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, U.S.A.

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 specified 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. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: memory model; operational specification; Java thread; formal verification

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.

*Correspondence to: Yue Yang, School of Computing, University of Utah, Salt Lake City, UT 84112, U.S.A.
†E-mail: syang@cs.utah.edu

Contract/grant sponsor: NSF; contract/grant number: CCR-0081406
Contract/grant sponsor: SRC; contract/grant number: 1031.001

Received 13 January 2003
Revised 31 August 2003
Accepted 14 October 2003
Initially, flag1 = flag2 = false, turn = 0

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

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

Figure 1. Peterson’s algorithm for mutual exclusion.

Initially, flag1 = flag2 = false, turn = 0

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>flag1 = true;</td>
<td>flag2 = true;</td>
</tr>
<tr>
<td>turn = 2;</td>
<td>turn = 1;</td>
</tr>
<tr>
<td>r1 = turn;</td>
<td>r3 = turn;</td>
</tr>
<tr>
<td>r2 = flag2;</td>
<td>r4 = flag1;</td>
</tr>
</tbody>
</table>

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

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

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. A careful study of these test programs can reveal critical memory model properties and help programmers make the right decisions in code selection and optimization. For simple cases, one can often follow a pencil-and-paper 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 *Uniform Memory Model* (UMM)\(^\ddagger\) 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 a *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 the 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.

### 1.1. 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 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.

### 1.2. 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 JMMMP in Section 6.

\(^\ddagger\)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 the *Unified Memory Model* to avoid confusion.
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 trade-off 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 the program order of each thread.

In general, memory models can be categorized according to their visibility ordering requirements. Early memory models, such as SC, 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 SC 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 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.

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, $a = 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) $a = 1$;</td>
<td>(3) $a = 2$;</td>
</tr>
<tr>
<td>(2) $r1 = a$;</td>
<td>(4) $r2 = a$;</td>
</tr>
</tbody>
</table>

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

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

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 JMM has turned out to be very difficult.

The existing JMM 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 JMM 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 JMM 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 JMMMP), another proposal is from Maessen et al. [17] (referred to as JMMCRF) based on the Commit/Reconcile/Fence (CRF) framework.
The JMM 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 JMM draft [19] for community review.

2.2. Overview of JMMMP

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

JMMMP 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. JMMMP 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. JMMMP essentially follows Location Consistency, which does not require Coherence.
- The synchronization mechanism should be relaxed to enable the removal of redundant synchronization blocks. JMMMP applies a release/acquire mechanism in which visibility states are only synchronized through the same lock.
- Java safety should be guaranteed even under race conditions. JMMMP enforces that all final fields should be initialized properly from a constructor.
- Reasonable semantics for final variables should be provided. In JMMMP, 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. JMMMP 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 JMMMP and JMM$_{CRF}$ have initiated promising improvements on Java thread semantics, they are not as easily comprehensible and comparable as first thought. JMM$_{CRF}$ inherits the design from its predecessor hardware model [20]. Memory instructions need to be divided into CRF instructions, making it harder to capture memory properties directly. The cache-based architecture also prohibits JMM$_{CRF}$ from describing more relaxed models. JMMMP 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 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 GIB 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 a 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], which is 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 unraveled through state enumeration). This helps make the translation process reliable.

Our Murphi program consists of two parts: (i) 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, and (ii) 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, and 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 a 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. SC 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.
Table I. Transition table for SC, Coherence, and PRAM.

<table>
<thead>
<tr>
<th>Event</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>( \exists i \in \text{LIB}_{t(i)} : )</td>
<td>( i, \text{data} := \text{data}(w) ; )</td>
</tr>
<tr>
<td></td>
<td>( \text{ready}(i) \land \text{op}(i) = \text{Read} \land )</td>
<td>( \text{LIB}<em>{t(i)} := \text{delete}(\text{LIB}</em>{t(i)}, i) ; )</td>
</tr>
<tr>
<td></td>
<td>(( \exists w \in \text{GIB} : \text{legalWrite}(i, w) ))</td>
<td></td>
</tr>
<tr>
<td>write</td>
<td>( \exists i \in \text{LIB}_{t(i)} : )</td>
<td>( \text{GIB} := \text{append}(\text{GIB}, i) ; )</td>
</tr>
<tr>
<td></td>
<td>( \text{ready}(i) \land \text{op}(i) = \text{Write} )</td>
<td>( \text{LIB}<em>{t(i)} := \text{delete}(\text{LIB}</em>{t(i)}, i) ; )</td>
</tr>
</tbody>
</table>

4.1. Instructions

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

- \( t(i) = t \): issuing thread;
- \( pc(i) = pc \): program counter;
- \( \text{op}(i) = \text{op} \): operation type, can be Read or Write;
- \( \text{var}(i) = \text{var} \): variable;
- \( \text{data}(i) = \text{data} \): data value;
- \( \text{target}(i) = \text{target} \): target thread observing a write;
- \( \text{time}(i) = \text{time} \): global time stamp, incremented each time an instruction is added to the GIB.

4.2. Initial conditions

The LIB initially contains all instructions from each thread in program order. For SC 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 \( k \) (\( \text{target}(i) = k \)), including the issuing thread. The sub-write instructions that originate from the same write share the same program counter \( pc \). The GIB initially contains the default write instructions for every variable \( v \) (with the default value of \( v \), a special thread ID \( t_{\text{init}} \), and a \( \text{time} \) field of 0). After the abstract machine is set up, it operates according to the transition table.

4.3. Transition table

The transition table for SC, 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 the GIB. A multithreaded program terminates when all instructions from all threads complete.

4.4. Bypassing table

Table II outlines the bypassing table \( \text{BYPASS} \) for SC, Coherence, and PRAM, where an entry \( \text{BYPASS}_{\text{op1}}[\text{op2}] \) determines whether an instruction with type \( \text{op2} \) can bypass a previous instruction with type \( \text{op1} \). Values used in table \( \text{BYPASS} \) 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 SC. 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 \( \text{ready} \).

4.5. Visibility ordering requirement

Condition \( \text{legalWrite} \) 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 \( \text{legalWrite} \) definition of PRAM is only slightly different from that of SC and Coherence because a reading thread \( t \) can only observe sub-writes targeting \( t \) or the default writes.

For SC and Coherence:

\[
\text{legalWrite}(r, w) \equiv \\
\text{op}(w) = \text{Write} \land \text{var}(w) = \text{var}(r) \land \\
(\neg \exists w' \in \text{GIB} : \text{op}(w') = \text{Write} \land \text{var}(w') = \text{var}(r) \land \\
\text{time}(r) > \text{time}(w') \land \text{time}(w') > \text{time}(w))
\]

For PRAM:

\[
\text{legalWrite}(r, w) \equiv \\
\text{op}(w) = \text{Write} \land \text{var}(w) = \text{var}(r) \land (\text{target}(w) = t(r) \lor t(w) = t_{\text{init}}) \land \\
(\neg \exists w' \in \text{GIB} : \text{op}(w') = \text{Write} \land \text{var}(w') = \text{var}(r) \land \text{target}(w') = t(r) \land \\
\text{time}(r) > \text{time}(w') \land \text{time}(w') > \text{time}(w))
\]
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 \textit{ready}) and visibility ordering rules (defined in condition \textit{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 SC, however, the execution in Figure 2 is not allowed.

5. AN ALTERNATIVE JMM SPECIFICATION

We develop an alternative JMM 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\textsubscript{MP} [15] as of 11 January 2002.

5.1. Variables and instructions

In the JMM, a \textit{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 \textit{normal}, \textit{volatile}, or \textit{final} variable. A \textit{local variable} corresponds to a Java local variable or an operand stack location. In our examples, we follow a convention that uses $a$, $b$ and $c$ to represent global variables, $r_1$, $r_2$ and $r_3$ to represent local variables, and $1$, $2$ and $3$ to represent primitive values.

The instruction tuple in the JMM is extended to carry local variable and locking information. An instruction $i$ is denoted by $(t, pc, op, var, data, local, useLocal, lock, time)$ 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 the 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 JMM, the conceptual architecture of the transition system is slightly extended. Figure 5 shows the abstract machine for modeling the JMM. 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 array $LK$ is also added. Each element $LK[l]$ is a tuple $(\text{count}, \text{owner})$, where $\text{count}$ is the number of recursive lock acquisitions and $\text{owner}$ 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 of 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 is further discussed in Section 5.5.
5.3. Initial conditions

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

5.4. Transition table for the JMM

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 \( r_1 = a \). It always stores the data value in the target local variable. A write operation on a global variable can have two formats, \( a = r_1 \) or \( a = 1 \), depending on whether the useLocal tag is set. The format \( a = r_1 \) allows one to examine the data flow implications caused by the non-determinism 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 key word 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 \( r_1 = 1 \) and \( r_1 = r_2 + r_3 \) 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 JMM. 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.

\[
\text{ready}(i) \equiv \\
\neg \exists j \in \text{LIB}_{(i)} : \text{pc}(j) < \text{pc}(i) \land \\
(\text{localDependent}(i, j) \lor \\
\text{BYPASS}([op(j)][op(i)]) = \text{No} \lor \\
\text{BYPASS}([op(j)][op(i)]) = \text{RdtLk} \land \text{notRedundant}(j))
\]
Table III. Transition table for the alternative JMM.

<table>
<thead>
<tr>
<th>Event</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>readNormal</td>
<td>$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \land \text{op}(i) = \text{ReadNormal} \land$\break ($\exists w \in \text{GIB} : \text{legalNormalWrite}(i, w))</td>
<td>$\text{LV}<em>{t(i)}[\text{local}(i)] := \text{data}(w)$; \break $\text{LIB}</em>{t(i)} := \text{delete}($\text{LIB}_{t(i)}, i$);</td>
</tr>
<tr>
<td>writeNormal</td>
<td>$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \land \text{op}(i) = \text{WriteNormal}$</td>
<td>if (useLocal($i$)) \break $i.data := \text{LV}<em>{t(i)}[\text{local}(i)];$ \break end: \break $\text{GIB} := \text{append}($\text{GIB}, i$); \break $\text{LIB}</em>{t(i)} := \text{delete}($\text{LIB}_{t(i)}, i$);</td>
</tr>
<tr>
<td>lock</td>
<td>$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \land \text{op}(i) = \text{Lock} \land$\break ($\text{LK}[\text{lock}(i)].\text{count} = 0 \lor$\break $\text{LK}[\text{lock}(i)].\text{owner} = t(i))$</td>
<td>$\text{LK}[\text{lock}(i)].\text{count} :=$\break $\text{LK}[\text{lock}(i)].\text{count} + 1$; \break $\text{LK}[\text{lock}(i)].\text{owner} := t(i)$; \break $\text{GIB} := \text{append}($\text{GIB}, i$); \break $\text{LIB}<em>{t(i)} := \text{delete}($\text{LIB}</em>{t(i)}, i$);</td>
</tr>
<tr>
<td>unlock</td>
<td>$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \land \text{op}(i) = \text{Unlock} \land$\break ($\text{LK}[\text{lock}(i)].\text{count} &gt; 0 \land$\break $\text{LK}[\text{lock}(i)].\text{owner} = t(i))$</td>
<td>$\text{LK}[\text{lock}(i)].\text{count} :=$\break $\text{LK}[\text{lock}(i)].\text{count} - 1$; \break $\text{GIB} := \text{append}($\text{GIB}, i$); \break $\text{LIB}<em>{t(i)} := \text{delete}($\text{LIB}</em>{t(i)}, i$);</td>
</tr>
<tr>
<td>readVolatile</td>
<td>$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \land \text{op}(i) = \text{ReadVolatile} \land$\break ($\exists w \in \text{GIB} : \text{legalVolatileWrite}(i, w))</td>
<td>if (useLocal($i$)) \break $i.data := \text{LV}<em>{t(i)}[\text{local}(i)];$ \break end: \break $\text{GIB} := \text{append}($\text{GIB}, i$); \break $\text{LIB}</em>{t(i)} := \text{delete}($\text{LIB}_{t(i)}, i$);</td>
</tr>
<tr>
<td>writeVolatile</td>
<td>$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \land \text{op}(i) = \text{WriteVolatile}$</td>
<td>if (useLocal($i$)) \break $i.data := \text{LV}<em>{t(i)}[\text{local}(i)];$ \break end: \break $\text{GIB} := \text{append}($\text{GIB}, i$); \break $\text{LIB}</em>{t(i)} := \text{delete}($\text{LIB}_{t(i)}, i$);</td>
</tr>
<tr>
<td>readFinal</td>
<td>$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \land \text{op}(i) = \text{ReadFinal} \land$\break ($\exists w \in \text{GIB} : \text{legalFinalWrite}(i, w))</td>
<td>$\text{LV}<em>{t(i)}[\text{local}(i)] := \text{data}(w)$; \break $\text{LIB}</em>{t(i)} := \text{delete}($\text{LIB}_{t(i)}, i$);</td>
</tr>
<tr>
<td>writeFinal</td>
<td>$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \land \text{op}(i) = \text{WriteFinal}$</td>
<td>if (useLocal($i$)) \break $i.data := \text{LV}<em>{t(i)}[\text{local}(i)];$ \break end: \break $\text{GIB} := \text{append}($\text{GIB}, i$); \break $\text{LIB}</em>{t(i)} := \text{delete}($\text{LIB}_{t(i)}, i$);</td>
</tr>
<tr>
<td>freeze</td>
<td>$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \land \text{op}(i) = \text{Freeze}$</td>
<td>$\text{GIB} := \text{append}($\text{GIB}, i$); \break $\text{LIB}<em>{t(i)} := \text{delete}($\text{LIB}</em>{t(i)}, i$);</td>
</tr>
</tbody>
</table>
Table IV. Bypassing table for the alternative JMM.

<table>
<thead>
<tr>
<th>2nd ⇒</th>
<th>Read</th>
<th>Write</th>
<th>Lock</th>
<th>Unlock</th>
<th>Read</th>
<th>Write</th>
<th>Read</th>
<th>Write</th>
<th>Freeze</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>Normal</td>
<td>Normal</td>
<td>Lock</td>
<td>Unlock</td>
<td>Volatile</td>
<td>Volatile</td>
<td>Final</td>
<td>Final</td>
<td>Freeze</td>
</tr>
<tr>
<td>Read Normal</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Write Normal</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Lock</td>
<td>No</td>
<td>RdtLk</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Unlock</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Read Volatile</td>
<td>No</td>
<td>RdtLk</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Write Volatile</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Read Final</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Write Final</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Freeze</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

localDependent(i, j) ≡

\[ t(j) = t(i) \land \text{local}(j) = \text{local}(i) \land \]
\[ (\text{isWrite}(i) \land \text{useLocal}(i) \land \text{isRead}(j) \lor \text{isWrite}(j) \land \text{useLocal}(j) \land \text{isRead}(i) \lor \text{isRead}(i) \land \text{isRead}(j)) \]

5.6. Visibility ordering requirement for the JMM

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;
3. there exists another operation that can transitively establish the order.

\( LCOrder(i_1, i_2) \equiv \)

\[ (t(i_1) = t(i_2) \land pc(i_1) > pc(i_2) \lor t(i_1) \neq t_{\text{init}} \land t(i_2) = t_{\text{init}}) \lor \]
\[ \text{synchronized}(i_1, i_2) \lor \]
\[ (\exists i' \in \text{GIB} : \text{time}(i') > \text{time}(i_2) \land \text{time}(i') < \text{time}(i_1) \land LCOrder(i_1, i') \land LCOrder(i', i_2)) \]

The synchronization mechanism is formally captured in condition \( \text{synchronized} \). Instruction \( i_1 \) can be synchronized with a previous instruction \( i_2 \) via a release/acquire process, where a lock is first released by \( t(i_2) \) after \( i_2 \) is issued and later acquired by \( t(i_1) \) before \( i_1 \) is issued. Release can be triggered by an Unlock or a ReadVolatile instruction. Acquire can be triggered by a Lock or a ReadVolatile instruction.

\( \text{synchronized}(i_1, i_2) \equiv \)

\[ \exists l, u \in \text{GIB} : \]
\[ (op(l) = \text{Lock} \land op(u) = \text{Unlock} \land lock(l) = lock(u) \land \]
\[ op(l) = \text{ReadVolatile} \land op(u) = \text{WriteVolatile} \land var(l) = var(u) \land \]
\[ t(l) = t(i_1) \land t(u) = t(i_2) \lor t(i_2) = t_{\text{init}} \land \]
\[ \text{time}(i_2) \leq \text{time}(u) \land \text{time}(u) < \text{time}(l) \land \text{time}(l) \leq \text{time}(i_1) \]
After establishing the ordering relationship by condition $LCO_{\text{Order}}$, the requirement of serialization is enforced in $\text{legalNormalWrite}$. Informally, a write $w$ cannot provide its value to a read $r$ if there exists another overwriting write $w'$ in the ordering path.

$$\text{legalNormalWrite}(r, w) \equiv \begin{align*}
\text{op}(w) &= \text{WriteNormal} \land \text{var}(w) = \text{var}(r) \land \\
(t(w) &= t(r) \rightarrow pc(w) < pc(r)) \land \\
(\neg \exists w' \in \text{GIB} : \text{op}(w') = \text{WriteNormal} \land \text{var}(w') = \text{var}(r) \land LCO_{\text{Order}}(r, w') \land LCO_{\text{Order}}(w', w))
\end{align*}$$

The mutual exclusion effect of $\text{Lock}$ and $\text{Unlock}$ 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$_{\text{MP}}$ 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$_{\text{MP}}$ 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 $\text{ReadVolatile}$ and $\text{WriteVolatile}$ operations. The bypassing table shown in Table IV prohibits any reordering among volatile operations. Condition $\text{legalVolatileWrite}$, which follows $\text{legalWrite}$ in Sequential Consistency, defines the legal results for $\text{ReadVolatile}$ operations.

$$\text{legalVolatileWrite}(r, w) \equiv \begin{align*}
\text{op}(w) &= \text{WriteVolatile} \land \text{var}(w) = \text{var}(r) \land \\
(\neg \exists w' \in \text{GIB} : \text{op}(w') = \text{WriteVolatile} \land \text{var}(w') = \text{var}(r) \land \\
& t(r) > t(w') \land \text{time}(w') > \text{time}(w))
\end{align*}$$

### 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$_{\text{MP}}$ 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$_{\text{MP}}$ 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 JMM 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 JMMMP in that it only requires the final field itself to be a
costant 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 final field is initialized.

$$legalFinalWrite(r, w) \equiv$$
$$op(w) = WriteFinal \land var(w) = var(r) \land$$
$$t(w) = t_{init} \rightarrow$$
$$(\neg \exists i_1 \in GIB : op(i_1) = Freeze \land var(i_1) = var(r)) \land$$
$$\neg \exists i_2 \in GIB : op(i_2) = WriteFinal \land var(i_2) = var(r) \land t(i_2) = t(r))$$

6. ANALYSIS OF JMMMP VIA MODEL CHECKING

After adapting JMMMP, 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 JMMCRF
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 JMM is available at

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 JMMMP 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 JMMMP 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 JMMMP but forbidden by JMMCRF. This reveals a difference between the
two models. A more thorough analysis of JMMCRF indicates that the requirement of write atomicity
in JMMCRF 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 JMMMP, a legal interleaving that allows such a behavior is detected, proving
that JMMMP does not enforce Causal Consistency.
Initially, \( a = 0 \)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = 1; )</td>
<td>( a = 2; )</td>
</tr>
<tr>
<td>( r1 = a; )</td>
<td>( r3 = a; )</td>
</tr>
<tr>
<td>( r2 = a; )</td>
<td>( r4 = a; )</td>
</tr>
</tbody>
</table>

Finally, can it result in \( r1 = 1, r2 = r3 = 2, \) and \( r4 = 1 \)?

Figure 6. Write atomicity test.

Initially, \( a = b = 0 \)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = 1; )</td>
<td>( r1 = a; )</td>
<td>( r2 = b )</td>
</tr>
<tr>
<td>( b = 1; )</td>
<td>( r3 = a )</td>
<td>( r3 = a )</td>
</tr>
</tbody>
</table>

Finally, can it result in \( r1 = r2 = 1 \) and \( r3 = 0 \)?

Figure 7. Causality test.

Initially, \( a = 0 \)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r1 = a; )</td>
<td>( r2 = a; )</td>
</tr>
<tr>
<td>( a = 1; )</td>
<td>( a = r2; )</td>
</tr>
</tbody>
</table>

Finally, can it result in \( r1 = 1 \) and \( r2 = 1 \)?

Figure 8. Prescient write test.

**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 JMMMP. 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 hypothetic 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 JMMCRF, where *Membar1* is a special *EndCon* instruction used in JMMCRF 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 JMMCRF but fails under JMMMP. In JMMMP, the default write
Initially, reference = field = 0

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>field = 1; Membar1; reference = 1;</td>
<td>r1 = reference; Membar2; r2 = field;</td>
</tr>
</tbody>
</table>

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

Figure 9. Constructor test.

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\textsuperscript{CRF} treats all fields uniformly and JMM\textsuperscript{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\textsuperscript{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 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 \( ops \equiv \)
\( \forall t \in T. (\exists \text{order}. \) 
requireProgramOrder (restrictThread \( ops \) \( t \)) \( \text{order} \land \) 
requireWeakTotalOrder (restrictThread \( ops \) \( t \)) \( \text{order} \land \) 
requireTransitiveOrder (restrictThread \( ops \) \( t \)) \( \text{order} \land \) 
requireAsymmetricOrder (restrictThread \( ops \) \( t \)) \( \text{order} \land \) 
requireReadValue (restrictThread \( ops \) \( t \)) \( \text{order} \))

requireProgramOrder \( ops \) \( \text{order} \) \( \equiv \)
\( \forall i, j \in \text{ops}. ((t_i = t_j \land \text{pc}_i < \text{pc}_j) \lor (t_i = t_{\text{init}} \land t_j \neq t_{\text{init}})) \Rightarrow \text{order} \ i \ j \)

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 the GIB in any arbitrary order and then imposes 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. [28–33]. 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 et al. [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 JMM. 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. This work was supported in part by Research Grant No. CCR-0081406 (ITR Program) of NSF and SRC Task 1031.001.

REFERENCES