-,W,-,=,=)$
RW	17	$(P,L,R,V,O,S) <_{RW} (=,-,W,-,-,-)$
WR	17	$(P,L,W,V,O,S) <_{WR} (=,-,R,-,-,-)$
RO	16	$(P,L,R,V,O,S) <_{RO} (=,-,R,-,-,=)$
WO	17	$(P,L,W,V,O,S) <_{WO} (=,-,W,-,-,-)$
$MB{-}RR$	55	$(P,L,R,V,O,S) <_{MB\text{–}RR} (=,-,MB-RR,-,-,-)$ $(P,L,MB-RR,-,-,-) <_{MB-RR} (=,-,R,-,-,-)$
$MB{-}RW$	55	$(P,L,R,V,O,S) <_{MB\text{–}RW} (=,-,MB-RW,-,-,-)$ $(P,L,MB-RW,-,-,-) <_{MB-RW} (=,-,W,-,-,-)$
$MB{-}WR$	56	$(P,L,W,V,O,S) <_{MB\text{–}WR} (=,-,MB-WR,-,-,-)$ $(P,L,MB-WR,-,-,-) <_{MB-WR} (=,-,R,-,-,-)$
$MB{-}WW$	56	$(P,L,W,V,O,S) <_{MB\text{–}WW} (=,-,MB-WW,-,-,-)$ $(P,L,MB-WW,-,-,-) <_{MB-WW} (=,-,W,-,-,-)$
$=_{WA\text{–}S}$	57	$(P,L,W,V,O,\neq P) =_{WA\text{–}S} (=,=,W,=,=,\neq P)$
$WA{-}S_{intra}$	58	$(P,L,W,V,O,=P) <_{WA\text{–}S_{intra}} (=,=,W,=,=,\neq P)$
CON	20	$(P,L,W,V,O,S) <_{CON} (-,-,W,-,=,=)$
WA	19	$(P,L,A,V,O,S) <_{WA} (-,-,-,-,-,-)$

Table 4.3. Architecture rules

Architecture rule	Page	Subrules
CMP	21	SRW, CRW, CWW, CWR
PO	17	WR, WO, RO, RW
$WA-S$	57	$=_{WA\text{–}S}$, $WA{-}S_{intra}$, CON
MB	53	$MB{-}WR$, $MB{-}WO$, $MB{-}RO$, $MB{-}RW$

Table 4.4 shows the memory models in our discussion and their specification in ARCHTEST's framework.

Table 4.4. Memory models specification in ARCHTEST's framework

Memory Model	Page	ARCHTEST specification
Sequential consistency	23	$A(CMP, PO, WA)$
IBM 370	61	$A(CMP, UPO, RO, WO,$ $RW, WA, MB-WR)$
Total Sorted Order (TSO)	56	$A(CMP, UPO, RO, WO,$ $RW, WA-S, MB-WR)$
Partial Sorted Order (PSO)	61	$A(CMP, UPO, RO,$ $RW, WA-S, MB-WR, MB-WW)$
Alpha Shared Memory Model	112	$A(CMP, UPO, ROO,$ $WA-S, MB, MB-WW)$

CHAPTER 5

CONCLUSION AND FUTURE WORK

We presented a new approach to verify multiprocessors for formal memory models that combines two existing powerful techniques: model-checking, and the testing method of ARCHTEST. From our results, we conclude that test model-checking can be of great value in detecting bugs during the early stages of the design cycle of modern microprocessors whose memory subsystems are complex. Our results of our URM of the HP PA/Runway bus attest to this. In effect, test automata offer good specifications to check for formal memory models such as sequential consistency.

We considered how test model-checking technique could be applied to verify weaker memory models. We proposed new architectural rules in the existing framework of ARCHTEST that enabled us to specify weaker memory models in ARCHTEST's framework. We proposed new tests similar to ARCHTEST tests for these weaker memory models. We developed test automata for such tests and presented formal proofs of the soundness of the abstraction used in these test automata. We have applied these test model-checking automata on operational model of TSO to demonstrate its applicability.

A Test model-checker tool for multiprocessor memory system verification can be created which provides a suite of various test model-checking automata developed. Such a told could provide a generic interface to memory systems which can be driven by a set of test model-checking automata. One can select a set of applicable test model-checking automata for a given memory model and run verification on memory systems using these automata. Test model-checking automata for various pure tests facilitate verification for various different weaker memory models which may differ from each other in subtle manners.

Some of the issues that could be explored in future work are :

- Most memory models explicitly talk about conformance of data space with address space. Many memory models explicitly address the issue of when the data written

in global operand space becomes visible during further instruction fetch. To ensure conformance between the data space and the address space, the SPARC architecture provides a FLUSH instruction and the Alpha memory model provides an explicit instruction memory barrier instruction (IMB). How to verify that a memory system implements such memory operations properly is an open question. This issue is relevant for self-modifying programs that share code through shared memory between multiple processors.

- Some memory models provide special hardware instructions for efficient implementation of semaphore and other synchronization operations. Examples of such instructions are load-store, compare-and-swap, etc. It could be interesting to explore the issues related to verifying the correct implementation of such memory operations.

- It would be interesting to apply all of the test model-checking automata developed for weaker memory models to a memory system similar to URM (suitably modified to provide a weaker memory model) and examine the experimental results.

- Since the test model-checking automata conditions are necessary but not sufficient conditions, it could be interesting to come up with complete test model-checking automata. Many new interesting issues may arise while extending the test model-checking automata for completeness for weaker memory models.

REFERENCES

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

[2] Adve, S. V., and Hill, M. D. Weak ordering—a new definition. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)* (May 1990), pp. 2–14.

[3] Afek, Y., Brown, G., and Merritt, M. Lazy caching. *ACM Transactions on Programming Languages and Systems 15*, 1 (Jan. 1993), 182–205.

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

[5] Ahamad, M., Neiger, G., Kohli, P., Burns, J. E., and Hutto, P. W. Casual memory: Definitions, implementation and programming. *Distributed Computing 9* (1995), 37–49.

[6] 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* (July 1996), pp. 219–228.

[7] Bershad, B. N., Zekauskas, M. J., and Sawdon, W. A. The midway distributed shared memory system. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)* (Feb. 1993), pp. 528–537.

[8] 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.

[9] Camilleri, A. A hybrid approach to verifying liveness in a symmetric multiprocessor. In *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97, Murray Hill, NJ* (Aug. 1997), pp. 49–67. Springer-Verlag LNCS 1275.

[10] Case, R. P., and Padegs, A. Architecture of the IBM System 370. *Communications of the ACM 21* (Jan. 1978), 73–96.

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

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

[13] Collier, W. W. Multiprocessor diagnostics. http://www.infomall.org/diagnostics/research.html.

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

[15] 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.

[16] 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.

[17] Friedman, R. *Consistency Conditions for Distributed Shared Memories.* PhD thesis, Computer Science Department, Technion–Israel Institute of Technology, June 1994.

[18] Frigo, M. The weakest reasonable memory. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, 1998.

[19] Frigo, M., and Luncangco, V. Computation-centric memory models. In *Proc. of the 10th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'98)* (June 1998).

[20] Gao, G. R., and Sarkar, V. Location consistency: Stepping beyond the barriers of memory coherence and serializability. Tech. Rep. ACAPS Technical Memo 78, School of Computer Science, McGill University, Dec. 1993.

[21] 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#Lazy.

[22] Gharachorloo, K. *Memory Consistency Models for Shared-Memory Multiprocessors.* PhD thesis, Stanford University, 1995.

[23] Gharachorloo, K., Adve, S. V., Gupta, A., Hennessy, J. L., and Hill, M. D. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing 15*, 4 (Aug. 1992), 399–407.

[24] Gharachorloo, K., Gupta, A., and Hennessy, J. L. Revision to "memory consistency and event ordering in scalable shared-memory multiprocessors". Tech. Rep. CSL-TR-93-568, Computer Systems Laboratory, Stanford University, Apr. 1993.

[25] Gharachorloo, K., Lenoski, D. E., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J. L. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)* (May 1990), pp. 15–26.

[26] 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.

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

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

[29] Goodman, J. R., and Woest, P. J. The wisconsin multicube: A new large-scale cache-coherent multiprocessor. In *Proc. of the 15th Annual Int'l Symp. on Computer Architecture (ISCA'88)* (May 1988), pp. 422–431.

[30] 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.

[31] Graf, S. Verification of a distributed cache memory by using abstractions. *Lecture Notes in Computer Science 818* (1994), 207–??

[32] Higham, L., Kawash, J., and Verwaal, N. Defining and comparing memory consistency models. In *Proc. of the 10th Int'l Conf. on Parallel and Distributed Computing Systems (PDCS-97)* (Oct. 1997).

[33] Hojati, R., and Brayton, R. Automatic datapath abstraction of hardware systems. In *Conference on Computer-Aided Verification* (1995).

[34] 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.

[35] Hu, W., Shi, W., and Tang, Z. A framework of memory consistency models. *Journal of Computer Science and Technology 13*, 2 (Mar. 1998), 110–124.

[36] Iftode, L., Singh, J. P., and Li, K. Scope consistency: A bridge between release consistency and entry consistency. In *Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96)* (June 1996), pp. 277–287.

[37] Kane, G. *PA-RISC 2.0 Architecture*. Prentice Hall, 1996. ISBN 0-13-182734-0.

[38] Keleher, P., Cox, A. L., and Zwaenepoel, W. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)* (May 1992), pp. 13–21.

[39] Kohli, P., Neiger, G., and Ahamad, M. A characterization of scalable shared memories. In *Proc. of the 1993 Int'l Conf. on Parallel Processing (ICPP'93)* (Aug. 1993).

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

[41] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. C-28*, 9 (Sept. 1979), 690–691.

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

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

[44] 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.

[45] 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.

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

[47] Mokkedem, A. Verification of three memory systems using test model-checking. http://www.cs.utah.edu/~mokkedem/vis/vis.html.

[48] Mosberger, D. Memory consistency models. *ACM Operating Systems Review 27*, 1 (Jan. 1993), 18–26.

[49] Park, S., and Dill, D. L. An executable specification, analyzer and verifier for RMO (relaxed memory order). In *Proc. 7th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'95* (Santa Barbara, California, July 1995), pp. 34–41. Stanford.

[50] Park, S., and Dill, D. L. Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In *SPAA* (Padua, Italy, June 24–26, 1996), pp. 288–296.

[51] Raynal, M., and Schiper, A. A suite of formal definitions for consistency criteria in shared memories. Tech. Rep. PI-968, IRISA, France, May 1995.

[52] Sites, R. L. *Alpha Architecture Reference Manual.* Digital Press and Prentice-Hall, 1992.

[53] Vis-1.2 release. http://www-cad.eecs.berkeley.edu/Respep/Research/vis/.

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