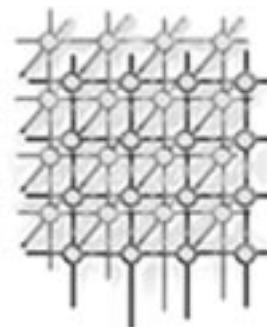


UMM: An Operational Memory Model Specification Framework with Integrated Model Checking Capability[†]



Yue Yang, Ganesh Gopalakrishnan, and
Gary Lindstrom*

School of Computing, University of Utah, Salt Lake City, UT 84112

SUMMARY

Given the complicated nature of modern shared memory systems, it is vital to have a systematic approach to specifying and analyzing memory consistency requirements. In this paper, we present the UMM specification framework, which integrates two key features to support memory model verification: (i) it employs a simple and generic memory abstraction that can capture a large collection of memory models as guarded commands with a uniform notation, and (ii) it provides built-in model checking capability to enable formal reasoning about thread behaviors. Using this framework, memory models can be developed in a parameterized style – designers can simply redefine a few bypassing rules and visibility ordering rules to obtain an executable specification of another memory model. We formalize several classical memory models, including Sequential Consistency, Coherence, and PRAM, to illustrate the general techniques of applying this framework. We then provide an alternative specification of the Java memory model, based on a proposal from Manson and Pugh, and demonstrate how to analyze Java thread semantics using model checking. We also compare our operational specification style with axiomatic specification styles and explore a mechanism that converts a memory model definition from one style to the other.

KEY WORDS: Memory model, Operational specification, Java thread, Formal verification

*Correspondence to: School of Computing, University of Utah, Salt Lake City, UT 84112; Email: {yyang|ganesh|gary}@cs.utah.edu

[†]This work was supported in part by Research Grant No. CCR-0081406 (ITR Program) of NSF and SRC Task 1031.001.



```

(Initially, flag1 = flag2 = false, turn = 0)

Thread 1                                Thread 2

flag1 = true;                            flag2 = true;
turn = 2;                                turn = 1;
while(turn == 2 && flag2)                 while(turn == 1 && flag1)
;                                         ;
< critical section >                     < critical section >
flag1 = false;                           flag2 = false;

```

Figure 1. Peterson's algorithm for mutual exclusion.

Initially, $flag1 = flag2 = false, turn = 0$

Thread 1	Thread 2
$flag1 = true;$	$flag2 = true;$
$turn = 2;$	$turn = 1;$
$r1 = turn;$	$r3 = turn;$
$r2 = flag2;$	$r4 = flag1;$

Finally, can it result in $r1 = 2, r3 = 1,$ and $r2 = r4 = false?$

Figure 2. An execution that breaks Peterson's algorithm.

1. INTRODUCTION

With the recent advances in multiprocessor shared memory architectures and integrated threading support from programming languages such as Java, multithreaded programming is becoming an increasingly popular technique for developing well structured and high performance applications. Unlike a sequential program, where each read simply returns the value written by the most recent write according to program order, a multithreaded program relies on the *memory model* (also known as the *thread semantics*) to define how threads interact in a shared memory system.

Memory consistency requirements can impact program correctness in a fundamental way. Consider, for example, Peterson's algorithm [1] for mutual exclusion shown in Figure 1. Each thread first sets its own flag indicating its intention to enter the critical section, and then asserts that it is the other thread's turn if appropriate. The eventual value of the variable *turn* determines which thread enters the critical section first. The correctness of this well known programming pattern, however, crucially depends on the allowed thread interleavings. Figure 2 abstracts the key memory operations from Peterson's algorithm and illustrates a specific thread behavior that breaks the algorithm: if both threads can still observe the default flag values while checking the loop conditions, they are able to enter the critical section at the



same time. Many memory systems permit this result due to optimization needs. Consequently, programs relying on Peterson's algorithm would be erroneous on those systems.

Program fragments such as the one in Figure 2 are generally known as *litmus tests*. Carefully studying these test programs can reveal critical memory model properties and help programmers make right decisions in code selection and optimization. For simple cases, one can often follow a pencil-and-pen approach to reason about the legality of a litmus test. But as bigger programs are used and more complex models are involved, thread interleavings quickly become non-intuitive and hand-proving program compliance can be very difficult and error-prone. The proliferation of various proposed memory models also poses a major challenge for programmers to reliably comprehend the differences as well as similarities among them, which are often subtle yet critical. For example, even experts have experienced difficulties in understanding the exact power of certain memory models [2].

Given that memory model compliance is a prerequisite for developing robust concurrent systems, it is crucial to support a rigorous methodology for analyzing memory model specifications. In this paper, we present the *UMM* (Uniform Memory Model) [‡] specification framework, which employs an abstract transition system to define memory operations in an operational style. Our main insight is that by using only two kinds of buffers, namely *local instruction buffer* (LIB) and *global instruction buffer* (GIB), we can separately capture requirements on *program order* and *visibility order*, two pivotal properties for understanding thread behaviors. Relaxations of the program order are configured through a bypassing table and rules in first-order logic are used to express these bypassing policies. Completed instructions in GIB are used to construct the legal visibility order subject to certain visibility ordering rules. With this approach, variations between memory models can be isolated into a few well-defined places such as the bypassing table and the visibility ordering rules, enabling easy comparison and configuration. Coupled with a model checking utility, the UMM framework can exhaustively exercise a test program to cover all thread interleavings.

Summary of Results We offer the following contributions. First, we develop a generic framework that can be used to produce executable memory model specifications. One main result of this paper is to show that this particular design can capture not only architectural level memory models but also language level memory models. No previous work has treated these two categories in a uniform framework; yet, the importance of doing so is growing, especially with the advent of multiprocessor machines on whose architectural designs one has to support the language level thread semantics in the most efficient manner. Second, we discuss the process of defining different types of consistency models according to a taxonomy based on the visibility ordering requirement. As a concrete demonstration, we formalize several classical memory models and use these executable specifications to perform program verification. Third, we adapt the Java thread specification by Manson and Pugh and show how to conduct rigorous analysis for a complex memory model design. Finally, we discuss the relationship of our

[‡]The term “uniform models” (in contrast to “hybrid models”) has occasionally been used for memory models without special synchronization instructions. For this reason, UMM can also be referred to as *Unified Memory Model* to avoid confusion.



operational specification style with trace-based axiomatic specification styles and propose a mechanism that transforms a memory model definition from one style to the other.

Road Map In the next section, we introduce the background issues related to memory model specifications. Then we present an overview of our framework in Section 3. In Section 4, we formalize several well known memory models to show our general approach. We provide an alternative formal specification of the Java memory model in Section 5. It is followed by a thorough analysis of JMM_{MP} in Section 6. In Section 7, we compare our operational specification style with axiomatic specification styles. In Section 8, we survey related work. Finally, we conclude and explore future research opportunities in Section 9.

2. BACKGROUND

Memory consistency requirements often place various restrictions on program order and visibility order among memory operations. Program order is the original instruction order determined by software. Visibility order (similar to the notions of “before order” in [3] and “visibility order” in [4]) is the final observable order of memory operations perceived by one or more threads.

Memory model designs typically involve a tradeoff between programmability and efficiency. As one of the earliest memory models, *Sequential Consistency* (SC) [5] is intuitive but restrictive. A memory system is sequentially consistent if the result of an execution is the same as if the operations of all the threads were executed in some sequential order, and the operations of each individual thread appear in this sequence according to program order.[§] Many weaker memory models (see [6] for a survey) have since been proposed to enable higher performance implementations. Some still require a property called *Coherence* [7] (also known as *Cache Coherence* or *Cache Consistency*), which requires all operations involving a given variable to exhibit a total order that also respects program order of each thread.

In general, memory models can be categorized according to their visibility ordering requirements. Early memory models, such as Sequential Consistency, are designed for single bus systems, where a common visibility order is enforced for all observing threads. Some other models, such as *Parallel Random Access Memory* (PRAM) [8], allow each individual thread to observe its own visibility order. Informally, PRAM requires that there exists an execution sequence for every thread, containing all operations from the observing thread and all write operations from other threads, such that it exhibits a total order that also respects program order. For example, the outcome of the program in Figure 3 is not allowed by Sequential Consistency and Coherence since there does not exist a common visibility order that can be agreed upon by both threads. However, the behavior is permitted by PRAM because each thread can perceive its own visibility order. That is, thread 1 and thread 2 may observe

[§]Architectural level memory models are usually described in terms of *processors* and *memory locations*. Language level memory models, on the other hand, are often discussed using *threads* and *shared variables*. In this paper, we adopt the latter terminology for our discussion.



Initially, $a = 0$

Thread 1	Thread 2
(1) $a = 1;$	(3) $a = 2;$
(2) $r1 = a;$	(4) $r2 = a;$

Finally, can it result in $r1 = 2$ and $r2 = 1$?

Figure 3. An execution prohibited by SC and Coherence but allowed by PRAM.

interleaving (1)(3)(2) and (3)(1)(4), respectively. Using our framework, we can formally analyze the behaviors of such concurrent programs under various memory models.

Many shared memory systems allow programmers to use special synchronization operations in addition to read and write operations. In *Lazy Release Consistency* [9], synchronization is performed by *release* and *acquire* operations. When release is performed, previous memory activities from the issuing thread need to be written to the shared memory. A thread reconciles with the shared memory to obtain the updated data when acquire is issued. *Lazy Release Consistency* requires Coherence. This requirement is further relaxed by *Location Consistency* [10]. Operations in *Location Consistency* are only partially ordered if they are from the same thread or if they are synchronized through locks.

2.1. The Existing Java Memory Model

Java is the first widely deployed programming language that provides built-in threading support at the language level. Java developers routinely rely on threads for structuring their programs, sometimes even without explicit awareness. As future hardware architectures become more aggressively parallel, multithreaded Java also provides an appealing platform for high performance software. The Java memory model (JMM) is a critical component of the Java threading system since it imposes significant implications on a broad range of activities, such as programming pattern development, compiler optimization, and Java virtual machine (JVM) implementation. Unfortunately, developing a rigorous and intuitive Java memory model has turned out to be very difficult.

The existing Java memory model is given in Chapter 17 of the Java Language Specification [11], where Java thread semantics is defined by eight different actions that are constrained by a set of informal rules. Due to the lack of rigor in specification, non-obvious implications can be deduced by combining different rules [12]. As a result, the existing Java memory model is flawed and hard to understand. Some of its major issues are listed as follows.

- The model requires Coherence. Because of this restriction, important compiler optimizations such as *fetch elimination* are prohibited.
- The model requires a thread to flush all variables to main memory before releasing a lock, imposing a strong restriction on visibility order. Consequently, some seemingly redundant synchronization operations (such as thread local synchronization blocks) cannot be optimized away.



- The ordering guarantee for a constructor is not strong enough. On weak memory architectures such as Alpha, uninitialized fields of an object can be observable under race conditions even after the object reference is initialized and made visible to other threads. This problem opens a security hole to malicious attacks via race conditions.
- Semantics of *final* variable operations is omitted.
- *Volatile* variable operations do not have synchronization effects on normal variable operations. As a result, volatile variables cannot be applied as synchronization flags to indicate the completion of non-volatile variable operations.

Several efforts [13, 14] have been conducted to formalize the existing Java memory model since its problems were pointed out in [12]. Improved Java thread semantics have also been proposed to replace the current one. One proposal is from Manson and Pugh [15, 16] (referred to as JMM_{MP}), another proposal is from Maessen, Arvind, and Shen [17] (referred to as JMM_{CRF}) based on the Commit/Reconcile/Fence (CRF) framework. The Java memory model is currently under an official revision process [18] and there is an ongoing discussion through the JMM mailing list [19]. Most recently, Manson and Pugh announced a new Java memory model draft [19] for community review.

2.2. Overview of JMM_{MP}

Since the core semantics of JMM_{MP} is adapted as a concrete case study in this paper, we briefly describe JMM_{MP} here as an overview. Readers are referred to [15] for more details.

JMM_{MP} is based on an abstract global system that executes one operation from one thread at each step. The actions occur in a total order that respects the program order from each thread. The only ordering relaxation explicitly allowed is for *prescient writes* under certain conditions. To define the thread semantics, various *sets* are used to track history information of memory activities. JMM_{MP} proposes the following thread properties in order to solve the problems listed in Section 2.1.

- The ordering constraint should be relaxed to enable common optimizations. JMM_{MP} essentially follows Location Consistency, which does not require Coherence.
- The synchronization mechanism should be relaxed to enable the removal of redundant synchronization blocks. JMM_{MP} applies an release/acquire mechanism in which visibility states are only synchronized through the *same* lock.
- Java safety should be guaranteed even under race conditions. JMM_{MP} enforces that all final fields should be initialized properly from a constructor.
- Reasonable semantics for final variables should be provided. In JMM_{MP} , a final field v is *frozen* at the end of the constructor before the reference of the object is returned. Another thread would always observe the initialized value of v unless it is improperly exposed to other threads before it is frozen.
- Volatile variables should be specified to be more useful for multithreaded programming. JMM_{MP} adds the release/acquire semantics to volatile variable operations as well. A write to a volatile field acts as a release and a read of a volatile field acts as an acquire.

Although JMM_{MP} and JMM_{CRF} have initiated promising improvements on Java thread semantics, they are not as easily comprehensible and comparable as first thought. JMM_{CRF}

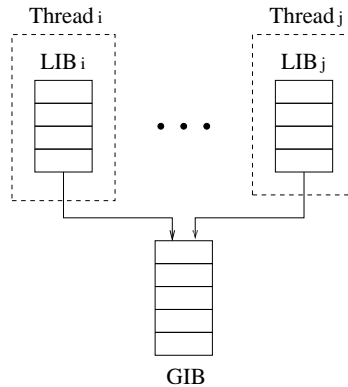


Figure 4. Basic conceptual architecture of the UMM specification framework.

inherits the design from its predecessor hardware model [20]. Memory instructions need to be divided into Commit/Reconcile/Fence instructions, making it harder to capture memory properties directly. The cache based architecture also prohibits JMM_{CRF} from describing more relaxed models. JMM_{MP} maintains the history of memory activities in various sets of memory actions. While this notation might be sufficient to express the proposed semantics, adjusting it to specify different properties is not trivial. Since designing a memory model involves a repeated process of testing and fine-tuning, a generic specification framework is needed to provide such flexibility.

3. OVERVIEW OF THE FRAMEWORK

The UMM framework consists of an abstract transition system with an associated transition table. Figure 4 illustrates the basic conceptual architecture of the transition system. Each thread k has a *local instruction buffer* LIB_k , which stores all pending memory operations in program order. Threads interact through a *global instruction buffer* GIB , which stores all previously completed memory operations that are necessary for fulfilling a future read request.

In contrast to most processor level memory architectures that apply a cache layer between processors and main memory, the UMM system only uses two layers – one for thread local information and the other for global trace information. This simple memory abstraction eliminates unnecessary complexities introduced by implementation specific data structures and helps clarify the essential semantics of the shared memory system. Instead of a fixed-size main memory, we apply a global instruction buffer whose size may be increased if necessary, which is needed for specifying relaxed memory models that require to keep a trace of multiple writes on a given variable.



3.1. Defining a Memory Model

Semantics of memory operations are precisely defined as *guarded commands* in a transition table, where memory operations are categorized as *events* that may be completed by carrying out some *actions* when certain *conditions* are satisfied. At a given step, any eligible event may be nondeterministically chosen and atomically completed by the abstract machine. For clarity, our framework applies a bypassing table called **BYPASS** to configure the ordering policy for issuing instructions. These bypassing rules used in the instruction selection process serve two purposes: (i) they impose an interleaving close to the memory model requirement, and (ii) they presciently enable certain operations when needed. Completed instructions in **GIB** are used to form the legal visibility order. The visibility ordering rules are imposed as a final filtering mechanism to guarantee proper *serialization*, i.e., a read returns the value from the latest write on the same variable.

A memory model \mathcal{M} defines what are the *legal executions* of a given program. Intuitively, a legal execution is a sequence of permissible actions from various threads that also forms a proper serialization. An actual implementation of \mathcal{M} , $\mathcal{I}_{\mathcal{M}}$, may choose different architectures and optimization techniques as long as the legal executions allowed by $\mathcal{I}_{\mathcal{M}}$ are also permitted by \mathcal{M} .

Our operational definition employs rules expressed in first-order logic to capture details. Thus, in a sense, it has a *dual status*: the big picture is captured operationally, while the details are captured in a declarative manner. This style is also found in some related efforts, e.g. [21]. Here, our contributions are twofold: we employ this style for a wide spectrum of memory models; and we integrate a model checking tool with the specification framework to support rigorous analysis.

3.2. Integrating the Model Checking Technique

To make a memory model specification executable, we encode it in Murphi [22], a description language with a syntax similar to C as well as a model checking system that supports exhaustive state space enumeration. Since Murphi naturally supports specifications based on guarded commands, this encoding process is straightforward. We retain the first-order logic style of the formal specification in our Murphi model (Murphi supports first-order logic quantifiers, which are unravelled through state enumeration). This helps make the translation process reliable.

Our Murphi program consists of two parts. The first part implements the formal specification of a memory model. The transition table is specified as Murphi rules. Bypassing and visibility ordering conditions are implemented as Murphi procedures. The second part comprises a collection of idiom-driven test programs. Each test program, defined by specific Murphi initial state and invariants, is designed to reveal a certain memory model property or to simulate a common programming idiom. When a test program is executed under the guidance of the UMM transition system, the Murphi model checker exhaustively exercises all possible executions allowed by the memory model. Our system can detect deadlocks and invariant violations. To examine test results, two techniques can be applied. The first one uses Murphi 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 when all threads are completed, to output all possible results. The Murphi implementation is highly configurable, allowing one to easily set up test programs, abstract machine parameters, and memory model properties. The executable memory model can also be treated as a “black box” whereby the users are not necessarily required to understand all the details of the model to benefit from the specification.

4. FORMALIZING CLASSICAL MEMORY MODELS

UMM can be configured to specify a large collection of memory models. The general strategy is to customize the bypassing table to control thread interleavings and impose proper visibility ordering constraints on the operations in GIB. This configuration process is typically trivial for memory models involving a common visibility order. If a model requires per-thread visibility orders, we apply a technique (inspired by similar methods in [4, 23]) that decomposes a write operation into multiple sub-write operations targeting each thread so that the single GIB can be used to retrieve unique visibility orders for every observing thread.

In this section, we demonstrate our approach by formalizing several well known memory models. Sequential Consistency and Coherence are presented to show the process of defining models with a single visibility order. PRAM serves as an example of models requiring per-thread visibility orders. Most other memory models can be defined in a similar way.

4.1. Instructions

For the common memory models discussed in this section, an instruction i is represented by a tuple $\langle t, pc, op, var, data, target, time \rangle$, where

$t(i) = t$:	issuing thread;
$pc(i) = pc$:	program counter;
$op(i) = op$:	operation type, can be <i>Read</i> or <i>Write</i> ;
$var(i) = var$:	variable;
$data(i) = data$:	data value;
$target(i) = target$:	target thread observing a write;
$time(i) = time$:	global time stamp, incremented each time when an instruction is added to GIB.

4.2. Initial Conditions

LIB initially contains all instructions from each thread in program order. For Sequential Consistency and Coherence, writes do not need to be decomposed. For PRAM, a write i is converted to a set of sub-write instructions for each thread k ($target(i) = k$), including the issuing thread. The sub-write instructions that originate from the same write share the same program counter pc . GIB initially contains the default write instructions for every variable v (with the default value of v , a special thread ID t_{init} , and a $time$ field of 0). After the abstract machine is set up, it operates according to the transition table.



Event	Condition	Action
read	$\exists i \in \text{LIB}_{t(i)} :$ $\text{ready}(i) \wedge \text{op}(i) = \text{Read} \wedge$ $(\exists w \in \text{GIB} : \text{legalWrite}(i, w))$	$i.\text{data} := \text{data}(w);$ $\text{LIB}_{t(i)} := \text{delete}(\text{LIB}_{t(i)}, i);$
write	$\exists i \in \text{LIB}_{t(i)} :$ $\text{ready}(i) \wedge \text{op}(i) = \text{Write}$	$\text{GIB} := \text{append}(\text{GIB}, i);$ $\text{LIB}_{t(i)} := \text{delete}(\text{LIB}_{t(i)}, i);$

Table I. Transition table for Sequential Consistency, Coherence, and PRAM.

	SC		Coherence		PRAM	
2nd \Rightarrow 1st \Downarrow	Read	Write	Read	Write	Read	Write
Read	No	No	DiffVar	DiffVar	No	DiffTgt
Write	No	No	DiffVar	DiffVar	DiffTgt	DiffTgt

Table II. Bypassing table for Sequential Consistency, Coherence, and PRAM.

4.3. Transition Table

The transition table for Sequential Consistency, Coherence, and PRAM is given in Table I. A read instruction completes when the return value is bound. A write instruction completes when it is added to GIB. A multithreaded program terminates when all instructions from all threads complete.

4.4. Bypassing Table

Table II outlines the bypassing table `BYPASS` for Sequential Consistency, Coherence, and PRAM, where an entry `BYPASS[op1][op2]` determines whether an instruction with type `op2` can bypass a previous instruction with type `op1`. Values used in table `BYPASS` include *Yes*, *No*, *DiffVar*, and *DiffTgt*. Informally, *Yes* permits the bypassing, *No* prohibits it, *DiffVar* conditionally enables the bypassing only if the variables are different and not aliased, and *DiffTgt* conditionally enables the bypassing when a sub-write targeting a different thread is involved. According to Table II, no bypassing is allowed for Sequential Consistency. For Coherence, instructions operated on different variables can be issued out of order. For PRAM, two sub-writes targeting different threads can be reordered, and a sub-write can be reordered with a read if the sub-write targets another thread. These bypassing rules are precisely defined in condition *ready*.



$$\begin{aligned} \text{ready}(i) \equiv & \\ & \neg \exists j \in \text{LIB}_{t(i)} : pc(j) < pc(i) \wedge \\ & (\text{BYPASS}[op(j)][op(i)] = \text{No} \vee \\ & \text{BYPASS}[op(j)][op(i)] = \text{DiffVar} \wedge var(j) = var(i) \vee \\ & \text{BYPASS}[op(j)][op(i)] = \text{DiffTgt} \wedge op(j) = \text{Write} \wedge op(i) = \text{Read} \wedge target(j) = t(i) \vee \\ & \text{BYPASS}[op(j)][op(i)] = \text{DiffTgt} \wedge op(j) = \text{Read} \wedge op(i) = \text{Write} \wedge t(j) = target(i) \vee \\ & \text{BYPASS}[op(j)][op(i)] = \text{DiffTgt} \wedge op(j) = \text{Write} \wedge op(i) = \text{Write} \wedge target(j) = target(i)) \end{aligned}$$

4.5. Visibility Ordering Requirement

Condition *legalWrite* is a guard for read events that guarantees the serialization requirement. Informally, it states that a write w is not eligible for a read r if there exists an overwriting write w' between r and w in the ordering path. The *legalWrite* definition of PRAM is only slightly different from that of Sequential Consistency and Coherence because a reading thread t can only observe sub-writes targeting t or the default writes.

For **Sequential Consistency** and **Coherence**:

$$\begin{aligned} \text{legalWrite}(r, w) \equiv & \\ & op(w) = \text{Write} \wedge var(w) = var(r) \wedge \\ & (\neg \exists w' \in \text{GIB} : op(w') = \text{Write} \wedge var(w') = var(r) \wedge \\ & time(r) > time(w') \wedge time(w') > time(w)) \end{aligned}$$

For **PRAM**:

$$\begin{aligned} \text{legalWrite}(r, w) \equiv & \\ & op(w) = \text{Write} \wedge var(w) = var(r) \wedge (target(w) = t(r) \vee t(w) = t_{init}) \wedge \\ & (\neg \exists w' \in \text{GIB} : op(w') = \text{Write} \wedge var(w') = var(r) \wedge target(w') = t(r) \wedge \\ & time(r) > time(w') \wedge time(w') > time(w)) \end{aligned}$$

4.6. Applying the Executable Specifications for Program Analysis

The above specifications of SC, Coherence, and PRAM clearly illustrate a key feature of the UMM framework – memory models are specified in a *parameterized* style, meaning designers can simply redefine a few bypassing rules (defined in condition *ready*) and visibility ordering rules (defined in condition *legalWrite*) to obtain an executable specification for another memory model. In fact, the corresponding Murphi implementation is written in a modular fashion with switches that select which memory model is being defined.

The executable specifications coded in Murphi can help one analyze common programming patterns against different memory models. For example, recall the litmus test shown in Figure 2, which reveals a scenario that would make Peterson's algorithm erroneous. If this program is executed under PRAM or Coherence, the tool immediately detects certain thread interleaving that would lead to the result of interest, indicating that Peterson's algorithm is



broken under these memory models. Under Sequential Consistency, however, the execution in Figure 2 is not allowed.

5. AN ALTERNATIVE JAVA MEMORY MODEL SPECIFICATION

We develop an alternative Java memory model specification for several reasons: (i) it illustrates how to resolve some of the language level memory model issues, such as the treatment of local variables, (ii) it shows how synchronization operations may be defined using our framework, and (iii) it demonstrates that our methodology can be scaled up for analyzing complex memory model designs. The core JMM semantics formalized in this section is primarily based on JMM_{MP} [15] as of January 11, 2002.

5.1. Variables and Instructions

In the Java memory model, a *global variable* refers to a static field of a loaded class, an instance field of an allocated object, or an element of an allocated array. It can be further categorized as a *normal*, *volatile*, or *final* variable. A *local variable* corresponds to a Java local variable or an operand stack location. In our examples, we follow a convention that uses a, b, c to represent global variables, $r1, r2, r3$ to represent local variables, and 1, 2, 3 to represent primitive values.

The instruction tuple in the Java memory model is extended to carry local variable and locking information. An instruction i is denoted by $\langle t, pc, op, var, data, local, useLocal, lock, time \rangle$, where

$t(i) = t$:	issuing thread;
$pc(i) = pc$:	program counter;
$op(i) = op$:	operation type;
$var(i) = var$:	variable;
$data(i) = data$:	data value;
$local(i) = local$:	local variable;
$useLocal(i) = useLocal$:	tag to indicate if the write value is provided by $local(i)$;
$lock(i) = lock$:	lock;
$time(i) = time$:	global time stamp, incremented each time when an instruction is added to GIB.

Since the proposed semantics does not enforce a *unique* per-thread visibility order for any observing thread, a write does not need to be decomposed into sub-writes.

5.2. The Extended Conceptual Architecture

To capture the additional requirements regarding local variables and locks in the Java memory model, the conceptual architecture of the transition system is slightly extended. Figure 5 shows the abstract machine for modelling the Java memory model. In addition to the local instruction buffer, each thread k also maintains a *local variable array* LV_k . Each element $LV_k[v]$ contains the data value of the local variable v . To maintain the locking status, a dedicated global *lock*

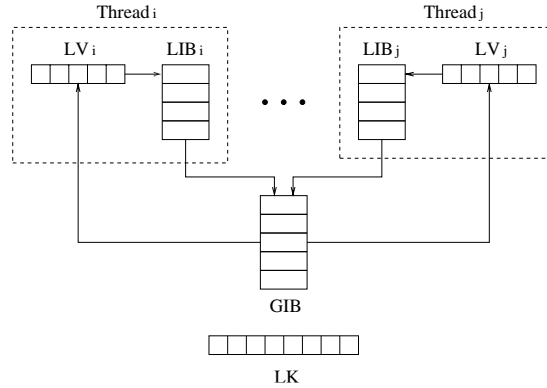


Figure 5. Extended conceptual architecture for the Java memory model.

array LK is also added. Each element $LK[l]$ is a tuple $\langle count, owner \rangle$, where *count* is the number of recursive lock acquisitions and *owner* is the owning thread.

Need for Local Variable Information In processor level memory model specifications, a read is usually retired immediately after the return value is obtained. Following the same style, neither JMM_{MP} nor JMM_{CRF} keeps track the return values from reads. However, most programming activities in Java, such as computation, flow control, and method invocation, are carried out using local variables. Therefore, it is desired to extend the scope of the memory model framework by recording the values committed to local variables as part of the global state. The addition of local variable arrays in the transition system also provides a clear separation of local data dependence and memory model ordering requirements, which will be further discussed in Section 5.5.

5.3. Initial Conditions

LIB initially contains instructions from each thread in program order. GIB initially contains the default write instructions for every variable v (with the default value of v , a special thread ID t_{init} , and a *time* field of 0). The *count* fields in LK are initially set to 0.

5.4. Transition Table for the Java Memory Model

Java memory operations are defined in the transition table given in Table III. A *read* operation on a global variable corresponds to the Java program instruction with a format of $r1 = a$. It always stores the data value in the target local variable. A *write* operation on a global variable can have two formats, $a = r1$ or $a = 1$, depending on whether the *useLocal* tag is set. The format $a = r1$ allows one to examine the data flow implications caused by the



nondeterminism of memory behaviors. If all writes have $useLocal = false$ and all reads use non-conflicting local variables, the system degenerates to traditional models that do not keep local variable information. *Lock* and *unlock* instructions are issued as determined by the Java keyword *synchronized*. They are used to model the mutual exclusion effect as well as the visibility effect. A special *freeze* instruction for every final field v is added at the end of the constructor that initializes v to indicate that v has been frozen. Since we are defining the memory model, only memory operations are identified in our transition system. Instructions such as $r1 = 1$ and $r1 = r2 + r3$ are not included. However, the UMM framework can be easily extended to a comprehensive program analysis system by adding semantics for computational instructions.

5.5. Bypassing Table and Local Data Dependence

Table IV specifies the bypassing rules for the Java memory model. Since JMM_{MP} respects program order except for *prescient writes*, Table IV only allows normal write instructions to bypass certain previous instructions. Although it might be desired to enable more reordering, e.g. between two normal read operations, the specification presented here follows the same guideline from JMM_{MP} to capture similar semantics.

In JMM_{MP} , different threads are only synchronized via the *same* lock. No ordering restriction is imposed by a *Lock* instruction if there is no synchronization effect associated with it. Since most redundant synchronization operations are caused by thread local and nested locks, Table IV uses a special entry *RdtLk* to enable optimization in the cases involving redundant locks. A *WriteNormal* instruction can bypass a previous *Lock* or *ReadVolatile* instruction when the previous instruction does not impose any synchronization effect. Condition *ready* enforces the bypassing policy of the memory model as well as local data dependence. The helper function $notRedundant(j)$ returns *true* if instruction j does have a synchronization effect.

Data dependence imposed by the usage of conflicting local variables is expressed in condition *localDependent*. The helper function $isWrite(i)$ returns *true* if the operation type of i is *WriteNormal*, *WriteVolatile*, or *WriteFinal*. Similarly, $isRead(i)$ returns *true* if the operation of i is *ReadNormal*, *ReadVolatile*, or *ReadFinal*.

$$\begin{aligned}
 ready(i) &\equiv \\
 &\neg \exists j \in LIB_{t(i)} : pc(j) < pc(i) \wedge \\
 &(localDependent(i, j) \vee \\
 &BYPASS[op(j)][op(i)] = No \vee \\
 &BYPASS[op(j)][op(i)] = RdtLk \wedge notRedundant(j))
 \end{aligned}$$

$$\begin{aligned}
 localDependent(i, j) &\equiv \\
 &t(j) = t(i) \wedge local(j) = local(i) \wedge \\
 &(isWrite(i) \wedge useLocal(i) \wedge isRead(j) \vee \\
 &isWrite(j) \wedge useLocal(j) \wedge isRead(i) \vee \\
 &isRead(i) \wedge isRead(j))
 \end{aligned}$$



Event	Condition	Action
readNormal	$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \wedge \text{op}(i) = \text{ReadNormal} \wedge (\exists w \in \text{GIB} : \text{legalNormalWrite}(i, w))$	$\text{LV}_{t(i)}[\text{local}(i)] := \text{data}(w);$ $\text{LIB}_{t(i)} := \text{delete}(\text{LIB}_{t(i)}, i);$
writeNormal	$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \wedge \text{op}(i) = \text{WriteNormal}$	if ($\text{useLocal}(i)$) $i.\text{data} := \text{LV}_{t(i)}[\text{local}(i)];$ end; $\text{GIB} := \text{append}(\text{GIB}, i);$ $\text{LIB}_{t(i)} := \text{delete}(\text{LIB}_{t(i)}, i);$
lock	$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \wedge \text{op}(i) = \text{Lock} \wedge (\text{LK}[\text{lock}(i)].\text{count} = 0 \vee \text{LK}[\text{lock}(i)].\text{owner} = t(i))$	$\text{LK}[\text{lock}(i)].\text{count} := \text{LK}[\text{lock}(i)].\text{count} + 1;$ $\text{LK}[\text{lock}(i)].\text{owner} := t(i);$ $\text{GIB} := \text{append}(\text{GIB}, i);$ $\text{LIB}_{t(i)} := \text{delete}(\text{LIB}_{t(i)}, i);$
unlock	$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \wedge \text{op}(i) = \text{Unlock} \wedge (\text{LK}[\text{lock}(i)].\text{count} > 0 \wedge \text{LK}[\text{lock}(i)].\text{owner} = t(i))$	$\text{LK}[\text{lock}(i)].\text{count} := \text{LK}[\text{lock}(i)].\text{count} - 1;$ $\text{GIB} := \text{append}(\text{GIB}, i);$ $\text{LIB}_{t(i)} := \text{delete}(\text{LIB}_{t(i)}, i);$
readVolatile	$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \wedge \text{op}(i) = \text{ReadVolatile} \wedge (\exists w \in \text{GIB} : \text{legalVolatileWrite}(i, w))$	$\text{LV}_{t(i)}[\text{local}(i)] := \text{data}(w);$ $\text{GIB} := \text{append}(\text{GIB}, i);$ $\text{LIB}_{t(i)} := \text{delete}(\text{LIB}_{t(i)}, i);$
writeVolatile	$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \wedge \text{op}(i) = \text{WriteVolatile}$	if ($\text{useLocal}(i)$) $i.\text{data} := \text{LV}_{t(i)}[\text{local}(i)];$ end; $\text{GIB} := \text{append}(\text{GIB}, i);$ $\text{LIB}_{t(i)} := \text{delete}(\text{LIB}_{t(i)}, i);$
readFinal	$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \wedge \text{op}(i) = \text{ReadFinal} \wedge (\exists w \in \text{GIB} : \text{legalFinalWrite}(i, w))$	$\text{LV}_{t(i)}[\text{local}(i)] := \text{data}(w);$ $\text{LIB}_{t(i)} := \text{delete}(\text{LIB}_{t(i)}, i);$
writeFinal	$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \wedge \text{op}(i) = \text{WriteFinal}$	if ($\text{useLocal}(i)$) $i.\text{data} := \text{LV}_{t(i)}[\text{local}(i)];$ end; $\text{GIB} := \text{append}(\text{GIB}, i);$ $\text{LIB}_{t(i)} := \text{delete}(\text{LIB}_{t(i)}, i);$
freeze	$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \wedge \text{op}(i) = \text{Freeze}$	$\text{GIB} := \text{append}(\text{GIB}, i);$ $\text{LIB}_{t(i)} := \text{delete}(\text{LIB}_{t(i)}, i);$

Table III. Transition table for the alternative Java memory model.



2nd \Rightarrow 1st \Downarrow	Read Normal	Write Normal	Lock	Unlock	Read Volatile	Write Volatile	Read Final	Write Final	Freeze
Read Normal	No	Yes	No	No	No	No	No	No	No
Write Normal	No	Yes	No	No	No	No	No	No	No
Lock	No	RdtLk	No	No	No	No	No	No	No
Unlock	No	Yes	No	No	No	No	No	No	No
Read Volatile	No	RdtLk	No	No	No	No	No	No	No
Write Volatile	No	Yes	No	No	No	No	No	No	No
Read Final	No	Yes	No	No	No	No	No	No	No
Write Final	No	Yes	No	No	No	No	No	No	No
Freeze	No	No	No	No	No	No	No	No	No

Table IV. Bypassing table for the alternative Java memory model.

5.6. Visibility Ordering Requirement for the Java Memory Model

JMM_{MP} applies an ordering constraint similar to Location Consistency. As captured in condition *LCOrder*, two instructions are ordered if one of the following cases holds:

1. they are ordered by program order;
2. they are synchronized by the same lock or the same volatile variable; or
3. there exists another operation that can transitively establish the order.

$$\begin{aligned}
 LCOrder(i1, i2) \equiv & \\
 & (t(i1) = t(i2) \wedge pc(i1) > pc(i2) \vee t(i1) \neq t_{init} \wedge t(i2) = t_{init}) \vee \\
 & synchronized(i1, i2) \vee \\
 & (\exists i' \in GIB : time(i') > time(i2) \wedge time(i') < time(i1) \wedge LCOrder(i1, i') \wedge LCOrder(i', i2))
 \end{aligned}$$

The synchronization mechanism is formally captured in condition *synchronized*. Instruction *i1* can be synchronized with a previous instruction *i2* via a release/acquire process, where a lock is first released by *t(i2)* after *i2* is issued and later acquired by *t(i1)* before *i1* is issued. Release can be triggered by an *Unlock* or a *WriteVolatile* instruction. Acquire can be triggered by a *Lock* or a *ReadVolatile* instruction.

$$\begin{aligned}
 synchronized(i1, i2) \equiv & \\
 & \exists l, u \in GIB : \\
 & (op(l) = Lock \wedge op(u) = Unlock \wedge lock(l) = lock(u) \vee \\
 & op(l) = ReadVolatile \wedge op(u) = WriteVolatile \wedge var(l) = var(u)) \wedge \\
 & t(l) = t(i1) \wedge (t(u) = t(i2) \vee t(i2) = t_{init}) \wedge \\
 & time(i2) \leq time(u) \wedge time(u) < time(l) \wedge time(l) \leq time(i1)
 \end{aligned}$$

After establishing the ordering relationship by condition *LCOrder*, the requirement of serialization is enforced in *legalNormalWrite*. Informally, a write *w* cannot provide its value to a read *r* if there exists another overwriting write *w'* in the ordering path.



$$\begin{aligned} \text{legalNormalWrite}(r, w) &\equiv \\ &op(w) = \text{WriteNormal} \wedge var(w) = var(r) \wedge \\ &(t(w) = t(r) \rightarrow pc(w) < pc(r)) \wedge \\ &(\neg \exists w' \in \text{GIB} : op(w') = \text{WriteNormal} \wedge var(w') = var(r) \wedge LCOrder(r, w') \wedge LCOrder(w', w)) \end{aligned}$$

The *mutual exclusion* effect of **Lock** and **Unlock** operations is enforced by tracking the *count* and *owner* fields of each lock as specified in the transition table.

5.7. Volatile Variable Semantics

When JMM_{MP} was proposed, the exact ordering requirement for volatile variable operations was still under debate. One suggestion was to require volatile variable operations to be sequentially consistent. Another suggestion was to relax Write Atomicity. Although JMM_{MP} provides a formal specification that allows non-atomic volatile write operations, recent consensus favors Sequential Consistency for all volatile variable operations. Therefore, we define volatile variable semantics based on Sequential Consistency in this paper.

With the uniform notation of our framework, pre-defined memory requirements can be easily reused. Hence, the formal definition of Sequential Consistency described in Section 4 is applied to define **ReadVolatile** and **WriteVolatile** operations. The bypassing table shown in Table IV prohibits any reordering among volatile operations. Condition *legalVolatileWrite*, which follows *legalWrite* in Sequential Consistency, defines the legal results for **ReadVolatile** operations.

$$\begin{aligned} \text{legalVolatileWrite}(r, w) &\equiv \\ &op(w) = \text{WriteVolatile} \wedge var(w) = var(r) \wedge \\ &(\neg \exists w' \in \text{GIB} : op(w') = \text{WriteVolatile} \wedge var(w') = var(r) \wedge \\ &time(r) > time(w') \wedge time(w') > time(w)) \end{aligned}$$

5.8. Final Variable Semantics

In Java, a final field can either be a primitive value or a reference to another object. When it is a reference, the Java language only requires that the reference itself cannot be modified in the program after its initialization but the fields of the object it points to do not have the same guarantee. JMM_{MP} proposes to add a special rule to those non-final fields that are referenced by a final variable: if such a field is assigned in the constructor, its default value cannot be observed by another thread after normal construction. To achieve this, JMM_{MP} uses a special mechanism to “synchronize” initialization information from the constructing thread to the final reference and eventually to the elements contained by the final reference. However, without explicit support for immutability from the Java language, this mechanism makes the memory semantics substantially more complicated because synchronization information needs to be carried by every variable.

Since the main goal of this paper is to illustrate our methodology, finding the most reasonable solution for final field semantics is an orthogonal task. To make our Java memory model specification complete, yet not to distract readers with the details specific to certain semantics, we provide a straightforward definition for final fields. It is different from JMM_{MP} in that it only requires the final field itself to be a constant after being frozen. The visibility criteria for final fields is shown in condition *legalFinalWrite*. The default value of the final field (when $t(w) = t_{init}$) can only be observed if the final field is not frozen. In addition, the constructing thread cannot observe the default value after the

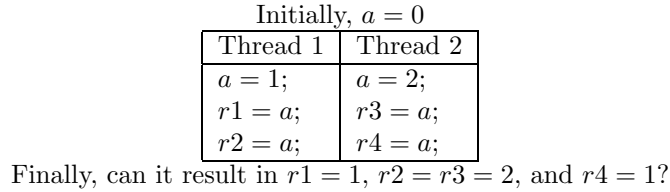


Figure 6. Write Atomicity test.

final field is initialized.

$$\begin{aligned}
 \text{legalFinalWrite}(r, w) \equiv & \\
 & op(w) = \text{WriteFinal} \wedge var(w) = var(r) \wedge \\
 & (t(w) = t_{init} \rightarrow \\
 & ((\neg \exists i1 \in \text{GIB} : op(i1) = \text{Freeze} \wedge var(i1) = var(r)) \wedge \\
 & (\neg \exists i2 \in \text{GIB} : op(i2) = \text{WriteFinal} \wedge var(i2) = var(r) \wedge t(i2) = t(r))))
 \end{aligned}$$

6. ANALYSIS OF JMM_{MP} VIA MODEL CHECKING

After adapting JMM_{MP} , we can systematically exercise it with test programs and gain substantial insight about the underlying semantics. Since we have also developed an executable model for JMM_{CRF} in a previous work [24], we can perform a comparison analysis by running the same test programs on both models. This helps us understand the subtle differences between the two models.

Running on a PC with a 900 MHz Pentium III processor and 256 MB of RAM, most of our tests complete in less than one second. Our Murphi implementation of the Java memory model is available at http://www.cs.utah.edu/formal_verification/umm.

6.1. Analyzing Memory Model Properties

Coherence Test Recall the litmus test shown in Figure 3, which reveals an execution prohibited by Coherence. An exhaustive enumeration of this test program under JMM_{MP} reports that the result of interest is permitted for normal variables. The UMM tool can output an interleaving that leads to such a result to help the users understand the scenario. Thus, based on this simple litmus test (without even looking at the details of the memory model), one can make an immediate conclusion that JMM_{MP} does not follow Coherence.

Write Atomicity Test The execution in Figure 6 illustrates a violation of Write Atomicity. When this test program (for a normal variable a) is run using our tool, one can quickly find out that the result in Figure 6 is allowed by JMM_{MP} but forbidden by JMM_{CRF} . This reveals a difference between the two models. A more thorough analysis of JMM_{CRF} indicates that the requirement of Write Atomicity in JMM_{CRF} is a direct consequence of the CRF architecture because it uses the shared memory as the rendezvous point between threads and caches.

Causality Test *Causal Consistency* [25] requires thread local orderings to be transitive through a causal relationship. The program shown in Figure 7 reveals a violation of causality. When it is executed for normal variables under JMM_{MP} , a legal interleaving that allows such a behavior is detected, proving



Initially, $a = b = 0$

Thread 1	Thread 2	Thread 3
$a = 1;$	$r1 = a;$ $b = 1;$	$r2 = b$ $r3 = a$

Finally, can it result in $r1 = r2 = 1$ and $r3 = 0$?

Figure 7. Causality test.

Initially, $a = 0$

Thread 1	Thread 2
$r1 = a;$ $a = 1;$	$r2 = a;$ $a = r2;$

Finally, can it result in $r1 = 1$ and $r2 = 1$?

Figure 8. Prescient write test.

Initially, $reference = field = 0$

Thread 1	Thread 2
$field = 1;$ $Membar1;$ $reference = 1;$	$r1 = reference;$ $Membar2$ $r2 = field;$

Finally, can it result in $r1 = 1$ and $r2 = 0$?

Figure 9. Constructor test.

that JMM_{MP} does not enforce Causal Consistency.

Prescient Write Test Figure 8 reveals an interesting case of *prescient write*, where $r1$ in Thread 1 can observe a write that is initiated by a later write on the same variable from the same thread. Our system detects that such a non-intuitive execution is indeed allowed by JMM_{MP} . Therefore, programmers should not assume *antidependence* (data dependence caused by Write after Read on the same variable) among global variable operations.

Constructor Property The constructor property is illustrated by the program in Figure 9. Thread 1 simulates the constructing thread, which initializes the field before releasing the object reference. Thread 2 simulates another thread accessing the object field without synchronization. *Membar1* and *Membar2* are some hypothetical memory barriers that prevents instructions from crossing them, which can be easily implemented in our program by simply setting some test specific bypassing rules. This program essentially simulates the object constructing mechanism used by JMM_{CRF} , where *Membar1* is a special *EndCon* instruction used in JMM_{CRF} to indicate the completion of a constructor and *Membar2* is due to data dependence when accessing *field* through *reference*. If *field* is a normal variable, this mechanism works under JMM_{CRF} but fails under JMM_{MP} . In JMM_{MP} , the default write



to *field* is still a valid write for the reading thread since there does not exist an ordering requirement on non-synchronized writes. However, if *field* is declared as a final variable and the **Freeze** instruction is used for *Membar1*, Thread 2 would never observe the default value of *field* if *reference* is initialized. This illustrates the different strategies used by the two models for preventing premature releases during object construction: JMM_{CRF} treats all fields uniformly and JMM_{MP} guarantees fully initialized fields only if they are final or pointed by final variables.

6.2. Verifying Programming Patterns

As demonstrated by Peterson's algorithm in Section 1, many popular programming patterns developed for certain memory models might break under other models. If the test program in Figure 2 is executed using normal variables under JMM_{MP}, it is immediately detected that such a scenario is allowed, indicating that the algorithm is unsafe in Java programs without applying additional synchronization operations. On the other hand, if volatile variables are used, such an execution would not be allowed. Carefully analyzing concurrent algorithms based on formal methods is an effective strategy for developing robust multithreaded programs.

7. COMPARISON WITH AXIOMATIC SPECIFICATIONS

The area of memory model specification has been pursued under different approaches. Some researchers have used *axiomatic* (also known as *non-operational*) specifications, in which the desired properties are directly defined. Other researchers have employed *operational* specifications, in which the update of the global state is defined step-by-step with the execution of each instruction.

An axiomatic approach divides the global ordering relation in terms of *facets*, each of which constrains a specific aspect of the memory system. In a separate research [26, 27], for instance, we define a memory model as a complete set of ordering rules, including a fully explicit description about general ordering properties, such as totality, transitivity, and circuit-freedom. To take a concrete example, PRAM can be defined in an axiomatic style as a set of constraints imposed on an execution trace *ops*, shown in predicate **legal**. Predicate **restrictThread** selects a subset of memory operations from *ops*, which contains all operations from the observing thread *t* and all writes from other threads. Each constraint is then precisely defined (the program order definition is given here as an example).

legal *ops* \equiv
 $\forall t \in T. (\exists \text{order}.$
 $\text{requireProgramOrder}(\text{restrictThread } ops \ t) \ \text{order} \wedge$
 $\text{requireWeakTotalOrder}(\text{restrictThread } ops \ t) \ \text{order} \wedge$
 $\text{requireTransitiveOrder}(\text{restrictThread } ops \ t) \ \text{order} \wedge$
 $\text{requireAsymmetricOrder}(\text{restrictThread } ops \ t) \ \text{order} \wedge$
 $\text{requireReadValue}(\text{restrictThread } ops \ t) \ \text{order})$

requireProgramOrder *ops order* \equiv
 $\forall i, j \in ops. ((\mathbf{t} \ i = \mathbf{t} \ j \ \wedge \ \mathbf{pc} \ i < \mathbf{pc} \ j) \vee (\mathbf{t} \ i = t_{init} \ \wedge \ \mathbf{t} \ j \neq t_{init})) \Rightarrow \mathbf{order} \ i \ j$

The UMM framework applies a two-layer architecture to generate operational memory model definitions. Variations of memory consistency properties are parameterized as different bypassing rules and visibility ordering rules. The total order property, if required, can be implicitly built up during the execution based on interleavings allowed by the bypassing rules. Despite its operational style, the UMM framework is closely related to axiomatic specification methods. If a memory model allows all instructions to be sent to **GIB** in any arbitrary order and then impose additional ordering constraints when read values are obtained, a UMM specification degenerates to an axiomatic one.



Each of the two styles has its own advantages. The axiomatic approach is declarative and more flexible. One can disable/enable the constraints and study the impact on the global ordering requirement. However, each constraint itself may involve aspects that pertain to both program orders and visibility orders, which cannot be easily distinguished. The operational style, on the other hand, can separate these matters clearly and often simplify the rules using its interleaving mechanism.

Understanding the different specification mechanisms can help one to transform a memory model definition from one style to the other. To capture an axiomatic definition using UMM, one needs to consider all the ordering rules and extract those that can be enforced using the front-end instruction selection process of the UMM framework – this would simplify the filtering process when read values are obtained. To convert a UMM specification to an axiomatic definition, one must encode all the ordering requirements implied by the UMM front-end process and impose them as axioms on the final execution trace.

8. RELATED WORK

Extensive research has been done in model checking Java programs, e.g. [28, 29, 30, 31, 32, 33]. These tools, however, assume sequentially consistent program behaviors and do not address memory model issues. Therefore, they cannot analyze fine-grained thread interleavings. We can imagine our method being incorporated into these tools to make their analysis more realistic.

In order to formalize memory models, Collier [23] described a theory of memory ordering rules. Using methods similar to Collier's, Gharachorloo [7] developed a generic framework for specifying the implementation conditions for various memory models. The shortcoming of these approaches is that it is not trivial to infer program behaviors from a combination of ordering constraints. In fact, the lack of a means for automatic execution is a noticeable limitation for most declarative specifications. To solve this problem, we proposed a method in [26, 27] that (i) captures the complete set of memory ordering rules as axioms, (ii) encodes the non-operational specification into a machine recognizable format, and (iii) makes it executable by checking the satisfiability of all constraints using a SAT solver.

To make an operational memory model executable, Park and Dill [34, 35] proposed to integrate a model checker with the Sparc *Relaxed Memory Order* [36] specification for verifying small assembly synchronization routines. In our previous work on the analysis of JMM_{CRF} [24], we extended this methodology to the domain of JMM and demonstrated its feasibility and effectiveness for analyzing language level thread semantics. After adapting JMM_{CRF} to an equivalent executable specification implemented in Murphi, we systematically exercised the underlying model with a suite of test programs to reveal pivotal properties and verify common programming idioms, such as the *Double-Checked Locking* algorithm [37]. Roychoudhury and Mitra [38] also applied techniques similar to [24] to study the existing Java memory model. They formalized the current JMM with an operational representation using a local cache and a set of read/write queues and implemented the specification with an invariant checker. Although [24] and [38] have improved their respective target models by making them executable, they are limited to the specific designs from the original proposals and the intuitions behind the memory consistency requirements based on those notations are not immediately apparent. As a result, they are not suited as a generic specification framework. Furthermore, the complexity of the special data structures demands more memory consumption during model checking, which would worsen the state space explosion problem. In [39], a formal operational framework was developed to verify protocol implementation against weak memory models using model checking. In that framework, however, data structures of the transition system vary depending on whether a single visibility order or multiple visibility orders need to be defined. These variations make it difficult to create a parameterized analysis tool. The UMM framework was originally applied to analyze JMM_{MP} in our previous work [40]. While [40] concentrated on the Java thread semantics, this paper provides a thorough description of the UMM framework.



9. CONCLUSION

We have discussed the design and application of the UMM framework, and demonstrated how to apply it to formalize memory models in general and the Java memory model in particular. With this framework, a designer can specify and analyze shared memory consistency requirements using a systematic approach as follows: (i) formally define thread semantics as guarded commands using the UMM transition system, (ii) encode such a specification, along with idiom-driven test programs, using a model checker, and (iii) automatically enumerate all executions of these test programs to verify pivotal memory model properties. Based on our experience, making memory models executable greatly reduces the likelihood of having overlooked corner cases.

A reliable specification framework may lead to several interesting future works. First, people currently need to develop test programs by hand to conduct verification. To automate this process, programming pattern annotation and inference techniques can play an important role. Second, traditional compilation techniques can be systematically analyzed for memory model compliance in a multithreaded environment and new optimization opportunities allowed by more relaxed consistency requirements should be explored. Lastly, architectural memory models and the Java memory model may be compared through refinement analysis to aid efficient JVM implementations. We hope our work can help pave the way towards future studies in these exciting areas.

ACKNOWLEDGEMENTS

We sincerely thank all contributors to the Java memory model mailing list for their inspiring discussions. We especially thank Bill Pugh for his insightful comments about our work. We also thank the anonymous referees of this paper for their detailed suggestions.

REFERENCES

1. G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, Volume 12, Number 3, June 1981.
2. Mustaque Ahamad, Rida Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of Processor Consistency. In *the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1993.
3. L. Lamport. Time, clocks and ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–565, July 1978.
4. A formal specification of Intel Itanium processor family memory ordering, Application Note, Document Number: 251429-001, October 2002.
5. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.
6. S. V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *IEEE Computer*, 29(12):66–76, 1996.
7. K. Gharachorloo. Memory consistency models for shared-memory multiprocessors. Technical Report CSL-TR-95-685, Stanford University, December 1995.
8. R. J. Lipton and J. S. Sandberg. PRAM: a scalable shared memory. Technical Report CS-TR-180-88, 1988.
9. P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *the 19th International Symposium of Computer Architecture*, pages 13-21, May 1992.
10. G. Gao and V. Sarkar. Location consistency - a new memory model and cache consistency protocol. Technical Report, 16, CAPSL, University of Delaware, February 1998.
11. J. Gosling, B. Joy, and G. Steele. The Java language specification, chapter 17. Addison-Wesley, 1996.
12. W. Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(1):1-11, 2000.
13. A. Gontmakher and A. Schuster. Java consistency: non-operational characterizations for Java memory model. In *ACM Transactions On Computer Systems*, vol. 18, No. 4, pages 333-386, November 2000.



14. Y. Gurevich, W. Schulte, and C. Wallace. Investigating Java concurrency using abstract state machines. Technical Report 2000-04, University of Delaware, December 1999.
15. J. Manson and W. Pugh. Semantics of multithreaded Java. Technical Report UMIACS-TR-2001-09, 2002.
16. J. Manson and W. Pugh. Core semantics of multithreaded Java. In *ACM Java Grande Conference*, June 2001.
17. J. Maessen, Arvind, and X. Shen. Improving the Java memory model using CRF. In *OOPSLA*, pages 1-12, October 2000.
18. Java Specification Request (JSR) 133: Java memory model and thread specification revision. <http://jcp.org/jsr/detail/133.jsp>.
19. Java memory model mailing list. <http://www.cs.umd.edu/~pugh/java/memoryModel/archive>.
20. 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*, May 1999.
21. R. Gerth. Introduction to Sequential Consistency and the lazy caching algorithm. *Distributed Computing*, 1995.
22. D. Dill. The Mur ϕ verification system. In *8th International Conference on Computer Aided Verification*, pages 390-393, 1996.
23. W. W. Collier. Reasoning about parallel architectures. Prentice-Hall, 1992.
24. Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. Analyzing the CRF Java memory model. In *the 8th Asia-Pacific Software Engineering Conference*, pages 21-28, 2001.
25. Mustaque Ahamad, Phillip W. Hutto, Gil Neiger, James E. Burns, and Prince Kohli. Causal memory: definitions, implementation and programming. Technical Report GIT-CC-93/55, Georgia Institute of Technology, July 1994.
26. Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Analyzing the Intel Itanium memory ordering rules using logic programming and SAT. In *the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2003)*, October 2003.
27. Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Nemos: a framework for axiomatic and executable specifications of memory consistency models. Technical Report UUCS-03-019, University of Utah, 2003.
28. Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA PathFinder. In *International Journal on Software Tools for Technology Transfer*, volume 2, number 4, pages 366-381, 2000.
29. 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.
30. James C. Corbett. Evaluating deadlock detection methods for concurrent software. In *IEEE Transactions on Software Engineering*, volume 22, number 3, pages 161-180, March 1996.
31. 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.
32. 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.
33. J. Moore, R. Krug, H. Liu, and G. Porter. Formal models of Java at the JVM level - a survey from the ACL2 perspective. In *Workshop on Formal Techniques for Java Programs*, in association with ECOOP 2001, June 2001.
34. D. Dill, S. Park, and A. Nowatzky. Formal specification of abstract memory models. In *the 1993 Symposium for Research on Integrated Systems*, pages 38-52, 1993.
35. S. Park and D. Dill. An executable specification and verifier for Relaxed Memory Order. *IEEE Transactions on Computers*, 48(2):227-235, 1999.
36. D. Weaver and T. Germond. The SPARC architecture manual version 9. Prentice Hall, 1994.
37. Philip Bishop and Nigel Warren. Java in practice: design styles and idioms for effective Java, chapter 9. Addison-Wesley, 1999.
38. A. Roychoudhury and T. Mitra. Specifying multithreaded Java semantics for program verification. In *International Conference on Software Engineering*, 2002.
39. Prosenjit Chatterjee, Hemantkumar Sivaraj, and Ganesh Gopalakrishnan. Shared memory consistency protocol verification against weak memory models: refinement via model-checking. In *Computer-Aided Verification (CAV'02)*, July, 2002.
40. Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. Specifying Java thread semantics using a uniform memory model. In *Joint ACM Java Grande ISCOPE Conference*, 2002.