

# **Overview of ARCHTEST**

**Presented at the  
Tutorial and Workshop  
on Formal Specification  
and Verification Methods  
for Shared Memory Systems**

**31 October 2000  
Austin, Texas, USA**

**by  
William W. Collier  
Multiprocessor Diagnostics  
collier@acm.org  
www.mpdiag.com**

Copyright (C) 2000 Multiprocessor Diagnostics. All rights reserved.

---

## **Table of Contents**

- **Cover Page .**
- **Table of Contents.**

- Sources.
- Abstract.
- Limits of Testing (I).
- Limits of Testing (II).
- Limits of Testing (III).
- Example. Test T7.
- Executions and Events.
- One Write Event Per Process.
- Rules.
- Computation Rules.
- The Statement-Read-Write Rule (SRW).
- The Compute-Read-Write Rule (CRW).
- The Compute-Write-Read Rule (CWR).
- The Compute-Write-Write Rule (CWW).
- The Uniprocessor Order Rules.(CWW).
- Order Rules.
- Recent Results (CWW).
- Cache Coherence Rules (CWW).
- Shorthand Notation.(CWW).
- Architectures.
- Indistinguishable Machines.
- Theorem:  $CC1 \Leftrightarrow CC2$ .
- How to Deduce a Relaxation.
- A Simple Example.
- Explanation of the Circuit.
- Test T7.
- Results of Running ARCHTEST.
- Work in Progress.
- Shared Files as Sink Operands.
- Testing Real Shared Memory Programs.
- Generate Relaxed Data (I).
- Generate Relaxed Data (II).
- Program to Prove  $CC1 \Leftrightarrow CC2$ .

---

## Sources.

The most recent version of these foils can be found at

[www.mpdia.com/overview.htm](http://www.mpdia.com/overview.htm).

**Info on ARCHTEST:** [www.mpdia.com/archtest.html](http://www.mpdia.com/archtest.html).

**Info on *Reasoning about Parallel Architectures*:**  
[www.mpdia.com/rapa.html](http://www.mpdia.com/rapa.html).

---

## **Abstract.**

**ARCHTEST is a program which runs on a shared memory multiprocessor and which seeks to determine what rules a machine fails to obey.**

**ARCHTEST has been used by many companies in the development and testing of new shared memory systems.**

**In addition, considerable research has been based on the formal structure underlying ARCHTEST by Prof. G. C. Gopalakrishnan and his group at the University of Utah.**

---

## **The Limits of Testing with ARCHTEST (I).**

**ARCHTEST cannot prove that a machine obeys a given architecture.**

**(You can fish in a lake all day and not catch any fish,**

**but that does not prove that there are no fish in the lake).**

**ARCHTEST can show only that a machine relaxes an architecture.**

**(However, the output from ARCHTEST can also suggest the frequency and severity of relaxations of particular architectures.)**

-----

## **The Limits of Testing with ARCHTEST (II).**

**ARCHTEST cannot prove that a machine obeys an arbitrary architecture.**

**There are only nine unique tests known.**

**Therefore, we can test only for the architectures associated with those nine tests.**

-----

## **The Limits of Testing with ARCHTEST (III).**

**There are some rules which every machine is expected to obey all the time, namely CMP and UPO.**

**There are some tests which involve only, say, rule R and CMP and UPO.**

**If such a test fails, then it is clear**

the machine relaxes rule R.

Such a test is called a *pure* test for R.

For many rules there exists no pure test.

---

## Example: Test T7 from ARCHTEST.

### Test T700. Seek a relaxation of A(CMP,UPO,RR,CC1).

Initially,  $A = B = U[i] = V[i] = 0, i = 0$  to  $K$ .

P1		P2	
L00:	A = 0;	L00:	B = 0;
L01:	- = A;	L01:	- = B;
L02:	U[0] = B;	L02:	V[0] = A;
L10:	A = 1;	L10:	B = 1;
L11:	- = A;	L11:	- = B;
L12:	U[1] = B;	L12:	V[1] = A;
L20:	A = 2;	L20:	B = 2;
L21:	- = A;	L21:	- = B;
L22:	U[2] = B;	L22:	V[2] = A;
L30:	A = 3;	L30:	B = 3;
L31:	- = A;	L31:	- = B;
L32:	U[3] = B;	L32:	V[3] = A; etc.

Seek 7.1.  $U[i] < j$  and  $V[j] < i$ .  $d1 = V[ U[i]+1 ] -$   
Seek 7.2.  $V[i] < j$  and  $U[j] < i$ .  $d2 = U[ V[i]+1 ] -$

To show: Not A(CMP,UPO,RR,CC1).

To understand the ideas behind this test, we need to understand *executions, events, rules, and architectures*.

---

## Executions and Events.

An *execution* is the act of executing a program P, which operates on some input I to produce a result R.

An execution is composed of one or more processes, each of which is composed of one or more statements.

A *statement* consists of one or more read events along with one write event for each process in the execution.

-----

## In Each Statement One Write Event Per Process.

Test T600.

Initially,  $A = B = U = V = X = Y = 0$ .

P1	P2	P3	P4
$A = 1;$	$U = A;$	$X = B;$	$B = 1;$
	$V = B;$	$Y = A;$	

Initially,  $A = B = U = X = 1, V = Y = 0$ .

The events for the first statement in P2 are:

(P2,L1,R,1,A,S2)  
(P2,L1,W,1,U,S1)  
(P2,L1,W,1,U,S2)  
(P2,L1,W,1,U,S3)  
(P2,L1,W,1,U,S4)

**The write events represent the times at which the new value of A becomes visible to the stores, S1, S2, S3, S4 for processes P1, P2, P3, P4, respectively.**

-----

## **Rules.**

**Programmers expect that machines will exhibit certain behaviors.**

**These behaviors called *rules*.**

**The *computation rules* are so obvious that they are usually taken for granted, but it is not possible to reason about the behavior of a machine without defining these rules explicitly.**

**The *uniprocessor order rules* describe the ordering rules that both uniprocessors and multiprocessors need to obey.**

**The *order rules* are simple. Program order (PO) requires that all events in a process occur in the order defined by the underlying program.**

**The *cache coherence rules* describe the patterns in time at which the output from a statement becomes visible to all processes in an execution.**

**Each rule induces a partial ordering on the events in an execution. (Actually it is more complicated than that.)**

---

## The Computation Rules.

The computation rules are so intuitively obvious that they are almost never stated explicitly. However, they must be defined formally and precisely in order for them to be used in reasoning about interactions between processes on shared data.

A failure to obey one of the rules of computation might look like this:

Initially,  $A = 0$ .

P1  
L0:  $A = 1$ ;

Terminally,  $A = 13$ .

Operand  $A$  is initially zero. Then it is set to one. But at the end of the execution it has the value 13. Clearly, the machine on which the execution occurred did not compute in the way one normally expects a machine to behave.

---

## The Statement-Read-Write Rule (SRW).

Write

$(P1, L0, R, 0, B, S1) <srw (P1, L0, W, 0, A, S1)$

to show that in

P1  
L0: A = B;

the read event(s) in a statement occur before the write event(s).

-----

## The Compute-Read-Write Rule (CWR).

Initially, A = X = 0.

P1                      P2  
L0: A = 1;              L0: X = A;

If X = 0 at the end of the execution, then it must be that the read event in process P2 occurred before the write event in process P1. Represent this by

(P2,L0,R,0,A,S2) <crw (P1,L0,W,1,A,S2)

to show that statement L0 in process P2 read a 0 from operand A in S2 before statement L0 in process P1 wrote a 1 value into operand A in S2.

-----

## The Compute-Write-Read Rule (CWR).

Initially, A = X = 0.

P1                      P2  
L0: A = 1;              L0: X = A;

Terminally,  $A = X = 1$ .

If  $X = 1$  at the end of the execution, then it must be that the write event in process P1 occurred before the read event in process P2. Represent this by

$$(P1, L0, W, 1, A, S2) <_{cwr} (P2, L0, R, 1, A, S2)$$

to show that statement L0 in process P1 wrote a 1 into operand A in S2 (S2 is P2's view of storage) before statement L0 in process P2 read a 1 value from operand A in S2.

-----

## The Compute-Write-Write Rule (CWW).

Initially,  $A = 0$ .

$$\begin{array}{ll} P1 & P2 \\ L0: A = 1; & L0: A = 2; \end{array}$$

If  $A = 2$  (in both S1 and S2, that is, in both P1's and P2's view of storage) after this program has executed, then it must be that the write events in process P1 (for both S1 and S2) occurred before the write events in process P2. Represent this by

$$\begin{array}{ll} (P1, L0, W, 1, A, S1) <_{cww} (P2, L0, W, 2, A, S1) \\ (P1, L0, W, 1, A, S2) <_{cww} (P2, L0, W, 2, A, S2) \end{array}$$

to show that statement L0 in process P1 wrote a 1 into operand A in S1 and in S2 before statement L0 in process P2 wrote a 2 into operand A in S1 and in S2, respectively.

---

## The Rules of Uniprocessor Order (UPO).

On a uniprocessor the parts of the following three code fragments (1) involving operations on X and (2) involving operations on the store for the process containing the code fragments, would always be executed in order.

```
X = 1;      X = 1;      Y = X;  
Y = X;      X = 2;      X = 1;
```

Thus for this fragment

```
    Pi  
X = 1;  
X = 2;
```

we would have

$$(P_i, -, W, 1, X, S_i) <_{\text{upo}} (P_i, -, W, 2, X, S_i)$$

but UPO would not mean that  $(P_i, -, W, 1, X, S_i)$  would necessarily occur before any of the other write events in the second statement.

The necessity of this behavior is so obvious that UPO is treated like CMP as being part of the rules that all machines, even multiprocessors, automatically obey.

---

## Order Rules.

## **Read-Read Order (RR)**

**Reads occur in order.**

## **Write-Write Order (WW)**

**Writes occur in order.**

## **Write-Read Order (WR)**

**If a machine obeys WR, and if a write operation on shared data logically occurs in a program before a read operation on shared data, then the write operation is executed before the read operation.**

**Many machines allow WR to be disobeyed.**

## **Read-Write Order (RW)**

**If a machine obeys RW, and if a read operation on shared data logically occurs in a program before a write operation on shared data, then the read operation is executed before the write operation.**

## **Program Order (PO)**

**A machine obeys PO if and only if it obeys RR, WW, RW, and WR.**

-----

## **Recent Results.**

**If a machine obeys WR, it obeys PO.**

**If a machine obeys RO, it obeys RW.**

**If a machine obeys WO, it obeys RW.**

**Ref: Rajnish Ghughal. *Test Model-Checking Approach to Verification of Formal Memory Models*. December, 1999.  
See [www.cs.utah.edu/formal\\_verification/](http://www.cs.utah.edu/formal_verification/)**

-----

## **Rules of Cache Coherence.**

### **CC1**

**All processes see each operand change value at exactly the same instant. CC1 is also called write atomicity (WA).**

### **CC2**

**All processes see exactly the same sequence of changes of values for all operands. CC2 is also called write synchronization (WS).**

### **CC3**

**All processes see exactly the same sequence of changes of values *for each separate operand*.**

## CC4

Different processes may see an operand assume different sequences of values.

---

### Shorthand Notation.

Suppose a system obeys CC1, and let  $w_1$  and  $w_2$  be any two write events in the same statement. If it is known that event  $x$  happens before  $w_1$ , say, then, due to the write atomicity of the system, it must be that  $x$  happens before  $w_2$ .

Symbolically, if  $x <_{hb} w_1$ , then  $x <_{hb} w_2$ .

This is inconvenient to write when one is trying to establish the existence of a circuit. A more economical notation is to write  $x$

$<_{hb} w_1 =_{cc1} w_2$

to show that  $x$  occurs before  $w_1$  which occurs at the same time as  $w_2$ .

---

### Architectures.

An *architecture* is the set of rules a machine obeys.

If a machine obeys rules  $R_1, R_2, \dots, R_n$ , then denote by  $A(R_1, R_2, \dots, R_n)$  the architecture for the machine.

-----

## Indistinguishable Machines.

Let machine  $M_1$  obey architecture  $A_1$ , and  $M_2$  obey  $A_2$ .

Suppose that every program which can calculate any given result on  $M_1$  can calculate the same result on  $M_2$ , and vice versa.

Then  $M_1$  and  $M_2$  are *indistinguishable*.

This is written:  $M_1 \Leftrightarrow M_2$ .

Alternatively,  $A_1$  and  $A_2$  are indistinguishable.

Otherwise,  $M_1$  and  $M_2$  ( $A_1$  and  $A_2$ ) are *distinguishable*.

-----

## Theorem: $CC_1 \Leftrightarrow CC_2$ .

Let  $A_1$  be any architecture containing  $CC_1$ , and  $A_2$  be any architecture containing  $CC_2$  (but not  $CC_1$ ).

Then  $A_1 \Leftrightarrow A_2$ .

The architecture for Sun machines requires machines to be

**CC2. The documents cite this theorem as support for the machines being in fact CC1.**

-----

## **How to Deduce a Relaxation.**

**How to test machine M to see if it relaxes the architecture A associated with test T?**

- 1. Assume that M obeys A.**
- 2. Run Test T.**
- 3. Analyze the output data.**  
Use the rules to attempt to show that a circuit of events existed in time.
- 4. If found, conclude that M does not obey A.**

**This is a *reductio ad absurdum* argument:**

- 1. Make an assumption.**
  - 2. Deduce a falsehood.**
  - 3. Conclude that the assumption was false.**
- 

## **A Simple Example.**

**Initially,  $A = B = U = V = 0$ .**

	<b>P1</b>			<b>P2</b>	
<b>L00:</b>	<b>A</b>	<b>= 1;</b>		<b>L00:</b>	<b>B = 1;</b>
<b>L01:</b>	<b>-</b>	<b>= A;</b>		<b>L01:</b>	<b>- = B;</b>
<b>L02:</b>	<b>U</b>	<b>= B;</b>		<b>L02:</b>	<b>V = A;</b>

**Terminally,  $A = B = 1, U = V = 0$ .**

```

(P1,L00,W,1,A,S1) <upo (P1,L01,R,1,A,S1)
                   <rr  (P1,L02,R,0,B,S1)
                   <crw (P2,L00,W,1,B,S1)
                   =cc1 (P2,L00,W,1,B,S2)
                   <upo (P2,L01,R,1,B,S2)
                   <rr  (P2,L02,R,0,A,S2)
                   <crw (P1,L00,W,1,A,S2)
                   =cc1 (P1,L00,W,1,A,S1)

```

---

## Explanation of the Circuit.

```

(P1,L00,W,1,A,S1) <upo (P1,L01,R,1,A,S1)

```

**By uniprocessor order the write into A in S1 in statement L00 occurred before the read from A in S1 in statement L01.**

```

(P2,L01,R,1,A,S1) <rr  (P2,L02,R,0,B,S1)

```

**By read order the read event in statement L01 occurred before the read event in statement L02.**

```

(P1,L02,R,0,B,S1) <crw (P2,L00,W,1,B,S1)

```

**Obvious.**

```

(P2,L00,W,1,B,S1) =cc1 (P2,L00,W,1,B,S2)

```

**By write atomicity the write into B in S1 occurred at the same instant as the write into B in S2.**

**The second half of the circuit repeats the reasoning for the first half.**

**The existence of the circuit shows that the machine failed to**

obey A(CMP,UPO,RR,CC1).

---

## Test T700. Seek a Relaxation of A(CMP,UPO,RR,CC1).

Initially,  $A = B = U[i] = V[i] = 0$ ,  $i = 0$  to  $K$ .

P1		P2	
L00:	A = 0;	L00:	B = 0;
L01:	- = A;	L01:	- = B;
L02:	U[0] = B;	L02:	V[0] = A;
L10:	A = 1;	L10:	B = 1;
L11:	- = A;	L11:	- = B;
L12:	U[1] = B;	L12:	V[1] = A;
L20:	A = 2;	L20:	B = 2;
L21:	- = A;	L21:	- = B;
L22:	U[2] = B;	L22:	V[2] = A;
L30:	A = 3;	L30:	B = 3;
L31:	- = A;	L31:	- = B;
L32:	U[3] = B;	L32:	V[3] = A; etc.

Seek 7.1.  $U[i] < j$  and  $V[j] < i$ .  $d1 = V[ U[i]+1 ] -$   
Seek 7.2.  $V[i] < j$  and  $U[j] < i$ .  $d2 = U[ V[i]+1 ] -$

To show: Not A(CMP,UPO,RR,CC1).

If a circuit is found, then  $d$  measures the length of the circuit.

Test T7 fails on many machines. The reason is that fetches are made from the store buffer, and the store buffer is not subjected to the same MESI discipline as the cache.

---

## **Results of Running ARCHTEST.**

**Of 26 machines tested so far, 6 were sequentially consistent (SC).**

**Eight machines relaxed the rule of WR, that is, they allowed reads to occur before logically preceding writes. Call these machines POK machines.**

**Twelve machines relaxed both WR and CC1. This has been called processor consistency (Pcon).**

**Details at: [www.mpdiag.com/results.html](http://www.mpdiag.com/results.html).**

=====

## **Work in Progress.**

### **Shared Files as Sink Operands.**

**Instead of writing into operands in shared memory, write into shared files.**

**Goal: Subject both software and hardware to the same tests.**

-----

## **Test Real Shared Memory Programs on Real SMP Systems.**

**What programs access shared memory?**

**Ans: Critical section routines.**

**(Are there others? Probably very, very few.)**

**Most critical section routines use atomic instructions but there are several in the literature which do not:**

- 1. Various notes in the CACM about 1965 and after.**
- 2. *Algorithms for Mutual Exclusion* by M. Raynal, MIT Press, 1986.**

**Hope to put this work on the web with a copyleft statement.**

**Question: can critsec routines run correctly on a PCon machine?**

-----

## **Generate Relaxed Data (I).**

**Problem: how to ensure that ARCHTEST correctly interprets the data generated by a machine?**

**It is easy to generate data that appears to come from a SC machine. ARCHTEST correctly interprets such data.**

**How to generate data that appears to come from a non-SC machine?**

**At present ARCHTEST generates valid data and then modifies it slightly at random points. Hand-checking shows**

that ARCHTEST correctly picks out the errors.

-----

## **Generate Relaxed Data (II).**

**Proposal: write a program to generate data for each test in ARCHTEST as if it came from a machine obeying any possible architecture (within the confines of the rules described above). Put it on the web with a copyleft statement.**

**Useful for testing ARCHTEST and possibly to others.**

-----

## **Program to Prove $CC1 \Leftrightarrow CC2$ .**

**In RAPA I proved that  $CC1 \Leftrightarrow CC2$ , that is, any machine that is  $CC2$  also appears to be  $CC1$ .**

**One important lemma necessary for the proof had 224 subcases. I wrote a program in Pascal to generate and to check each subcase.**

**I will rewrite this program in C and make it available with a copyleft statement.**