A Generic Operational Memory Model Specification Framework for Multithreaded Program Verification *

Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom

School of Computing, University of Utah
{yyang, ganesh, gary}@cs.utah.edu

Abstract. Given the complicated nature of modern architectural and language level memory model designs, it is vital to have a systematic approach for specifying memory consistency requirements that can support verification and promote understanding. In this paper, we develop a specification methodology that defines a memory model operationally using a generic transition system with integrated model checking capability to enable formal reasoning about program correctness in a multithreaded environment. Based on a simple abstract machine, our system can be configured to define a variety of consistency models in a uniform notation. We then apply this framework as a taxonomy to formalize several well known memory models. We also provide an alternative specification for the Java memory model based on a proposal from Manson and Pugh and demonstrate how to conduct computer aided analysis for Java thread semantics. Finally, we compare this operational approach with axiomatic approaches and discuss a method to convert a memory model definition from one style to the other.

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 better structured and high performance applications. Unlike a sequential program, where each read simply returns the value written by the most recent write according to program order, a multithreaded program relies on the *memory model* (also known as the *thread semantics*) to define how threads interact in a shared memory system.

The memory model plays a critical role in developing and debugging concurrent programs. Consider, for example, Peterson's algorithm [1] shown in Figure 1 designed to solve the two-thread mutual exclusion problem. 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

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

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

Thread 1                          Thread 2

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

Fig. 1. Peterson's algorithm for mutual exclusion.

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

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

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

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

turn determines which thread enters the critical section first. The correctness of this well known programming pattern, however, heavily depends on certain assumptions about 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. In Figure 2, if both threads can still observe the default flag values when the loop conditions are checked, they are able to enter the critical section at the same time. Suppose the underlying memory model permits such a program behavior (which happens to be the case for many shared memory systems), programs relying on Peterson's algorithm would be erroneous.

Program fragments such as the one in Figure 2 are generally known as *litmus tests*. Carefully studying these test programs can reveal critical memory model properties, which is very helpful for programmers and compiler writers to make right decisions in code selection and optimization. For simple cases, one can often follow a pencil-and-pen approach to reason about the legality of a litmus test. But as bigger programs are involved and more complex models are used, 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].

Multithreaded programming is notoriously difficult. Developing efficient and reliable compilation techniques for multithreaded programs is also hard. Given that memory model compliance is a prerequisite for developing robust concurrent systems, it is crucial to have a rigorous methodology for analyzing memory model specifications. In this paper, we present a generic specification framework that provides integrated model checking capability. We also demonstrate how to apply this framework to perform multithreaded program verification.

1.1 Memory Model Overview

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* is the final observable order of memory operations perceived by one or more threads (this is similar to the notion of visibility order used in [3] and the notion of *before order* used in [4]).

Memory model designs typically involve a tradeoff between programmability and efficiency. For example, 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.[1] Many weaker memory models (see [6] for a survey) have since been proposed to enable higher performance implementations. Some of these models still require *Coherence* [7] (also known as *Cache Coherence* or *Cache Consistency*). Informally, Coherence requires all operations involving a given variable to exhibit a total order that also respects program order of each thread.

Early memory models, such as Sequential Consistency, are designed for single bus systems, where a common visibility order is enforced for all observing threads. Some other models, such as *Parallel Random Access Memory (PRAM)* [8], allow each individual thread to observe its own visibility order. Informally, PRAM requires the execution sequences containing all operations from the reading thread and all write operations from other threads to exhibit a total order that also respects program order of each thread. For example, the outcome of the program shown in Figure 3 is not allowed by Sequential Consistency and Coherence since there does not exist a common visibility order that can be agreed upon by both threads. However, the behavior is permitted by PRAM because each thread can perceive its own visibility order. That is, thread 1 and thread 2 may observe interleaving (1)(3)(2) and (3)(1)(4), respectively. Using our framework, we can formally analyze the behaviors of such test programs guided by various memory models.

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

Initially, $a = 0$

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

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

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

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.

1.2 The Existing Java Memory Model

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

The existing Java memory model is given in Chapter 17 of the Java Language Specification [11]. It specifies that every variable has a *working copy* stored in the *working memory*. Threads communicate via 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, however, non-obvious implications can be deduced by combining different rules [12]. As a result, the existing Java memory model is flawed and hard to understand. Some of its major issues are listed as follows.

- The model requires Coherence. Because of this restriction, important compiler optimizations such as *fetch elimination* are prohibited (see [12] for a detailed example).
- 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 for *final* variable operations is omitted.
– *Volatile* variable operations specified by the existing Java memory model do not have synchronization effects on normal variable operations. Consequently, volatile variables cannot be applied as synchronization flags to indicate the completion of non-volatile variable operations.

1.3 A Java Memory Model Proposed by Manson and Pugh

Several improved Java thread semantics have been proposed, including a proposal from Manson and Pugh [13, **?**] (referred to as JMM_{MP} in this paper). Since the core semantics of JMM_{MP} is adapted as a concrete case study in this paper, we briefly describe JMM_{MP} here as an overview. Readers are referred to [13] for more details.

JMM_{MP} is based on an abstract global system that executes one operation from one thread at 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. A *write* is defined as a tuple of $\langle variable, value, GUID \rangle$, uniquely identified by its global ID $GUID$. JMM_{MP} uses *set* 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. A write w may be performed early under certain situations. 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. To solve the Java memory model problems listed in Section 1.2, JMM_{MP} proposes the following thread properties and mechanisms for achieving them.

– The ordering constraint should be relaxed to enable common optimizations. JMM_{MP} essentially follows Location Consistency, which does not require Coherence.
– The synchronization mechanism should be relaxed to enable the removal of redundant synchronization blocks. In JMM_{MP}, visibility states are only synchronized through the *same* lock. The thread local *overwritten* and *previous* sets are synchronized between threads through release/acquire actions. An

unlock acts as a release, which passes the local sets from a thread to a monitor. A *lock* acts as an acquire, which passes the sets associated with a monitor to a thread.
- 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. JMM_{MP} allows volatile write operations to be non-atomic. 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.

1.4 Summary of Results

We present a Uniform Memory Model (UMM) specification framework, which uses an abstract machine associated with a transition table to execute thread instructions in an operational style. Coupled with a model checking utility, it can exhaustively exercise a test program to cover all thread interleavings. Furthermore, the simple and flexible design of the system enables one to define different thread semantics by crafting a customized transition table. These advantages make UMM suitable as a generic formalism for creating executable memory model definitions. Our main insight is that by using two separate kinds of buffers, namely *local instruction buffers* (LIB) and *global instruction buffers* (GIB), we can separately capture requirements on program order and visibility order, two *pivotal* properties for understanding shared memory thread behaviors. Relaxations of the program order are configured through a bypassing table and rules in first-order logic are used to express these bypassing policies. Completed instructions in GIB are used to construct the legal visibility order subject to certain visibility ordering rules. With this approach, variations between memory models can be isolated into a few well-defined places such as the bypassing table and the visibility ordering rules, enabling easy comparison and configuration.

We offer the following contributions in this paper. First, we develop a generic transition system that can be used to produce executable memory model specifications. One main result in this paper is to show that this particular design of an operational model can capture not only architectural level memory models but also language level memory models. No previous work has treated both these categories of memory models in a uniform framework; yet, the importance of doing so is growing, especially with the advent of multiprocessor machines

on whose architectural memory models one has to support the language level memory model in the most efficient manner. Second, we apply this framework as a taxonomy to formalize a variety of well known memory model properties and use those executable specifications to perform program verification. Third, we provide an alternative Java thread specification, based on the semantics proposed by Manson and Pugh, and demonstrate how to conduct automated verification for a complex language level memory model. Inconsistencies from the proposed semantics are also uncovered. Finally, we discuss the relationship of our operational specification approach with trace-based axiomatic specification approaches and propose a mechanism to transform a memory model definition from one style to the other.

Road Map In the next section, we discuss the related work. Then we present an overview of our specification framework in Section 3. In Section 4, we show how to apply our approach to formalize several well known memory model properties. Our alternative formal specification of the Java memory model is described in Section 5. In Section 6, we provide a thorough analysis of JMM_{MP}. We conclude and explore future research opportunities in Section 7.

2 RELATED WORK

Extensive research has been conducted in the area of model checking based verification of multithreaded Java programs, for example [15, **?**,**?**,**?**,**?**]. A tool was also developed in [20] to analyze Java byte code. These efforts, however, do not specifically address the Java memory model issues. The memory operations analyzed in these tools are interpreted using sequentially consistent behaviors instead of a strict memory model. Therefore, they cannot be used to analyze fine-grained thread interleavings under race conditions. We can imagine our proposed Uniform Memory Model being incorporated into these tools to make their analysis more realistic. In [21], a type-based framework for race analysis is proposed. These authors also seem to tacitly assume Sequential Consistency.

The Java memory model problems were pointed out in [12]. Several efforts have been conducted to formalize the existing Java memory model [22, **?**,**?**]. Improved Java thread semantics have also been proposed to replace the current Java memory model. Besides JMM_{MP}, there was another proposal from Maessen, Arvind, and Shen [24] (referred to as JMM_{CRF} in this paper) based on the Commit/Reconcile/Fence (CRF) framework. The Java memory model is currently under an official revision process [25]. There is an ongoing discussion through the Java memory model mailing list [26]. Most recently, Manson and Pugh announced a new Java memory model draft [26] for community review.

Although JMM_{MP} and JMM_{CRF} have initiated promising improvements on Java thread semantics, they are not as easily comprehensible and comparable as first thought. JMM_{CRF} inherits the design from its predecessor hardware model [27]. Java operations have to be divided into fine grained Commit/Reconcile/Fence instructions to capture the precise thread semantics. On

the one hand, this translation process adds complexities for describing memory properties. On the other hand, the dependency on a cache based architecture also prohibits JMM_{CRF} from describing more relaxed models. JMM_{MP} uses sets of memory operations to record the history of memory activities. Instead of specifying the intrinsic memory model properties such as the ordering rules, it resorts to nonintuitive mechanisms to enforce the desired behaviors. 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.

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

As an example of non-operational approaches, Collier [28] described memory requirements based on a formal 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 their approaches is that it is nontrivial to infer program behaviors from a combination of several ordering constraints. In fact, the lack of a means for automatic execution is a noticeable limitation for most specifications with a declarative style. In a separate research effort [29], we developed a method to capture memory ordering rules as axioms and encode these non-operational models into a machine recognizable format. We then made the specifications executable by applying a constraint solver or a boolean SAT solver to check the existence of a legal execution for a given test program.

To make an operational memory model executable, Park and Dill [30, **?**] proposed a method to integrate a model checker with the Sparc *Relaxed Memory Order* [32] specification for verifying small assembly synchronization routines. In our previous work on the analysis of JMM_{CRF} [33], we extended this methodology to the domain of Java thread semantics and demonstrated its feasibility and effectiveness for analyzing language level memory models. After adapting JMM_{CRF} to an equivalent executable specification implemented with the Murphi [34] model checking tool, we systematically exercised the underlying model with a suite of test programs to reveal pivotal properties and verify common programming idioms, e.g. the *Double-Checked Locking* [35] algorithm. Roychoudhury and Mitra [36] also applied techniques similar to [33] to study the existing Java memory model. They formalized the current Java memory model with an operational representation using a local cache and a set of read/write queues and implemented the specification with an invariant checker. Although [33] and [36] have improved their respective target models by making them executable and more rigorous, they are limited to the specific designs from the original proposals. As a result, they are not suited as a generic specification framework, and the intuitions for the memory model requirements based on those notations

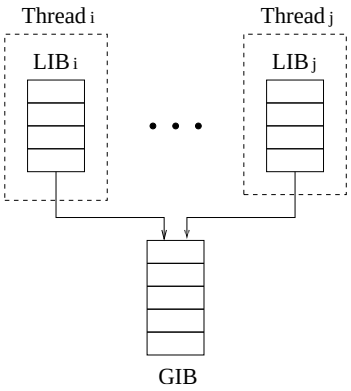

Fig. 4. Basic conceptual architecture of the UMM specification framework.

are not immediately apparent. Furthermore, the complexity of the specific data structures demands more memory consumption during model checking, which would worsen the state space explosion problem. In [37], 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 parameterizable analysis tool. This paper improves the method by providing a generic abstraction mechanism for executable models based on a simple transition system.

Non-operational and operational specification styles are complementary techniques. The non-operational approach is often more intuitive due to its declarative nature. But it does not directly help people (such as the compiler writers) to build a mental process of how the desired properties can be achieved. While operational descriptions often mirror a system's behavior and can be exploited by a model checker, they tend to emphasize the *how* aspects through their usage of specific data structures, not the *what* aspects that formal specifications are supposed to stress. Hence, it is essential for an operational specification framework to employ a generic abstract machine that can represent primitive consistency properties as opposed to specific architectural designs. In Section 4.7, we also explore a method to transform a memory model definition from one style to the other.

3 OVERVIEW OF THE FRAMEWORK

The UMM specification framework consists of an abstract transition system with an associated transition table. Figure 4 illustrates the basic conceptual architecture of the UMM transition system. Each thread k has a *local instruction buffer* LIB_k that stores its pending instructions in program order. Thread interactions

are communicated through a *global instruction buffer* GIB, which stores all previously completed memory instructions that are necessary for fulfilling a future read request.

The specification of a memory system is precisely defined in a transition table based on guarded commands. 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. The sequence of permissible actions from various threads 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 subject to the ordering constraints specified by the transition table. A memory model $\mathcal{M}$ is defined by the 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 $\mathcal{M}$.

Our notation based on guarded commands has been widely used in architectural models [38], making it familiar to hardware designers. In contrast to most processor level memory models that apply a cache structure, only two layers are used in the UMM system, one for thread local information and the other for global trace information. 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. One is to impose an interleaving close to the memory model requirement. The other is to presciently enable certain operations when needed. Completed instructions in GIB are used to form the legal visibility order. The visibility ordering rules are imposed as a final filtering mechanism to guarantee proper serialization. Instead of a fixed size main memory, we apply 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.

Integrating the Model Checking Technique To make the memory models executable, we encode them in Murphi [34], 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 translation process is straightforward. The Murphi program consists of two parts. The first part implements the formal specification of a memory model. The transition table is specified as Murphi rules. Bypassing conditions and visibility conditions are implemented as Murphi procedures. The second part comprises a large collection of idiom-driven test programs. These test programs are designed to reveal specific memory model properties or to simulate common programming idioms. Each test program is defined by specific Murphi initial state and invariants, which can be executed with the guidance of the transition system to reveal pivotal properties of the model. When a test program is executed under the guidance of the UMM transition system, the Mur-

phi 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 only when all threads are completed, to output all possible final results.

The Murphi implementation is highly configurable. It allows one to easily set up different test programs, abstract machine parameters, and memory model properties. The executable memory model can also be treated as a "black box" whereby the users are not necessarily required to understand all the details of the model to benefit from the specification. 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. [38]. Here, our contributions are twofold: (i) we employ this style for a wide spectrum of memory models; (ii) we *retain* the first-order logic style in our Murphi model which supports first-order logic quantifiers (they are unravelled through state enumeration). This also helps make the implementation of the memory model reliable.

4 FORMALIZING COMMON MEMORY MODEL PROPERTIES

The UMM framework can be configured to produce executable specifications for a variety of memory consistency properties. The general strategy is to customize the bypassing table to control the interleaving and impose proper visibility ordering constraints on the execution trace in GIB. This configuration process is typically very straightforward for memory models involving a single visibility order. For models requiring per-thread visibility orders, a write instruction needs to be decomposed into multiple sub-write instructions targeting each different thread (including the issuing thread) so that the single GIB can be used to retrieve a unique visibility order for every observing thread. This technique is inspired by similar methods applied in [3, **?**].

We demonstrate our approach by formalizing several memory model requirements representing different categories. In this section, Sequential Consistency and Coherence are presented to show the method of defining models involving a single visibility order and PRAM is described to serve as an example of models requiring per-thread visibility orders. In Section 5, the core semantics of JMM_{MP} is captured to illustrate how synchronization instructions may be handled.

4.1 Instructions

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

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

4.2 Initial Conditions

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

4.3 Transition Table

The generic transition table for Sequential Consistency, Coherence, and PRAM is given in Table 1. A read instruction completes when the return value is bound. A write instruction completes when it is added to GIB. A multithreaded program terminates when all instructions from all threads complete.

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

Table 1. Transition table for Sequential Consistency, Coherence, and PRAM.

4.4 Bypassing Rules

Table 2 outlines the bypassing rules for Sequential Consistency, Coherence, and PRAM. An entry BYPASS[$op1$][$op2$] in the bypassing table determines whether an instruction with type $op2$ can bypass a previous instruction with type $op1$. Values used in table BYPASS include *Yes*, *No*, *DiffVar*, and *DiffTgt*. Informally, *Yes*

	SC		Coherence		PRAM	
2nd ⇒ 1st ⇓	Read	Write	Read	Write	Read	Write
Read	No	No	DiffVar	DiffVar	No	DiffTgt
Write	No	No	DiffVar	DiffVar	DiffTgt	DiffTgt

Table 2. Bypassing table for Sequential Consistency, Coherence, and PRAM.

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 2, no bypassing is allowed for Sequential Consistency. For Coherence, instructions operated on different variables can be issued out of order. For PRAM, to allow each thread to perceive an independent visibility order, two sub-writes targeting different threads can be reordered. In addition, a sub-write can be reordered with a read if the sub-write targets another thread. To make these bypassing rules precise, they are formally defined in condition *ready*.

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

4.5 Visibility Ordering Requirement

Condition *legalWrite* is a guard that guarantees the serialization requirement. It specifies that a write instruction w is not eligible for a read instruction r if there exists an intermediate write instruction w' on the same variable between r and w in the ordering path. The $legalWrite$ definition of PRAM is slightly different from that of Sequential Consistency and Coherence because a reading thread t can only observe a sub-write targeting t or the default write.

For Sequential Consistency and Coherence:

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

For PRAM:

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

4.6 Verifying Programming Patterns with Executable Specifications

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 allow the results to occur, indicating that Peterson's algorithm is broken for these memory models. Under Sequential Consistency, however, the execution in Figure 2 would not be possible.

4.7 Relationship to Axiomatic Specification Methods

Despite its operational style, the UMM framework is closely related to axiomatic specification methods such as [29]. An axiomatic approach divides the global ordering relation in terms of *facets*, each of which constraints a specific aspect of the global ordering. In [29], the visibility order of a memory model is defined as a complete set of ordering rules, including a fully explicit description about general ordering properties, such as totality, transitivity, and circuit-freeness. To take a concrete example, the PRAM memory model can be defined in an axiomatic style as a set of constraints imposed to 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.

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

Each constraint is then precisely defined. For example, the program order definition can be specified as follows:

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

The UMM framework applies a two-layer architecture to make memory models operational. Variations of memory consistency properties are parameterized as different bypassing rules (defined in condition *ready*) and visibility ordering rules (defined in condition $legalWrite$). As illustrated by the common memory model properties defined in this section, the totality requirement can be implicitly built up during the execution based on interleavings allowed by the bypassing rules. If a model allows all instructions to be sent to GIB in any arbitrary order and then impose additional ordering constraints when read values are obtained, a UMM specification degenerates to an axiomatic one.

Each of the two styles has its own advantages. The axiomatic approach is declarative and more flexible. One can disable/enable the constraints and study the impact on the global orderings. However, each constraint itself may involve aspects that pertain to both program orderings and global visibility, 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 imposed using the front-end instruction selection process of the UMM framework. To convert a UMM specification to an axiomatic definition, one must encode all the ordering requirements implied by the UMM front-end process and add them as axioms to the final execution trace.

5 AN ALTERNATIVE SPECIFICATION OF THE JAVA MEMORY MODEL

We provide an alternative Java memory model specification to show how memory models involving synchronization operations may be defined using our framework. In addition, it illustrates how to resolve some of the language level memory model issues, such as the treatment of local variables. Lastly, it demonstrates the feasibility of our methodology for analyzing non-trivial memory model designs. The core Java memory model semantics formalized in this section, including definitions of normal memory operations and synchronization operations, is primarily based on JMM_{MP} [13] as of January 11, 2002.

5.1 Variables and Instructions

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

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

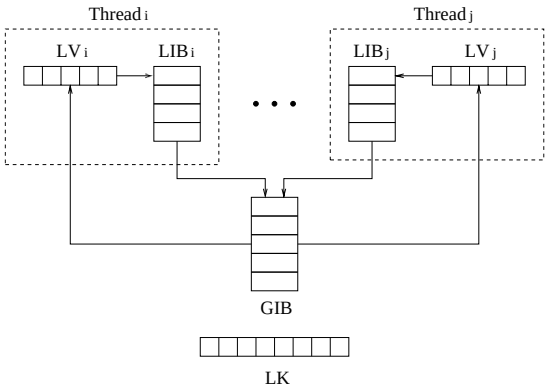

Fig. 5. Extended conceptual architecture for the Java memory model.

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

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

5.2 The Extended Conceptual Architecture

To capture the additional requirements regarding local variables and locks in the Java memory model, the conceptual architecture of the transition system is slightly extended. Figure 5 shows the abstract machine for modelling the Java memory model. In addition to the local instruction buffer, each thread k also maintains a 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. To maintain the locking status, a dedicated global *lock array* LK is also added. 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.

Need for Local Variable Information Because traditional memory models are designed for processor level architectures, aiding software 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 track the return values from read operations. However, most programming activities in Java, such as computation, flow control, and method invocation, are carried out using local variables. In order to analyze straight-line code, 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 dependency and memory model ordering requirements, which will be further discussed in Section 5.5.

5.3 Initial Conditions

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

5.4 The Transition Table for the Java Memory Model

Java memory operations are defined in the transition table given in Table 3. 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 non-conflicting local variables, the system degenerates to traditional models that do not keep local variable information. *Lock* and *unlock* instructions are issued as determined by the Java keyword *synchronized*. They are used to model the mutual exclusion effect as well as the visibility effect. A special *freeze* instruction for every final field v is added at the end of the constructor that initializes v to indicate v has been frozen. Since we are defining the memory model, only memory operations are identified in our transition system. Instructions such as $r1 = 1$ and $r1 = r2 + r3$ are not included. However, the UMM framework can be easily extended to a comprehensive program analysis system by adding semantics for computational instructions.

5.5 Bypassing Policy and Data Dependency

Table BYPASS shown in Table 4 specifies the bypassing rules for the Java memory model. Since JMM_{MP} respects program order except for *prescient writes*, Table 4 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.

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** $(useLocal(i))$ $\quad i.data := \text{LV}_{t(i)}[local(i)];$ **end**; $\text{GIB} := append(\text{GIB}, i);$ $\text{LIB}_{t(i)} := delete(\text{LIB}_{t(i)}, i);$
lock	$\exists i \in \text{LIB}_{t(i)} : ready(i) \ \wedge \ op(i) = \text{Lock} \wedge$ $(\text{LK}[lock(i)].count = 0 \vee$ $\text{LK}[lock(i)].owner = t(i))$	$\text{LK}[lock(i)].count :=$ $\quad \text{LK}[lock(i)].count + 1;$ $\text{LK}[lock(i)].owner := t(i);$ $\text{GIB} := append(\text{GIB}, i);$ $\text{LIB}_{t(i)} := delete(\text{LIB}_{t(i)}, i);$
unlock	$\exists i \in \text{LIB}_{t(i)} : ready(i) \ \wedge \ op(i) = \text{Unlock} \wedge$ $(\text{LK}[lock(i)].count > 0 \wedge$ $\text{LK}[lock(i)].owner = t(i))$	$\text{LK}[lock(i)].count :=$ $\quad \text{LK}[lock(i)].count - 1;$ $\text{GIB} := append(\text{GIB}, i);$ $\text{LIB}_{t(i)} := delete(\text{LIB}_{t(i)}, i);$
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** $(useLocal(i))$ $\quad i.data := \text{LV}_{t(i)}[local(i)];$ **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** $(useLocal(i))$ $\quad i.data := \text{LV}_{t(i)}[local(i)];$ **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 3. Transition table for the alternative Java memory model.

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	RdtLk	No	No	No	No	No	No	No
Unlock	No	Yes	No	No	No	No	No	No	No
Read Volatile	No	RdtLk	No	No	No	No	No	No	No
Write Volatile	No	Yes	No	No	No	No	No	No	No
Read Final	No	Yes	No	No	No	No	No	No	No
Write Final	No	Yes	No	No	No	No	No	No	No
Freeze	No	No	No	No	No	No	No	No	No

Table 4. Bypassing table for the alternative Java memory model.

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 4 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 dependency. The helper function $notRedundant(j)$ returns *true* if instruction j does have a synchronization effect.

The 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.

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

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

5.6 Visibility Ordering Requirement for the Java Memory Model

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

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

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

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

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

After establishing the ordering relationship by condition $LCOrder$, the requirement of serialization is enforced in *legalNormalWrite*. 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.

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

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

5.7 Volatile Variable Semantics

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

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

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

5.8 Final Variable Semantics

In Java, a final field can either be a primitive value or a reference to another object. When it is a reference, the Java language only requires that the reference itself cannot be modified in the program after its initialization but the fields of the object it points to do not have the same guarantee. JMM_{MP} proposes to add a special rule to those non-final sub-fields 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. 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 to support weak memory architectures such as Alpha since it involves run-time reachability analysis.

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

$$
\begin{aligned}
&legalFinalWrite(r, w) \equiv \\
&\quad op(w) = \text{WriteFinal} \ \wedge \ var(w) = var(r) \ \wedge
\end{aligned}
$$

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 $r4 = 1$?

Fig. 6. Write Atomicity test.

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

6 Analysis of JMM_{MP}

After adapting JMM_{MP} using our system, we are able to systematically exercise it with idiom-driven test programs and gain substantial insight about the underlying semantics. Since we have also developed formal executable models for JMM_{CRF} [33], 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 test programs complete in less than one second. Our Java memory model Murphi implementation is available at `http://www.cs.utah.edu/formal_verification/umm`.

6.1 Analyzing Memory Model Properties

Coherence Test Recall the litmus test shown in Figure 3, which reveals an execution prohibited by Coherence. When variable a is configured as a normal variable, an exhaustive enumeration of this test program under JMM_{MP} reports that the outcome is indeed permitted. To further help the users understand the scenario, the UMM tool can output an interleaving that would allow such a result. Thus, based on this simple litmus test (without even looking into the internals of the memory model), one can make an immediate conclusion that JMM_{MP} does not enforce Coherence.

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

Causality Test *Causal Consistency* [39] 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 with our verification system, a legal interleaving that allows such a behavior is immediately detected. This proves that JMM_{MP} does not

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 $r1 = r2 = 1$ and $r3 = 0$?

Fig. 7. Causality test.

Initially, $a = 0$

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

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

Fig. 8. Prescient write test.

enforce Causal Consistency for normal variable operations.

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

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 $r2 = 0$?

Fig. 9. Constructor test.

Constructor Property The constructor property is illustrated by the program in Figure 9. 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 object constructing mechanism used by JMM_{CRF}, where $Membar1$ is a special EndCon instruction used in JMM_{CRF} to indicate the completion of a constructor and $Membar2$ is due to data dependency enforced by program semantics when accessing $field$ through $reference$. If $field$ is a normal variable, this mechanism works under JMM_{CRF} but fails under JMM_{MP}. In JMM_{MP}, the default write to $field$ is still a valid write for the reading thread since there does not exist an ordering requirement on non-synchronized writes. However, if $field$ is declared as a final variable and the Freeze

instruction is used for $Membar1$, Thread 2 would never observe the default value of $field$ if $reference$ is initialized. This illustrates the different strategies used by the two models for preventing premature releases during object construction. JMM_{CRF} treats all fields uniformly and JMM_{MP} guarantees fully initialized fields only if they are final or pointed by final fields.

6.2 Verifying Programming Patterns

In [33], we have proposed to apply the model checking technique for verifying common synchronization idioms, such as the Double-Checked Locking algorithm, for memory model compliance. As also demonstrated by Peterson's algorithm in Section 1, many popular programming patterns developed under certain memory consistency assumptions might break for more relaxed memory models. An effective strategy for developing robust multithreaded programs is to carefully analyze them based on formal methods. For example, when the test program in Figure 2 is executed using normal variables, it is shown that such a violation scenario is indeed allowed by the Java memory model. Based on this experiment, one can immediately conclude that the algorithm is unsafe to use in Java programs without applying additional synchronization operations. On the other hand, if one use volatile variables and run the test again, the violation scenario would not occur.

6.3 Inconsistencies in JMM_{MP}

Non-Atomic Volatile Writes JMM_{MP} provides a formal definition that 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 [13, 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 specification and the intended goal.

Final Variable Semantics A design flaw in the 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 6.3. After the final field a is initialized, it is read by a local variable in the same constructor. The readFinal definition [13, 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.

7 CONCLUSIONS

We have presented a specification methodology for formalizing memory consistency models in general, and the Java memory model in particular. Coupled with a model

```
class foo {
    final int a;

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

Fig. 10. A flaw in the final variable semantics.

checking tool, it provides a powerful framework for conducting memory model analysis and multithreaded program verification. The flexibility of the transition system provides a generic abstraction mechanism for executable consistency models. Our operational specifications are written in a *parameterizable* style. Users can redefine the bypassing table and the visibility ordering rules to obtain an executable specification for another memory model. In addition, the simple abstract machine architecture eliminates unnecessary complexities introduced by implementation specific data structures. Hence, it helps clarify the essential semantics of the shared memory system.

Future Works A reliable specification framework may lead to several interesting future works. First, currently people 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 should 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.

Acknowledgments

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

References

1. G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, Volume 12, Number 3, June 1981.
2. Mustaque Ahamad, Rida Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of Processor Consistency. In *the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1993.

3. A formal specification of Intel Itanium processor family memory ordering, Application Note, Document Number: 251429-001, October 2002.
4. Rajeev Joshi, Leslie Lamport, John Matthews, Serdar Tasiran, Mark Tuttle, Yuan Yu. Checking cache-coherence protocols with TLA+, Volume 22, Issue 2, Pages 125-131, March 2003.
5. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.
6. S. V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *IEEE Computer*, 29(12):66–76, 1996.
7. K. Gharachorloo. Memory consistency models for shared-memory multiprocessors. Technical Report CSL-TR-95-685, Stanford University, December 1995.
8. R. J. Lipton and J. S. Sandberg. PRAM: a scalable shared memory. Technical Report CS-TR-180-88, 1988.
9. P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *the 19th International Symposium of Computer Architecture*, pages 13-21, May 1992.
10. G. Gao and V. Sarkar. Location consistency - a new memory model and cache consistency protocol. Technical Report, 16, CAPSL, University of Delaware, February 1998.
11. J. Gosling, B. Joy, and G. Steele. The Java language specification, chapter 17. Addison-Wesley, 1996.
12. W. Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(1):1-11, 2000.
13. J. Manson and W. Pugh. Semantics of multithreaded Java. Technical Report UMIACS-TR-2001-09, 2002.
14. J. Manson and W. Pugh. Core semantics of multithreaded Java. In *ACM Java Grande Conference*, June 2001.
15. Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA PathFinder. In *International Journal on Software Tools for Technology Transfer*, volume 2, number 4, pages 366-381, 2000.
16. W. Visser, K. Havelund, G. Brat, and S. Park. Java PathFinder - second generation of a Java model checker. In *Post-CAV Workshop on Advances in Verification*, Chicago, 2000.
17. James C. Corbett. Evaluating deadlock detection methods for concurrent software. In *IEEE Transactions on Software Engineering*, volume 22, number 3, pages 161–180, March 1996.
18. James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*, pages 439-448, 2000.
19. D. Park, U. Stern, and D. Dill. Java model checking. In *the First International Workshop on Automated Program Analysis, Testing and Verification*, Limerick, Ireland, 2000.
20. J. Moore, R. Krug, H. Liu, and G. Porter. Formal models of Java at the JVM level – a survey from the ACL2 perspective. In *Workshop on Formal Techniques for Java Programs*, in association with ECOOP 2001, June 2001.
21. Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. PLDI'03.
22. A. Gontmakher and A. Schuster. Java consistency: non-operational characterizations for Java memory model. In *ACM Transactions On Computer Systems, vol. 18, No. 4*, pages 333-386, November 2000.

23. Y. Gurevich, W. Schulte, and C. Wallace. Investigating Java concurrency using abstract state machines. Technical Report 2000-04, University of Delaware, December 1999.
24. J.-W. Maessen, Arvind, and X. Shen. Improving the Java memory model using CRF. In *OOPSLA*, pages 1-12, October 2000.
25. Java Specification Request (JSR) 133: Java memory model and thread specification revision.
http://jcp.org/jsr/detail/133.jsp.
26. Java memory model mailing list.
http://www.cs.umd.edu/ pugh/java/memoryModel/archive.
27. 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.
28. W. W. Collier. Reasoning about parallel architectures. Prentice-Hall, 1992.
29. Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Analyzing the Intel Itanium memory ordering rules using logic programming and SAT. In *the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2003)*, Springer Verlag LNCS, October 2003.
30. 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, 1993.
31. S. Park and D. Dill. An executable specification and verifier for Relaxed Memory Order. *IEEE Transactions on Computers*, 48(2):227-235, 1999.
32. D. Weaver and T. Germond. The SPARC Architecture Manual Version 9. Prentice Hall, 1994.
33. Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. Analyzing the CRF Java memory model. In *the 8th Asia-Pacific Software Engineering Conference*, pages 21-28, 2001.
34. D. Dill. The Murφ verification system. In *8th International Conference on Computer Aided Verification*, pages 390-393, 1996.
35. Philip Bishop and Nigel Warren. Java in pratice: design styles and idioms for effective Java, chapter 9. Addison-Wesley, 1999.
36. A. Roychoudhury and T. Mitra. Specifying multithreaded Java semantics for program verification. In *International Conference on Software Engineering*, 2002.
37. Prosenjit Chatterjee, Hemanthkumar Sivaraj, and Ganesh Gopalakrishnan. Shared memory consistency protocol verification against weak memory models: refinement via model-checking. In *Computer-Aided Verification (CAV'02)*, July, 2002.
38. R. Gerth. Introduction to Sequential Consistency and the lazy caching algorithm. *Distributed Computing*, 1995.
39. Mustaque Ahamad, Phillip W. Hutto, Gil Neiger, James E. Burns, and Prince Kohli. Causal memory: definitions, implementation and programming. Technical Report GIT-CC-93/55, Georgia Institute of Technology, July 1994.