Specifying Java Thread Semantics Using a Uniform Memory Model

Yue Yang
School of Computing
University of Utah
yyang@cs.utah.edu

Ganesh Gopalakrishnan
School of Computing
University of Utah
ganesh@cs.utah.edu

Gary Lindstrom
School of Computing
University of Utah
gary@cs.utah.edu

ABSTRACT

Standardized language level support for threads is one of the most important features of Java. However, defining the Java Memory Model (JMM) has turned out to be a major challenge. Several models produced to date are not as easily comprehensible and comparable as first thought. Given the growing interest in multithreaded Java programming, it is essential to have a sound framework that would allow formal specification and reasoning about the JMM.

This paper presents the Uniform Memory Model (UMM), a generic memory model specification framework. Based on guarded commands, UMM can be easily integrated with a model checking utility, providing strong built-in support for formal verification and program analysis. With a flexible and simple architecture, UMM is configurable to capture different memory consistency requirements for both architectural and language level memory models. An alternative formal specification of the JMM, primarily based on the semantics proposed by Manson and Pugh, is implemented in UMM. Systematic analysis has revealed interesting properties of the proposed semantics. Inconsistencies from the original specification have also been uncovered.

1. INTRODUCTION

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 development, especially for server applications. The Java Memory Model (JMM), which specifies how threads interact in a concurrent environment, is a critical component of the Java threading system. It imposes significant implications on a broad range of engineering activities such as programming pattern development, compiler optimization, Java virtual machine (JVM) implementation, and architectural design.

Unfortunately, developing a rigorous and intuitive JMM has turned out to be a major challenge. The existing JMM, given in Chapter 17 of the Java Language Specification [11], is flawed and very hard to understand [18]. Its strong ordering constraint prohibits many common optimizations and the lack of guarantee for constructors may cause security problems. Two replacement semantics have been proposed to improve the JMM, one by Manson and Pugh [15] [16], the other by Maessen, Arvind, and Shen [14]. We refer to these two proposals as JMM_{MP} and JMM_{CRF} respectively in this paper. The JMM is currently under an official revisionary process [2]. There is also an ongoing discussion through the JMM mailing list [1].

Although the new proposals have initiated promising improvements on Java thread semantics, their specification frameworks can be enhanced in several ways. One area of improvement is the support for formal verification. Multithreaded programming is notoriously hard. The difficulty of being able to understand and reason about the JMM has become a major obstacle for Java threading to realize its full potential. Although finding an ultimate solution is not an easy task, integrating formal verification techniques with the specification framework does provide an encouraging first step towards this goal.

Another problem is that none of the proposals can be easily configured to support different desired consistency requirements since they are somewhat limited to their specific data structures. JMM_{CRF} inherits the design from its predecessor hardware model [20]. Java operations have to be divided into fine grained Commit/Reconcile/Fence (CRF) instructions to capture the precise thread semantics. This translation process adds unnecessary complexities for describing memory properties. The dependency on cache based architecture also prohibits JMM_{CRF} from describing more relaxed models. JMM_{MP} uses *sets* to record the history of memory operations. Instead of explicitly specifying the intrinsic memory model properties, e.g., the ordering rules, it resorts to nonintuitive mechanisms such as splitting a write instruction and using assertions to enforce proper conditions. While the notation using *sets* is sufficient to express the proposed semantics, adjusting it to specify different properties is not trivial. Similar to any software engineering activities, designing a memory model involves a repeated process of testing and fine-tuning. Therefore, a generic specification framework is needed to provide such flexibilities. In addition, a uniform notation is desired to help people understand similarities as well as distinctions between different models.

In this paper, we present the Uniform Memory Model (UMM), a formal framework for specifying the abstraction of

.

memory consistency requirements. Integrated with a model checking tool, it facilitates formal analysis of corner cases. It also extends the scope of traditional memory models by including the state information of thread local variables, enabling source level reasoning about program behaviors. Using a simple architecture, UMM explicitly specifies the primitive memory model properties and allows one to configure them at ease. An alternative specification of the JMM, primarily based on the semantics from JMM_{MP}, is captured in UMM. Subtle inconsistencies in JMM_{MP} have been revealed.

We discuss the related work in the next section. Then we review the problems of the current JMM specification. It is followed by an introduction of JMM_{MP}. Our formal specification of the JMM in UMM is described in Section 5. In Section 6, we discuss interesting results and compare JMM_{MP} with JMM_{CRF}. We conclude and explore future research opportunities in Section 7.

2. RELATED WORK

A *memory model* describes how a memory system behaves on memory operations such as reads and writes. Much previous research has concentrated on the processor level consistency models. One of the strongest memory models is *Sequential Consistency* [13]. Many weaker models [3] have been proposed to enable optimizations. One of them is *Lazy Release Consistency* [12], where synchronization operations such as *release* and *acquire* are treated differently from normal memory operations. When *release* is performed, previous memory activities need to be recorded to the shared memory. A reading process reconciles with the shared memory to obtain the updated data when *acquire* is issued. *Lazy Release Consistency* enforces an ordering property called *Coherence* that requires a total order among all write instructions at each individual address. The requirement of *Coherence* is further relaxed by *Location Consistency* [7]. Operations in *Location Consistency* are only partially ordered if they are from the same process or if they are synchronized through a lock. With the verification capability in UMM, we can formally compare the JMM with the conventional models.

To categorize different memory models, Collier [4] described them based on a formal theory of memory ordering rules. Using methods similar to Collier's, Gharachorloo et al. [9] [10] developed a generic framework for specifying the implementation conditions for different memory consistency models. The shortcoming of their approach is that it is nontrivial for people to infer program behaviors from a compound of ordering constraints with a declarative specification style. Park and Dill [6] developed an executable model with formal verification capabilities for the *Relaxed Memory Order* [17]. We extended this approach to the domain of the JMM in our previous work on the analysis of JMM_{CRF} [22]. After adapting JMM_{CRF} to an executable specification implemented with the Murφ model checking tool, we systematically exercised the underlying model with a suite of test programs to reveal pivotal properties and verify common programming idioms. Our analysis demonstrated the feasibility and effectiveness of using the model checking technique to understand language level memory models. Roychoudhury and Mitra [19] also applied the same technique to study the existing JMM, achieving similar success. Limited to the specific designs of the target model, however, none of the existing executable specification is generic. UMM improves the method by providing a generic abstraction mechanism for capturing different memory consistency requirements into a formal executable specification.

3. PROBLEMS OF THE EXISTING JMM

In the existing JMM, every variable has a *working copy* stored in the *working memory*. Threads communicate through the *main memory*. 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. As a result, this framework is flawed and hard to understand. Some of the major issues are listed as follows.

- The existing JMM requires a total order for memory operations on each individual variable. Because of the strong ordering restriction, important compiler optimization techniques such as *fetch elimination* are prohibited [18].
- The existing JMM requires a thread to flush all variables to main memory before releasing a lock, imposing a strong requirement on visibility effect. Consequently, some seemingly redundant synchronization operations such as the thread local synchronization blocks cannot be optimized away.
- The ordering guarantee for a constructor is not strong enough. On some 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. This loophole compromises Java safety since it opens the security hole to malicious attacks via race conditions.
- Semantics for *final* variable operations is omitted.
- *Volatile* variable operations specified by the existing JMM do not have synchronization effects for normal variable operations. Therefore, volatile variables cannot be effectively applied as synchronization flags to indicate the completion of normal operations.

4. PROPOSAL BY MANSON AND PUGH

To improve the JMM, JMM_{MP} is proposed as a replacement semantics for Java threads. After extensive discussions and debates through the JMM mailing list, some of the thread properties have emerged as leading candidates to appear in the new JMM.

4.1 Desired Properties

- The ordering constraint should be relaxed to enable common optimizations. JMM_{MP} essentially follows *Location Consistency* that does not require *Coherence*.
- The synchronization requirement should be relaxed to enable the removal of redundant synchronization blocks. In JMM_{MP}, 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. If the final variable is "improperly" exposed to other threads before it is frozen, v is said to be a *pseudo-final* field. Another thread would always observe the initialized value of v unless it is pseudo-final, in which case it can also obtain the default value.

- Volatile variables should be specified to be more useful for multithreaded programming. JMM_{MP} proposes to add the release/acquire semantics to volatile variable operations to achieve synchronization effects for normal variables. A write to a volatile field acts as a release and a read of a volatile field acts as an acquire.

  The exact ordering requirement for volatile operations is still under debate. Some people have proposed to relax it from *Sequential Consistency* to allow *non-atomic* volatile write operations.

4.2 JMM_{MP} Notations

JMM_{MP} is based on an abstract global system that executes one operation from one thread in each step. An operation corresponds to a JVM opcode, which occurs 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.

4.2.1 Data Structures

A *write* is defined as a tuple of $\langle variable, value, GUID \rangle$, uniquely identified by its global ID *GUID*. JMM_{MP} uses the *set* data structure to store history information of memory activities. In particular, *allWrites* is a global set that records all write events that have occurred. Every thread, monitor, or volatile variable k also maintains two local sets, $overwritten_k$ and $previous_k$. The former stores the obsolete writes that are known to k. The latter keeps all previous writes that are known to k. When a new write is issued, writes in the thread local *previous* set become obsolete to that thread and the new write is added to the *previous* set as well as the *allWrites* set. When a read action occurs, the return value is chosen from the *allWrites* set. But the writes stored in the *overwritten* set of the reading thread are not eligible results.

4.2.2 Prescient Write

A write w may be performed presciently, i.e., executed early, if (a) w is guaranteed to happen, (b) w cannot be read from the same thread before where w would normally occur, and (c) any premature reads of w from other threads must not be observable by the thread that issues w via synchronization before where w would normally occur. To capture the prescient write semantics, a write action is split into initWrite and performWrite. A special assertion is used in performWrite to ensure that proper conditions are met.

4.2.3 Synchronization Mechanism

The thread local *overwritten* and *previous* sets are synchronized between threads through the release/acquire process. A release operation passes the local sets from a thread to a monitor. An acquire operation passes the sets associated with a monitor to a thread.

4.2.4 Non-atomic Volatile Writes

JMM_{MP} does not require *write atomicity* for volatile write operations. To capture this relaxation, a volatile write is split into two consecutive instructions, initVolatileWrite and performVolatileWrite. Special assertions are also used to impose proper conditions.

4.2.5 Final Field 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 elements it points to do not have the same guarantee. A very tricky issue arises from the fact that Java does not allow array elements to be declared as final fields. JMM_{MP} proposes to add a special guarantee to those non-final subfields that are referenced by a final field: if such a sub-field is assigned in the constructor, its default value cannot be observed by another thread after normal construction. Using this constraint, an immutable object can be implemented by declaring all its fields as final.

Adding this special requirement substantially complicates JMM_{MP} because synchronization information needs to be passed between the constructing thread and every object pointed by a final field. The visibility status of the subfields must be captured when the final field is frozen and later synchronized to another thread when these elements are dereferenced. Therefore, every final variable v is treated as a special lock. A special release is performed when v is frozen. Subsequently, an acquire is performed when v is read to access its sub-fields. The variable structure is extended to a *local*, which is a tuple of $\langle a, oF, kF \rangle$, where a is the reference to an object or a primitive value, oF is the overwritten set caused by freezing the final fields, and kF is a set recording what variables are known to have been frozen. Whenever a final object is read, its $knownFrozen$ set associated with its initializing thread is synchronized to the reference of the final object. This allows any subsequent read operations to know if the sub-field has been initialized.

5. SPECIFYING JMM_{MP} USING UMM

Given the complicated nature of the JMM, it is vital to have a systematic approach to analyze a complex proposal. We have defined an alternative formal specification of the JMM using the UMM framework. The core semantics, including definitions of normal memory operations and synchronization operations, is based on JMM_{MP} [15] as of January 11, 2002.

5.1 Overview

The UMM uses an abstract transition system to define thread interactions in a shared memory environment. Memory instructions are categorized as *events* that may be completed by carrying out some *actions* when certain *conditions* are satisfied. A *transition table* defines all possible events along with their corresponding conditions and actions. At a given step, any legal event may be nondeterministically chosen and atomically completed by the abstract machine. The sequence of permissible actions from various threads based on proper ordering constraints constitutes an *execution*. A *legal execution* is defined as a *serialization* of these memory operations, i.e., a read operation returns the value from the most recent previous write operation on the same variable.

A memory model $\mathcal{M}$ is defined by the legal results perceived by each read operation in legal executions. 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 $\mathcal{M}$, i.e., the return values for read operations generated by $\mathcal{I}_{\mathcal{M}}$ are also valid according to $\mathcal{M}$.

5.2 The Architecture

As shown in Figure 1, each thread k has a *local instruction buffer* LIB_k that stores its pending instructions in program order. It also maintains a set of local variables in a *local variable array* LV_k. Each element $\mathsf{LV}_k[v]$ contains the data value of the local variable v. Thread interactions are communicated through a *global instruction buffer* GIB, which is visible to all threads. GIB stores all previously completed memory instructions that are necessary for fulfilling a future read request. A read instruction completes when the return value is bound to its target local variable. A write or synchronization instruction completes when it is added to the global instruction buffer. A multithreaded program terminates when all instructions from all threads complete. For clarity, UMM also uses a dedicated global *lock array* LK to maintain locking status. Each element $\mathsf{LK}[l]$ is a tuple $\langle count, owner\rangle$, where *count* is the number of recursive lock acquisitions and *owner* is the owning thread.

Only two layers are used in UMM, one for thread local information and the other for global trace information. The devices applied in UMM, such as instruction buffers and arrays, are standard data structures that are easy to understand. Some traditional frameworks use multiple copies of the shared memory modules to represent non-atomic operations [9]. In UMM, these multiple modules are combined into a single global buffer that substantially simplifies state transitions. Any completed instructions that may have any future visibility effects are stored in the global instruction buffer along with the time stamps of their occurrence. This allows one to plug in different selection algorithms to observe the state. In a contrast to most processor level memory models that use a fixed size main memory, UMM applies a global instruction buffer whose size may be increased if necessary, which is needed to specify relaxed memory models that require to keep a trace of multiple writes on a variable.

The usage of LIB and GIB is motivated by the need for expressing *program order* and *visibility order*, two pivotal properties for understanding thread behaviors. *Program order* is the original instruction order of each thread determined by software. It can be relaxed in certain ways by a memory model to allow optimizations. UMM uses a bypassing table that controls the program order relaxation policy to enable necessary interleaving. *Visibility order* is the observable order of memory activities perceived by one or more threads. The legal executions of a memory model are enforced by imposing corresponding ordering requirements on the elements stored in GIB. Based on these ordering constraints, illegal results are filtered out by the requirement of serialization. With this mechanism, 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.

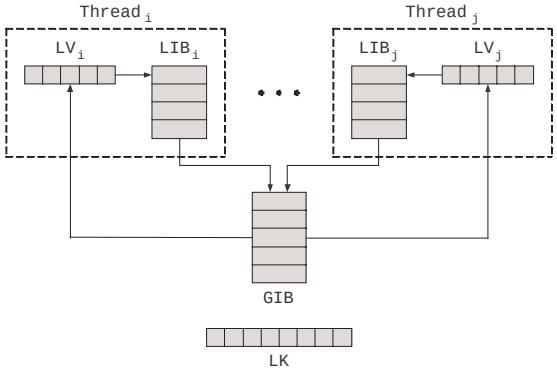

Figure 1: The UMM Architecture

5.3 Variables and Instructions

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 in Java. It can be further categorized as a *normal*, *volatile*, or *final* variable. A *local variable* in UMM corresponds to a Java local variable or an operand stack location. An *instruction* i is represented by a tuple $\langle t, pc, op, var, data, local, useLocal, lock, time\rangle$, where

$t(i) = t$:	issuing thread;
$pc(i) = pc$:	program counter of i;
$op(i) = op$:	operation type of i;
$var(i) = var$:	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.

In Java, a variable has a form of object.field, where object is the object reference and field is the field name. The object.field entity can be represented by a single variable v. In our examples, we also 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.

5.4 Need for Local Variable Information

Because traditional memory models are designed for processor level architectures, aiding software program analysis is not a common priority in those specifications. Consequently, a read instruction is usually retired immediately when the return value is obtained. Following the same style, neither JMM_{MP} nor JMM_{CRF} keeps the return values from read operations. However, Java poses a new challenge to memory model specification by integrating the threading system as part of the programming language. In Java, most programming activities, such as computation, flow control, and method invocation, are carried out using local variables. Programmers have a clear need for understanding memory model implications caused by the nondeterministically returned values in local variables. Therefore, it is desired to extend the scope of the memory model framework by recording the values obtained from read instructions as part of the global state of the transition system.

Based on this observation, we include local variable arrays in UMM. This allows us to define the JMM at both the byte code level and the source code level, providing an end-to-end view of the memory consistency requirement. It also provides a clear delimitation between local data dependency and memory model ordering requirements.

The Java language specification requires that local variables must be assigned before being used. Combined with the data dependency constraint, it guarantees that a value observed by a read operation on a global variable must have been written by some write operation.

5.5 Memory Operations

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 write instructions have $useLocal = false$ and all read instructions use nonconflicting local variables, the UMM degenerates to the traditional models that do not keep local variable information.

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 full blown program analysis system by adding semantics for computational instructions.

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 v has been frozen.

All memory operations are defined in the transition table in Table 1. Our notation based on guarded commands has been widely used in many architectural models [8] [21], making it familiar to hardware designers.

5.6 Initial Conditions

Initially, LIB contains instructions from each thread in their original program order. GIB is initially cleared. Then for every variable v, a write instruction with the default value of v and a special thread ID t_{init} is added to GIB. The *count* fields in LK are set to 0.

After the abstract machine is set up, it operates according to the transition table.

5.7 Data Dependency and Bypassing Rules

Data dependency 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.

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

The bypassing rules provide the flexibility for generating certain interleaving such as *prescient writes*. Table BYPASS as shown in Table 2 specifies the ordering policy between every pair of instructions. 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*, and *redundantLock*. The value *yes* permits the bypassing and the value *no* prohibits it. The value *redundantLock* conditionally enables the bypassing only if the first instruction operating on a lock or a volatile variable does not impose any synchronization effect, e.g., when the lock is thread local or nested.

Since JMM_{MP} respects program order except for *prescient writes*, the UMM specification follows the same guideline by only allowing normal write instructions to bypass certain previous instructions. Because different threads are only synchronized through the *same* lock in JMM_{MP}, a thread local or nested lock can be removed and no memory ordering restrictions should be imposed in such cases. Hence, a WriteNormal instruction can bypass a previous Lock or ReadVolatile instruction when the Lock or ReadVolatile instruction does not impose any synchronization effect. As a result, decisions about reordering across a synchronization boundary have to involve global analysis in order to take advantage of all relaxations allowed by JMM_{MP}.

Condition *ready*, required by every event in the transition table, enforces local data dependency as well as the bypassing policy of the memory model. The helper function $notRedundant(j)$ returns *true* if instruction j does have a synchronization effect.

$$
\begin{aligned}
&ready(i) \iff \\
&\neg \exists j \in \text{LIB}_{t(i)} : pc(j) < pc(i) \ \wedge \\
&(localDependent(i,j) \ \vee \\
&\text{BYPASS}[op(j)][op(i)] = no \ \vee \\
&\text{BYPASS}[op(j)][op(i)] = \ redundantLock \ \wedge \ notRedundant(j))
\end{aligned}
$$

5.8 Visibility Rules

JMM_{MP} applies an ordering constraint similar to *Location Consistency*. As captured in condition *LCOrder*, two instructions are ordered if and only 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 is another operation that can transitively establish the order.

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

The synchronization mechanism, formally captured in condition *synchronized*, plays an important role in selecting legal return values for normal variables. 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

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

Table 1: Transition Table

2nd ⇒ 1st ⇓	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	redundantLock	no	no	no	no	no	no	no
Unlock	no	yes	no	no	no	no	no	no	no
Read Volatile	no	redundantLock	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 2: The Bypassing Table (Table BYPASS)

instruction. Acquire can be triggered by a Lock or a ReadVolatile instruction.

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

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

The requirement of serialization is enforced by condition *legalNormalWrite*. A write instruction w cannot provide its value to a read instruction r if there exists another intermediate write instruction w' on the same variable between r and w in the ordering path.

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

5.9 Volatile Variable Semantics

Consensus of the ordering requirement for volatile variable operations has not been reached. While some people suggest to follow *Sequential Consistency*, other people propose to relax *write atomicity* to support future memory architectures. Because it is still not clear what will eventually be used by the future JMM, we define volatile variable operations based on *Sequential Consistency* in this paper. The flexibility of UMM should allow it to capture different semantics using alternative ordering rules.

Volatile variable operations are issued according to their program order since the bypassing table prohibits any reordering among them. Condition *legalVolatileWrite* defines the legal results for ReadVolatile operations. Similar to condition *legalNormalWrite*, it guarantees serialization by eliminating obsolete write instructions in the ordering path. Instead of *LCOrder*, *SCOrder* is used to enforce *Sequential Consistency*. This illustrates how different memory models can be parameterized by the ordering rules.

$legalVolatileWrite(r, w) \iff$
$op(w) = \text{WriteVolatile} \ \wedge\ var(w) = var(r) \ \wedge$
$(\neg\exists w' \in \text{GIB} : op(w') = \text{WriteVolatile} \ \wedge\ var(w') = var(r) \ \wedge$
$SCOrder(r, w') \ \wedge\ SCOrder(w', w))$

$SCOrder(i1, i2) \iff$
$time(i1) > time(i2)$

5.10 Final Variable Semantics

As mentioned in Section 4.2.5, JMM_{MP} proposes to add a special guarantee for the normal sub-fields pointed by a final field. 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 difficult to understand because synchronization information needs to be carried by every variable. It is also not clear how the exact proposed semantics can be efficiently implemented by a Java compiler or a JVM since it involves run-time reachability analysis.

While still investigating this issue and trying to find the most reasonable solution, we implement a straightforward definition for final fields in the current UMM. It is slightly 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 final field is initialized.

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

5.11 Control Dependency Issues

In UMM, thread local data dependency is formally defined in *localDependent*. In addition, control dependency on local variables should also be respected to preserve the meaning of the Java program. However, how to handle control dependency is a tricky issue. A compiler might be able to remove a branch statement if it can determine the control path through program analysis. A policy about reordering instructions across a control boundary needs to be set by the JMM.

JMM_{MP} identifies some special cases and adds two more read actions, guaranteedRedundantRead and guaranteedReadOfWrite that can suppress prescient writes to enable *redundant load elimination* and *forward substitution* under specific situations involving control statements. For example, the need for guaranteedRedundantRead is motivated by a program in Figure 2. In order to allow $r2 = a$ to be replaced by $r2 = r1$ in Thread 1, which would subsequently allow the removal of the if statement, $r2 = a$ must be guaranteed to get the previously read value.

Initially, a = b = 0

Thread 1	Thread 2
r1 = a;	r3 = b;
r2 = a;	a = r3;
if(r1 == r2)	
b = 2;	

Finally,
can r1 = r2 = r3 = 2?

Figure 2: Need for guaranteedRedundantRead

Although we could follow the same style in UMM, we do not believe it is a good approach to specify a memory model by enumerating special cases for every optimization need. Therefore, we propose a clear and uniform policy regarding control dependency: the compiler may remove a control statement only if the control condition can be guaranteed in *every* possible execution, including all interleaving results

caused by thread interactions. This approach should still provide plenty of flexibility for compiler optimizations. If desired, global data flow analysis may be performed. UMM offers a great platform for such analysis. One can simply replace a branch instruction with an assertion and let the model checker verify its validity.

5.12 Murφ Implementation

To make our JMM specification executable, our formal specification is implemented in Murφ [5], a description language with a syntax similar to C as well as a model checking system that supports exhaustive state space enumeration. Due to the operational style of UMM, it is straightforward to translate the formal specification to the Murφ implementation.

Our Murφ program consists of two parts. The first part implements the formal specification of JMM_{MP}, which defines Java thread semantics. The transition table in Table 1 is specified as Murφ rules. Bypassing conditions and visibility conditions are implemented as Murφ procedures. The second part comprises a suite of test cases. Each test program is defined by specific Murφ initial state and invariants. It is executed with the guidance of the transition system to reveal pivotal properties of the underlying model. Our system can detect deadlocks and invariant violations. The output can display a violation trace or show all possible interleaving results. Our executable specification is highly configurable, enabling one to easily set up different test programs, abstract machine parameters, and memory model properties. Running on a PC with a 900 MHz Pentium III processor and 256 MB of RAM, most of our test programs terminate in less than one second.

6. ANALYSIS OF JMM_{MP}

By systematically exercising JMM_{MP} with idiom-driven test programs, we are able to gain a lot of insights about the model. Since we have developed formal executable models for both JMM_{CRF} [22] and JMM_{MP}, we are able to perform a comparison analysis by running the same test programs on both models. This can help us understand subtle differences between them.

As an ongoing process of evaluating the Java Memory Model, we are continuing to develop more comprehensive test programs. In this section we highlight some of the interesting findings based on our preliminary results.

6.1 Ordering Properties

6.1.1 Coherence

JMM_{MP} does not require *Coherence* for normal variable operations. This can be detected by the program shown in Figure 3. If $r1 = 2$ and $r2 = 1$ is allowed, the two threads have to observe different orders of writes on the same variable a, which violates *Coherence*. For a normal variable a, this result is allowed by JMM_{MP} but prohibited by JMM_{CRF}.

Initially, $a = 0$

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

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

Figure 3: Coherence Test

6.1.2 Write Atomicity for Normal Variables

JMM_{MP} does not require *write atomicity* for normal variables. This can be revealed from the test in Figure 4. For a normal variable a, the result in Figure 4 is allowed by JMM_{MP} but forbidden by JMM_{CRF}. Because the CRF model uses the shared memory as the rendezvous point between threads and caches, it has to enforce *write atomicity*.

Initially, $a = 0$

Thread 1	Thread 2
$a = 1;$	$a = 2;$
$r1 = a;$	$r3 = a;$
$r2 = a;$	$r4 = a;$

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

Figure 4: Write Atomicity Test

6.1.3 Causality

Causal consistency requires thread local orderings to be transitive through a causal relationship. Using the test program shown in Figure 5 that illustrates a violation of causality, our verification system proofs that JMM_{MP} does not enforce *causal consistency* for normal variable operations.

Initially, $a = b = 0$

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

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

Figure 5: Causality Test

6.1.4 Prescient Write

Figure 6 reveals an interesting case of *prescient write* allowed by JMM_{MP}, where $r1$ in Thread 1 can observe a write that is initiated by a later write on the same variable from the same thread. Therefore, programmers should not assume that *antidependence* between global variable operations is always enforced.

Initially, $a = 0$

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

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

Figure 6: Prescient Write Test

6.2 Synchronization Mechanism

JMM_{MP} follows *Location Consistency*. When a read instruction is issued, any unsynchronized writes on the same variable issued by other threads can be observed, in any order. Therefore, JMM_{MP} is strictly weaker than *Lazy Release Consistency*. Without synchronization, thread interleaving

may result in very surprising results. An example is shown in Figure 7.

Initially, $a = 0$

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

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

Figure 7: Legal Result Under Location Consistency

6.3 Constructor Property

The constructor property is studied by the program in Figure 8. Thread 1 simulates the constructing thread. It 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 construction mechanism used by JMM_{CRF}, where $Membar1$ is a special EndCon instruction indicating the completion of the constructor and $Membar2$ is the data dependency enforced by program semantics when accessing $field$ through $reference$.

Initially, $reference = field = 0$

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

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

Figure 8: Constructor Test

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 exists 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} only guarantees fully initialized fields if they are final or pointed by final fields.

6.4 Inconsistencies in JMM_{MP}

6.4.1 Non-Atomic Volatile Writes

JMM_{MP} allows volatile write operations to be non-atomic. One of the proposed requirements for non-atomic volatile write semantics is that if a thread t has observed the new value of a volatile write, it can no longer observe the previous value. In order to implement this requirement, a special flag $readThisVolatile_{t,\langle w,info_t\rangle}$ is initialized to $false$ in initVolatileWrite [15, Figure 14]. When the new volatile value is observed in readVolatile, this flag should be set to $true$ to prevent the previous value from being observed again by the same thread. However, this critical step is missing and the flag is never set to $true$ in the original proposal. This omission causes inconsistency between the formal specification and the intended goal.

6.4.2 Final Semantics

A design flaw for final variable semantics has also been discovered. This is about a corner case in the constructor that initializes a final variable. The scenario is illustrated in Figure 9. After the final field a is initialized, it is read by a local variable in the same constructor. The readFinal definition [15, Figure 15] would allow r to read back the default value of a. This is because at that time a has not been "synchronized" to be known to the object that it has been frozen. But the readFinal action only checks that information from the kF set that is associated with the object reference. This scenario compromises program correctness because data dependency is violated.

```
class foo {
    final int a;

    public foo() {
        int r;
        a = 1;
        r = a;
        // can r = 0?
    }
}
```

Figure 9: Flaw in Final Variable Semantics

7. CONCLUSIONS

UMM provides a uniform framework for specifying memory consistency models in general, and the JMM in particular. It permits one to conduct formal analysis on the JMM and pave the way towards future studies on compiler optimization techniques in a multithreaded environment. UMM possesses many favorable characteristics as a specification and verification framework.

1. The executable specification style based on guarded commands enables the integration of model checking techniques, providing strong support for formal verification. Formal methods can help one to better understand the subtleties of the model by detecting corner cases that would be very hard to find through traditional simulation techniques. Because the specification is executable, the memory model can be treated as a "black box" and the users are not necessarily required to understand all the details of the model to benefit from the specification. In addition, the mathematical rules in the transition table makes the specification more rigorous, which eliminates any ambiguities.

2. UMM addresses the special need for a language level memory model by including the local variable states in the transition system. Not only does this reduce the gap between memory model semantics and program semantics, it also provides a clear delimitation between them.

3. The flexible design of UMM provides generic support for consistency protocols and enables easy configuration. The abstraction mechanism in UMM offers a feasible common design interface for executable memory models. Different bypassing rules and visibility ordering rules can be carefully developed so that a user can select from a "menu" of primitive memory properties to assemble a desired formal specification.

4. The simple architecture of UMM eliminates unnecessary complexities introduced by implementation specific data structures. Hence, it helps clarify the essential semantics of the memory system.

Designed as a specification framework, however, UMM is not intended to be actually implemented in a real system since it chooses simplicity over efficiency in its architecture. For example, UMM does not support some of the *nonblocking* memory operations that could be expressed by adding intermediate data structures and fine-grained memory activities.

Our approach also has some limitations. Based on model checking techniques, UMM is exposed to the state explosion problem. Effective abstraction and slicing techniques need to be applied in order to use UMM to verify real Java programs. In addition, our UMM prototype is still under development. The optimal definition for final variables needs to be identified.

A reliable specification framework may lead to many interesting future works. First, currently people need to develop the 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 methods can be systematically analyzed for JMM compliance. The UMM framework can also help one explore new optimization opportunities allowed by the relaxed consistency requirement. Third, architectural memory models can be specified in UMM under the same framework so that memory model refinement analysis can be performed to aid efficient JVM implementations. Finally, we plan to apply UMM to study the various proposals to be put forth by the Java working group in their currently active discussions regarding Java shared memory semantics standardization. The availability of a formal analysis tool during language standardization will provide the ability to evaluate various proposals and foresee pitfalls.

Acknowledgments

We sincerely thank all contributors to the JMM mailing list for their insightful and inspiring discussions about improving the Java Memory Model.

8. REFERENCES

[1] The Java Memory Model mailing list. http://www.cs.umd.edu/~pugh/java/memoryModel.

[2] Java Specification Request (JSR) 133: Java Memory Model and Thread Specification Revision. http://jcp.org/jsr/detail/133.jsp.

[3] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[4] W. W. Collier. *Reasoning about Parallel Architectures*. Prentice-Hall, 1992.

[5] D. Dill. The Murφ verification system. In *8th International Conference on Computer Aided Verification*, pages 390–393, 1996.

[6] D. Dill, S. Park, and A. Nowatzyk. Formal specification of abstract memory models. In *the 1993 Symposium for Research on Integrated Systems*, pages 38–52, March 1993.

[7] G. Gao and V. Sarkar. Location consistency - a new memory model and cache consistency protocol. Technical report, 16, CAPSL, University of Delaware, February, 1998.

[8] R. Gerth. Introduction to sequential consistency and the lazy caching algorithm. *Distributed Computing*, 1995.

[9] K. Gharachorloo. Memory consistency models for shared-memory multiprocessors. Technical report, CSL-TR-95-685.

[10] K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy, and M. D. Hill. Specifying system requirements for memory consistency models. Technical report, CSL-TR93-594.

[11] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*, chapter 17. Addison-Wesley, 1996.

[12] 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.

[13] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[14] J.-W. Maessen, Arvind, and X. Shen. Improving the Java Memory Model using CRF. In *OOPSLA*, pages 1–12, October 2000.

[15] J. Manson and W. Pugh. Semantics of multithreaded Java. Technical report, UMIACS-TR-2001-09.

[16] J. Manson and W. Pugh. Core semantics of multithreaded Java. In *ACM Java Grande Conference*, June 2001.

[17] S. Park and D. L. Dill. An executable specification and verifier for Relaxed Memory Order. *IEEE Transactions on Computers*, 48(2):227–235, 1999.

[18] W. Pugh. Fixing the Java Memory Model. In *Java Grande*, pages 89–98, 1999.

[19] A. Roychoudhury and T. Mitra. Specifying multithreaded Java semantics for program verification. In *International Conference on Software Engineering*, 2002.

[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*, Atlanta, Georgia, May 1999.

[21] D. Weaver and T. Germond. *The SPARC Architecture Manual Version 9*. Prentice Hall, 1994.

[22] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Analyzing the CRF Java Memory Model. In *the 8th Asia-Pacific Software Engineering Conference*, pages 21–28, 2001.