

An Overview of Test Model-Checking for Verifying Conformance to Shared Memory Models*

Ganesh Gopalakrishnan
University of Utah, School of Computing
ganesh@cs.utah.edu

1 Introduction

Test model-checking [NGMG98, GMNG98] is a method for debugging finite-state models of shared memory coherence protocols for conformance to desired memory ordering rules (memory models). When a shared memory concurrent program consisting of reads and writes is executed, the coherence protocol performs internal actions on behalf of reads and writes, manifesting all observable effects through read return values. The original concurrent program with all reads annotated with their returned values is called a concurrent execution. Even for finite-state models of memory systems, the set of illegal concurrent executions with respect to a given memory model is infinite. We obtain a handle on verification by classifying the violations for a memory model into a finite number of classes and looking for representatives of each class. This procedure is carried out by test automata that take the place of CPUs, administering non-deterministic sequences of writes and looking for the violation of a safety property with respect to the read values. The situation is analogous to the use of a catalog of Temporal Logic formulas in traditional model-checking. In fact, expressing the test automata as Temporal Logic formulas is possible, albeit at the risk of producing a virtually unreadable formula. Test model-checking is an adaptation of the ARCHTEST [Col92] (hereafter referred to as RAPA, standing for “Reasoning about Parallel Architectures”) style of testing parallel architectures.

Test model-checking provides several practical advantages over other methods. The fact that the architectural tests are conducted with respect to finite-state models of the memory system using model-checking means that it can be applied during design time. Since test automata

*Virtually all the material in this paper is based on the works of Ratan Nalumasu [Nal98], Rajnish Ghughal [Ghu99], Michael Jones, Ali Sezgin, Ritwik Bhattacharya, and Prosenjit Chatterjee. Their efforts are gratefully acknowledged. Support by NSF through award MIP-9987516 as well as Intel through a cash and equipment award are gratefully acknowledged.

depend only on the memory model for which they are written, and not on the internal properties of the memory system which might change during design iterations, verification can be rapidly re-invoked following bug fixes. Another significant advantage comes from our modeling of memory models using elementary architectural rules such as read orderings, write orderings, etc. By creating focussed *pure tests* for these elementary rules that are independent of the other ordering rules, a designer is able to pinpoint the source of violations. They can also apply test model-checking runs as putative queries to understand the memory system as well as memory model better. On the negative side, by not taking advantage of the specific implementation details, we may be unable to provide completeness in a formal sense. In the past, we have reported on several aspects of test model-checking. In this extended abstract, we try to provide more concrete examples as well as indicate results that may lead to complete test automata for interesting practical situations. A companion extended abstract in these proceedings by Jones and Gopalakrishnan presents a case study where a subtle coding error introduced by us in a model of the MIT HCN coherence protocol was automatically detected via a combination of test model-checking and parameterized verification.

2 Examples

We illustrate test model-checking through a simple example. We create a simple operational model of a TSO [WG94] memory system in Promela [Hol91] and test the model for conformance to the PRAM [LS88] memory model for one address. In actual use of test model-checking, the memory system model will not be such an operational model, but instead a finite-state model of the memory system being verified. In the style of RAPA, we define PRAM to be (CMP,POS)¹.

2.1 An Example

The TSO operational model is specified in the standard way by having write buffers that go out to the memory, permitting reads to first look for matching addresses in the write buffer, and if not found, reading from main memory. One can model the write buffer and main memory as follows:

```
chan WRITE_BUFFER[PROCESSORS] = [WRITE_BUFFER_CAPACITY] of {Addr,Data};
MEMORY memory_data[NUMBER_OF_ADDRESSES];
```

We can now express the condition under which writes may happen as well as what reads return:

¹Our definition for POS appears to be slightly different from that used by RAPA. Given two writes w1 and w2 occurring in that order in a sequential program, our definition of POS says that all the events caused by these writes in every store respect the w1 < w2 order. Using this definition of POS, it can be shown that PRAM is the same as (CMP,POS). Also we define CMP without using the SRW rule of RAPA, as we consider reads and writes to be primitives, as opposed to the assignment statement, as in RAPA.

```

#define may_write (len(write_buffer) < WRITE_BUFFER_CAPACITY)
#define read_value(a,d)
( ((len(write_buffer)==0)&&(memory_data[a]==d))
  ||((len(write_buffer)>0) && (write_buffer?[a,d])) )

```

Writes can be enqueued so long as there is room in the write buffer. A read instruction returns the value from the buffer in case of an address match, or the value from memory.

The main memory non-deterministically updates itself from the write buffers. Since we are modeling only one address, the data always goes into location `memory_data[0]`:

```

proctype Memory_Update(chan write_buffer)
{do
  :: write_buffer?0,memory_data[0];
od}

```

For verifying one-address executions against the PRAM memory model, we employ two test automata called P and Q. For an N processor system, N-1 copies of P and one copy of Q are run, taking the place of the processors:

```

proctype process_P(chan write_buffer)
{P0: do :: may_write->write_buffer!0,2;
      :: atomic{read_value(0,1) }-> goto P1;
  od;
  P1: do :: may_write->write_buffer!0,2;
      :: assert(!read_value(0,0));
  od;}

```

```

proctype process_Q(chan write_buffer)
{Q0:do ::may_write->write_buffer!0,0;
      ::may_write->write_buffer!0,1->goto Q1;
      ::assert(!read_value(0,1));
  od;
  Q1: do ::may_write->write_buffer!0,1;
      ::atomic{read_value(0,1)}->goto Q2;
      ::assert(!read_value(0,0));
  od;
  Q2: do ::may_write->write_buffer!0,1;
      ::assert(!read_value(0,0));
  od;}

```

```

init
{run process_P(WRITE_BUFFER[P]);
 run process_Q(WRITE_BUFFER[Q]);
 run Memory_Update(WRITE_BUFFER[P]);
 run Memory_Update(WRITE_BUFFER[Q]);}

```

SPIN [Hol91] verifies the above model in 106 states, reaching a depth of 37. Our own model-checker for a subset of Promela called PV² verifies it in 102 states.

A snippet of the behavior realized by P and Q is as follows. Suppose process Q moved from state Q0 to state Q1 by writing a 1 in location 0. After this, neither process should be able to read a 1 followed by a 0 from location 0. Basically, Process P and Process Q are trying to detect one of the following five violation classes that are believed to be exhaustive for 1-address PRAM violations. Some of the pertinent details have been published in the participant's edition of the WAVE'00 workshop as well as [Nal98]. A detailed proof is in progress.

Interesting practical observations with respect to the above example are as follows:

- When the operational model for PSO was used in lieu of TSO, a violation for PRAM was easily flagged. Such experiments can help designers understand memory ordering rules at an intuitive level.
- We have created a finite-state model for the HP Runway bus which is believed to be sequentially consistent. While the 1-address PRAM verification finishes on modestly endowed (main memory size is the critical parameter) machines, the 2-address PRAM verification ran into state explosion and could run to completion only on a capacious machine. However, seeded errors are detected easily even on modestly sized machines. Our future work will examine avenues to contain state explosion in a domain-specific manner.
- The above one-address automata detected a subtle error in our encoding of the HCN protocol, as is reported in the paper by Jones and Gopalakrishnan in these proceedings.

3 Issues in Advancing Test Model-checking

The following are the top level issues to be addressed before we make further progress on test model-checking.

Formal specification of memory models

The main attraction of the framework in RAPA is that it allows many memory models to be defined in a uniform manner. Two (of the several) issues to be tackled in carrying this program forward are:

²You may obtain details as well as download a copy of PV from http://www.cs.utah.edu/formal_verification. For all larger test model-checking runs (as well as many other examples) to date, PV has been observed to finish in far less number of states.

- *What primitive operations to use:* RAPA uses assignment statements of the form $A := B$ as primitives. In [Nal98], we have gravitated towards the more traditional approach of considering `read` and `write` operations as primitives. While this difference appears superficial, in reality it is not: RAPA uses the SRW rule as part of CMP that orders the events due to B before those due to A . Many of the tests used in RAPA as well as many of the pure tests proposed in [Ghu99] (which also uses assignments as primitives) critically depend on whether SRW is used or not. Under the presence of the RW and RO rules, however, it appears obvious that SRW is un-necessary; this fact remains to be proved.
- *How to specify memory models:* In [Ghu99], it was shown that specifying memory models such as TSO requires new elementary ordering rules, such as weakened write atomicity (WA-S) as well as several memory barrier rules. Based on these, the following memory models have been formally specified and informally argued to be correct:
 - SC = (CMP,POS,WA)
 - IBM 370 = (CMP,UPO,RO,WO,RW,WA,MB-WR)
 - TSO = (CMP,UPO,RO,WO,RW,WA-S,MB-WR)
 - PSO = (CMP,UPO,RO,RW,WA-S,MB-WR,MB-WW)
 - (Subset of) Alpha = (CMP,UPO,ROO,WA-S,MB,MB-WW)

A formal proof remains to be done.

Characterization of Violation Classes

For each memory model (such as SC=(CMP,POS,WA)) as well as elementary ordering rule (such as (CMP,POS) - POS in the presence of the inevitable CMP), either heuristically good violations are to be obtained, possibly supported by formal proofs of completeness. In [Ghu99], several such new tests are proposed, including for (CMP,RO) for reads happening on different operands. An advantage of the test model-checking framework is that such tests can be accumulated over time and shared by the community. This can permit proof reuse and modular reasoning, instead of having to extricate equivalent lemmas from one-off proofs. We also have complete finite-state characterizations of violations of SC for two processors and addresses assuming POS. This way, verification modulo clear-cut assumptions becomes possible.

Other issues in characterizing violation classes include questions of address saturation: how many addresses are sufficient to be considered in executions to verify various memory models? In [Nal98], it is shown that all (CMP,RO,WOS) violations can be detected within two addresses. It however appears that to verify PRAM (or equivalently (CMP,POS)), N distinct addresses must be modeled. A violation involving four distinct addresses appears below. Here, we tacitly assume unambiguous executions (executions in which each write into a location writes a different value than previously written into that location). For brevity, we use “`wa3`” to indicate a write

of 3 into location a. We use “wa3*” to indicate a write of a value different from 3 into location a.

P1	P2	P3	P4
wa1	rb1	wc2	wd2
wb1	wc2*	wd1	wa1*
	rc2	wc3	wc4
	wc3*		
	rc3		
	rd2		
	rc4		
	ra1		

The reason this is a PRAM violation is as follows. The writes issued in P1, P3, and P4 must maintain their local orders when posted into the store of P2. The reads and writes issued in P2 must maintain their local program order. Any explanation must be serial. This forces P1:wb1 before P2:rb1, P2:wc2* to be before P3:wc2, P3:wd1 before P4:wd2, and P4:wa1* before P1:wa1, thus completing a cycle involving four addresses.

Similarly, sequential consistency is the same as (CMP,POS, WA). We have shown in [Nal98] that address saturation occurs at N, the number of processors.

Sound Abstractions

Arriving at finite-state abstractions of violation scenarios in a soundness preserving manner is in itself an interesting challenge. Complete examples are provided in [Ghu99]; an example below illustrates our point. The following execution is a test that reveals a violation of RO on distinct operands (assuming SRW). The corresponding test automaton given in Figure 1 is based on powerful manually discovered abstract interpretations. It will be essential to automate the discovery and/or verification of such abstractions:

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

Creation of finite-state models of memory systems

A good language in which to specify architectural design models can not only promote the expression of cache coherence protocols but may also provide certain assumptions syntactically.

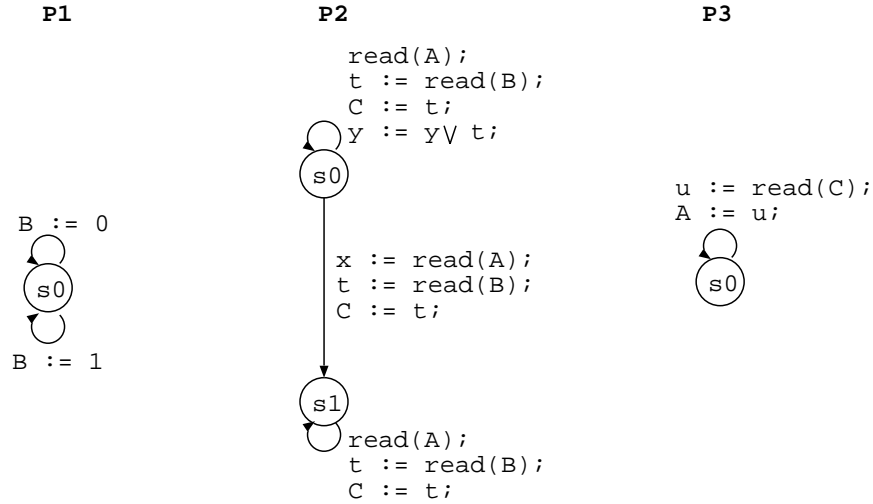


Figure 1: Test automata for RO on different operands

The TLA+ notation (see these proceedings) provides an expressive notation in which to formally specify coherence protocol actions. In [Nal98], a formal guarded-command notation is introduced to specify coherence protocols whereby certain properties such as data independence and address projectability are automatically obtained.

Conquering state explosion and meaningful error reporting

While these problems are endemic to all model-checkers, focussed domain-specific attacks might be possible as well as desired.

Some Conjectures

We were surprised to find that the PRAM=(CMP,POS) test (where POS is defined different from that in RAPA as pointed out earlier) does not fail on the TSO operational model. In [Ghu99], we have defined TSO to be definable as (CMP,UPO,RO,WO,RW,WA-S,MB-WR) where

- CMP,UPO,RO,WO, and RW are defined as in RAPA;
- WA-S is a weakened write-atomicity rule which we introduce to adequately model TSO;
- MB-WR is a membar rule we introduce.

Our conjecture is that (CMP,UPO,WO,RO,RW) is the same as (CMP,POS) where POS is defined as in RAPA. Many additional properties pertaining to WA-S are stated and proved in [Ghu99].

4 Conclusions

We believe that the effort begun in RAPA allows one to study memory systems in terms of elementary ordering rules that can be effectively composed to obtain several practical models. We have found that even incomplete test automata are valuable for debugging (although we certainly intend to pursue formal completeness results). For instance, for the Alpha model, our test automata properly subsume the “litmus tests” proposed in the Alpha architecture manual. We look forward to gaining further practical experience before we know whether such views are indeed true. We are working on building a tool supporting test model-checking and performing case studies.

References

- [Col92] W. W. Collier. *Reasoning About Parallel Architectures*. Prentice-Hall, Englewood Cliffs, NJ, 1992.
- [Ghu99] Rajnish Ghughal. *Test Model-checking Approach to Verification of Formal Memory Models*. PhD thesis, Department of Computer Science, University of Utah, 1999. Also available on www.cs.utah.edu/formal_verification.
- [GMNG98] Rajnish Ghughal, Abdel Mokkedem, Ratan Nalumasu, and Ganesh Gopalakrishnan. Using “test model-checking” to verify the runway-pa8000 memory model. In *Tenth ACM Symposium on Parallel Algorithms and Architectures*, pages 231–239, Puerto Vallarta, Mexico, June 1998. ACM Press.
- [Hol91] Gerard Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
- [LS88] R. J. Lipton and J. S. Sandburg. PRAM: A scalable shared memory model. Technical Report CS-TR-180-88, Princeton University, September 1988.
- [Nal98] Ratan Nalumasu. *Formal Design and Verification Methods for Shared Memory Systems*. PhD thesis, Department of Computer Science, University of Utah, 1998. Also available on www.cs.utah.edu/formal_verification.
- [NGMG98] Ratan Nalumasu, Rajnish Ghughal, Abdel Mokkedem, and Ganesh Gopalakrishnan. The ‘test model-checking’ approach to the verification of formal memory models of multiprocessors. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification, CAV ’98*, volume 1427 of *Lecture Notes in Computer Science*, pages 464–476, Vancouver, BC, Canada, June/July 1998. Springer-Verlag.
- [WG94] David L. Weaver and Tom Germond. *The SPARC Architecture Manual – Version 9*. P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1994.