A Unified Memory Model Proposal for Java

Jeremy Manson, William Pugh and Sarita Adve

March 17, 2004, 3:50pm

1 Introduction

We have developed a new memory model that combines many features of the earlier M/P model and the SC- model. It seems to be simpler than either, possess the properties that we know are required, and also forbid causality tests cases 5 and 10, which were deemed undesirable.

`http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html` is where test cases can be found.

We give a brief informal description of the model in Section 2. We are considering several different ways of describing the formalism that we believe to be provably equivalent; at the moment our focus is on whether this model has the important properties we need rather than the best way to present it.

In Section 3, we describe test cases U1-U7 that show some of the behaviors allowed and prohibited by this model. Some authors were surprised and unhappy to find that the behaviors of U1-4 were prohibited, and that the behavior of U7 is allowed (different authors were unhappy about different test cases). On further investigation, we found that both the M/P model and the stronger version of the SC- model (which forbids #5 and #10) forbid the behaviors of U1, U2 and U4 as well.

Our current belief is that the advantages of a clear and simple model clearly outweigh, for the moment, any potential unknown disadvantages from accepting the behaviors of these test cases. Certainly, these are cases that could be addressed in later revisions of the JMM if necessary.

2 Informal Description of Model

An execution consists of a set of actions observable at the processor/shared memory boundary, such as a read, a write, a lock or unlock. For each thread, we have a **program order** that is a total order over actions performed by that thread.

Each execution has a **synchronization order**, which is a total order over all synchronization actions in the execution. For each thread t, the synchronization order of synchronization actions performed by t is consistent with the program order of action performed by t. There is a **happens-before** relation $\stackrel{hb}{\rightarrow}$ defined on actions $i \stackrel{hb}{\rightarrow} j$ if i is before j in program order, if i is an unlock or volatile write and j is a matching lock or volatile read that comes after it in the total order over synchronization actions, or if $i \stackrel{hb}{\rightarrow} k \stackrel{hb}{\rightarrow} j$ for some k.

A read r is **allowed** to see a write w to the same variable v if r does not happen-before w and if there is no other write w' to v such that $w \stackrel{hb}{\rightarrow} w' \stackrel{hb}{\rightarrow} r$. A volatile read always sees the result of the most recent volatile write of the same variable in the synchronization order.

In all executions, reads can only see writes that they are allowed to see and the actions must respect intra-thread semantics.

We justify/build/commit an execution in justification/commitment order. At each step, we have a sequence α of **committed** actions that is a prefix of the justification order. Note that the justification order does *not* have to respect happens-before ordering.

We start with an empty sequence of committed actions.

To commit additional actions, we must demonstrate an execution E that has a justification order consisting of $\alpha\beta$, where each read in β sees a write that happens-before it.

Given this, we can now commit any or all actions in β. For example, if both x and y are contained in β, we can commit both of them, giving αxy as our new committed prefix.

Let N be a set of non-redundant conflicting release/acquire action pairs in E. A set N is non-redundant if the transitive closure of the happens-before edges induced by N and the program order edges in E gives all of the happens-before edges in E. If we commit an action a such that an action s in N happens-before a in E, we must also commit s (if it is not already committed).

To commit a read r, it must see a write w in α. If there is another write w' in α such that it would be happens-before consistent for r to see w', then we can instead commit r', where r' is the same as r except that it sees w' rather than w.

Actions occurring in more than one execution

We have assumed that we can freely talk about a sequence of actions in one execution occurring in another execution. If you want to get very deep into it, here is a more formal explanation of what that entails.

Given two sequences of actions, α and α', we say that $\alpha \equiv \alpha'$ if for all i, α_i is tagged with all the same information as α'_i (they are both performed by the same thread, both the same kind of action, access the same variables, read/write the same value). Furthermore, for all i and j, $\alpha_i \stackrel{hb}{\rightarrow} \alpha_j$ if and only if $\alpha'_i \stackrel{hb}{\rightarrow} \alpha'_j$, and $\alpha_i \stackrel{so}{\rightarrow} \alpha_j$ if and only if $\alpha'_i \stackrel{so}{\rightarrow} \alpha'_j$ (where $\stackrel{so}{\rightarrow}$ denotes synchronization order).

Then, if we have $E = \alpha\beta x\gamma$, where α is committed and we justify committing x next, then an execution is said to start with the justification order prefix αx if it starts with a justification order prefix α' such that $\alpha x \equiv \alpha'$.

Initially, x = 1 and y = 0

Thread 1	Thread 2
`r1 = x`	`r2 = y`
`y = r1`	`if r2 > 0`
	`  x = 2`

Behavior in question: r1 == r2 == 2

Figure 1: Test case U1 – Simple case of behavior prohibited by new model

3 Additional test cases forbidden by the model

The new model forbids some test cases that we don't have strong reasons for forbidding. These cases have something of the flavor of test cases #5 and #10, but do not seem as undesirable.

The fact that the behaviors in U1-U4 are forbidden was initially surprising. However, on further investigation, we found that both the M/P model and the stronger version of the SC- model (which forbids #5 and #10) forbid these behaviors as well.

Test case U1 (Figure 1) is probably the simplest example of a behavior that is disallowed. Test case U2 (Figure 2) is the example introduced by Vijay as test case 14a. Test case U3 (Figure 3) is a hybrid between 14 and 14a; it uses a while loop (like case 14), but `y` isn't volatile (as in test 14a). Test case U4 (Figure 4) shows an example of behavior that is not allowed. If threads 1-3 are inlined in U4, we get U5 (Figure 5), in which the behavior is possible. This shows that thread inlining is not always legal.

Test case U6 (Figure 6) is very similar to cases #5 and #10. Unlike those cases, if there is a write of 1 to x and y, then there is a happens-before relationship between the initial volatile write by Thread 3 and the performance of the actions in Threads 1 and 2. It is similar enough to these cases that it is reasonably clear why we disallow it. It is presented here mostly as an example of why we force non-redundant synchronization actions to be committed. If we didn't have this requirement, then it would be possible to commit a write of 1 to x in Thread 2, and then change the happens-before order so that the write in Thread 4 never occurred. With this requirement, the happens-before relationships among synchronization actions have to remain the same.

4 Additional test cases allowed by the model

The new model allows some test cases that we don't have strong reasons for allowing. These cases also have something of the flavor of test cases #5 and #10.

Test case U7 (Figure 7) is similar to U6, but there are only synchronization edges between the initial writes by Threads 1 and 2 and the actions in Thread 3. Because of the lack of synchronization edges to enforce ordering, the write to z in Thread 3 can be in a self-justifying causal loop with the actions in Thread 4, and x can still see the value 1.

Acknowledgments

Many people have made substantial contributions to this work, too many to name. These contributors include many of the people participating in the Java memory model mailing list, the JSR-133 expert group and those who were involved in the JSR-133 review process. Doug Lea deserves special acknowledgment for significant contributions. Thanks to all.

Initially, a = b = y = 0

Thread 1	Thread 2
`r1 = a;`	`r2 = y;`
`if (r1 == 0)`	`r3 = b;`
`  y = 1;`	`if (r2 + r3 != 0)`
`else`	`    a = 1;`
`  b = 1;`	

Behavior in question: r1 == r3 == 1; r2 == 0

Figure 2: Test case U2 – Same as Vijay's 14a; prohibited by new model

Initially, a = b = y = 0

Thread 1	Thread 2
`r1 = a;`	`do {`
`if (r1 == 0)`	`  r2 = y;`
`  y = 1;`	`  r3 = b;`
`else`	`  } while (r2 + r3 == 0);`
`  b = 1;`	`a = 1;`

Behavior in question: r1 == r3 == 1; r2 == 0

Figure 3: Test case U3 – Same as case 14, except that y is not volatile; behavior prohibited by new model. Note that the difference with U2 is largely the lack of the loop

Initially, a = b = c = d = 0

Thread 1	Thread 2	Thread 3	Thread 4
			`r4 = d;`
`r1 = a;`	`r2 = b;`	`r3 = c;`	`if (r4 == 1)  {`
`if (r1 == 0)`	`if (r2 == 1)`	`if (r3 == 1)`	`  c = 1;`
`  b = 1;`	`  c = 1;`	`  d = 1;`	`  a = 1;`
			`  }`

Behavior in question: r1 == r3 == r4 == 1; r2 == 0

Figure 4: Test case U4 – behavior prohibited by new model

Initially, a = b = c = d = 0

Thread 1/2/3	Thread 4
`r1 = a;`	
`if (r1 == 0)`	
`  b = 1;`	`r4 = d;`
`r2 = b;`	`if (r4 == 1)  {`
`if (r2 == 1)`	`  c = 1;`
`  c = 1;`	`  a = 1;`
`r3 = c;`	`  }`
`if (r3 == 1)`	
`  d = 1;`	

Behavior in question: r1 == r3 == r4 == 1; r2 == 0

Figure 5: Test case U5 – result of thread inlining of U4; behavior allowed by new model

Initially, x = y = z = 0, z is volatile

Thread 1	Thread 2	Thread 3	Thread 4
			`r3 = z`
`join Thread 4`	`join Thread 4`		`if (r3 == 1) {`
`r1 = x`	`r2 = y`	`z = 1`	`x = 1`
`y = r1`	`x = r2`		`y = 1`
			`}`

Behavior in question:r1 = r2 = 1, r3 = 0

Figure 6: Test case U6 – behavior disallowed by new model

Initially, x = y = z = 0

Thread 1	Thread 2	Thread 3	Thread 4
		`join thread 1`	
		`join thread 2`	
`x = 1`	`x = 2`	`r1 = x`	`r3 = z`
		`y = r1`	`y = r3`
		`r2 = y`	
		`z = r2`	

Behavior in question: r1 == 1; r2 == r3 == 2

Figure 7: Test case U7 – behavior allowed by new model