Java Memory Model aware Software Validation

Arnab De
Indian Inst. of Science
arnabde03@gmail.com

Abhik Roychoudhury
National Univ. of Singapore
abhik@comp.nus.edu.sg

Deepak D'Souza
Indian Inst. of Science
deepakd@csa.iisc.ernet.in

ABSTRACT

The Java Memory Model (JMM) provides a semantics of Java multi-threading for any implementation platform. The JMM is defined in a declarative fashion with an allowed program execution being defined in terms of existence of "commit sequences" (roughly, the order in which actions in the execution are committed). In this work, we develop an operational approximation of the JMM. The immediate motivation of this work lies in integrating a formal specification of the JMM with software model checkers. We show how our operational description of the JMM can be integrated into a Java Path Finder (JPF) style model checker for Java programs.

1. INTRODUCTION

With increased push towards concurrent and parallel programming, developing tools and techniques for reliable concurrent programming is a must. Programmers tend to find parallel/concurrent programming harder than sequential programming — owing to the many possible program executions for any given program input. Moreover, programming languages like Java and C# describe the semantics of multi-threading via a language level Memory Model. The language level memory model is somewhat synonymous to the semantics of multi-threading in the concerned programming language — it describes which writes can be visible to a program read operation. As a simple example, one may consider the following program fragment with shared variables `A`, `B` and local variables `r1, r2`, all initially 0.

Thread 1	Thread 2
`A = 1;`	`r1 = B;`
`B = 1;`	`r2 = A;`

We would normally assume that at the end of the program we cannot have $r1 == 1 \wedge r2 == 0$ since `B` is set after `A` in Thread 1. However, in order to allow underlying compiler/hardware optimizations the language level memory model may allow re-ordering of the writes in Thread 1, thereby making the result $r1 == 1 \wedge r2 == 0$ possible at the end of the program. Such intricacies in the memory model makes formal reasoning about programs particularly difficult.

The Java Memory Model [12] essentially describes all allowed behaviors of a multi-threaded Java program on any implementation platform. The formal description of the Java Memory Model is *declarative* — it describes a notion of execution (a partial order of actions) and then defines what is an "allowed execution". However, the test of whether an execution is allowed is not directly executable. The model defines an execution as an allowed execution if there exists a commit sequence (sequence of sets of committing actions in the execution) satisfying certain properties. However, given an execution, even testing whether it is an allowed execution is non-trivial. It will require the construction of a sequence of commit-sets. Unfortunately, no algorithm on how to construct these commit sets is given in the Java Memory Model.

In this paper, we develop an operational approximation of the Java Memory Model (JMM) and use it for automated software validation via model checking. Our memory model is an under-approximation of the JMM in terms of allowed executions. We note that in (explicit-state) model checking we need to (a) traverse program executions, and (b) on-the-fly evaluate whether these are "allowed" executions. No efficient algorithm for construction and traversal of allowed program executions from the declarative description is known. This is where our operational under-approximation fits in. We integrate our operational characterization of the JMM with an on-the-fly software model checker that checks for assertion violations. *We call our operational Memory Model as OpMM*. Since OpMM is a strict under-approximation of the JMM, an OpMM-aware model checker can be used to find bugs in a given program, rather than verify it.

Note that although the JMM guarantees Sequential Consistency (SC) for race-free programs [12], programs in general may have races for high performance or other reasons. A memory model aware model checker like ours is most suitable for validating programs which are not proven to be race-free.

2. BACKGROUND AND RELATED WORK

Memory consistency models have been used in shared memory multiprocessors for many years. Details of hardware memory models appear in [1, 5].

The interest in programming language memory models is relatively new. Among the different programming languages, work on the Java Memory model (JMM) has received the most attention. A formal description of the JMM has been presented in [7, 12] with inputs from researchers and developers alike. This model is presented in a non-operational, declarative style. Even checking whether a given program execution is allowed by the programming language semantics is not straightforward – it involves showing the existence of certain "commit sets" (capturing the order in which program actions are committed). No procedure for efficiently checking (say in polynomial time) whether a given execution of a given program is an allowed behavior is known.

Program validation in a memory model aware fashion was first studied in the context of hardware memory models. Park and Dill [13] developed executable memory models for SPARC architectures and used them to verify synchronization routines. Gopalakrishnan et. al. [8] have used SAT solving to check whether an execution is allowed by a multi-processor memory model, in particular the Intel Itanium memory model. In recent works, Burckhardt, Alur and Martin [3] study bounded model checking of concurrent data types under relaxed hardware memory models.

In our past work [10], we have developed a bytecode level invariant checker for C#, which proceeds in a memory model aware fashion. This required us to formally specify the C# memory model and integrate inside a bytecode level model checker. However, the C# memory model is simpler than the JMM, in that it can be specified as a re-ordering table describing which pairs of operations can be re-ordered. As we will describe now, the JMM specification is more complex, and cannot be captured by a re-ordering table. Hence our first task is to develop an operational version of the JMM, which we then use for bytecode level model checking.

3. OVERVIEW OF OUR APPROACH

We now describe the JMM and OpMM models via several examples. As our first example, we present the following program fragment, where `r1` - `r6` are locals, `p`, `q` are shared variables and initially `p == q` and `p.x == 0`:

Thread 1:	Thread 2:
`r1 = p;`	`r6 = p;`
`r2 = r1.x;`	`r6.x = 3;`
`r3 = q;`	
`r4 = r3.x;`	
`r5 = r1.x;`	

Figure 1: Example 1

A compiler optimization might replace the last statement of Thread 1 by `r5 = r2;` as they are assigned same expression `r1.x` and the value of the expression is not changed in Thread 1. But this optimization may lead to the behavior `r2 == r5 == 0, r4 == 3` after the execution, which is *not* allowed by Sequential Consistency on the original program.

In order to support such optimizations, JMM allows this behavior. In the resulting execution, the reads into `r2` and `r5` see the write of the initial value and the read into `r4` sees the write by Thread 2. The commit sequence for validating this execution (as required by JMM [12]) is the following. In the first step, we can commit all actions except `r4 = r3.x`. As all non-committed reads must see a write that *happens-before* it, the read into `r4` sees the initial write of value `0`. In the next step, the read into `r4` is committed, but it still gets the value `0` as the committing reads must see a write which is committed earlier as well as happens-before the read. In the next step, when the read is committed at least one step earlier, it can see any write that has been committed before (but not necessarily happens-before) — hence the read now can see the write `r6.x = 3` from Thread 2, resulting in the desired behavior.

As mentioned, validating an execution in JMM requires finding a commit sequence according to the rules given in [12]. There is no efficient algorithm for this purpose other than generating and testing all possible commit sequences. The problem of generating all legal executions of a given Java program is even more difficult.

Now we show how we can *generate* an execution in OpMM that displays the desired behavior in the program of Figure 1. In OpMM, actions are executed in a total-order consistent with the *program-order*, but it is *not* necessary for a read to see the last write in that total order (thus differing from SC). In fact, a *state* in OpMM consists of different *views* of the heap, one by each thread, instead of a single global heap. Each location of the heap contains a set of values which a read is allowed to see. In this example, in an execution where the write to `r6.x` in Thread 2 is scheduled before the last two reads of Thread 1, the heap location for `r3.x` and `r1.x` contains both the values `0` and `3`. Hence it is possible for `r4` to read the value `3` and `r5` to read the value `0`, and thus producing the desired behavior.

OpMM uses *synchronization-actions* to mask the writes that should not be seen by a later read according to JMM. Let us consider the following example where `r1` is a local and `x`, `l` are shared variables, `x` initialized to `0` and `l` is initially unlocked.

Thread 1:	Thread 2:
`lock l;`	`lock l;`
`x = 1;`	`x = 2;`
`r1 = x;`	`unlock l;`
`unlock l;`	

Figure 2: Example 2

In any execution of this program, the only value `r1` can get is `1`, from the write in Thread 1. In any execution in OpMM, two cases might happen: Thread 1 can execute entirely before Thread 2 or vice versa. No other interleaving is possible as OpMM obeys *mutual exclusion* of lock operations. In the first case, the write `x = 1` masks the initial value of `x` (i.e., `0`) as in the same thread, a newer write masks the older one. Hence when the read `r1 = x` is executed, only one value `1` can be seen. In the second case, when Thread 1 locks the same monitor unlocked by Thread 2, their views of the heap are synchronized. Consequently the write of `1` in Thread 1 masks the write by Thread 2 as well as the initial value of `x`, resulting in the desired behavior.

It should be noted that OpMM is a strict under-approximation of JMM i.e. there are executions which are allowed by JMM but not by OpMM. Let us consider the following example where all variables are initialized to `0`.

Thread 1:	Thread 2:
`r3 = x;`	`r2 = y;`
`if(r3 == 0)`	`x = r2;`
` x = 1;`	
`r1 = x;`	
`y = r1;`	

Figure 3: Example 3

The behavior `r1 == r2 == r3 == 1` after the execution is *allowed* by JMM [12], but it is not allowed in OpMM. To allow this behavior, JMM needs the reads of `x` to see a different write before and after committing. In fact, the read `r1 = x;` initially sees the write `x = 1;`, which does not even occur in the final execution. As a result, it becomes difficult to model such behaviors operationally because it loses a clear notion of trace — a sequence of actions that modify the states. Hence OpMM does not model the entire JMM. We believe this operational under-approximation can useful to system designers like compiler writers who may not wish to follow the intricacies of the full JMM [12].

4. DESCRIPTION OF OPMM

4.1 Programs

The programming language we consider is an abstract version of the Java bytecode. Local variables are of the form rn. The shared variables can be accessed through (static or non-static) field access of a reference type local variable. C, f, v denote any class, field, volatile field, respectively. All statements are prefixed with a thread id while describing the semantics (written as $t\colon c$). Threads, classes, objects, monitors and volatiles have unique ids.

Statements. The statements we are considering are one of the following forms: (1) Writing to shared variables: `ri.f = rj`, (2) Reading from shared variables: `ri = rj.f`, (3) Writing to volatile variables: `ri.v = rj`, (4) Reading from volatile variables: `ri = rj.v`, (5) Locking a monitor: `lock ri`, (6) Unlocking a monitor: `unlock ri`, (7) Creating an object: `ri = new C`, (8) Starting a thread: `start ri`, (9) Interrupting a thread, (10) Determining whether a thread is interrupted, (11) Detecting whether a thread is alive, (12) `skip`, (13) Assignment statements involving only local variables, (14) Control statements, (15) Method calls and returns, and (16) Throwing and catching of exceptions. In this paper we concentrate on statements of type 1 - 11 as they are relevant to the memory model. Other statements must follow the intra-thread semantics as defined in [9].

Program. The program consists of one or more threads. Each thread has one or more methods. There is one thread with `main` method (referred as *main* thread hereafter). Other threads must have a `run` method.

Structure of the Heap. The heap can be seen to be made of *object areas*, each object area allocated to a particular object. As Java is a strongly typed language, reference type local variables point to one of these object areas or `null`. The object areas are divided into *cells*, one cell for each field of the object allocated in that object area. The cells do not contain a *value*, but a *write-list*. A *write-list* can contain three types of elements: (i) *write-item (WI)* which is a $\langle value, tid\rangle$ pair, (ii) a *release-item (RI)* which is a $\langle lock, tid\rangle$ pair or a $\langle volatile, tid\rangle$ pair, and (iii) an *acquire-item (AI)* which is again a $\langle lock, tid\rangle$ pair or a $\langle volatile, tid\rangle$ pair.

4.2 Structure of States and Transitions

Program State. The **program state** (denoted by Ω) consists of one local state for each *active* thread, and a global state for monitors and volatiles. Each local state L_t (t is the corresponding *tid*) consists of the following parts:

• A stack of frames, one frame for each outstanding method call in the thread (as mentioned in [11]), denoted by S_t. Each frame, among other things (as specified in The Java Virtual Machine Specification [11]), contain values of the local variables of the corresponding method. We denote the local variable map of the top frame by σ_t.

• A view of the heap (denoted by H_t) which is a function from cells to a *write-list*, as specified in Section 4.1.

The global state consists of monitor states and volatile states. The monitor state is a mapping M from monitors to tuples of the form $\langle tid, lockcount\rangle$. When the *lockcount* is 0, the *tid* part also has a value 0 (we assume that there is no thread with tid 0). The volatile state V is a mapping from volatile ids to the value of the corresponding volatile variable. It should be noted that *our notion of program state has no global heap*, except for the values of volatiles and monitors which are globally maintained. The state might contain other information like thread status, class information for objects etc which are required by Java Virtual Machine [11]. As this information is not directly related to the memory model, we do not include it in our description here.

State Transitions. A program statement can cause state transitions. If a statement c changes the program state from Ω to Ω', we write it as $\langle c, \Omega\rangle \rightarrow \Omega'$. We did not include *program counter(pc)* in our description of states as that is not relevant for our work. Execution order of instructions is described informally in Section *4.3.1*.

4.3 Operational Semantics

We present the operational semantics of OpMM which is a sound but incomplete approximation of JMM. The semantics is given in form of inference rules. The consequence of any of these rules is a state transition by a statement. The premises are the conditions that must be satisfied to enable the transition given as consequence.

The following **conventions** are used while describing the semantics for the sake of simplicity: In the inference rules, k ranges over thread ids and h ranges over cells in the universal quantifiers. Field access of an object returns the corresponding cell. $\Omega[A \rightarrow B]$ denotes a state same as Ω but the component A has been changed to B. $\Omega[F|x\colon v]$ is a shorthand expression representing a state which is equivalent to Ω except $F(x)$ has a new value v. Here we give a formal semantics of the statements relevant to the memory model. Other statements follow the intra-thread semantics in [9].

Algorithm 1 Mask&Read

Input: *write-list wl, thread-id tid*
Output: *Set of write-items ws*

*/*Lock id, Vol id and Tid are the sets of lock ids, volatile ids and thread ids*/*
WriteSet: Set of write-items
ThSet: Tid $\rightarrow$ {*marked, unmarked, undef*}
AcSet: (Lock id $\cup$ *Vol id)* $\rightarrow$ {*marked, unmarked, undef*}

```
$WriteSet \leftarrow \emptyset$
$\forall k \in Lock\ id \cup Vol\ id\colon AcSet(k) \leftarrow undef$
$\forall t \in Tid\colon ThSet(t) \leftarrow undef$
$ThSet(tid) \leftarrow unmarked$
for all element e in wl, starting from the newest do
    if e is a write-item $\langle v, t\rangle$ then
        if ThSet(t) = undef then
            $WriteSet \leftarrow WriteSet \cup \{\langle v, t\rangle\}$
        else if ThSet(t) = unmarked then
            $WriteSet \leftarrow WriteSet \cup \{\langle v, t\rangle\}$
            $ThSet(t) \leftarrow marked$
        else if ThSet(t) = marked then
            skip
        end if
    else if e is an acquire-item $\langle k, t\rangle$ then
        if AcSet(k) = marked or ThSet(t) = undef then
            skip
        else
            $AcSet(k) \leftarrow ThSet(t)$
        end if
    else if e is a release-item $\langle k, t\rangle$ then
        if ThSet(t) = marked or AcSet(k) = undef then
            skip
        else
            $ThSet(t) \leftarrow AcSet(k)$
        end if
    end if
end for
$ws \leftarrow WriteSet$
return ws
```

4.3.1 Restrictions on Execution Order.

The execution should start at the `main` method of the *main* thread. At each step, only one statement is executed from the active threads, if the statement is *enabled* according to the semantics as given in

(Wr)

$$\frac{v = \sigma_t(\texttt{rj}) \quad h = \sigma_t(\texttt{ri}).\texttt{f} \quad \Omega' = \Omega[\forall k\colon H_k(h) \to \text{Append}(H_k(h), \text{WI}(\langle v, t\rangle))]}{\langle t\colon \texttt{ri.f} = \texttt{rj}, \Omega\rangle \to \Omega'}$$

(Rd)

$$\frac{\langle v, t'\rangle \in \mathit{Mask\&Read}(H_t(\sigma_t(\texttt{rj}).\texttt{f}, t))}{\langle t\colon \texttt{ri} = \texttt{rj.f}, \Omega\rangle \to \Omega[\sigma_t|\texttt{ri}\colon v]}$$

(Ul)

$$\frac{\begin{array}{c} M(\sigma_t(\texttt{ri})) = \langle t, n\rangle \\ \Omega' = \Omega[\forall k, \forall h\colon H_k(h) \to \text{Append}(H_k(h), \text{RI}(\langle \sigma_t(\texttt{ri}), t\rangle)); M \to \text{Unlock}(\sigma_t(\texttt{ri}))] \end{array}}{\langle t\colon \texttt{unlock ri}, \Omega\rangle \to \Omega'}$$

(L-1)

$$\frac{\begin{array}{c} M(\sigma_t(\texttt{ri})) = \langle 0, 0\rangle \\ \Omega' = \Omega[\forall k, \forall h\colon H_k(h) \to \text{Append}(H_k(h), \text{AI}(\langle \sigma_t(\texttt{ri}), t\rangle)); M \to \text{Lock}(\sigma_t(\texttt{ri}), t)] \end{array}}{\langle t\colon \texttt{lock ri}, \Omega\rangle \to \Omega'}$$

(L-2)

$$\frac{\begin{array}{c} M(\sigma_t(\texttt{ri})) = \langle t, n\rangle \\ \Omega' = \Omega[\forall k, \forall h\colon H_k(h) \to \text{Append}(H_k(h), \text{AI}(\langle \sigma_t(\texttt{ri}), t\rangle)); M \to \text{Lock}(\sigma_t(\texttt{ri}), t)] \end{array}}{\langle t\colon \texttt{lock ri}, \Omega\rangle \to \Omega'}$$

(V-Wr)

$$\frac{v = \sigma_t(\texttt{rj}) \quad \Omega' = \Omega[\forall k, \forall h\colon H_k(h) \to \text{Append}(H_k(h), \text{RI}(\langle \sigma_t(\texttt{ri}).\texttt{v}, t\rangle))]}{\langle t\colon \texttt{ri.v} = \texttt{rj}, \Omega\rangle \to \Omega'[V|\sigma_t(\texttt{ri}).\texttt{v}\colon v]}$$

(V-Rd)

$$\frac{v = V(\sigma_t(\texttt{rj}).\texttt{v}) \quad \Omega' = \Omega[\forall h\colon H_t(h) \to \text{Append}(H_t(h), \text{AI}(\langle \sigma_t(\texttt{rj}).\texttt{v}, t\rangle))]}{\langle t\colon \texttt{ri} = \texttt{rj.v}, \Omega\rangle \to \Omega'[\sigma_t|\texttt{ri}\colon v]}$$

Figure 4: OpMM rules for read/write, lock/unlock and volatile read/write statements

Section *4.3.2*, producing a total-order of *actions*. Statements from a thread are executed in program order. Statements from threads other than *main* can by executed only after they are started (using a `start` statement). Executions in other threads should start at the `run` method of the corresponding thread.

4.3.2 Semantics of Statements.

We now present the operational semantics rules of OpMM for the different language constructs of Java.

Write statement. A write statement appends the write to the write-list for the corresponding cell of every local state. Rule (Wr) in Figure 4 is the operational semantics rule for the write statement. Here Append(l, i) appends the element i to the list l.

Read statement. A read statement updates the local variable with a value from the set of write-items returned by the *Mask&Read* function applied on the corresponding local write-list and the reading thread-id. Rule (Rd) in Figure 4 is the operational semantics rule for the read statement. Note that the *Mask&Read* function is defined as an algorithm in Algorithm 1. The main function of the algorithm is to prevent a read r from seeing a write w if there is another write w' such that $w \xrightarrow{hb} w' \xrightarrow{hb} r$ where *hb* is the happens-before relation as defined in [12]. Informally, it maintains two functions: *ThSet* and *AcSet* maps the thread ids and synchronization object ids (locks/ volatiles) respectively to *undef*, *unmarked* or *marked*. Whenever a thread t is not synchronized with the input thread, $\mathit{ThSet}(t) = \mathit{undef}$. Similarly when a volatile/monitor k is not acquired by any thread t in *ThSet* (i.e. $\mathit{ThSet}(t) \neq \mathit{undef}$), the Mask&Read algorithm treats it as $\mathit{AcSet}(k) = \mathit{undef}$. Algorithm 1 initially puts the input thread id into *ThSet* as *unmarked* and traverses the input write-list, starting from the newest item. When it finds an *acquire-item* whose thread id belongs to *ThSet*, it puts the lock/ volatile id into the *AcSet* with the same marking. Similarly, when it sees a *release-item* whose lock/ volatile id belongs to *AcSet*, it puts the thread id into the *ThSet* with the same marking. A write is put into the *write-set* if the thread id is *unmarked* or *undef* in *ThSet*. If the thread id is *unmarked*, it is *marked* after seeing the write. If a thread id is already marked, the writes by that thread are not put into *write-set*.

Unlock statement. An unlock statement can execute only when the executing thread holds the lock. It appends the corresponding *release-item* to all the write-lists. It also changes the monitor state. Rule (Ul) in Figure 4 is the operational semantics rule for the unlock statement. Here Unlock(m) reduces the lockcount of $M(m)$ and if it reaches 0, changes the tid of $M(m)$ to 0. Recall that M is a mapping from monitors to the threads locking a monitor.

Lock statement. The lock statement can execute if the corresponding monitor is not locked or locked by the same thread. It appends the corresponding *acquire-item* to all the write-lists, and changes the monitor state. Rules (L-1) and (L-2) in Figure 4 are the rules for the lock statement. Here Lock(m, t) increments the lockcount of $M(m)$ and if was 0, changes the tid of $M(m)$ to t.

Volatile write statement. Volatile writes directly update the global state for volatiles. It also appends the corresponding *release-item* to each write-list of each local state. Rule (V-Wr) in Figure 4 is the operational semantics rule for the volatile write statement.

Volatile read statement. Volatile reads read the value directly from the global state of volatiles and update the local variable. It also appends the corresponding *acquire-item* to each write-list of each local state. Rule (V-Rd) in Figure 4 is the operational semantics rule for volatile reads.

Object creation statement. The object creation statement `ri = new C` creates a new object area, extending H_k for all threads k. A unique id is assigned to it. The local map for `ri` is updated to hold reference to the new object. After the object is created, the constructor call can be treated as a normal method call.

Thread creation statement. When a thread t' is spawned by a thread t, the spawned thread inherits the object areas from the spawning thread, i.e. $H_{t'}$ becomes equal to H_t. Moreover, all the write-lists of the state are appended by a *release-item* $\langle l, t\rangle$ and an *acquire-item* $\langle l, t'\rangle$, in that order, where l is a unique lock id that does not occur in the execution.

Thread termination and interruption. When a thread t is terminated, then after modification of the local state (such as thread status), a *release item* $\langle l, t\rangle$ is appended to all the write-lists. When an action from some other thread t' detects that t has terminated, an *acquire-item* $\langle l, t'\rangle$ is appended to all the write-lists, before any other modification in the state by that action. l is a unique lock id that does not occur in the execution. Similar steps are followed for thread interruption.

5. EXPRESSIVE POWER OF THE MODEL

In this section, we show that the memory model semantics proposed in Section 4.3 is an under-approximation of the Java Memory Model in terms of allowed executions. We have also shown that OpMM is strictly weaker than the hardware memory model TSO — details appear in [6].

5.1 Traces and Occurs-Before Relation

A program *trace* allowed by the proposed semantics is an interleaving of the actions $c_0, \dots c_n$ such that

$$\Omega_0 \xrightarrow{c_0} \Omega_1 \dots \Omega_n \xrightarrow{c_n} \Omega_{n+1}$$

where Ω_0 is an initial state. The order of execution must obey the restrictions given in Section *4.3.1* and each transition $\langle c_j, \Omega_j\rangle \rightarrow \Omega_{j+1}$ must be allowed by the semantics given in Section *4.3.2*.

Given a trace t, we say $c_i \xrightarrow{ob} c_j$, if c_i appears before c_j in the trace t. We call $\xrightarrow{ob}$ as the occurs-before relation.

5.2 Construction of an Execution from a Trace

Given a *trace* (as defined in Section 5.1), we first need to construct an *execution* (according to the definition of execution given in [12]) corresponding to it.

The execution is $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb}\rangle$ where P is the program, A is the set of actions in the trace, $a \xrightarrow{po} b$ if they belong to the same thread and $a \xrightarrow{ob} b$, $a \xrightarrow{so} b$ if both of them are synchronization actions and $a \xrightarrow{ob} b$, $W(a) = b$ if a is a read action and it reads a write by action b from its write-list, $V(a)$ is the value written by a write action a, the synchronizes-with relation $\xrightarrow{sw}$ and the happens-before $\xrightarrow{hb}$ are as defined as in [12]. For example, an unlock action on a monitor m synchronizes with all "subsequent" (as per the synchronization order relation $\xrightarrow{so}$) lock actions on m that were performed on any thread. A write to a volatile variable u synchronizes-with all 'subsequent" (as per the synchronization order relation $\xrightarrow{so}$) reads of u performed by any thread. The happens-before relation is the transitive closure of program order and synchronizes-with order, that is, $\xrightarrow{hb} = (\xrightarrow{po} \cup \xrightarrow{sw})^*$. It should be noted that $\xrightarrow{po}$ is a total order among the actions from the same thread and $\xrightarrow{so}$ is a total order among synchronization actions, as required by JMM. If a trace allowed by OpMM leads to a state that violates an assertion, the corresponding execution constructed from the trace will also violate the assertion. As our notion of under-approximation is based on allowed executions, it is enough to show that the execution constructed from the trace is allowed by JMM.

5.3 Some Properties of OpMM

We now establish some key properties of the OpMM model, which will be useful for establishing that OpMM is a strict under-approximation of the JMM.

THEOREM 1. *If $a \xrightarrow{hb} b$ in the constructed execution, then $a \xrightarrow{ob} b$ in the trace.*

PROOF. As $\xrightarrow{hb} = (\xrightarrow{po} \cup \xrightarrow{sw})^*$, it is enough to show that po and sw of the constructed execution are consistent with the *ob* relation of the trace. If $a \xrightarrow{po} b$, then by construction of po edges (in Section 5.2), $a \xrightarrow{ob} b$. Let $a \xrightarrow{sw} b$. Then either $a \xrightarrow{po} b$ or $a \xrightarrow{so} b$ or a changes the thread status which is observed by b. As all of these relations are consistent with *ob*, *sw* is also consistent with *ob*. □

LEMMA 2. *During the execution of* Mask&Read *algorithm (Algorithm 1), for all tid t and sync id (lock id or volatile id) l, $ThSet(t)$ and $AcSet(l)$ can change their values from* undef *to* unmarked *to* marked*, but in no other order.*

This can be proved by induction on the number of iterations of the main loop of Alg. 1.

LEMMA 3. *Let $t'\colon a \xrightarrow{hb} t\colon r$ where a is an action from thread t', r is a read action from thread t and the result of a is present in the write-list for r in H_t. Then after a is encountered during the execution of* Mask&Read *algorithm for r (Algorithm 1), $ThSet(t') \neq undef$.*

PROOF. We have $t'\colon a \xrightarrow{hb} t\colon r$ and $\xrightarrow{hb} = (\xrightarrow{po} \cup \xrightarrow{sw})^*$. We prove by induction on number of po or sw edges between a and r.
Base case: If the number of edges is one, then $t\colon a \xrightarrow{po} t\colon r$ (as r cannot be target of an sw edge). $ThSet(t)$ is set to *unmarked* in line 4. By Lemma 2, it cannot change to *undef* later when we encounter a.
Induction step: By induction hypothesis, if $\hat{t}\colon \hat{a} \xrightarrow{hb} r$ using n po or sw edges, then after encountering $\hat{a}$, $ThSet(\hat{t}) \neq undef$. If $a \xrightarrow{po} \hat{a}$, then $t' = \hat{t}$, hence proved. Otherwise, if $a \xrightarrow{sw} \hat{a}$, a is a *release-item* and corresponding *AcSet* is not *undef* (as $ThSet(\hat{t}) \neq undef$ while encountering $\hat{a}$). Hence by lines 22 - 26, $ThSet(t') \neq undef$ after encountering a. □

COROLLARY 4. *If $t'\colon w \xrightarrow{hb} t\colon r$, where w is a write action to the same location as read r, then before w is encountered during the execution of* Mask&Read *algorithm for r (Algorithm 1), $ThSet(t') \neq undef$.*

The proof follows from Lemma 3 and the fact that the last edge in the happens-before path will always be *po* in case of a write.

LEMMA 5. *If $t_1\colon w_1 \xrightarrow{hb} t_2\colon w_2 \xrightarrow{hb} t\colon r$, where w_1 and w_2 are writes to the same location as read r then before w_1 is encountered during the execution of* Mask&Read *algorithm for r (Algorithm 1), $ThSet(t_1) = marked$.*

PROOF. w_2 will be encountered before w_1. After w_2 is encountered, by Lemma 4 and lines 9 - 13 of Algorithm 1, $ThSet(t_2) = marked$. Using Lemma 2, by induction on the number of po and sw edges between w_1 and w_2, this Lemma can be proved. □

THEOREM 6. *If* $t_1 : w_1 \xrightarrow{hb} t_2 : w_2 \xrightarrow{hb} t : r$, *where* w_1 *and* w_2 *are writes to the same location as read* r, *then* $w_1 \notin ws$ *(returned by* Mask&Read *algorithm (Algorithm 1)) executed during processing of* r.

This follows directly from Lemma 5 and lines 12 - 13 of the Algorithm 1.

5.4 Well-formedness of Executions

Using the above properties we can show that our executions are well-formed as per the JMM [12]. Details of these arguments appear in [6]. The key to our argument is showing that any execution obeys happens-before consistency. It can be shown by contradiction. Two cases can happen:

(A) *There is a* r *such that* $r \xrightarrow{hb} W(r)$ In our model, if $r \xrightarrow{hb} W(r)$, then we have $r \xrightarrow{ob} W(r)$ in the trace (by Theorem 1). But this means when r was executed, $W(r)$ was not part of the write-list, implying r cannot read $W(r)$.

(B) *There is a* r *such that* $W(r) \xrightarrow{hb} w \xrightarrow{hb} r$ This is not possible by Theorem 6.

5.5 Causality Requirements

Finally, we need to show that the executions generated by our model meet the causality requirements given in JMM (see Section 4.5 of the JMM paper [12]). For this purpose, we need to find set of actions to commit in each step and a validating execution for each of them. We can simply follow the *occurs-before* order of the actions of the thread. We commit the instructions in that order and use the execution up to that point to validate it. It must be noted that when we commit a read, in that step, the read sees a write that happens before it (there is at least one such write due to initialization), but in the next step we can change it to the write it finally sees as that write has already been committed.

This completes the proof of the fact that OpMM is a strict under-approximation of the JMM.

5.6 Disallowed Behavior

Note that OpMM is a strict under-approximation of JMM. This naturally raises the following question: what kind of program behaviors are allowed by the JMM, but disallowed by OpMM? We address this issue in the following.

A *write-seen edge* (denoted as *ws*) is an edge from a write action w to a read action r such that $W(r) = w$. In some executions allowed by JMM, the *happens-before* and *write-seen* edges form a cycle. For example, in the example of Fig. 3, there is a cycle $\texttt{r1} = \texttt{x}; \xrightarrow{hb} \texttt{y} = \texttt{r1}; \xrightarrow{ws} \texttt{r2} = \texttt{y}; \xrightarrow{hb} \texttt{x} = \texttt{r2}; \xrightarrow{ws} \texttt{r1} = \texttt{x};$.

As $a_1 \xrightarrow{hb} a_2$ and $a_1 \xrightarrow{ws} a_2$ both imply $a_1 \xrightarrow{ob} a_2$ and *ob* is a partial order, such cycles are not allowed in OpMM. As a result, any execution containing such cycles are disallowed in OpMM.

6. EXPERIMENTS ON MODEL CHECKING

We have integrated OpMM into a bytecode level model checker for Java. This allows us to use OpMM for property checking of Java programs. This clearly goes beyond conventional software validation, which ignores the language level memory model, and implicitly assume Sequential Consistency as the execution model.

Implementation issues. Our model checker is an explicit state on-the-fly model checker in the style of the Java Path Finder (JPF) [15]. It takes the bytecode representation of a program as input and detects whether any assertion specified in the program can be violated under OpMM. As the state space of the input program can be infinite, we do not guarantee termination in general, but a depth-bound can be imposed on the depth-first search to force termination. While integrating OpMM to a model checker, we observe that in OpMM the write-lists can grow unboundedly and thus, the size of a state can also grow unboundedly. We developed a technique to prune the write-lists in many cases (but not all). We modify the *Mask&Read* algorithm (presented in Algorithm 1) to delete the write-items if the corresponding *ThSet* value is *marked* when the item is encountered during the traversal of the write-list. We also delete the release-items and acquire-items if the corresponding *ThSet* value is *marked* after processing of the items. The modified algorithm appears in [6], where we also prove that the modified algorithm returns the same results as Algorithm 1 (as a result of which the proof that OpMM is a strict under-approximation of JMM is unaffected).

Results. We present the results of running our tool on some subject programs in Table 1. For each subject program we check one invariant assertion encoding the correctness of the program. The subject program `dcl` is the original version of double-checked locking [14]. This program fragment is used for efficient lazy instantiation of a singleton class. 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. The assertion checks whether the singleton object reference is correctly constructed after lazy initialization. This assertion can be violated under JMM [2]. Our tool successfully detects the bug.

The subject program `dcl-vol` is the version of double-checked locking where the singleton object reference is declared `volatile`. This version works correctly under JMM, and this is validated by our tool. In other words, we check that the singleton object is properly constructed before it is referenced. The subject program `singleton` represents the same singleton pattern as `dcl`, but here, instead of using double-checked locking, the lock is taken every time the reference is requested. This version works correctly under JMM and our tool validates it.

Subject programs `peterson` and `dekker` are traditional algorithms for establishing mutual exclusion. The assertion checks whether mutual exclusion property of these programs can be violated. Both of them fail under JMM. Our tool detects the assertion violations.

Program	#bytecode	#threads	time (sec)	#states explored
`dcl`	70	2	0.25	87
`dcl-vol`	70	2	0.31	267
`singleton`	67	2	0.31	297
`peterson`	108	2	0.25	99
`dekker`	135	2	0.27	101

Table 1: Experimental Results

Table 1 shows the results of our experiments. All experiments were done on a 1.6 GHz dual core machine with 1 GB main memory. The time reported is in seconds. In cases where the assertion is violated, the first counterexample is reported. Otherwise, the entire state space is exhausted.

We note that our checker is currently a prototype. In future, there exists scope for (a) optimizing the checker's implementation and (b) integrating state space reduction methods like partial order reduction [4].

7. REFERENCES

[1] S.V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, pages 66–76, Dec. 1996.

[2] D. Bacon et al. The "double-checked locking is broken" declaration. http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.

[3] S. Burckhardt, R. Alur, and M. Martin. Checkfence: Checking consistency of concurrent data types on relaxed memory models. In *PLDI*, 2007.

[4] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[5] D.E. Culler and J. Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1998.

[6] A. De, A. Roychoudhury, and D. D'Souza. Java memory model aware software verification. Technical report, 2008. http://people.csa.iisc.ernet.in/~arnabde/opmm-full.pdf.

[7] JSR-133 expert group. Jsr-133: Java memory model and thread specification, August 2004.

[8] G. Gopalakrishnan, Y. Yang, and G. Lindstrom. QB or not QB: An efficient execution verification tool for memory orderings. In *CAV*, 2004.

[9] J. Gosling, W. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[10] T.Q. Huynh and A. Roychoudhury. A memory model sensitive checker for C#. In *Formal Methods Symposium (FM)*, 2006.

[11] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[12] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, New York, NY, USA, 2005. ACM.

[13] S. Park and D.L. Dill. An executable specification and verifier for relaxed memory order. *IEEE Transactions on Computers*, 48(2), 1999.

[14] D.C. Schmidt and T. Harrison. Double-checked locking. *Pattern languages of program design 3*, pages 363–375, 1997.

[15] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *IEEE international conference on Automated software engineering (ASE)*, 2000.