

# Analyzing the CRF Java Memory Model

Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom  
School of Computing, University of Utah  
{yyang | ganesh | gary}@cs.utah.edu

## Abstract

*The current Java Memory Model [1] is flawed and has many unintended implications [2]. It is a very challenging task to design or understand a memory model at the programming language level. As multithreaded programming becomes increasingly popular in Java and hardware memory architectures become more aggressively parallel, it is of significant importance to provide a framework for formally analyzing the Java Memory Model.*

*The Mur $\varphi$  verification system is applied to study the Commit/Reconcile/Fence (CRF) memory model [3], one of the proposed thread semantics to replace the present Java Memory Model. The CRF proposal is formally specified using the Mur $\varphi$  description language. A suite of test programs is designed to reveal interesting properties of the model. The results demonstrate the feasibility of applying model checking techniques to language level memory model specifications. Not only can it help the designers to debug their designs, it also provides a formal mechanism for Java programmers to understand the subtleties of the Java Memory Model.*

## 1 Introduction

Java is the first popular programming language that integrates thread support at the language level. As a type-safe and robust language, Java needs a solid memory model to specify how threads interact through shared memory. Unfortunately, the current Java Memory Model as given in Chapter 17 of the Java Language Specification [1], has many flaws. The present model has been summarized by Pugh [2] as both being too weak and too strong. It is too weak because safety guarantees can be violated and no semantics are given for final fields. On the other hand, the constraints of the current Java Memory Model are too strong in the sense they prohibit common compiler optimization techniques and introduce many unintended implications.

To fix the problems, the Java Memory Model is currently under review [4] and will be replaced in the future. Two replacement semantics for Java threads have been proposed. Manson and Pugh [5] use a notation based on multimap to update and record histories of Java memory operations. Maessen, Arvind, and Shen [3] use the Commit/Reconcile/Fence (CRF) [6] framework to provide an operational semantics. Although notations are very different, the two proposals have similar design goals. They both present a general guideline and a detailed specification. However, neither of them has applied formal analysis using model checking to ensure that the low level semantics actually achieve the projected guidelines.

While formal verification techniques have achieved some success in verifying architecture level shared memory systems and cache coherence protocols, a language level memory model has not been formally analyzed to date. In this paper, we present a formal framework for analyzing the CRF Java Memory Model. We choose the CRF notation because its precise rewrite rules can be naturally transformed into more rigorous mathematical rules. Our method is based on the *executable specification* idea developed by Dill, Park, and Nowatzky [7]. The CRF model is formally specified in *Mur $\varphi$*  [8], a description language as well as a verification system. This forms the verification engine. A suite of test cases is executed with this engine to reveal pivotal properties of the model. This mechanism enables one to exhaustively exercise a language level memory model on a particular program. Our *Mur $\varphi$*  verification program for the CRF Java Memory Model is available at [9].

In the next section, we review the related work in memory model verifications. It is followed by a discussion of some of the problems of the current Java Memory Model. The CRF model, upon which our executable model is based, is described in Section 4. Section 5 outlines how to formally specify the memory model and construct test programs. In Section 6, our preliminary results are presented. We conclude by discussing the limitations of our approach and proposing

some areas of future work that could help advance the software model checking techniques.

## 2 Related Work

A *memory model* describes how a memory system behaves on memory operations such as reads and writes. Much previous research has concentrated on the processor level memory models. One of the strongest memory models for multiprocessor memory systems is *Sequential Consistency* [10]. Many weaker memory models [11] have been proposed to enable optimizations. One of them is *Coherence* [12]. Informally, *Coherence* requires all memory operations at a given memory location to exhibit a total order that respects the program order of memory operations at that location in each processor. Another weak memory model, *PRAM* [13], requires the execution sequences containing all instructions from a given processor and all write instructions from all other processors to exhibit a total order which also respects the program orders in each processor. With our verification system, we have formally compared the CRF model with some of these conventional models.

Several model checking tools, such as VeriSoft [14], Java PathFinder [15] [16], JCAT [17], Bandera [18], and the SAL Model Checker [19], have been developed for verifying multithreaded Java programs. These tools, however, are intended for real Java applications and they do not specifically address Java Memory Model issues. The memory operations in these applications are interpreted using intuitive behaviors instead of a strict memory model.

Formal verification techniques have also been applied for verifying processor level memory design protocols. The idea of *executable specification* of abstract memory models is proposed in [7]. Using this methodology, one can specify processor memory models using Mur $\phi$ . Such a description provides a precise specification of the machine architecture, both for implementors and programmers. The model can be executed with test programs to verify certain properties. Although this approach has been applied to some processor architectures such as the Relaxed Memory Order (RMO [20]) [21], it has not been explored for software level memory models which have to deal with more complicated issues. In our work, we have followed similar techniques but applied them to the domain of Java Memory Model, with encouraging initial results.

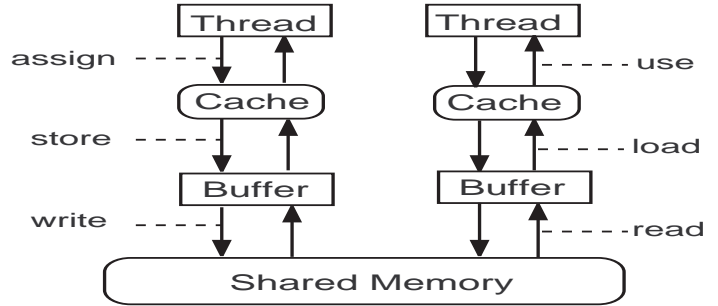


Figure 1. Memory hierarchy in the current Java Memory Model

Initially, p = null

Thread 1	Thread 2
<code>Synchronized(this) {                p = new Point(1,2);          }</code>	<code>if(p != null) {                a = p.x;          }</code>

Finally,  
can result in a = 0

Figure 2. Premature release of object reference

## 3 Problems with the Existing Java Memory Model

The current Java Memory Model [1] uses a memory hierarchy illustrated in Figure 1. It defines possible actions that may occur at each layer. Besides the actions shown in Figure 1, two additional actions, *lock* and *unlock*, are cooperatively used by both the threads and the memory system to achieve synchronization. The specification further outlines sets of rules to impose constraints to the actions.

There are several shortcomings of this specification. The style of the constraints is to prescribe the program behaviors, which is very hard to be transformed to formal notations. In addition, the multiple layers used between threads and memory introduce unnecessary complexities. As a result, the model has some hidden flaws and is very hard to understand.

Under this model, an object reference is allowed to escape prematurely during the object construction if there exists a race condition. Take figure 2 as an example, when Thread 2 fetches the object field without locking, it might obtain uninitialized data in the statement `a = p.x`. Consequently, final fields are not truly final and immutable objects are not truly immutable since other threads might see their default values.

Because of this loophole, some popular synchronization idioms are unsafe. For example, the *double-checked locking* [22] algorithm is often used for creating a Singleton object, which is only constructed once. As shown in Figure 3, this programming pattern only uses locking when the object is created to avoid unnecessary synchronization overhead. Unfortunately, because the assignment to `helper` may be visible to other threads before the constructor completes, this algorithm is broken under the existing Java Memory Model.

Another problem of the current model is that it prohibits important compiler optimization techniques such as fetch elimination. In [23], a proof is given to show that the Java Memory Model requires *Coherence*. Consider figure 4, the instructions in Thread 1 eventually operate on the same variable due to aliasing. Therefore, the statement `k = p.x` can not be replaced by `k = i` by the compiler because *Coherence* requires a total order for operations on the same variable. Many existing JVM implementations violate this requirement.

Compared with a processor level memory model, the Java Memory Model has many new challenges. First, it must maintain safety guarantees which can not be compromised even under race conditions. Second, the Java Memory Model is mapped to many different processor architectures, some of which have very weak memory models. For the Java Memory Model designers, it is a nontrivial task to identify the optimal requirement that can be supported by all major architectures in a safe and efficient manner. For the JVM implementers, the compliance with the Java Memory Model need to be ensured. Third, the Java Memory Model needs to address the semantics of many programming language level constructs, such as final fields, volatile fields, constructors, and finalizers. Finally, run-time effects such as aliasing can introduce more implications of the language level specification. Because of these issues, a mechanism for formally reasoning about the

```

class foo {
    private static Helper helper = null;
    public static Helper get() {
        if (helper == null) {
            synchronized(this) {
                if(helper == null)
                    helper = new Helper();
            }
        }
        return helper;
    }
}

```

Figure 3. Double-Checked Locking Idiom

Initially, p.x = 0

Thread 1	Thread 2
i = p.x;	p.x = 1;
j = q.x;	q = p;
k = p.x;	

Figure 4. Coherence prohibits fetch elimination

Java Memory Model would be highly valuable.

## 4 The CRF Java Memory Model

The CRF framework is designed to describe instruction reordering and data replication (caching) in a memory model. It captures the precise semantics of memory operations with or without race conditions. The CRF memory hierarchy has three layers, as shown in Figure 5. The elimination of the buffer layer in Figure 1 simplifies the design. Each thread has a local semantic cache. Each cache entry maintains a tag with a state of either Clean, Dirty, Locked, or Frozen. We present the outline of the CRF Java Memory Model in this section. The full definition can be found in [3].

### 4.1 CRF Instructions

To provide enough fine granularity to specify Java memory activities, eight CRF instructions are used. These CRF instructions are summarized as follows:

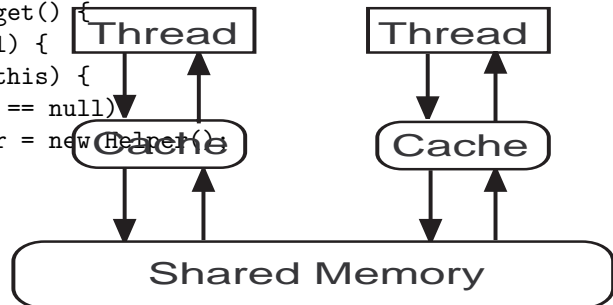


Figure 5. Memory hierarchy in the CRF model

Operation	Regular	Final	Volatile
Store $a, v$ ;	Storel $a, v$ ; Commit $a$ ;	Storel $a, v$ ; Commit $a$ ; Freeze $a$ ;	Fence <sub>rw</sub> $*V_R, a$ ; Fence <sub>ww</sub> $*V_R, a$ ; Storel $a, v$ ; Commit $a$ ;
$v = \text{Load } a$ ;	Reconcile $a$ ; $v = \text{Loadl } a$ ;	Reconcile $a$ ; $v = \text{Loadl } a$ ; Freeze $a$ ;	Fence <sub>wr</sub> $*V, a$ ; Reconcile $a$ ; $v = \text{Loadl } a$ ; Fence <sub>rr</sub> $a, *V_R$ ; Fence <sub>wr</sub> $a, *R$ ;

**Table 1. CRF translations for Java loads and stores**

Operation	Translation
Enter $l$ ;	Fence <sub>ww</sub> $*L, l$ ; Lock $l$ ; Fence <sub>wr</sub> $l, *V_R$ ; Fence <sub>ww</sub> $*V_R, l$ ;
Exit $l$ ;	Fence <sub>ww</sub> $*V_R, l$ ; Fence <sub>rw</sub> $*V_R, l$ ; Unlock $l$ ;
EndCon;	Fence <sub>ww</sub> $*, *V_R$ ;

**Table 2. CRF translations for Java synchronization operations**

CRF Instruction	Description
Loadl	Load value from local cache
Storel	Store value to local cache
Commit	Ensure write back from cache to memory
Reconcile	Ensure update from memory to cache
Fence	Ensure ordering restrictions
Lock	Acquire a lock
Unlock	Release a lock
Freeze	Complete a final field operation

reordering rules and rewrite rules. The former dictates how the CRF instructions can be reordered locally and the latter defines the effects of memory activities.

A reordering table [3, Figure 6] is provided to specify ordering rules between any CRF instructions. It is designed to allow maximum reordering flexibility. For example, memory access instructions can be reordered if they access different addresses or if they are both Loadl instructions. Memory rendezvous instructions (Commit and Reconcile) can always be swapped. Memory fence instructions can always be swapped. All locking operations can be ordered freely unless Fence instructions are applied. As long as the reordering table is followed, any instructions of a given thread can be arbitrarily reordered.

In addition to the ordering table, a set of rewrite rules [3, Figure 4 and Figure 5] is specified. These rewrite rules include *local rules*, which define the effects of executing the CRF instructions, and *background rules*, which maintain the synchronization between cache and shared memory.

## 5 Modeling the CRF Memory Model in Mur $\varphi$

To better understand the CRF model, we have developed a formal executable Java Memory Model in Mur $\varphi$  based on the CRF specification. Our verification system consists of two parts. The first part is the formal specification of the CRF Java Memory Model using Mur $\varphi$  rules, which provides a “black box” that defines the thread semantics. The second part comprises a suite of test cases. Each test program defines the corresponding Mur $\varphi$  initial state and invariants.

### 4.2 Translating Java into CRF

Java memory operations are categorized as four groups depending on the type of variables they act upon: *regular*, *volatile*, *final*, and *monitor*. These operations are translated to the fine-grained CRF instructions according to Table 1 and Table 2. For example, the source level Store instruction for a regular variable is decomposed into a Storel (store-local) followed by a Commit.

The fence operation takes two parameters as the pre-address and post-address and specifies an ordering constraint between them. It can also use a wildcard  $*$  to denote all addresses. For example, Fence<sub>rw</sub>  $*V_R, a$ ; requires that all previous Loadl instructions on all *volatile* and *regular* fields must be completed before allowing Storel to a.

The EndCon instruction indicates the completion of a constructor. It is used to solve the reference escape problem, which will be further discussed in Section 6.2.

### 4.3 CRF Reordering Rules and Rewrite Rules

In the CRF model, the semantics of the Java Memory Model is specified collectively from two aspects: the

## 5.1 Data Structures

The global state of our system includes information about all threads and the shared memory. A variable can be declared as one of the following: `REGULAR`, `VOLATILE`, `FINAL`, or `LOCK`, which is referred to as the variable type in this paper. Each thread maintains its own local cache and an instruction buffer. There is a tag associated with each cache entry, which can be `UNMAPPED`, `CLEAN`, `DIRTY`, `FROZEN`, or `LOCKED`. The instruction buffer records all CRF instructions in program order. Each instruction uses a completion flag to indicate whether it has been completed. It also maintains a result field to store the value obtained from a read operation. The scale of our program can be adjusted for different tests. For example, key parameters can be declared as follows:

```
THREAD_NUM:    2;    -- maximum number of threads
ADDRESS_NUM:   2;    -- maximum number of different addresses
VALUE_NUM:     2;    -- data range
LABEL_NUM:     6;    -- maximum number of instructions
LOCK_NUM:      1;    -- maximum number of locks
```

## 5.2 Formal Specification of the Java Memory Model

Our `Mur $\phi$`  program models all the interleaving behaviors of a set of running threads. These threads can be executed concurrently on multiple processors. The nature of the processor architecture is abstracted out and thread semantics is only governed by the high level memory model.

The CRF model uses rewrite rules to describe the effect of completing any CRF instruction or background action. These rewrite rules are implemented using `Mur $\phi$`  rules. For the local rules, an eligible instruction from any thread can be nondeterministically chosen and executed. A background rule may also be fired when its condition is satisfied. The `Mur $\phi$`  verification system automatically enumerates the state space and examines all the possible interleaving between these rules.

In addition to the rewrite rules, the CRF ordering constraints are also specified. A `Mur $\phi$`  function `Ordered(t: Thread; i, j: InstructionIndex)` is implemented following the CRF reordering table to check if there exists any ordering restriction between instruction `i` and `j`. It is used as one of the conditions for local rules to ensure an instruction can be executed. An instruction is only eligible to be selected when it can bypass all previous pending instructions.

Although the process of transforming CRF to `Mur $\phi$`  was relatively straightforward, implementing the operational semantics with strict mathematical rules forced

us to think harder about the CRF rules, which in turn helped us clarify many details.

## 5.3 Constructing Test Programs

A test case consists of small concurrent programs comprised of two or more threads. The programs are abstracted to their memory operations. They accept source level instructions including `Load`, `Store`, `EnterMonitor`, `ExitMonitor`, and `EndCon`. The type of each variable is initialized in `Mur $\phi$`  startstate. A procedure `AddInstruction` is used to translate the high level instructions into the fine-grained CRF instructions, which are later added to the instruction buffer of the corresponding thread.

Our system can detect deadlocks and invariant violations. To examine test results, two techniques can be applied. The first one uses `Mur $\phi$`  invariants to specify that a particular scenario can never occur. If it does occur, a violation trace can be generated to help understand the cause. The second technique uses a special “thread completion” rule, which is triggered only when all threads are completed, to output all possible final results. We use a configuration flag `SHOW_ALL_RESULTS` to select this output mode.

Another configuration flag `ALIASING` is used to indicate if the variables might be aliased. This enables one to easily experiment with both scenarios. In the CRF reordering table, some ordering rules depend on if the two addresses are the same and every single reordering decision may be made dynamically. Since static analysis at the language level does not have discrimination power comparable to the architecture level where an address is always concrete, aliasing analysis could be hard and sometimes people have to make conservative assumptions. As a verification tool for high level reasoning, our system relies on the users to do the aliasing analysis themselves but enables them to know what to expect in either case.

A third flag `DATA_DEPENDENCE` is used to specify an additional ordering restriction during object dereference. When accessing an object field, the object reference must be obtained before fetching the content of the field. Since we model the reference and field as separate global variables, we turn on the flag `DATA_DEPENDENCE` when simulating the scenario of object dereference. We add the following restrictions to the original CRF reordering table when `DATA_DEPENDENCE` is set: the CRF instruction `Loadl a` can not be bypassed by `Reconcile b` or `Storel b` if `a` and `b` has the reference/field type of data dependency.

Initially,  $A = 0$

Thread 1	Thread 2
$A = 1;$	$X = A;$
$A = 2;$	$Y = A;$

Finally,  
can it result in  $X = 2 \ \& \ Y = 1$ ?

**Figure 6. Test program revealing violation of *Coherence***

Initially,  $A = B = 0$

Thread 1	Thread 2
$A = 1;$	$X = B;$
$B = 1;$	$Y = A;$

Finally,  
can it result in  $X = 1 \ \& \ Y = 0$ ?

**Figure 7. Test program revealing violation of *PRAM***

## 6 Analysis of the CRF Model

By exercising our Java Memory Model engine with test programs, we are able to gain insights about the CRF memory model. To demonstrate the feasibility and usability of this approach, we present our preliminary study of the ordering rules, the constructor properties, and synchronization idioms.

### 6.1 Ordering Rules

One can better understand a memory model by comparing it to other well known memory models. We have designed some test programs to reveal ordering properties of the CRF Java Memory Model.

#### 6.1.1 Comparison with *Coherence*

Figure 6 illustrates a test program that reveals violation of *Coherence*. When we declare  $A$  as a regular variable and exhaustively execute this program with our memory model engine, a trace leading to the contradicting final result is discovered. The violation is due to the fact that a `Loadl` instruction can be freely reordered with another `Loadl` instruction, even if they use the same address. Therefore, this simple test program formally proves that the CRF specification does not enforce *Coherence*. Since *Coherence* is strictly weaker than *Sequential Consistency*, it can be concluded immediately that the CRF Java Memory Model does not enforce *Sequential Consistency* either.

#### 6.1.2 Comparison with *PRAM*

Figure 7 is a program to test *PRAM* ordering properties. Our verification system reveals that the contradicting result exists for regular variables. A study of the violation trace indicates it is also because the `Loadl` instructions can be reordered. The result proves that *PRAM* is not obeyed by the CRF Java Memory Model.

Initially,  $A = B = 0$

Thread 1	Thread 2
$X = A;$	$Y = B;$
$B = 1;$	$A = 1;$

Finally,  
can it result in  $X = 1 \ \& \ Y = 1$ ?

**Figure 8. Test program for prescient store**

#### 6.1.3 Prescient Store

In some circumstances, the CRF model can relax the read/write ordering such that a write may be performed before a read completes. Figure 8 is used to study such effects. The results turn out to depend on the setting of `ALIASING`. It can be concluded that the CRF model only allows the prescient store optimization if  $A$  and  $B$  can never be aliased.

#### 6.1.4 Write Atomicity

Figure 9 verifies write atomicity, which has passed successfully. Because the CRF model uses the shared memory as the rendezvous between threads and caches, it is impossible to have  $X = 2$  and  $Y = 1$  in Figure 9. This example shows a main difference between the CRF model and some of the more relaxed memory models. For example, Pugh's model [5] would allow  $X = 2$  and  $Y = 1$ .

Initially,  $A = B = 0$

Thread 1	Thread 2
$A = 1;$	$A = 2;$
$X = A;$	$Y = A;$

Finally,  
can it result in  $X = 2 \ \& \ Y = 1$ ?

**Figure 9. Test program verifying write atomicity**

Initially,  $A = B = 0$

Thread 1	Thread 2	Thread 3
$A = 1;$	$X = A;$ $B = 1;$	$Y = B$ $Z = A$

Finally,  
can it result in  $X = 1 \ \& \ Y = 1? \ \& \ Z = 0?$

**Figure 10. Test program for Causality**

Initially,  $A = B = 0$

Thread 1	Thread 2
$B = 1;$ <i>EndCon</i> ; $A = 1;$	$X = A;$ $Y = B;$

Invariant:  
it should never result in  $X = 1 \ \& \ Y = 0$

**Figure 11. Test program for *EndCon***

### 6.1.5 Causality

Causality requires thread-local orderings to be transitive, which might be expected by programmers intuitively. Consider Figure 10, if A and B are aliased at run time, the CRF model prohibits the reordering between the two instructions in Thread 2. If  $X = 1$  and  $Y = 1$ , Z must return 1 if Thread 3 respects the ordering of Thread 2. But our verification system reveals that even when ALIASING is set, the program can still result in  $X = 1$ ,  $Y = 1$  and  $Z = 0$ . Therefore, the CRF Java Memory Model does not enforce causality.

The above examples only demonstrated tests for *regular* variables. By simply changing the type declaration of the variables, the same approach can be applied to analyze properties of different variable types, such as *final* and *volatile*.

## 6.2 Constructor Properties

One of the problems of the current Java Memory Model is the premature release of an object reference in the constructor. To prevent such an early release, the *EndCon* instruction is introduced in CRF to denote the completion of a constructor.

Figure 11 shows the test program we designed to study the effectiveness of this approach. We use A to represent the object reference and B to represent the object field. Thread 1 simulates the essential activities in the constructor. The reference is assigned only after the field is initialized and *EndCon* is issued. Thread 2 simulates a race situation by attempting to access the object field without synchronization. The invariant ensures the premature release of the object reference can never occur.

There is no violation found by this test. However, several subtle issues have been discovered:

- If the flag DATA\_DEPENDENCE is not turned on, the test would fail. This implies the ordering restriction due to data dependence between the reference and field is absolutely required. Although seemingly a reasonable assumption, this constraint is not enforced by all architectures. Under a very aggressive memory model, the content pointed by a reference might come out of a stale cache line. In effect, this breaks the data dependence restriction. Therefore, we propose to strengthen the CRF reordering table by explicitly adding the constraints due to this kind of data dependency, as mentioned in Section 5.3.
- If we move the instruction  $A = 1$  ahead of the *EndCon* instruction, the test would fail. Therefore, the *EndCon* operation must be inserted between initializing the object field and assigning the object reference. In the original CRF specification, it is not clearly stated when the *EndCon* instruction should be issued.
- If the above items are taken care of, the test succeeds regardless of the variable types. The same level of restriction is imposed to both final and non-final fields. To make sure the requirement of *EndCon* is correctly implemented in an actual JVM implementation, additional memory barriers are required on some architectures with aggressive memory models, which will result in a performance penalty. Pugh’s proposal [5] is different in this regard as it only impose safety constraints to *final* fields.

## 6.3 Synchronization Idioms

As multithreaded programming becomes increasingly popular, many synchronization idioms have been established. However, an algorithm working under one memory model is not necessarily safe to use under another memory model. As previously illustrated, the *Double-Checked Locking* idiom is a good example.

Our verification system can be used for testing the validity of a synchronization idiom. The idiom needs to be abstracted to a test program. Violations can be detected by any contradicting results. To demonstrate this strategy, we abstract the *Double-Checked Locking* algorithm in Figure 3 to a test program shown in Figure 12. Again, A represents the object reference and B represents the object field. We use Thread 1 to simulate the thread that grabs the lock and creates the Helper class. After the thread enters the monitor, it

Initially,  $A = B = 0$

Thread 1	Thread 2
EnterMonitor; $X = A$ ; $B = 1$ ; EndCon; $A = 1$ ; ExitMonitor;	$Y = A$ ; $Z = B$ ;

Invariant:

it should never result in  $X = 0 \ \& \ Y = 1 \ \& \ Z = 0$

**Figure 12. Test for *Double-Checked Locking* idiom**

first double-checks if the reference is still null. Then it completes the constructor before exiting the monitor. Thread 2 simulates another thread that obtains a non-null helper by calling the get method. When it reads the object field, it should never see uninitialized content.

This test has passed successfully. Although this is not a complete proof of validity, it does help Java programmers to gain some level of confidence in the programming pattern. And we expect many broken idioms can be discovered this way.

It should be pointed out that the EndCon instruction plays a crucial role to make the *Double-Checked Locking* algorithm work. In a closely related version, the object reference is obtained from a pre-allocated object instead of being created from scratch in the “double-checked” slot. To simulate this programming pattern, EndCon should be taken out from Thread 1 in Figure 12. This would cause the failure of our invariant.

## 6.4 Performance Statistics

The above test programs are executed on a Dell PC with a 366 MHz Pentium processor and 128 MB RAM running Windows 2000. The performance statistics are listed below:

Test	State Size (Bytes)	Total States	Rules Fired	Time (Sec)
Coherence	148	875	2971	0.21
PRAM	152	2942	9676	0.57
Prescient Store (without aliasing)	152	2834	9142	0.50
Prescient Store (with aliasing)	152	1148	3484	0.23
Write Atomicity	148	937	3037	0.22
Causality	228	31526	10347	2.06
EndCon	188	1584	4690	0.36
Double-Checked Locking	500	6992	24790	3.69

## 7 Conclusion

Specifying the Java Memory Model with a Model Checking framework has several advantages over the original CRF specification. First, the specification is executable. It can benefit the model designers to gain extra level of confidence in their designs. It is also appealing to programmers, JVM implementers, and compiler writers because it can help them better understand the subtleties of the model. The beauty of this mechanism is that the memory model can be provided as a “black box”. The user of the verification system is not necessarily required to understand all the details of the memory model. Second, the Model Checking technique enables one to exhaustively exercise simple programs. This can reveal some corner cases which would be very hard to find through traditional simulation techniques. Finally, the specification in Mur $\phi$  is more rigorous, which further eliminates any ambiguities. The process of formally specifying the model forces one to think harder about the specification. Our experience of translating CRF to Mur $\phi$  has helped us clarify many details about the original design.

On the other hand, it is inevitable that the Mur $\phi$  program comes with some internal data structures and implementation details. Without a higher level description, this specification alone might be hard to follow. It would be a desirable feature to have a common design interface for any executable memory model with all the implementation details encapsulated.

Our approach also has some limitations. Based on the Model Checking techniques, it is exposed to the state explosion problem. We have to limit our test program to a small scale. A test with four variables and eight instructions per thread has run out of memory on our 128 MB PC. Even though our examples demonstrated how simple tests can help people understand complex memory models, state explosion problem nonetheless limits our ability to verify bigger commercial Java code. Because our system is bounded, it is not designed for guarantee of correctness. Rather, it provides a debugging tool for understanding the underlying memory model.

The “advanced” features, such as thread creation, termination, and interruption, are not discussed in detail in the CRF specification. Pugh has suggested that hidden volatile fields can be used to describe these kinds of thread communication semantics. This strategy can be integrated to our verifier to model these scenarios. In our on-going process for studying the Java Memory Model, we are continuing to develop more comprehensive test programs to cover more interesting properties. A well designed suite of test programs will be a

valuable benchmark for understanding any given memory model. We will also explore better state reduction techniques and try to verify real thread-safe Java library code based on our framework.

## Acknowledgments

We would like to thank Jan-Willem Maessen and Xiaowei Shen for their insightful comments and suggestions about our work. Their feedback has helped us enhance our verification system.

## References

- [1] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*, chapter 17. Addison-Wesley, 1996.
- [2] William Pugh. Fixing the Java Memory Model. In *Java Grande*, pages 89–98, 1999.
- [3] Jan-Willem Maessen, Arvind, and Xiaowei Shen. Improving the Java Memory Model using CRF. In *OOPSLA*, pages 1–12, October 2000.
- [4] Jsr-133: Java Memory Model and Thread Specification Revision.  
URL: <http://jcp.org/jsr/detail/133.jsp>.
- [5] Jeremy Manson and William Pugh. Semantics of multithreaded Java. Technical report, UMIACS-TR-2001-09.
- [6] X. Shen, Arvind, and L. Rudolph. Commit-Reconcile & Fences (CRF): A New Memory Model for Architects and Compiler Writers. In *the 26th International Symposium On Computer Architecture*, Atlanta, Georgia, May 1999.
- [7] D. Dill, S. Park, and A. Nowatzyk. Formal specification of abstract memory models. In *the 1993 Symposium for Research on Integrated Systems*, pages 38–52, March 1993.
- [8] David Dill. The Mur $\phi$  verification system. In *8th International Conference on Computer Aided Verification*, pages 390–393, 1996.
- [9] Source code of the Mur $\phi$  verification program for the CRF Java Memory Model.  
URL: <http://www.cs.utah.edu/~yyang/research/crf.m>.
- [10] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.
- [11] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.
- [12] Mustaque Ahamad, Rida Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. In *5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 93)*, 1993.
- [13] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, 1988.
- [14] Patrice Godefroid. Model checking for programming languages using Verisoft. In *Symposium on Principles of Programming Languages*, pages 174–186, 1997.
- [15] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.
- [16] W. Visser, K. Havelund, G. Brat, and S. Park. Java PathFinder - second generation of a Java Model Checker. In *Post-CAV Workshop on Advances in Verification, Chicago*, 2000.
- [17] James C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, March 1996.
- [18] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.
- [19] D. Park, U. Stern, and D. Dill. Java model checking. In *the First International Workshop on Automated Program Analysis, Testing and Verification, Limerick, Ireland*, 2000.
- [20] D. Weaver and T. Germond. *The SPARC Architecture Manual Version 9*. Prentice Hall, 1994.
- [21] Seungjoon Park and David L. Dill. An executable specification and verifier for Relaxed Memory Order. *IEEE Transactions on Computers*, 48(2):227–235, 1999.
- [22] Philip Bishop and Nigel Warren. *Java in Practice: Design Styles and Idioms for Effective Java*, chapter 9. Addison-Wesley, 1999.

- [23] A. Gontmakher and A. Schuster. Java consistency: Non-operational characterizations for Java memory behavior. In *the Workshop on Java for High-Performance Computing, Rhodes*, June 1999.