Specifying Multithreaded Java Semantics for Program Verification

Abhik Roychoudhury
Department of Computer Science
School of Computing
National University of Singapore
Singapore 117543
abhik@comp.nus.edu.sg

Tulika Mitra
Department of Computer Science
School of Computing
National University of Singapore
Singapore 117543
tulika@comp.nus.edu.sg

ABSTRACT

The Java programming language supports multithreading where the threads interact among themselves via read/write of shared data. Most current work on multithreaded Java program verification assumes a model of execution that is based on interleaving of the operations of the individual threads. However, the Java language specification (which any implementations of Java multithreading must follow) supports a weaker model of execution, called the Java Memory Model (JMM). The JMM allows certain reordering of operations within a thread and thus permits more behaviors than the interleaving based execution model. Therefore, programs verified by assuming interleaved thread execution may not behave correctly for certain Java multithreading implementations.

The main difficulty with the JMM is that it is informally described in an abstract rule-based declarative style, which is unsuitable for formal verification. In this paper, we develop an equivalent formal executable specification of the JMM. Our specification is operational and uses guarded commands. We then use this executable model to verify popular software construction idioms (commonly used program fragments/patterns) for multithreaded Java. Our prototype verifier tool detects a bug in the widely used "Double-Checked Locking" idiom, which verifiers based on interleaving execution model cannot possibly detect.

1. INTRODUCTION

The Java programming language supports multithreaded programming where multiple threads can communicate via reads/writes of shared objects (see [21] for a detailed discussion on software design using multithreaded Java). Multithreading is a useful technique as it allows the programmer to structure different parts of the program into different threads. Implementing the user interface of a software as a separate thread is a common example of such structuring. Concrete real-life uses and applications of Java multithreading are presented in [18].

Java threads can be run on multiple hardware processors or on a single processor through a thread library (such as POSIX threads [7]). As the implementations of multithreading are varied, the Java Language Specification (JLS) prescribes certain abstract rules which *any* implementation of Java multithreading must follow [16]. These rules are called the *Java Memory Model (JMM)*. However, the JMM is more complex than an interleaved execution of the threads, where each thread executes in program order. The operations in any Java thread include read/write of shared variables and synchronization operations like lock/unlock. In order to allow standard compiler and hardware optimizations, the JMM permits these operations within a thread to be completed out-of-order. Thus, the permitted set of execution traces under the JMM is a superset of the simple interleaved execution of the individual threads. This makes the debugging and verification of multithreaded Java software very difficult as we have to consider:

- arbitrary interleaving of the threads,
- certain (not all) re-orderings of operations in the individual threads.

There is currently a huge body of ongoing work on employing static analysis and model checking techniques [10] for concurrent Java program verification [14, 19, 25, 26, 31]. Some of these techniques translate the program to a formal model [19, 25] and then use dataflow analysis/ model checking to search the state space of this model. Others [14, 31] directly analyze program source code by employing techniques such as stateless search and persistent sets. However, a commonality among all these techniques is that they assume the underlying execution model of a multithreaded program to be *sequentially consistent* [20].

Sequential Consistency. Before proceeding any further, let us elaborate on this point. An execution model for multithreaded programs is sequentially consistent if for any program P (a) any execution of P is an interleaving of the operations in the constituent threads (b) the operations in each constituent thread execute in program order. Thus, in the following program with two threads

$$(Op_1\ ;\ Op_2) \quad \| \quad (Op'_1\ ;\ Op'_2)$$

Op_1, Op'_1, Op'_2, Op_2 is a sequentially consistent execution but Op_1, Op'_2, Op_2, Op'_1 is not. As sequential consistency denotes the programmer's intuitive understanding of the execution model, it is generally an useful model to assume for purposes of program verification.

Unfortunately, it is not sufficient to assume a sequentially consistent execution model for verifying multithreaded Java programs. The reason for this lies in the JMM. The current JMM (which any implementation of Java multithreading must follow) is weaker than sequential consistency, that is, it allows more behaviors than simple interleaving of the operations of the individual threads. Thus, assuming sequential consistency during program verification of an invariant property might lead to an observable violation of the invariant in a verified-correct program on some execution platforms !! Examples of such programs even include some popular multithreaded Java software construction idioms such as the "Double-Checked Locking" idiom [29].

There could be several solutions to this problem. First, we could develop a restricted fragment of Java programs for which the JMM guarantees sequentially consistency [2]. Programmers are then encouraged to write programs only in this fragment. Secondly, we could change the JMM altogether (this is being seriously considered by an expert group, see [1]). Finally, we could develop an executable formal description of the JMM and incorporate it into program verification.

Let us now study each of these solutions in depth. In the first solution, the fragment of Java programs for which the execution will always appear to be sequentially consistent are the so-called "properly synchronized programs" or "data-race-free programs" [3]. Intuitively, these programs ensure that whenever a thread accesses a shared object, it possesses a lock to the object. There are two difficulties with this solution. First, even if the user's program is "properly synchronized", he/she might use software libraries from untrusted sources which are not guaranteed to be "properly synchronized". Moreover, demanding synchronization for every shared object access is known to cause unacceptable performance overheads in practice [3, 21].

For the purposes of program verification, the second solution also has certain difficulties. First of all, even if the JMM is changed, the new memory model will not be implemented for some time to come. Existing Java Virtual Machines (JVM) will continue to be used on uniprocessor as well as multiprocessor platforms. Therefore, changing the JMM now will not solve the problem of the Java programmer for many years to come. Moreover, the two concrete proposals for an improved JMM [22, 23] (which were proposed very recently, and are now being hotly debated) are also weaker than sequential consistency. In fact since the Java memory model describes all possible program behaviors on all possible platforms, it is unrealistic to define a Java memory model which enforces sequential consistency.

In this paper, we advocate the third solution. We compose a formal description of the JMM along with a formal model of the program for the purposes of program verification. Java's "write-once, run-everywhere" strategy makes it important to develop programs which do not behave differently on different platforms. Moreover, a verified-correct program behaving incorrectly on certain platforms is of particular concern! As the JMM captures all possible behaviors, incorporating it into program verification allows platform independent reasoning.

However to include the JMM in program verification, we have to take note of the following issues. First, the JMM [16] is informally described in a declarative style. It specifies certain rules that must never be violated in a multithreaded execution. In other words, the model is neither operational nor executable. This makes the JMM almost impossible to reason with (see [28] for the complexities of informal reasoning about the JMM). We develop an executable specification of the JMM in this paper.

Secondly, program verification via model checking suffers from the state space explosion problem. Composing the memory model along with the program model further blows up the state space to be explored. However, note that the Java memory consistency model coincides with sequential consistency for "properly synchronized" programs, that is, programs where any access to shared data is preceded by explicit synchronization. In reality, a significant portion of a multithreaded Java program is "properly synchronized". The "unsynchronized" portion using the performance enhancing features of the JMM (which allow reordering of operations within a thread) mostly appear in "low-level" program fragments. These are widely used *software construction idioms* performing a specific task. These program fragments are typically executed many times in the course of program executions. Hence they are optimized by avoiding explicit synchronization for every shared data access. We have used our executable memory model to debug and verify such program fragments.

Summary of Results. Concretely, our contributions are:

- We develop a formal *executable* specification of the JMM: the rules for implementing multithreading in the Java programming language. Our operational style specification describes *all* possible behaviors that *any* implementation of Java multithreading can exhibit.

- Our approach of (a) constructing an executable JMM and (b) composing it with the program model for program verification, is completely generic. It is *not* tied to the current JMM.

- We have used our executable model of JMM to develop a prototype invariant checker. This tool is particularly useful for verifying unsynchronized program fragments. Our checker has been used to detect a bug in the widely used "Double-Checked Locking" software construction idiom [29].

- Finally, our formal JMM specification alleviates the difficulty in understanding the current JMM [16]. The rule-based JMM has been described as "very hard to understand" [3, 28] and most reasoning has been done via informal counter-example construction.

Organization. The rest of the paper is organized as follows. In Section 2, we discuss related work on the JMM and program verification. Section 3 discusses the informal specification of the JMM, while Section 4 presents the formal specification. Section 5 discusses applications of our JMM specification in program verification. Finally, we discuss the broad implications of our work in Section 6.

2. RELATED WORK

Verification of Java programs has been studied extensively. Specifically, significant progress has been achieved recently in multithreaded Java program verification [6, 11, 14, 19, 25, 26, 30]. Out of these works, [11, 19, 25] extract formal model from Java source code and analyze the formal model, while [6, 14, 30] propose techniques to directly analyze the source code by modifying the state space search algorithm. These works appear at a higher level of abstraction than our work. They assume a simple execution model of sequential consistency and develop algorithms to analyze sequentially consistent execution traces of a program. Our work concentrates on formalizing the underlying execution model, but does not address the issue of state-space search algorithms.

Recently, some research has been directed towards developing executable models of the JVM [4, 24]. In particular, Moore [24] develops a formal model of a multithreaded JVM and advocates its use for verifying Java programs. Here, the only difference from conventional program verification is that instead of source code verification, the byte-code is verified. This work still suffers from the problems we discussed earlier: the reasoning performed is platform dependent because a specific JVM is formalized (which enforces sequential consistency). Any platform independent verification of Java programs *must* take into account the JMM.

The JMM has been a topic of intense research in the past few years. The informal model was first developed in the Java Language Specification [16]. Pugh [28] first pointed out the difficulties in informally reasoning about the model and suggested changes. Subsequently, researchers have proposed several improvements to the model [22, 23]. Contrary to these works, our work does not address the question: "What should be the semantics for multithreaded Java" ? Instead, it argues that multithreaded Java semantics (the current one or any future improvement) should be incorporated into Java program verification.

Since the inception of the JMM, several formalizations of Java concurrency have been proposed, [5, 8, 15, 17] to name a few. Some of these [5] focus only on language level concurrency constructs without considering the memory model. Some others [8, 15] construct non-executable specifications of the memory model. Most importantly, the goal of all these works is to have a clear understanding of Java concurrency (via formal specification) and then perform human reasoning. Our goal is different. We have developed an executable formal JMM specification for (semi)-automated reasoning about Java programs. This allows us to verify nontrivial software fragments, which would be extremely cumbersome to perform with human reasoning.

Developing executable memory models has been studied in the context of hardware multiprocessors [13, 27]. Similar to Java threads, hardware shared-memory multiprocessors also impose a consistency model which dictates the allowed interactions among the processors via a shared memory.

3. THE JAVA MEMORY MODEL

In this section, we present the Java Memory Model (JMM) given in [16]. The model is abstract and is not constructed as a guide for implementing Java multithreading. Rather any Java multithreading implementation is supposed to allow only behaviors allowed by the model. We construct a formal executable description of the model in the next section.

The Java threads interact among themselves via shared variables. For any shared variable v, each thread (a) possesses a local copy of v and (b) is allowed to access the global master copy of v in main memory. The JMM essentially imposes constraints on the interaction of the threads with the master copy of the variables and thus with each other. The model defines the following *actions* for reading/writing the local/master copy of v in thread t.

- $\texttt{use}_t(v)$: Read from the local copy of v in t
- $\texttt{assign}_t(v)$: Write into the local copy of v in t
- $\texttt{read}_t(v)$: Initiate reading from master copy of v to local copy of v in t.
- $\texttt{load}_t(v)$: Complete reading from master copy of v to local copy of v in t.
- $\texttt{store}_t(v)$: Initiate writing the local copy of v in t into master copy of v
- $\texttt{write}_t(v)$: Complete writing the local copy of v in t into master copy of v

Apart from the above actions, each thread t may perform lock/unlock on shared variables, denoted $\texttt{lock}_t$ and $\texttt{unlock}_t$ respectively.

When a thread executes a virtual machine instruction that uses/assigns the value of a variable, it accesses the local copy of that variable. Before `unlock`, the local copy is transferred to the master copy through `store` and `write` actions. Similarly, after `lock` action the master copy is transferred to the local copy through `read` and `load` actions. Given the above definitions, we can now consider a multithreaded program as "properly synchronized", if every access to a shared variable occurs between a `lock` and its corresponding `unlock`.

Two important points need to be noted here. First, the local copies of shared variables conceptually form a thread's private "cache". Secondly, data transfer between the local and the master copy is not modeled as an atomic action. This is to model the realistic transit delay when the master copy is located in the hardware shared memory and the local copy is in the hardware cache.

Among the eight actions mentioned above, a thread in a Java program invokes only four of them: `use`, `assign`, `lock`, and `unlock`. Each thread invokes these actions in its program order. The other four (`load`, `store`, `read`, and `write`) are invoked arbitrarily by the multithreading implementation, subject to *temporal ordering constraints* specified in the JMM. A major difficulty in reasoning about the JMM (as reported in literature [28]) seems to be these ordering constraints. They are given in an informal, rule-based, declarative style. It is difficult to reason how multiple rules determine the applicability/non-applicability of an action. Our operational specification avoids this difficulty by modeling each action as a guarded command. Details appear in the next section.

We conclude this section by briefly explaining why the JMM is weaker than sequential consistency. Note that the threads cannot directly invoke actions which modify the master copy of a shared variable. Therefore, modifications to the master copy of a shared variable can complete out-of-order. As a result, writes to shared variables are not seen by all threads in the same order. For example in the following program with two threads:

$$(\texttt{assign}\ u,\ 1\ ;\ \texttt{assign}\ v,\ 2)\ \|\ Op$$

The following is a legal trace of the program:

```
assign u, 1;               % Local writes to u
assign v, 2;               % Local writes to v
store v, 2; write v, 2;    % Write master copy of v
Op                         % thread 2 executes here
store u, 1; write u, 1     % Write master copy of u
```

In this trace, the write operations of the first thread do not complete in program order. In fact, when the second thread executes Op it can observe (via reads) the old value of u, and new value of v. This is never possible under sequential consistency.

4. JMM SPECIFICATION

This section presents a formal executable specification of the Java Memory Model (JMM). Our specification style is operational. In particular, we describe each action in the JMM as a guarded command. First we present an executable specification of the core memory model consisting of eight actions. A proof of equivalence of our executable formal description of the JMM with the rule-based declarative description in the Java language specification [16] appears in the appendix.

4.1 Core Memory Model

Our model is an asynchronous concurrent composition of n Java threads $\texttt{Th}_1, \ldots, \texttt{Th}_n$ and a single main memory process `MM`. Communication among processes takes place via shared data. Each process can perform a set of *actions*, each of which is modeled by a guarded command. The asynchronous concurrent composition of these processes is the union of the guarded commands of the constituent processes.

Local States. We now proceed to describe the local states of the $\texttt{Th}_i$ and `MM` processes. Then, we formally describe the actions which the threads and the main memory processes execute. Note that the threads communicate via shared program variables $\{v_1, \ldots, v_m\}$; we denote the type of v_j as τ_j. The program variables are not part of the local states of $\texttt{Th}_i$ or `MM`. Rather thread $\texttt{Th}_i$ maintains a local copy of each program variable v_j, while MM maintains the master copy.

$$\text{Set of Shared Variables} = \{v_1, v_2, \ldots, v_m\}$$
$$v_1 : \tau_1,\ v_2 : \tau_2,\ \ldots,\ v_m : \tau_m$$
$$Thread\_state_i = (Cache_i,\ Rd\_qs_i,\ Wr\_qs_i)$$

$$\begin{array}{l} Cache_i = [cache_{i,1}, \ldots, cache_{i,m}] \\ Rd\_qs_i = [rd\_q_{i,1}, \ldots, rd\_q_{i,m}] \\ Wr\_qs_i = [wr\_q_{i,1}, \ldots, wr\_q_{i,m}] \\ cache_{i,j} = (rvalue_{i,j}, dirty_{i,j}, stale_{i,j}) \\ rvalue_{i,j} : \tau_j \qquad dirty_{i,j}, stale_{i,j} : \mathsf{Boolean} \\ rd\_q_{i,j}, wr\_q_{i,j} : \mathsf{Queue\ of}\ \tau_j \end{array}$$

The local state of a thread process $\texttt{Th}_i$ can be described by a 3-tuple $(Cache_i,\ Rd\_qs_i,\ Wr\_qs_i)$ as shown above. $Cache_i$ contains the local copy of the shared variables (it need not correspond to a physical cache). Rd_qs_i and Wr_qs_i each denote exactly m queues, one for each shared variable.

The local copy of the shared variable v_j in $\texttt{Th}_i$ is described by $cache_{i,j} = (rvalue_{i,j}, dirty_{i,j}, stale_{i,j})$. The first component $rvalue_{i,j}$ is the value of v_j in the local copy of $\texttt{Th}_i$. The second component $dirty_{i,j}$ is a bit indicating whether the local copy of v_j is dirty, that is, there is an assignment to v_j by $\texttt{Th}_i$ which is not yet visible to other threads (via `store`, `write` actions). The third component $stale_{i,j}$ is a bit indicating whether the local copy is stale, that is, the local copy does not reflect recent write(s) which is (are) visible to some other threads.

As mentioned before, read/write of the master copy of a variable is not modeled as atomic operation. A `read` action need not immediately precede its corresponding `load` action and a `write` action need not immediately follows its corresponding `store` action. The set of queues Rd_qs_i and Wr_qs_i model this transit delay. Queue $rd\_q_{i,j}$ contains values of the variable v_j as obtained (from master copy) by $\texttt{Th}_i$'s `read` actions, but for which the corresponding `load` actions (to update the local copy) are yet to be performed. Similarly, queue $wr\_q_{i,j}$ contains values of the variable v_j as obtained (from local copy) by $\texttt{Th}_i$'s `store` actions, but for which the corresponding `write` actions (to update the master copy) are yet to be performed.

The local state of the main memory process `MM` is a pair $(Memvals,\ Lock\_state)$. $Memvals$ are the values of the master copy of shared variables: $mval_j$ denotes the value of the shared variable v_j in the main memory. The variable $Lock\_state$ records, for each thread, the number of `lock` actions executed for which the matching `unlock` actions are yet to occur; $lock\_cnt_i$ is a natural number. If thread i has executed l `lock` actions for which the matching `unlock` actions have not occurred, then $lock\_cnt_i = l$.

$$MM\_state = (Memvals,\ Lock\_State)$$
$$\begin{array}{l} Memvals = [\ mval_1,\ mval_2,\ \ldots,\ mval_m\ ] \\ Lock\_state = [\ lock\_cnt_1,\ lock\_cnt_2,\ \ldots,\ lock\_cnt_n\ ] \\ mval_j : \tau_j \qquad lock\_cnt_i : \mathsf{nat} \end{array}$$

The JMM enforces

$$\forall i, j\,(i \neq j \Rightarrow (lock\_cnt_i = 0 \vee lock\_cnt_j = 0))$$

as an invariant. In other words, at most one thread can possess a lock at any given time. This does *not* prevent unsafe accesses of shared variables, because a thread may not acquire a lock before accessing shared variables (as is the case in unsynchronized program fragments). In this paper, we consider only a single lock. This can be extended straightforwardly to the case of multiple locks.

Actions. Figure 1 formally describes the eight different actions performed by $\texttt{Th}_i$ and `MM` as mentioned in JLS [16]. At any time step, these processes can execute either a *program action* or a *platform action* which are defined below.

DEFINITION 1 (PROGRAM ACTION). *An action invoked by the program running as thread* $\texttt{Th}_i$ *is called a program action. The actions* $\texttt{use}_i$, $\texttt{assign}_i$, $\texttt{lock}_i$, *and* $\texttt{unlock}_i$ *are program actions.*

DEFINITION 2 (PLATFORM ACTION). *An action which is performed by the underlying multithreading implementation is called a platform action. The actions (*$\texttt{load}_i$, $\texttt{store}_i$, $\texttt{read}_i$, *and* $\texttt{write}_i$*) are platform actions.*

Typically, the purpose of executing platform actions is to enable those program actions which are currently disabled.

Action $\texttt{use}_i(j)$:
$\neg stale_{i,j} \rightarrow \mathsf{return}\ cache_{i,j}$

Action $\texttt{assign}_i(j, val)$:
$\mathsf{empty}(rd\_q_{i,j}) \rightarrow cache_{i,j} := val;\ dirty_{i,j} := true;\ stale_{i,j} := false$

Action $\texttt{load}_i(j)$:
$\neg\mathsf{empty}(rd\_q_{i,j}) \rightarrow cache_{i,j} := \mathsf{dequeue}(rd\_q_{i,j});\ stale_{i,j} := false$

Action $\texttt{store}_i(j)$:
$dirty_{i,j} \wedge \mathsf{empty}(rd\_q_{i,j}) \wedge \neg\mathsf{full}(wr\_q_{i,j}) \rightarrow \mathsf{enqueue}(cache_{i,j},\ wr\_q_{i,j});\ dirty_{i,j} := false$

Action $\texttt{read}_i(j)$:
$\neg dirty_{i,j} \wedge \mathsf{empty}(wr\_q_{i,j}) \wedge \neg\mathsf{full}(rd\_q_{i,j}) \rightarrow \mathsf{enqueue}(mval_j,\ rd\_q_{i,j})$

Action $\texttt{write}_i(j)$:
$\neg\mathsf{empty}(wr\_q_{i,j}) \rightarrow mval_j := \mathsf{dequeue}(wr\_q_{i,j})$

Action $\texttt{lock}_i$:
$(\forall k \neq i\ lock\_cnt_k = 0)\ \bigwedge\ \forall 1 \leq j \leq m\ (\mathsf{empty}(rd\_q_{i,j}) \wedge \neg dirty_{i,j}) \rightarrow$
$lock\_cnt_i := lock\_cnt_i + 1;\ \mathsf{for}\ j := 1\ \mathsf{to}\ m\ \mathsf{do}\ stale_{i,j} := true$

Action $\texttt{unlock}_i$:
$lock\_cnt_i > 0\ \bigwedge\ \forall 1 \leq j \leq m\ (\mathsf{empty}(wr\_q_{i,j}) \wedge \neg dirty_{i,j}) \rightarrow lock\_cnt_i := lock\_cnt_i - 1$

Initial Conditions:
$\forall 1 \leq i \leq n, lock\_cnt_i = 0$
$\forall 1 \leq i \leq n, \forall 1 \leq j \leq m\ \neg dirty_{i,j} \wedge stale_{i,j} \wedge \mathsf{empty}(rd\_q_{i,j}) \wedge \mathsf{empty}(wr\_q_{i,j})$

Figure 1: Actions in the core memory model

We model each action as a guarded command of the form $\mathcal{G} \rightarrow \mathcal{B}$, where the guard $\mathcal{G}$ is first evaluated; if $\mathcal{G}$ is true, then the body $\mathcal{B}$ is executed atomically. The guarded-command notation for describing concurrent systems has been popularized by many researchers including Chandy and Misra in their Unity programming language [9]. We denote action $\texttt{use}_i(j)$ as a `use` action on shared variable v_j by $\texttt{Th}_i$; similarly for `assign`, `load`, `store`, `read`, and `write`. The action $\texttt{lock}_i$ denotes locking of *all* shared variables by $\texttt{Th}_i$; similarly for $\texttt{unlock}_i$.

Understanding the JMM. We now explain the difficulty in understanding/reasoning about the rule-based JMM and how our guarded-command specification overcomes that difficulty. Typically several rules of the rule based JMM contribute to the applicability of an action. Thus it is difficult to comprehend the applicability condition of an action. Our formal model makes this applicability condition explicit via the guards in each action. In the following, we give one example to illustrate this point. We use the notation $<$ to denote the temporal ordering relation among actions.

In the JMM, no rule directly prevents $\texttt{assign}_i(j)$ to take place between a $\texttt{read}_i(j)$ and the corresponding $\texttt{load}_i(j)$. However, it is prevented by the interaction among three different rules of the JMM. One rule requires `read`, `load` and `store`, `write` to be uniquely paired, where:

$$\texttt{read}_i(j) < \texttt{load}_i(j) \quad \text{and} \quad \texttt{store}_i(j) < \texttt{write}_i(j).$$

Another rule states that a `store` must invervene between an `assign` and a `load` action.

$$\texttt{assign}_i(j) < \texttt{load}_i(j) \Rightarrow \texttt{assign}_i(j) < \texttt{store}_i(j) < \texttt{load}_i(j)$$

Yet another rule ensures that

$$\texttt{store}_i(j) < \texttt{load}_i(j) \Rightarrow \texttt{write}_i(j) < \texttt{read}_i(j)$$

where $\texttt{write}_i(j)$ ($\texttt{read}_i(j)$) is the write (read) corresponding to $\texttt{store}_i(j)$ ($\texttt{load}_i(j)$). Thus, from these three rules we get

$$\texttt{assign}_i(j) < \texttt{load}_i(j) \Rightarrow \texttt{assign}_i(j) < \texttt{store}_i(j) < \texttt{write}_i(j) < \texttt{read}_i(j) < \texttt{load}_i(j)$$

In other words, we infer that an $\texttt{assign}_i(j)$ cannot take place between a $\texttt{read}_i(j)$ and the corresponding $\texttt{load}_i(j)$. This restriction is explicitly stated in our specification with $\mathsf{empty}(rd\_q_{i,j})$ as the guard for $\texttt{assign}_i(j)$ action.

4.2 Volatile Variables

In this section, we extend our memory model to handle volatile variables. The Java Language Specification (JLS) [16] describes a variable v as volatile, if every access of v by a thread leads to an access of the master copy of v in the main memory. In other words, the notion of volatile variables disables the effect of caching.

In addition to the shared program variables described in the previous section, let $\{v_{m+1}, \ldots, v_{m+k}\}$ be volatile variables of type τ_{vol} [1]. First, we extend the local states of the thread and main memory processes to include states for the volatile variables. Here the main difference is that we do not have separate read and write queues for each volatile variable. Instead, the reads of all volatile variables for $\texttt{Th}_i$ are recorded in a single queue $vol\_rd\_q_i$, similarly for writes. This models the requirement that not only the memory accesses of the same volatile variable but also those of different volatile variables should proceed in order.

[1]All volatile variables are assumed to be of same type; the model can be easily extended if they are of different types.

$Cache_i = [\ cache_{i,1}, \ldots, cache_{i,m+k}\ ]$
$Memvals = [\ mval_1, mval_2, \ldots, mval_{m+k}\ ]$
$Rd\_qs_i = [\ rd\_q_{i,1}, \ldots, rd\_q_{i,m}, vol\_rd\_q_i\ ]$
$Wr\_qs_i = [\ wr\_q_{i,1}, \ldots, wr\_q_{i,m}, vol\_wr\_q_i\ ]$
$vol\_rd\_q_i, vol\_wr\_q_i$: Queue of (VolVarId, τ_{vol})
VolVarId : $m+1, \ldots, m+k$

We describe the actions on volatile variables. We denote the `use` of volatile variable v_j by $\mathtt{Th}_i$ as $\mathtt{use\_volatile_i(j)}$; similarly for other actions. The extension of `read` and `write` in presence of volatile variables is straightforward. Instead of updating the read (write) queue of an individual non-volatile variable we now update $vol\_rd\_q_i$ ($vol\_wr\_q_i$). For $\mathtt{lock_i}$ and $\mathtt{unlock_i}$ actions, instead of checking the read and write queues of only non-volatile variables, we check the read and write queues of both volatile and non-volatile variables.

The extensions for other actions (shown in Figure 2) are more involved. Note that each `use`/`assign` of a volatile variable requires a main memory access, that is, `load`/`store`. Moreover, the `load` must immediately precede an `use` and the `store` must immediately follow an `assign`. Thus, $stale_{i,j}$ is true after *every* access of the local copy of the volatile variable v_j in $\mathtt{Th}_i$; this forces the next access to go to main memory. Also, $dirty_{i,j}$ is true if $\mathtt{Th}_i$ has performed *exactly one* update (via `assign` action) on the local copy of volatile variable v_j which is not yet propagated to the master copy. Multiple updates of the local copy of a volatile variable is not possible without updating the master copy.

4.3 Prescient Stores

The JLS also allows prescient stores — that is, a `store` which occurs before the `assign`. This optimization is allowed only if the value that is written by `assign` is known beforehand. We define a prescient store as *pending* if the `store` has taken place, but the corresponding `assign` has not yet taken place. Prescient stores are allowed only for non-volatile variables.

To incorporate prescient stores into our memory model of Section 4.1, we extend the thread state. We add to $cache_{i,j}$ an extra state variable $prescient_{i,j}$. The type of $prescient_{i,j}$ is $\{nil\} \cup \tau_j$. Thus $prescient_{i,j} = nil$ if there is no pending prescient store on variable v_j by thread $\mathtt{Th}_i$; otherwise $prescient_{i,j}$ holds the value of the pending prescient store on v_j by $\mathtt{Th}_i$. We define a new action $\mathtt{prescient\_store_i(j)}$

Action $\mathtt{prescient\_store}_i(j)$:
$\mathsf{empty}(rd\_q_{i,j}) \wedge \neg\mathsf{full}(wr\_q_{i,j}) \rightarrow$
  pick $val \in \tau_j$; $\mathsf{enqueue}(val,\ wr\_q_{i,j})$;
  $prescient_{i,j} := val$; $dirty_{i,j} := false$

Note that we have weakened the guard of $\mathtt{store_i(j)}$ by removing the condition $dirty_{i,j} = true$. This is because a prescient store precedes an `assign` which sets the *dirty* bit. The `assign` action is modified to ensure that the `assign` writes the same value as the corresponding prescient store. The modification reflects both: (a) a normal `assign` as shown in Section 4.1 when $prescient_{i,j} = nil$, (b) a delayed `assign` for a preceding `prescient_store` where $prescient_{i,j} \neq nil$. In the second case, we do not set the $dirty_{i,j}$ bit; this prevents an unnecessary `store` action following the delayed `assign`.

Action $\mathtt{assign}_i(j, val)$:
$prescient_{i,j} = val \ \bigvee\ (prescient_{i,j} = nil \wedge \mathsf{empty}(rd\_q_{i,j})) \rightarrow$
  $cache_{i,j} := val$; $stale_{i,j} := false$;
  if $prescient_{i,j} = nil$ then $dirty_{i,j} := true$;
  $prescient_{i,j} := nil$

Note that, $prescient_{i,j} \neq nil$ is true only between a prescient store and the corresponding delayed assign. According to the JLS [16], no `lock`, `load`, or `store` actions can occur between a prescient store and a delayed assign. This is ensured in our model by strengthening the guards of $\mathtt{lock}_i$, $\mathtt{load}_i(j)$ and $\mathtt{store}_i(j)$ with the condition $prescient_{i,j} = nil$.

4.4 Waiting and Notification

Java supports the feature of *waiting* and *notification*. A thread $\mathtt{Th}_i$, which has acquired the lock, may voluntarily release it via a `wait`. $\mathtt{Th}_i$ is added to the set of waiting threads. Subsequently, $\mathtt{Th}_j$ ($i \neq j$) acquires the lock and decides to notify one (or more) of the threads from the list of waiting threads, possibly $\mathtt{Th}_i$. Thread $\mathtt{Th}_i$, however, can proceed only after $\mathtt{Th}_j$ (the current owner of the lock) releases the lock. To model waiting and notification, we extend the local state of `MM` with a state variable $Wait\_set$: set of $Thread\_id$. Also, we conjoin the condition $i \notin Wait\_set$ to the guards of the actions $\mathtt{use}_i$, $\mathtt{assign}_i$, and $\mathtt{lock}_i$. This prevents a waiting thread from progressing. The guard of $\mathtt{unlock}_i$ is not changed, as a waiting thread must be allowed to unlock (so that other threads can progress). The other actions (`load`, `store`, `read`, and `write`) correspond to actions taken by the JVM implementation. They are not directly fired by the Java program and therefore their guards are unaffected.

To model the three well-known synchronization constructs `wait`, `notify`, and `notifyAll`, we add actions or guarded commands to our model. The description of these actions follows directly from the standard notions of waiting, resumption and notification. Details are omitted for space considerations.

5. VERIFYING PROGRAMS

In this section, we discuss how our executable Java Memory Model can be used for verifying concurrent Java programs. For this purpose, we first discuss how each thread is modeled and how the threads are scheduled. Subsequently we discuss techniques for alleviating the state space explosion problem.

Modeling each thread. Given a multithreaded program $\mathtt{Th}_1 \parallel \mathtt{Th}_2 \parallel \ldots \parallel \mathtt{Th}_n$, we model each thread as follows:

- read of a variable `a` is converted to action `use(a)`
- write of a variable `a` is converted to action `assign(a)`
- any code fragment marked as `synchronized` is preceded by `lock` and is succeeded by `unlock`.

The reader will observe that we have not discussed the modeling of program expressions through `use` and `assign` actions. To model the statement $c = a + b$, we must execute `use(a); use(b)` followed by `assign(c, v)` where c is the addition of the values returned by `use(a)` and `use(b)`. This can be accommodated by (1) extending the executable model with a register set $\mathcal{R}$ to hold the values returned by `use` actions, (2) extending `assign` to be of the form $\mathtt{assign}(V,\ E_{\mathcal{R}})$ where V is a shared variable and $E_{\mathcal{R}}$ is an expression containing registers in $\mathcal{R}$. The exact description of possible expressions depends on the type of the shared variables.

Action $\mathtt{use\_volatile}_i(j)$:
$\neg stale_{i,j} \wedge \neg dirty_{i,j} \rightarrow stale_{i,j} := true$; return $cache_{i,j}$

Action $\mathtt{assign\_volatile}_i(j, val)$:
$\neg dirty_{i,j} \wedge stale_{i,j} \wedge \mathsf{empty}(vol\_rd\_q_i) \rightarrow cache_{i,j} := val;\ dirty_{i,j} := true;\ stale_{i,j} := false$

Action $\mathtt{load\_volatile}_i(j)$:
$\neg dirty_{i,j} \wedge stale_{i,j} \wedge \neg\mathsf{empty}(vol\_rd\_q_i) \rightarrow (j, val) := \mathsf{dequeue}(vol\_rd\_q_i);\ cache_{i,j} := val;\ stale_{i,j} := false$

Action $\mathtt{store\_volatile}_i(j)$:
$dirty_{i,j} \wedge \neg stale_{i,j} \wedge \mathsf{empty}(vol\_rd\_q_i) \wedge \neg\mathsf{full}(vol\_wr\_q_i) \rightarrow$
$\mathsf{enqueue}((j, cache_{i,j}), vol\_wr\_q_i);\ dirty_{i,j} := false;\ stale_{i,j} := true$

Figure 2: Actions for accessing volatile variables

Scheduling the threads. At each time step, any one thread executes either a program action or a platform action (refer Definitions 1 and 2 in Section 4.1). Because there can be several enabled actions at any given time, we can adopt a scheduling strategy to rule out certain behaviors. For example, given a multithreaded program $\mathtt{Th}_1, \ldots, \mathtt{Th}_n$ our scheduling can proceed as follows:

If the next program action of any thread is enabled
  then pick one such thread $\mathtt{Th}_i$;
    execute the next program action of $\mathtt{Th}_i$
  else pick a thread $\mathtt{Th}_j$ with enabled platform action;
    execute any enabled platform action of $\mathtt{Th}_j$.

The above policy portrays the situation that platform actions are executed only to enable program actions.[2] The above scheduling algorithm *does not guarantee* sequential consistency, even though the program actions are started in program order in each thread. Recall that in the JMM, execution of a program action does not update the shared memory. The "effect" of the program actions are updated to the shared memory via the platform actions which may complete out-of-order.

Invariant Checker. To verify an *invariant* (property that must hold in every state of every execution trace), we exhaustively check the states of every execution trace. Our invariant checker has been implemented on top of a memoized logic programming system XSB [32]. Because our model is expressed in guarded-command notation, the Murφ model checker [12] is a candidate implementation vehicle as it supports a guarded-command–based specification language. However, note that in the verification of any multithreaded program, it is sufficient to check *only* those executions which are generated by our scheduling strategy. In other words, we want to program (*i.e. prune*) the traversal strategy of the search space of multithreaded executions. This programming capability is very naturally supported in a general purpose logic programming system where computation proceeds by search. A prototype checker based on our executable memory model has been built using the XSB logic programming system. The checker could be used in two modes. Either we could search the entire search space consisting of all allowed execution traces of program actions and platform actions in the threads of a program; or we could input rules to prune the search space based on some scheduling algorithm. In the following, we discuss some techniques for further reducing the state space explosion.

State Space Reduction. Our executable memory model maintains elaborate state information for each thread: the local cache, as well as the read/write queues. Therefore, composing the model of full-blown Java programs along with the underlying Java memory model can result in a tremendous state space explosion. To alleviate this problem, we propose the use of our executable model for program verification as follows.

In a multithreaded program $\mathtt{Th}_1 \parallel \mathtt{Th}_2 \parallel \ldots \parallel \mathtt{Th}_n$ the user chooses only one program path in each thread $\mathtt{Th}_i$. A program path essentially encodes a choice in every control branch, and each of these choices impose constraints on program variables. Note that the program path that is chosen in thread $\mathtt{Th}_i$ need not be bounded, for example, a finitely represented unbounded loop can be chosen in $\mathtt{Th}_i$.

To represent the constraints on program variables imposed by a program path, we extend the `use` action. The syntax of `use` is extended to represent constraints on the value returned by a `use` action, for example, `use(a) = 0`. We do not specify the constraint domain here as this is not central to our methodology. For the purposes of this paper's illustration, it suffices to consider arithmetic equality and inequality constraints. Thus, if the constraint `use(a) = 0` appears in a thread, then it means that (1) `use(a)` is executed to return a value `v`, and (2) the check `v = 0` is performed, all in *one atomic step.*

Then we exhaustively check all possible execution traces made from the chosen program paths of $\mathtt{Th}_1, \ldots, \mathtt{Th}_n$, which are allowed by the JMM. Our approach is motivated by the fact that reasoning about the execution traces allowed by the JMM requires low-level understanding of the actions/data-structures of the JMM, and needs to be automated. However, reasoning about the program paths of a thread $\mathtt{Th}_i$ requires understanding the source code of $\mathtt{Th}_i$. In particular, the user will choose program paths $\pi_1, \ldots, \pi_n$ in threads $\mathtt{Th}_1, \ldots, \mathtt{Th}_n$ if he/she *suspects* a legal trace of $\pi_1, \ldots, \pi_n$ to violate the invariant being verified. This is a creative step, but still does not require the user to reason about the JMM. This task is left to the invariant checker which automatically confirms/refutes the user's suspicion.

Case Study. We now illustrate the checker's use in finding a bug in a commonly used software construction idiom

[2] We can relax this strategy to allow certain platform actions (such as `store`, `write`) to proceed even in the presence of enabled program actions.

```
Thread 1
use(Inst) = null.
lock.
use(Inst) = null.
assign(Data, newval).
assign(Inst, newptr).
unlock.
Y := use(Data).
assign(Ret, Y).
```

```
Thread 2
use(Inst)≠ null.
Y := use(Data).
assign(Ret, Y).
```

Figure 3: Program paths in two threads running Double-Checked Locking

of multithreaded Java: the Double-Checked Locking idiom. Double-Checked Locking [29] is a widely used pattern in multithreaded Java programs (see [18] for a discussion of its use). This program fragment is used for efficient lazy instantiation of a singleton class. A singleton class is a class with only one instance; for multithreaded programs this instance is shared by multiple threads. Double-Checked Locking is a program fragment for instantiation in which (a) only one instance is generated, and (b) the instance is generated only on-demand.

Consider a method `getInstance` which instantiates a singleton class `Singleton`. Clearly, `getInstance` must check whether an instance already exists, before creating an instance. This is to ensure that only one instance is generated. In a multithreaded program however, this is not enough. To avoid multiple instantiations of the Singleton class by multiple threads, the `getInstance` method must be executed as a critical section. This is achieved by synchronization, as shown in the following program fragment. Note that this program fragment will be run by multiple threads.

```
private static Singleton instance = null;
....    // the other fields
public static synchronized Singleton getInstance()
{
   if (instance == null)
          instance = new Singleton();
   return instance;
}
```

However, there is a substantial performance overhead for synchronizing on *every* invocation of `getInstance`. Double-Checked Locking [18, 29] is an efficient scheme which avoids such synchronization. Note that after the creation of an instance of the `Singleton` class is completed, there is no need to synchronize; any invocation of `getInstance` should simply return this instance. Double-Checked Locking avoids these redundant synchronizations. A program fragment implementing Double-Checked Locking is shown in Figure 4.

Any thread which invokes `getInstance` will execute this program fragment. Note that if `instance` is `null` (*i.e.*, an instance of the `Singleton` class has not yet been created), then the program fragment forces synchronization and checks whether `instance` is `null` again within the critical section. In between the first `instance == null` check and the synchronization, another thread may invoke the method `getInstance`, find that `instance` is `null`, and then create an instance of the `Singleton` class. Hence the need for the second `instance == null` check.

We have not shown the other fields of `Singleton`, which get initialized in the constructor of the `Singleton` class. We

```
private static Singleton instance = null;
....    // the other fields
public static Singleton getInstance()
{
   if (instance == null){
          synchronized (Singleton.class) {
              if (instance == null)
                  instance = new Singleton();
          }
    }
    return instance;
}
```

Figure 4: Double-Checked Locking

want to check that when multiple threads run `getInstance` concurrently, any invocation of `getInstance` always returns an initialized object, that is, the fields of the object returned by `getInstance` are not uninitialized garbage. For this purpose, it is sufficient to consider only one field of the object called `datafield`, which we assume to be initialized in the constructor.

When several threads run `getInstance` concurrently, one thread allocates a `Singleton` object and returns it, while other threads simply return the already allocated object. To show this, we can construct the program paths shown in Figure 3 with two threads running concurrently. Thread 1 allocates the `Singleton` and returns it, while Thread 2 returns the `Singleton` which has been allocated by Thread 1. In figure 3, `Inst` and `Data` denote two shared memory locations containing `instance` and `datafield` of the `Singleton` object. The location `Ret` holds the value of `o.datafield` where `o` is the object returned by `getInstance`. We could have modeled two different locations `Ret1`, `Ret2` to hold the values returned by the different threads. Modeling them as the same location only simplifies the invariant to be proved.

Initially, `Inst` = `null`, `Ret` = `null` and `Data` = `garbage`. We need to prove that `Ret` ≠ `garbage` is an invariant. To ensure that this property holds in *every* multithreaded implementation, we must show that for every execution traces allowed by the JMM, a state in which `Ret` = `garbage` is never reachable. This is accomplished automatically by our invariant checker. The program paths shown in Figure 3 are input to the checker. The checker yields a counterexample in only 0.15 seconds on a Pentium-4 1.3 GHz workstation with 1 GB of memory. In other words, the checker generates a trace where the object returned by `getInstance` can contain garbage values in the datafields. This shows that the

Double-Checked Locking program fragment is unsafe to use in a multithreaded environment. A counter-example trace constructed by the checker is:

`read(Inst), load(Inst), use(Inst) = null`	Thread 1
`lock`	Thread 1
`read(Inst), load(Inst), use(Inst) = null`	Thread 1
`assign(Data, newval), assign(Inst, newptr)`	Thread 1
`store(Inst), write(Inst)`	Thread 1
`read(Inst), load(Inst), use(Inst)` $\neq$ `null`	Thread 2
`read(Data), load(Data), Y := use(Data)`	Thread 2
`assign(Ret, Y), store(Ret), write(Ret)`	Thread 2

This corresponds to the situation where Thread 1 creates an instance by setting `Data` and `Inst`. This updates the local copies of `Data` and `Inst`. The master copy of `Inst` is then updated. Because now `Inst` $\neq$ `null`, thread 2 executes; it reads the master copy of `Data` and assigns this value to `Ret`. However, the master copy of `Data` has not been updated yet (*i.e.*, the write on `o.datafield` by Thread 1 has not completed), which causes `Ret` = `garbage`.

Thus, if the writes of `Data` and `Inst` are re-ordered then the Double-Checked Locking program fragment is unsafe to use for multithreaded programs. This re-ordering is allowed by the JMM and will be performed in many multi-processor implementations, for example, SUN SPARC, DEC Alpha. To safely use the Double-Checked Locking program fragment on such implementations, we need to turn off this re-ordering by explicitly inserting a *memory-barrier* instruction in the constructor of `Singleton`. This *memory-barrier* instructs the underlying implementation not to re-order operations across the barrier.

6. DISCUSSIONS

In this paper, we have used formal specification and verification techniques to analyze multithreaded Java programs. Our work is concentrated on formally specifying the Java Memory Model(JMM), the rules imposed by the Java language specification for any implementation of multithreading. We demonstrate (with a concrete case study) why reasoning about the JMM is necessary to verify multithreaded Java programs in a platform-independent fashion.

Even though this paper has focused on the JMM, the approach can apply to any multithreaded programming discipline. Typically, verification techniques for multithreaded programs assume a sequentially consistent execution model. The focus there is on the automation/efficiency of searching the sequentially consistent execution traces. However, multithreaded programming languages (such as Java) might impose weaker consistency models in order to allow for efficient implementations. This raises the question of generating a formal executable specification of these weak consistency models.

Weak memory consistency models [3] have traditionally been described declaratively as a set of rules. Constructing an equivalent executable formal model serves many purposes: understanding the consistency model, using the consistency model to aid verification of multithreaded programs. In this paper, we have undertaken this approach for a realistic multithreaded programming language (Java), and explored its utility. Our technique can be used to detect reorderings which produce counter-intuitive results in the execution of a multithreaded program — that is, break the programmer's intuition of sequential consistency. These reorderings can then be explicitly disabled. The rest of the re-orderings, whose effect is not visible by other threads, are allowed to proceed. This provides the efficiency of a weak memory consistency model while maintaining the programmer's intuitive abstraction of a single shared memory, as in sequential consistency.

7. ACKNOWLEDGMENTS

This work was partially supported by National University of Singapore Research Project R-252-000-095-112.

8. REFERENCES

[1] Java Specification Request (JSR) 133. Java Memory Model and Thread Specification revision. In `http://jcp.org/jsr/detail/133.jsp`, 2001.

[2] S. Adve. Memory model tutorial. In *Revising the Java Thread Specification Workshop, OOPSLA*, 2000.

[3] S.V. Adve, V.S. Pai, and P. Ranganathan. Recent advances in memory consistency models for hardware shared-memory systems. *IEEE special issue on distributed shared-memory*, 87(3), 1999.

[4] G. Barthe et al. A formal executable semantics of the Javacard platform. In *European Symposium on Programming, LNCS 2028*, 2001.

[5] E. Borger and W. Schulte. A programmer friendly modular definition of the semantics of Java. In *Formal Syntax and Semantics of Java, LNCS 1523*, 1999.

[6] D.L. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, MIT, 1999.

[7] David R. Butenhof. *Programming with POSIX threads*. Addison Wesley, 1997.

[8] P. Cenciarelli et al. An event based structural operational semantics of multithreaded Java. In *Formal Syntax and Semantics of Java, LNCS 1523*, 1999.

[9] K. Mani Chandy and J. Misra. *Parallel Program Design: a foundation*. Addison Wesley, 1988.

[10] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2), 1986.

[11] J. Corbett et al. Bandera: Extracting finite state models from Java source code. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2000.

[12] D. L. Dill. The Murφ verification system. In *Computer Aided Verification (CAV), LNCS 1102*, 1996.

[13] D.L. Dill, S. Park, and A. Nowatzyk. Formal specification of abstract memory models. In *Symposium on Research on Integrated Systems*. MIT Press, 1993.

[14] P. Godefroid. Model checking for programming languages using VeriSoft. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1997.

[15] A. Gontmakher and A. Schuster. Java consistency: non-operational characterizations for Java memory behavior. *ACM Transactions on Computer Systems*, 18(4), 2000.

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

[17] Y. Gurevich, W. Schulte, and C. Wallace. Investigating Java concurrency using Abstract State Machines. In *Abstract State Machines Workshop, LNCS 1912*, 2000.
[18] A. Holub. *Taming Java Threads*. Berkeley CA, APress, 2000.
[19] G. Holzmann and M. Smith. A practical method for verifying event driven software. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 1999.
[20] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9), 1979.
[21] Douglas Lea. *Concurrent Programming in Java : Design Principles and Patterns*. Addison Wesley, 1997.
[22] J. Maessen, Arvind, and X. Shen. Improving the Java Memory Model using CRF. In *ACM OOPSLA*, 2000.
[23] J. Manson and W. Pugh. Core semantics of multithreaded Java. In *ACM Java Grande Conference*, 2001.
[24] J.S. Moore. 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.
[25] G. Naumovich, G.S. Avrunin, and L.A. Clarke. Data flow analysis for checking properties of concurrent Java programs. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 399–410, 1999.
[26] G. Naumovich, G.S. Avrunin, and L.A. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. In *ESEC/FSE, LNCS 1687*, pages 338–354, 1999.
[27] S. Park and D.L. Dill. An executable specification and verifier for relaxed memory order. *IEEE Transactions on Computers*, 48(2), 1999.
[28] W. Pugh. Fixing the Java Memory Model. In *ACM Java Grande Conference*, 1999.
[29] D. Schmidt and T. Harrison. Double-checked locking: An optimization pattern for efficiently initializing and accessing thread-safe objects. In *3rd Annual Pattern Languages of Program Design conference*, 1996.
[30] S.D. Stoller. Model checking multithreaded distributed Java programs. In *SPIN Workshop on Model Checking of Software, LNCS 1885*, 2000.
[31] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *IEEE International Conference on Automated Software Engineering*, 2000.
[32] XSB. The XSB logic programming system v2.2, 2000. Available for downloading from `http://xsb.sourceforge.net/`.

APPENDIX

We present the proof of *equivalence* of our executable Java Memory model and the rule-based memory model given in the Java Language Specification (JLS) [16]. The proof follows from two lemmas of soundness and completeness. First we formalize the notion of a trace.

DEFINITION 3 (TRACE). *Given a program with $n > 1$ threads, an execution trace is a mapping*

$$\mathbb{N} \to \mathit{Act} \times \{1, \ldots, n\}$$

where Act is the set of permissible actions by any thread and $\{1, \ldots, n\}$ denotes the thread id. Thus, an execution trace is a sequence of actions of the various threads.

The permissible actions are {`use`, `assign`, `load`, `store`, `read`, `write`, `lock`, `unlock`}. Some of these actions (`read`, `write`, `lock` and `unlock`) involve interaction between a thread and the main memory process. Note that the notion of an execution trace does not distinguish between the program actions (`use`, `assign`, `lock`, `unlock`), and platform actions (`load`, `store`, `read` and `write`) for a particular thread. Given the above definition, we can prove soundness of our JMM specification as follows.

LEMMA 1 (SOUNDNESS). *Any execution trace of a multithreaded Java program which is allowed by our executable memory model is also allowed by the rules of the JLS [16].*

Proof: We prove this by showing that any execution trace in our executable model obeys all the rules in the JLS. The detailed proof given below is a case-by-case analysis of the rules in JLS.

Execution Order Rules. Each trace allowed by our model satisfies the first four rules of execution order in JLS by definition (Definition 3). Also, $\mathtt{lock}_i/\mathtt{unlock}_i$ actions are performed jointly by $Thread_i$ and MM since they are invoked by $Thread_i$ and they modify the state of the MM process. To guarantee that each $\mathtt{load}_i$ is uniquely paired with a preceding $\mathtt{read}_i$, note that $\mathtt{load}_i$ dequeues an entry from some $rd\_q_{i,j}$. Hence it is uniquely paired with the $\mathtt{read}_i$ instruction which enqueued this entry to $rd\_q_{i,j}$ previously. The pairing of $\mathtt{store}_i$ with $\mathtt{write}_i$ is guaranteed by our executable model similarly.

Rules about Variables. Let α_P be a trace of a program P, s.t. α_P is allowed by our executable memory model. Because our model invokes the `use`, `assign` actions as per their occurrence in threads of program P, the first rule is satisfied by α_P. The second rule requires a $\mathtt{store}_i(j)$ to intervene between $\mathtt{assign}_i(j)$ and $\mathtt{load}_i(j)$. This is ensured in our model by the $dirty_{i,j}$ bit. An $\mathtt{assign}_i(j)$ will always set $dirty_{i,j}$ to true. Now, a $\mathtt{load}_i(j)$ can be applied only if $rd\_q_{i,j}$ is non-empty, that is, there is a pending $\mathtt{read}_i(j)$. As $rd\_q_{i,j}$ is empty before the execution of $\mathtt{assign}_i(j)$, a $\mathtt{read}_i(j)$ must intervene between $\mathtt{assign}_i(j)$ and $\mathtt{load}_i(j)$. Now $\mathtt{read}_i(j)$ can be applied only if $dirty_{i,j}$ is false. Since $\mathtt{store}_i(j)$ is the only action which can set $dirty_{i,j}$ to false, therefore we guarantee that $\mathtt{store}_i(j)$ must intervene.

The third rule requires $\mathtt{assign}_i(j)$ to intervene between $\mathtt{load}_i(j)$ / $\mathtt{store}_i(j)$ and a subsequent $\mathtt{store}_i(j)$. This is also ensured by the $dirty_{i,j}$ bit. After the execution $\mathtt{load}_i(j)$ / $\mathtt{store}_i(j)$, the bit $dirty_{i,j}$ is guaranteed to be false (refer Figure 1). Also, the guard of a subsequent $\mathtt{store}_i(j)$ requires $dirty_{i,j}$ to be true. Therefore, there must be an intervening action which sets $dirty_{i,j}$ to be true. Since $\mathtt{assign}_i(j)$ is the only such action, it must intervene.

The fourth and fifth rules require $\mathtt{assign}_i(j)$ or $\mathtt{load}_i(j)$ to precede the first occurrence of $\mathtt{use}_i(j)$ or $\mathtt{store}_i(j)$. The $\mathtt{use}_i(j)$ requires $stale_{i,j}$ to be false. Since all *stale* bits are initially true, therefore actions setting $stale_{i,j}$ to false must precede $\texttt{use}_i(j)$. The only actions setting $stale_{i,j}$ to false are $\texttt{assign}_i(j)$, $\texttt{load}_i(j)$. Similarly, we can show that the first occurrence of $\texttt{store}_i(j)$ must be preceded by $\texttt{assign}_i(j)$ by taking the $dirty_{i,j}$ bit into consideration.

The sixth rule requires every $\texttt{load}_i(j)$ to be preceded by a corresponding $\texttt{read}_i(j)$ which transmits the same r-value as $\texttt{load}_i(j)$. As mentioned before, this is ensured in our model by the $rd\_q_{i,j}$ queue. Since $\texttt{load}_i(j)$ dequeues values which were enqueued into $rd\_q_{i,j}$ by a preceding $\texttt{read}_i(j)$, we can see that this rule is always satisfied by traces in our executable model.

The dependence between $\texttt{store}_i(j)$ and $\texttt{write}_i(j)$ as demanded by the seventh rule is shown similarly (by considering the $wr\_q_{i,j}$ queue).

The last rule requires (a) $\texttt{read}_i(j)$ to be in the same order as corresponding $\texttt{load}_i(j)$ actions, (b) $\texttt{write}_i(j)$ actions to be in the same order as the corresponding $\texttt{store}_i(j)$ actions, (c) if a $\texttt{store}_i(j)$ precedes $\texttt{load}_i(j)$, then the corresponding $\texttt{write}_i(j)$ precedes the corresponding $\texttt{read}_i(j)$, and (d) if a $\texttt{load}_i(j)$ precedes $\texttt{store}_i(j)$, then the corresponding $\texttt{read}_i(j)$ precedes the corresponding $\texttt{write}_i(j)$. Requirement (a) and (b) are ensured by the FIFO discipline of $rd\_q_{i,j}$ and $wr\_q_{i,j}$ respectively. Requirement (c) is ensured by the guard of $\texttt{read}_i(j)$ which is enabled only if $wr\_q_{i,j}$ is empty, that is, there is no pending $\texttt{write}_i(j)$. Similarly, requirement (d) is ensured by disabling $\texttt{store}_i(j)$ if $rd\_q_{i,j}$ is non-empty. Note that the guard $\neg dirty_{i,j}$ is used for the $\texttt{read}_i(j)$ action to prevent a deadlock. Without that guard, a $\texttt{read}_i(j)$ can follow an $\texttt{assign}_i(j)$. But after that the $\texttt{load}_i(j)$ can be enabled only if there is an intervening $\texttt{store}_i(j)$ and the $\texttt{store}_i(j)$ can be enabled only if there is an intervening $\texttt{load}_i(j)$. The guard ensures that a $\texttt{read}_i(j)$ cannot follow an $\texttt{assign}_i(j)$ without intervening $\texttt{store}_i(j)$ and $\texttt{write}_i(j)$.

Rules about Locks. The first rule requires that only one thread at a time owns a lock. This is ensured in our model, since the condition $\forall k \neq i\ lock\_cnt_k = 0$ in the guard of $\texttt{lock}_i$ ensures that all threads other than i do not own the lock. The second rule requires that only a thread owning a lock can execute an unlock. This is ensured in our model since $\texttt{unlock}_i$ is executed only if $lock\_cnt_i > 0$ *i.e.* thread i owns the lock.

Rules about Interaction of Locks and Variables. The first rule requires $\texttt{store}_i(j)$ and the corresponding $\texttt{write}_i(j)$ to intervene between an $\texttt{assign}_i(j)$ and an $\texttt{unlock}_i$. This is ensured in our model since the guard of $\texttt{unlock}_i$ is true provided $\mathsf{empty}(wr\_q_{i,j})$ holds (no pending $\texttt{write}_i(j)$) and $\neg dirty_{i,j}$ holds (no pending $\texttt{store}_i(j)$).

The second rule requires $\texttt{assign}_i(j)$ or $\texttt{read}_i(j)/\texttt{load}_i(j)$ pair to intervene between a $\texttt{lock}_i$ and a subsequent $\texttt{use}_i(j)$ / $\texttt{store}_i(j)$. In our model, after the $\texttt{lock}_i$ action is executed we must have $\forall j\ stale_{i,j} \wedge \neg dirty_{i,j}$. Therefore $\texttt{use}_i(j)$ / $\texttt{store}_i(j)$ actions are not enabled. The only action setting $dirty_{i,j}$ is $\mathsf{assign}_i(j)$, and the only actions resetting $stale_{i,j}$ are $\mathsf{load}_i(j)$ and $\mathsf{assign}_i(j)$. Therefore, one of these actions must intervene. Furthermore, $load_i(j)$ can intervene only if $rd\_q_{i,j}$ is non-empty. Since $rd\_q_{i,j}$ is empty when $\texttt{lock}_i$ is executed, the $\texttt{read}_i(j)$ corresponding to the intervening $\texttt{load}_i(j)$ must also take place after $\texttt{lock}_i$. □

We prove completeness of our JMM specification w.r.t. the model described in [16].

Lemma 2 (Completeness). *Any execution trace of a multithreaded Java program which is allowed by the rules of the Java language specification [16] is also allowed by our executable memory model.*

Proof: Consider some execution trace α allowed by [16] which is not allowed by our model. Consider the *first* action a in α which is disallowed by our model but is allowed by [16]. Since there are only eight actions in both models, a can only be one of them. The eight cases are shown below, and for each of them a contradiction is obtained. Thus, no such trace α may exist.

- $a = \texttt{use}_i(j)$: Then $stale_{i,j}$ must be true. Then, a occurs before any $\texttt{assign}_i(j)$ / $\texttt{load}_i(j)$ or after occurrence of $\texttt{lock}_i$ (by induction on application of our rules). This is disallowed by [16] as well.
- $a = \texttt{assign}_i(j)$: Then $rd\_q_{i,j}$ is non-empty. If this trace is allowed by [16] then the trace α cannot have a subsequent $\texttt{load}_i(j)$ until $dirty_{i,j}$ is reset, which is only possible by $\texttt{store}_i(j)$. But $\texttt{store}_i(j)$ again cannot be executed by [16] if $rd\_q_{i,j}$ is non-empty (because then $\texttt{store}_i(j)$ will precede a $\texttt{load}_i(j)$ but the corresponding `read`, `write` will be in reverse order). Thus thread i cannot progress and such a trace α is disallowed.
- $a = \texttt{load}_i(j)$: If $rd\_q_{i,j}$ is empty, no value can be loaded. Execution of $\texttt{load}_i(j)$ is disallowed by [16] also in such cases.
- $a = \texttt{store}_i(j)$: If $dirty_{i,j}$ is false, then there is no preceding $assign_i(j)$ without a subsequent $store_i(j)$ (by induction on our rules). This is disallowed in [16] by the rules about variables (third rule). If $rd\_q_{i,j}$ is non-empty then $\texttt{store}_i(j)$ will precede a $\texttt{load}_i(j)$ but the corresponding `read`, `write` will be in reverse order. Again if $wr\_q_{i,j}$ is full, no value can be stored. All these cases are again disallowed by [16].
- $a = \texttt{read}_i(j)$: If $dirty_{i,j}$ is true or $wr\_q_{i,j}$ is non-empty, then there is a preceding `assign` leading to a pending $store_i(j)$ or $write_i(j)$ operation. This will lead to a `store` preceding a `load` where the corresponding `read` and `write` are in reverse order, violating the last rule about variables in [16]. Again non-full $rd\_q_{i,j}$ must trivially hold.
- $a = \texttt{write}_i(j)$: Non-empty $wr\_q_{i,j}$ must trivially hold.
- $a = \texttt{lock}_i$: If $lock\_cnt_k > 0$ where $k \neq i$, then thread k contain has executed `lock` for which the corresponding `unlock` has not been performed yet (by induction). If $rd\_q_{i,j}$ is non-empty then there will be a `load` after `lock` whose `read` has been executed before `lock`. If $dirty_{i,j}$ is true, then there will be a `store` after `lock` whose `assign` is executed before `lock`. All of these situations violate the rules for locks and their interaction with variables in [16].
- $a = \texttt{unlock}_i$: If $lock\_cnt_i = 0$ then thread i does not own the lock (by induction). If $dirty_{i,j}$ is true or $wr\_q_{i,j}$ is non-empty for some variable j, then there is a preceding `assign` leading to a pending $store_i(j)$ or $write_i(j)$ operation. Each of these situations are again disallowed by the rules for locks and their interaction with variables in [16]. □