Verification Methods for 
Weaker Shared Memory Consistency Models

Rajnish P. Ghughal\(^1,2\) * and Ganesh C. Gopalakrishnan\(^2\)**

\(^1\) Formal Verification Engineer, Intel, Oregon. rajnish.ghughal@intel.com 
\(^2\) Department of Computer Science, University of Utah, Salt Lake City, UT 84112-9205. ganesh@cs.utah.edu

**Abstract.** The problem of verifying finite-state models of shared memory multiprocessor coherence protocols for conformance to weaker memory consistency models is examined. We start with W.W. Collier's architectural testing methods and extend it in several non-trivial ways in order to be able to handle weaker memory models. This, our first contribution, presents the construction of architectural testing programs similar to those constructed by Collier (e.g. the ARCHITEST suite) suited for weaker memory models. Our own primary emphasis has, however, been to adapt these methods to the realm of model-checking. In an earlier effort (joint work with Nalumasu and Mokkedem), we had demonstrated how to adapt Collier's architectural testing methods to model-checking. Our verification approach consisted of abstracting executions that violate memory orderings into a fixed collection of automata (called Test Automata) that depend only on the memory model. The main advantage of this approach, called Test Model-checking, is that the test automata remain fixed during the iterative design cycle when different coherence protocols that (presumably) implement a given memory model are being compared for performance. This facilitates 'push-button' re-verification when each new protocol is being considered. Our second contribution is to extend the methods of constructing test automata to be able to handle architectural tests for weaker memory models. After reviewing prior work, in this paper we mainly focus on architectural tests for weaker memory models and the new abstraction methods thereof to construct test automata for weaker memory models.

1 Introduction

Virtually all high-end CPUs are designed for multiprocessor operation in systems such as symmetric multiprocessor servers and distributed shared memory systems. As processors are getting faster faster than memories are, modern CPUs employ *shared memory consistency models* that permit more optimizations at the hardware and compiler levels. As weaker memory models (weaker relative

* The author is currently at Intel, Oregon and was at University of Utah during the course of the research work presented here.

** Supported in part by NSF Grant No. CCR-980928.
to sequential consistency [27]) permit more hardware/compiler optimizations, virtually all modern processors employ a weak memory model such as total store ordering (TSO, [36]), partial store ordering (PSO, [36]), or the Alpha Shared Memory Model [34]. Most past work in verifying processors for conformance to memory models has, however, focussed on sequential consistency verification. The upshot of these facts is that there is very limited understanding in the formal verification community on verifying conformance to weaker memory models, and to do it in a way that fits in a modern design cycle in which design changes, and hence verification regressions, are very important.

**Contribution 1: Architectural tests for Weaker Memory Models**

Our first contribution is in formally characterizing several weaker memory models and presenting new architectural tests for them. In our approach, a formal memory model is viewed as a conjunction of elementary ordering “rules” (relations) such as read ordering and write ordering, as defined by Collier [6] in conjunction with architectural testing methods for multiprocessor machines developed by him. For example, sequential consistency can be viewed as a conjunction of computational ordering (CMP), program ordering (PO), and write atomicity (WA). This is written “SC=(CMP,PO,WA)” where the right-hand side of the equation is called a compound rule, with CMP, PO, WA, etc., then called elementary rules. Collier’s work was largely geared towards strong memory models, as well as certain atypical weaker memory models. For these memory models, it turns out that it is sufficient to verify for conjunctions of ‘classical’ memory ordering rules such as PO, WA, etc. However, weaker memory models relax these classical ordering rules (often PO and WA) in subtle ways. For example, as we show later, TSO relaxes the write-to-read ordering (WR) aspect of PO. TSO also relaxes WA slightly. Therefore, in a memory system that is supposed to implement TSO, a violation of the classical PO rule does not mean that the memory system is erroneous. The memory system is erroneous with respect to PO only if it violates an aspect of PO other than WR orderings. Specifically, given that PO is made up of four sub-rules, namely RO (read ordering), WO (write ordering), WR (write-read ordering), and RW (read-write ordering), it means we must be prepared to look for violations of RO, WO, or RW. Generalizing this idea, to extend Collier’s method to cover practical weaker memory models, pure tests that test for violations of a single elementary architectural rule or limited combinations of elementary rules would be good to have. In this paper, we outline an example pure test. This example presents a test that checks whether (CMP,RO) (the conjunction of CMP and RO) is violated. We have developed several other such pure tests for other rules to facilitate testing for different weak memory models - some of considerably more complexity than the example presented. We will not be presenting all the tests but provide a brief summary of our results at the end of this paper.

In this paper, we explain the technique by which we arrive at pure tests, and examine various aspects of this process, including many non-obvious special cases as well as a few limitations. As one example, we show that sometimes we
need to limit the degree to which we leave out rules from a compound rule. For example, we show that the combination (CMP, WO) (WO is “write ordering”) is irrelevant in practice; instead, the minimal pure rule worthy of study is (CMP, UPO, WO) where UPO denotes uniprocessor ordering. As another example, we show that WO is indistinguishable from WOS if CMP, and a relaxed write atomicity condition WA-S are provided.

The practical implications of these results is that they allow us to explore various tests for a combination of elemental ordering rules and reason about whether an elemental rule is obeyed in presence of other rules. This also enables us to examine a weaker memory model for all aspects of its behavior, come up with different tests to stress these aspects separately, and to correlate the test results. In our work, we have obtained such characterizations for PSO, the Alpha Shared Memory Model, and the IBM 370 memory model. We investigate various pure tests to facilitate verification of conformance to these weaker memory models. In a nutshell, our contribution allows the ArchTest methodology to apply to several practical weaker memory models.

**Contribution 2: New Abstraction Methods for Architectural Tests**

Our second contribution pertains to new abstraction methods in test model-checking as explained below. In our earlier work [14,31], we reported our test model-checking approach to verify finite-state models of shared memory systems for conformance to sequential consistency. Test model-checking is basically a reachability analysis technique in which the model of the memory system being verified is closed with test automata playing the roles of the CPUs. The test automata administer a predetermined sequence of write operations involving only a limited range of addresses as well as data values. These writes are interspersed with reads over the same addresses. The test automata were constructed in such a way that when the reads return “unexpected” values, they move to error states, flagging ordering rule violations.

Test model-checking can be carried out in the framework of temporal logic (say, LTL) model-checking by converting each test automaton into a temporal logic formula and checking for the safety property □(¬inErrorState). In a practical setting, however, specialized reachability analysis algorithms may perform better. The fact that the test automata remain the same despite changes in the shared memory system implementation is a significant advantage, as the test model-checking algorithm can be automatically reapplied after each design iteration. In contrast, previous methods required the characterization of the reference specification, namely the desired formal memory model, in terms of very complex temporal logic specifications involving internal details of the memory system under design. This requires the error-prone step of rewriting the temporal logic specification following each design iteration. Many previous efforts also involved manual proofs which are not needed in our approach. For these reasons, test model-checking is eminently suited for use in actual design cycles.

Our earlier reported work on test model-checking [14,31] serves as the background for the work reported here. Our contributions in these works were the
following. We demonstrated that test automata can be derived through sound abstractions of architectural tests similar to ArchTest. The abstractions were based on data independence and address semi-dependence. These notions are defined with respect to executions, where executions are shared memory programs with reads annotated with the read data values. Under data independence, executions are closed under function applications to the involved data values; in other words, changing the data values does not affect the behavior of the memory system. Under address semi-dependence [22], no other operations may be performed on addresses other than comparison for equality. In our earlier work, we showed that test automata give the effect of running architectural tests for all possible addresses, data values, architectural test-program lengths, and interleavings.

The specific contribution we make with regard to test model-checking is in developing additional abstraction methods that help apply test model-checking for more general varieties of architectural tests. To give a few motivating details, the new pure tests we have developed for handling weaker memory models involve architectural tests that examine a finite unbounded history of read values. To handle these situations, we employ data abstraction in conjunction with properties of Boolean operators to derive a finite summary of these histories. Details of these abstraction methods as well as their soundness proofs are provided in [13].

Another related contribution we make is in handling memory barriers. Given that the test-automata administer a non-deterministic sequence of memory operations, a question that arises in connection with ‘membar’ instructions is how many membar instructions to consider. We show that under reasonable assumptions — specifically that the memory system does not decode the number of membar instructions it has seen — we need to consider only a limited number of membar instructions. Again, this paper only attempts to provide a self-contained overview of these results, with details appearing in [13].

Related Work.

In [19], abstract interpretation is employed to reduce infinite-system verification to finite ∀CTL* model checking. Besides the use of the more complex logic, in their technique: (i) the properties being verified are quite involved even for simple academic examples, and (ii) the properties are intimately related to the internal design details of the memory system, thus not supporting easy verification regressions. In [15,16], 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. In [5], it is shown that the set of sequentially consistent traces is not recursive. In [20], the problem of verifying parameterized memory systems is considered. They propose a method for verifying sequential consistency in parameterized
memory systems using a combination of manual invariant discovery and automated model-checking.

Various weaker memory models have been proposed by researchers to take advantage of various architectural optimizations in the design or otherwise improve performance [2–4, 11, 12, 17, 18, 24, 28]. Many weaker memory models have been proposed for use in commercial architectures also. The SPARC V9 architecture [36] proposed Total Store Ordering (TSO) and Partial Store Ordering (PSO) weaker memory models, which can be selectively implemented. Alpha architecture [34] 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, 29]. There have been many approaches proposing a formal framework to define and compare various weaker memory models [8–10, 21, 23, 26, 33].

In summary, many researchers have proposed weaker memory models, and many existing commercial systems also provide weaker memory model architectures. However, in spite of the great interest in weaker memory models, very little has been reported on formal verification methods for verifying conformance to these weak memory models. Most existing work in weak memory models 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.

2 ARCHTEST and Test Model-checking

In this section, we will examine an example ARCHTEST test for $A(CMP, RO, WO)$ and introduce test model-checking technique and test automata for this test.

2.1 ARCHTEST formal framework

We provide a brief summary of ARCHTEST’s formal framework for reasoning about memory systems and memory models. The complete framework is presented in [6].

ARCHTEST introduces the notion of an event that is intended to represent the activity of each process of accessing various stores 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 (i.e., per processor in the system). For example, consider the statement $X := B$ labeled $L_1$ of process $P_1$. 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 $(P_1, L_1, W, 0, X, S_2)$ (assuming there are two processors
in the system). * 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
- \(S\) 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.

ARCHTEST defines various elemental rules which essentially constrain the
order among various events. Let us consider rule of read ordering \(RO\) as an
example. Rule of read ordering \(RO\) is intended to capture that read events from
the same process happen in the program order. In ARCHTEST framework, rule
of read ordering \(RO\) specifies the following constraint on the order of two read
events.

\[(P, L, R, V, O, S) <_{RO} (\_\_\_\_\_\_)\]

The above template specifies that two read events are ordered by read order-
ing, \(<_{RO}\) if and only if they come from instructions in the same program (\(P\) in
the left tuple and = in the corresponding entry of the second tuple), come from
any program “label” (\(L\) in the first tuple and =, i.e., don’t care, in the second
tuple), both are read instructions (\(R, R\) ), the values (\(V\) ) and operands (\(O\) ) are
immaterial, and the read events are observed in the same store (\(S, =\)). Note that
it is implicit that the right read event occurs in program order later than the
left read event. Similarly rule of write ordering \(WO\) is defined as follows.

\[(P, L, W, V, O, S) <_{WO} (\_\_\_\_\_\_)\]

Many other rules are specified in [6].

* 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.
2.2 Test\textsubscript{ROWO}: ArchTest Test for $A(CMP, RO, WO)$

In ArchTest framework, compound rule $A(CMP, RO, WO)$ essentially represents a memory model which requires that all read are executed in program order and all writes are executed in program order. The test of ArchTest for architecture $A(CMP, RO, WO)$, is shown in Figure 1. 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 is true.

**Condition 1 (Monotonic)** The sequence of $X$ values is monotonically increasing, i.e.,:
\[ \forall i, j : 1 \leq i \leq j : X[i] \leq X[j] \text{ or equivalently } \forall i : 1 \leq i \leq k - 1 : X[i] \leq X[i + 1]. \]

If Monotonic condition is violated then at least one of the $CMP$, $RO$ and $WO$ rules is violated. In [13], we provide a formal proof for the correctness of this test.

All ArchTest test programs such as Test\textsubscript{WA}, Test\textsubscript{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\textsubscript{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.

2.3 Test Model-checking

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 [35] to capture the memory system models as well as the test automata. More precisely, the automata are modeled as (Verilog) processes which run in

\[
\text{Initially, } A = 0 \\
P_1 \quad P_2 \\
L_1 : A := 1; \quad X[1] := A; \\
L_2 : A := 2; \quad X[2] := A; \\
L_3 : A := 3; \quad X[3] := A; \\
\ldots \\
L_k : A := k \quad X[k] := A;
\]

Fig. 1. Test\textsubscript{ROWO}: ArchTest test for $A(CMP, RO, WO)$. 

---

---
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. 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.

**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 semidependent [22], 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.

2.4 Creation of Test Automata

As illustrated in Figure 2, 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 1 into that of (nondeterministically) writing all possible ascending values over $\{0,1\}$, as shown in $P_1$ of Figure 2. 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 $x_1$ and $x_2$ that are nondeterministically recorded by $P_2$. Hence, the memory rule safety property we model-check for is: $P_2$ in final state $\Rightarrow x_2 \geq x_1$.

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

![Fig. 2. TestROWO test automata: test automata for A(CMP, RO, WO).](image-url)
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 1 can be viewed as repeating the initialization and then repeating the instruction at label $L_1$ of $P_1$ of Figure 1.

2.5 Abstracting TestROWO

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

$$\exists i : 1 \leq i < k : X[i] > X[i + 1]$$

$$\iff \exists i, \alpha : 1 \leq i < k : (X[i] > \alpha) \land (X[i + 1] \leq \alpha)$$

$$\iff \exists i, \alpha : 1 \leq i < k : (X[i] > \alpha) \land \neg (X[i + 1] > \alpha)$$

Since, the last formula compares $X[i]$ and $X[i + 1]$ only to $\alpha$, we can rewrite the test program as shown in Figure 3(a) assuming data independence, and rewrite the last formula as

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

Note that in Figure 3(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(b). Figure 3(c) is obtained by simplifying Figure 3(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 2. Intuitively the automaton finds the violation as follows. $P_1$ remains in the initial state for $\alpha$ 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(c) except that the test automaton gives the effect of taking $k$ to $\infty$. Also notice that $x1$ and $x2$ get the values corresponding to $X[i]$ and $X[i + 1]$. Also, corresponding to $X[i] = 1 \land X[i + 1] = 0$, we have $x1 = 1 \land x2 = 0$. Hence the memory rule safety property corresponding to condition MONOTONIC is found violated by the test automaton exactly when TestROWO 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 $\alpha$ and $i$ corresponding to the violation.
Initially, $A = 0$

$P_1$

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

$P_2$

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

$P_1$

$L_1 : A := 1; \quad X[1] := A$
$L_2 : A := 2; \quad X[2] := A$ (a)
$L_3 : A := 3; \quad X[3] := A$ (b)
\ldots
$L_k : A := k \quad X[k] := A$ (c)

$P_2$

\ldots
$L_{\alpha-1} : A := \alpha \quad X[\alpha] := A$
$L_{\alpha} : A := 1 \quad X[\alpha + 1] := A$
$L_{\alpha+1} : A := 1 \quad X[\alpha + 2] := A$
\ldots
$L_k : A := 1 \quad X[k] := A$

Fig. 3. Abstraction of Test_{ROWO}: (a) 1 bit captures the ordering information. (b) 1 bit is written by $P_1$. (c) Writing values just 0 and 1.

3 Test Model-checking for weaker memory models

Suppose a memory system which provides some weaker memory model obeys RO (read ordering) but does not obey WO (write ordering). Test automata for Test_{ROWO} could not be used to verify conformance of such memory systems as violations of WO are indeed legal. We need to provide new tests for such weaker memory models. There are also other issues related to weaker memory model (such as memory barrier etc. instructions) that we need to address while developing test model-checking solutions for weaker memory models. We have developed a suite of new ArchTest style tests and corresponding test model-checking automata along with formal proofs of correctness of them. We will provide one example from this set of tests and will provide summary of the other tests. The details of the other tests and test automata is reported elsewhere [13].

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.

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 \( R_1 \) or \( R_2 \)). For example, \( Test_{ROWO} \) 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.

3.1 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 \( A(CMP, RO) \).

Initially, \( A = X = Y = Z = 0 \)

\[
P_1 \\
P_2 \\
L_1 : A := 1; \\
L_1 : X := A; \\
L_2 : A := 2; \\
L_2 : Y := A; \\
L_3 : Z := A; \\
\]

Finally, \( A = 2; X = Z = 1; Y = 2 \)

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 \( P_3 \). We provide a formal proof in [13].

Formally, we could show a cycle in every explanation of the order of events in this execution assuming \( RO \).

If we assume that \( RO \) is obeyed, then any explanation of the events of this execution will involve the following cycle:

\[
(P_1, L_2, W_2, A, S_2) <_{CRW} (P_2, L_2, R_2, A, S_2) \\
<_{RO} (P_2, L_3, R_1, A, S_2) \\
<_{CRW} (P_1, L_2, W_2, A, S_2) \\
\]

A graphical view of this cycle is shown in Figure 4. 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.
Fig. 4. A cycle corresponding to violation of $A(CMP, RO)$.

- 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.

$Test_{RO}$ A pure test for $A(CMP, RO)$ is proposed in Figure 5. The test is similar to $Test_{ROWO}$ in structure. If the memory system correctly realizes $A(CMP, RO)$, then Condition 2 is satisfied.

CONDITION 2 (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 2 implies that architecture $A(CMP, RO)$ is violated.

Consider a violation of condition 2:

$\exists p, q, r : 1 \leq p < q < r \leq k : X[p] = X[r] \land X[p] \neq X[q]$  

$\iff \exists p, q, r, \alpha, \beta : 1 \leq p < q < r \leq k : X[p] = X[r] = \alpha \land X[q] = \beta \land \alpha \neq \beta$

**Initially,** $A = 0$

$\begin{align*}
P_1 & \quad P_2 \\
L_1 : A := 1; & \quad L_1 : X[1] := A; \\
L_2 : A := 2; & \quad L_2 : X[2] := A; \\
L_3 : A := 3; & \quad L_3 : X[3] := A; \\
\cdots & \quad \cdots \\
L_k : A := k & \quad L_k : X[k] := A;
\end{align*}$

Fig. 5. $Test_{RO}$: a new test for $A(CMP, RO)$. 
Consider an execution corresponding to a violation of the condition. To show that the execution does not obey \( A(CMP, RO) \), we show a cycle in any explanation of the events involved if the memory system obeys read ordering. The exact cycle and the events involved in the cycle depends on the order in which \( \alpha \) and \( \beta \) 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 \( \alpha \) of A becomes visible to \( P_2 \) before value \( \beta \) does. The events and the arcs involving these events are shown in Figure 6(a). In this case, the analysis is similar to that of the example execution described earlier. The following cycle exists.

\[
(P_1, L_\beta, W; \beta, A, S_2) <_{CWR} (P_2, L_q, R; \beta, A, S_2)
\]

\[
<_{RO} \quad (P_2, L_r, R; \alpha, A, S_2)
\]

\[
<_{CWR} \quad (P_1, L_\beta, W; \beta, A, S_2)
\]

The cycle is indicated by bold edges in Figure 6(a).

**Case II:** Consider the case when value \( \alpha \) of A becomes visible to \( P_2 \) after value \( \beta \) does. The events and the arcs involving these events are shown in Figure 6(b). The \( CWR \) 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 \( \alpha \) as that is the latest value seen by that R event.

Thus, the following cycle exists.

\[
(P_1, L_\beta, W; \beta, A, S_2) <_{CWR} (P_2, L_q, R; \beta, A, S_2)
\]

\[
<_{RO} \quad (P_2, L_r, R; \alpha, A, S_2)
\]

\[
<_{CWR} \quad (P_1, L_\beta, W; \beta, A, S_2)
\]

The cycle is indicated by bold edges in Figure 6(b).

**Fig. 6.** A cycle corresponding to violation of \( Test_{RO} \) condition: (a) \( \alpha \) becomes visible before \( \beta \) (b) \( \beta \) becomes visible before \( \alpha \).
3.2 Test Automata for Test\textsubscript{RO}

We can obtain a test automata for Test\textsubscript{RO} as shown in Figure 7. The process of obtaining a test automata is similar to that for Test\textsubscript{ROWO}; Test\textsubscript{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 $x_1$, $x_2$ and $x_3$. The memory rule safety property to be model-checked is: $P_2$ in final state $\Rightarrow (x_1 = x_3 = 1 \Rightarrow x_1 = x_2 = x_3)$

![Diagram](image.png)

Now, we show that if the test program in Test\textsubscript{RO} shows that condition 2 is violated, then the test automata also reveals the error. Since condition 2 is violated,

$$\exists p, q, r : 1 \leq p < q < r \leq k : X[p] = X[r] \land X[p] \neq X[q]$$

$$\iff \exists p, q, r, \alpha, \beta : 1 \leq p < q < r \leq k : X[p] = X[r] = \alpha \land X[q] = \beta \land \alpha \neq \beta$$

We need to consider two cases.

**Case I**: $\alpha > \beta$. In this case, the violation of condition 2 could be captured by a nondeterministic run of test automata for Test\textsubscript{RO} when $P_1$ switched to $s_1$
after $\alpha - 1$ iterations on $s_0$. Variables $x_1$, $x_2$ and $x_3$ get the value which would have been stored in $X[p]$, $X[q]$ and $X[r]$ respectively. In this case, $x_2$ gets the value of the write event executed in $s_1$ state. The memory rule safety property of test automata would be violated with $x_1$, $x_2$ and $x_3$ values being 1, 0 and 1.

**Case II**: $\alpha < \beta$. In this case, the violation of condition 2 could be captured by a nondeterministic run of test automata for $Test_{RO}$ when $P_1$ switched to $s_1$ after $\alpha - 1$ iterations on $s_0$. Variables $x_1$, $x_2$ and $x_3$ store the value which would have been stored in $X[p]$, $X[q]$ and $X[r]$ respectively. In this case, $x_2$ gets the value of the write event executed in $s_0$ state. The memory rule safety property of test automata would be violated with $x_1$, $x_2$ and $x_3$ values being 1, 0 and 1.

Hence, in either case, a corresponding violation would occur in test automata for $Test_{RO}$. Thus, the test automata memory rule safety property corresponding to condition 2 would be found violated whenever the condition is found violated in $Test_{RO}$.

4 Summary of Results

4.1 ‘Sanity-checks’ against the Operational Semantics of TSO

We now summarize our key results in the form of tables and provide an overview (details are in [13]). In Table 1, we summarize the results of test model-checking an operational model of TSO implemented in Berkeley VIS Verilog [35]. This operational model is similar to that used in [7], and usually corresponds to the reference specification of TSO. The two ‘fail’ entries in the table correspond to program ordering, (CMP,PO), and write-to-read orderings, (CMP,WR). Since these orderings are not obeyed in TSO, we obtain ‘fail’ correctly. The other architectural tests in the tables indicate ‘pass’ which means that TSO obeys them. These pass/fail results provide added assurance (a ‘sanity check’) that our characterization of weaker memory models is consistent with the popular understanding of weaker memory models.

**Note that in both cases, the corresponding cycle could be of either type as shown in Figure 6.**

<table>
<thead>
<tr>
<th>test automata</th>
<th>#states</th>
<th>#bld nodes</th>
<th>runtime (mn:sec)</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP, RO, WO</td>
<td>3819</td>
<td>4872</td>
<td>&lt; 1s</td>
<td>pass</td>
</tr>
<tr>
<td>CMP, PO</td>
<td>6.50875e+06</td>
<td>30051</td>
<td>2:38</td>
<td>fail</td>
</tr>
<tr>
<td>CMP, WR</td>
<td>6.50875e+06</td>
<td>30051</td>
<td>1:25</td>
<td>fail</td>
</tr>
<tr>
<td>CMP, RW</td>
<td>6.50875e+06</td>
<td>30051</td>
<td>3:02</td>
<td>pass</td>
</tr>
<tr>
<td>CMP, RO</td>
<td>10187</td>
<td>2463</td>
<td>0:37</td>
<td>pass</td>
</tr>
</tbody>
</table>

Table 1. Verification results on an operational model of TSO using VIS
4.2 Architectural Rules Covered

Table 2 shows various architecture rules and their transition templates. The idea of transition templates introduced in [6] specified a summary of the ordering rule. Many of the entries in this table were specified in [6]. We have defined new architectural rules (MB—RR through WA—S intra), defined tests and test automata for them, as well as provided more complete tests for many of the previously existing rules.

Table 3 shows the architecture rules in our discussion and the subrules each of them consists of. In particular, note WA—S, which is a relaxed write-atomicity rule that is one of the central sub-rules of TSO. Briefly, write events become visible to the processor issuing the write first, and then the events become atomically visible to all other processors. In contrast, in sequential consistency, each write becomes atomically visible (at the same time) to all the processors.

<table>
<thead>
<tr>
<th>Architecture rule</th>
<th>Transition template</th>
</tr>
</thead>
<tbody>
<tr>
<td>MB—RR</td>
<td>((P, L, R, V, O, S) &lt;_{MB-RR} (=, =, MB - RR, -, -, -))</td>
</tr>
<tr>
<td>WA—S</td>
<td>((P, L, W, V, O, \neq P) =_{WA-S} (=, =, W, =, \neq P))</td>
</tr>
<tr>
<td>WA—S intra</td>
<td>((P, L, W, V, O, = P) &lt;<em>{WA-S</em>{intra}} (=, =, W, =, \neq P))</td>
</tr>
<tr>
<td>CON</td>
<td>((P, L, W, V, O, S) &lt;_{CON} (=, =, W, =, =))</td>
</tr>
<tr>
<td>WA</td>
<td>((P, L, A, V, O, S) &lt;_{WA} (=, =, =, =))</td>
</tr>
</tbody>
</table>
Table 3. Architecture rules

<table>
<thead>
<tr>
<th>Architecture rule</th>
<th>Subrules</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP</td>
<td>$RW$, $CRW$, $CWW$, $CWR$</td>
</tr>
<tr>
<td>PO</td>
<td>$WR$, $WO$, $RO$, $RW$</td>
</tr>
<tr>
<td>$WA - S$</td>
<td>$=WA - S$, $WA - S_{intra}$, $CON$</td>
</tr>
<tr>
<td>MB</td>
<td>$MB - WR$, $MB - WO$, $MB - RO$, $MB - RW$</td>
</tr>
</tbody>
</table>

4.3 Architectural Rules for Practical Memory Systems

Table 4 shows the memory models in our discussion and their specification in the ARCHTEST framework. These results provide, to the best of our knowledge, the first formal characterization, in one consistent framework, of several practical weaker memory ordering rules. For example, by contrasting TSO and the Alpha Shared Memory Model, it becomes clear that the later is much weaker than the former in terms of read/write orderings, but provides more safety-net operations to recover these orderings.

The Alpha architecture manual [34] describes a number of executions called Litmus tests to illustrate which shared memory behavior is allowed and not allowed by the Alpha Shared Memory Model. In [13], we show that all these litmus tests are (often trivially) covered by our characterization of the Alpha Shared Memory architectural compound rule. In addition to sanity-checking our results, these results indicate that a developer of a modern memory system can use our architectural rules to debug the memory system focusing on each facet (sub-rule) at a time.

Table 4. Memory models specification in the ARCHTEST framework

<table>
<thead>
<tr>
<th>Memory Model</th>
<th>ARCHTEST specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential consistency</td>
<td>$A(CMP, PO, WA)$</td>
</tr>
<tr>
<td>IBM 370</td>
<td>$A(CMP, UPO, RO, WO, RW, WA, MB - WR)$</td>
</tr>
<tr>
<td>Total Sorted Order (TSO)</td>
<td>$A(CMP, UPO, RO, WO, RW, WA - S, MB - WR)$</td>
</tr>
<tr>
<td>Partial Sorted Order (PSO)</td>
<td>$A(CMP, UPO, RO, RW, WA - S, MB - WR, MB - WW)$</td>
</tr>
<tr>
<td>Alpha Shared Memory Model</td>
<td>$A(CMP, UPO, R0O, WA - S, MB, MB - WW)$</td>
</tr>
</tbody>
</table>
5 Conclusions and Future Work

We formally characterize the problem of verifying finite-state models of shared memory multiprocessor coherence protocols for conformance to weaker memory consistency models in terms of Collier’s architectural testing methods. We extend Collier’s framework in several non-trivial ways in order to be able to handle weaker memory models. This permits the construction of architectural testing programs similar to those constructed by Collier (e.g. the ARCHTEST suite). We then adapt these tests to the realm of model-checking, to permit early life-cycle formal verification of design descriptions. Our approach consists of abstracting executions that violate memory orderings into a fixed collection of automata (called Test Automata) that depend only on the memory model. The main advantage of our test model-checking approach is that the test automata remain fixed during the iterative design cycle when different coherence protocols that (presumably) implement a given memory model are being compared for performance. This facilitates ‘push-button’ re-verification when each new protocol is being considered. Here, we report our new results that extend the methods of constructing test automata to be able to handle architectural tests for weaker memory models. We achieve this as follows: (i) we define new abstraction techniques for summarizing execution histories; (ii) we prove the soundness of these abstractions; and (iii) we provide practical means to reduce the number of memory barrier instructions used in test automata.

We provide a formal characterization, in one consistent framework, of several practical weaker memory ordering rules, including the TSO and PSO models [36], the IBM 370 memory model, and the Alpha Shared Memory Model [34]. We show that ‘Litmus tests’ that practitioners employ for the Alpha memory model are covered by our formalism. We define a suite of architectural tests and corresponding test model-checking automata to facilitate verification of weaker memory models. We report on VIS based model-checking results that clearly show how developers of modern memory systems can use our architectural rules to debug the memory system focusing on each facet (sub-rule) at a time. This helps cut down verification complexity and helps pinpoint errors.

We believe that test automata can provide leverage in attacking the shared memory system verification problem, as they avoid many redundant test cases that would otherwise have been used in a simulation framework. We are working on overcoming a limitation of the present work, namely that we do not yet have complete test automata for the weaker memory models examined here. Promising completeness results obtained in another work [30] may provide the necessary directions to pursue.

We are also investigating ways to mitigate state explosion. Since test model-checking can be cast as finite-state reachability analysis, efficient techniques under development by many groups for exact- as well as approximate reachability analysis are believed to be one promising direction to pursue. Once we get a handle on state explosion through reachability analysis based test model-checking, we plan to work on overcoming many other sources of inefficiency, including symmetries that, as yet, stand unexploited. Works on symmetry exploitation
(e.g., [25]) and the use of symbolic state descriptors (e.g., [32]) will be examined for possible answers. The integration of these ideas back into the realm of reachability analysis is expected to be a long-term direction.

References


