Java Memory Model Aware Sofware Verification

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

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

Deepak D'Souza
CSA Department
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

Modern programming languages such as Java and C# have embraced concurrency as a mainstream feature. Furthermore, with the increasing push towards multi-core platforms in the coming years, the importance of concurrent programming is steadily increasing. As an evidence of the growing importance of concurrent/parallel programming on multi-processor platforms, one may notice the launching of two Universal Parallel Computing Research Centers by Microsoft and Intel in collaboration with academia in March 2008. Increasingly, it is felt that computer architects will be moving towards processors with many cores (Intel already has a research processor with 80 cores!), and concurrent/parallel programming will become much more mainstream.

With this 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;`

In this example, we would normally assume `r1` $\leq$ `r2` at the end of the program. In other words, 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 [13] 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 report, we develop an operational approximation of the Java Memory Model (JMM) and use it for automated software verification 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. Clearly, a declarative description of allowed executions is not amenable to efficient construction and traversal of allowed program executions. 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 an under-approximation of the JMM, an OpMM-aware model checker can be used to find bugs in a given program, rather than verify it. In other words, given a program P — any assertion violation reported by our OpMM-aware model checker is indeed a violation in P, but not vice-versa. Thus, our method and tool can be viewed as an aid for finding subtle concurrency bugs, rather than as a concurrent program verifier.

It should be noted that although the JMM guarantees Sequential Consistency (SC) for race-free programs [13], 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.

The technical contributions of this report can be summarized as follows.

- We develop an *operational* under-approximation of the JMM (the JMM itself is described in declarative style). Our formalization, called the OpMM, is based on structured operational semantics rules of the language constructs. The core issue of which writes can be seen by a read operation in a program is clarified through an algorithm.

  We believe that our operational description of the JMM can be useful for understanding of the model by system designers such as compiler and JVM writers. Our model is stronger than the JMM (allows less behaviors) and the system designers can possibly use this model to implement the JMM in practice.

- We have also integrated our OpMM model with a Java Path Finder (JPF) style on-the-fly model checker for Java programs. We discuss specific optimizations which can be used to improve the scalability of our checker. We believe such memory model specific optimizations are key to building memory model aware software model checkers.

The rest of this report is organized as follows. The next section discusses related work, and the general background about the recent developments in memory models for programming languages. Section 3 presents a general overview of our approach. Section 4 presents our memory model OpMM. Section 5 establishes that our model OpMM is an under-approximation of the Java memory model, and also explores the relationship of OpMM with well-known hardware memory models like SPARC's Total Store Order (TSO). Section 6 discusses the issues in implementing OpMM inside a software model checker, whereas Section 7 discusses experimental results obtained from an OpMM-aware model checker for Java bytecode. Section 2 discusses related work and concludes the report.

2. BACKGROUND AND RELATED WORK

2.1 Memory Models

Memory consistency models have been used in shared memory multiprocessors for many years. The simplest model of memory consistency was proposed by Lamport and is called Sequential Consistency [11]. This model allows operations across threads to be interleaved in any order. Operations within each thread are however constrained to proceed in program order. SC serves as a very simple and intuitive model of execution to the programmer. However, it disallows most compiler and hardware optimizations. For this reason, shared memory multiprocessors have employed relaxed memory models, which allow certain reordering of operations within a thread. The Total Store Order (TSO) and Partial Store Order (PSO) models, supported by SUN SPARC architectures [6], differ from SC in that they allow memory write operations to be bypassed. By bypassing we mean that even if a write operation is stalled, a succeeding memory operation can execute. Memory read operations, however, are blocking in TSO and PSO. The TSO model only allows reads to bypass previous writes. PSO is a more relaxed model than TSO as it allows both reads and writes to bypass previous writes. Unlike TSO and PSO, the WO and RC models allow non-blocking reads (i.e., reads may be bypassed). However they classify memory operations as data operations (normal reads/writes) and synchronization operations (lock/unlock). Details of hardware memory models appear in [1, 5]. Note that all the memory models only allow reorderings which do not violate the uniprocessor program dependencies within a thread.

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, 13] 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). There appears to be no procedure for even efficiently checking (say in polynomial time) whether a given execution of a given program is an allowed behavior under the JMM. Consequently, any attempt to reason about programs in a memory model aware fashion remains difficult — an important motivation which drives our work.

2.2 Memory Model aware Verification

Program validation in a memory model aware fashion was first studied in the context of hardware memory models. Park and Dill [14] 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. Their work uses a SAT checker to verify observational equivalence between sequentially consistent behaviors of a concurrent program and behaviors allowed by 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. Such a memory model specification can be directly integrated into a state space search, since if a pair of operations could be re-ordered we could simply allow both possibilities (reorder them or not) during state space exploration. As we will describe now, the JMM specification is more complex, and cannot be captured by a re-ordering table. Allowed executions in the JMM are only defined by means of certain global properties which effectively restrict which write operations can be seen by a read operation in the program. Hence our first task is to develop an operational version of the JMM, which we then use for bytecode level program verification.

3. OVERVIEW OF OUR APPROACH

In this section, we present an overview of the OpMM, a memory model defined in terms of its operational semantics. OpMM is strictly *stronger* than the Java Memory Model (JMM), that is, any execution is allowed by OpMM is also allowed by JMM but the converse is not true.

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`:

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 ==`

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

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 [13]) 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 we have seen, validating an execution in JMM requires finding a commit sequence according to the rules given in [13]. 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 because of the declarative nature of JMM.

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 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 [13], 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. The reasons behind choosing this particular under-approximation for OpMM are following:

1. It can be presented as an easily implementable and intuitive operational semantics. 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 [13].

2. It is more relaxed than many of the traditional relaxed memory models (we show its relationship with the hardware memory model TSO), but still it can be proved to be an under approximation of the JMM.

4. DESCRIPTION OF OPMM

In this section, we present a detailed description of OpMM. It is defined in terms of its operational semantics, which we present here. We concentrate on a selected set of programming language constructs relevant to the memory model. We also describe the model of the heap and structure of the states and configurations as required by the semantics.

4.1 Programming Language under Consideration

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 simple statements are prefixed with a thread id while describing the semantics (written as $t : c$). 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: `interrupt ri`.
10. Detecting whether a thread is interrupted: `isinterrupted ri`.
11. Detecting whether a thread is alive: `isalive ri`.
12. `skip`.
13. Assignment statements involving only local variables.
14. Control statements.
15. Method calls and returns.
16. Throwing and catching of exceptions.

In this report 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.

4.2 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.

As we will see in Section 4.3, the cells do not contain a *value*, but a *write-list*.

A *write-list* can contain three types of elements:

1. A *write-item (WI)* which is a $\langle value, tid \rangle$ pair.
2. A *release-item (RI)* which is a $\langle lock, tid \rangle$ pair or a $\langle volatile, tid \rangle$ pair.
3. An *acquire-item (AI)* which is again a $\langle lock, tid \rangle$ pair or a $\langle volatile, tid \rangle$ pair.

4.3 Structure of States and Configurations

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 [12]), denoted by S_t. Each frame, among other things (as specified in The Java Virtual Machine Specification [12]), 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 4.2.

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[1].
- 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 [12]. As these information are not directly related to the memory model, we do not include them 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 *4.4.1*.

4.4 Operational Semantics

In this section, we present the operational semantics of OpMM which is a sound but incomplete approximation of JMM (the proof of under-approximation is given in Section 5). 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.
- $\Omega[F|x: v]$ is a shorthand expression representing a state which is equivalent to Ω except $F(x)$ has a new value v.

In this report, we give a formal semantics of the statements that are relevant to the memory model. Other statements should follow the intra-thread semantics given in [9].

[1]We assume that there is no thread with tid 0.

4.4.1 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 Section 4.4.2, 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.

4.4.2 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 [13]. 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, $ThSet(t) = undef$. Similarly when a volatile/monitor k is not acquired by any thread t in *ThSet* (i.e. $ThSet(t) \neq undef$), the Mask&Read algorithm treats it as $AcSet(k) = undef$. 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 the corresponding 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 operational semantics rule 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.

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*}

1: $WriteSet \leftarrow \emptyset$
2: $\forall k \in Lock\ id \cup Vol\ id\colon AcSet(k) \leftarrow undef$
3: $\forall t \in Tid\colon ThSet(t) \leftarrow undef$
4: $ThSet(tid) \leftarrow unmarked$
5: **for all** *element e in wl, starting from the newest* **do**
6: **if** *e is a write-item* $\langle v, t\rangle$ **then**
7: **if** *ThSet(t) = undef* **then**
8: *WriteSet* $\leftarrow$ *WriteSet* $\cup \{\langle v, t\rangle\}$
9: **else if** *ThSet(t) = unmarked* **then**
10: *WriteSet* $\leftarrow$ *WriteSet* $\cup \{\langle v, t\rangle\}$
11: *ThSet(t)* $\leftarrow$ *marked*
12: **else if** *ThSet(t) = marked* **then**
13: *skip*
14: **end if**
15: **else if** *e is an acquire-item* $\langle k, t\rangle$ **then**
16: **if** *AcSet(k) = marked or ThSet(t) = undef* **then**
17: *skip*
18: **else**
19: *AcSet(k)* $\leftarrow$ *ThSet(t)*
20: **end if**
21: **else if** *e is a release-item* $\langle k, t\rangle$ **then**
22: **if** *ThSet(t) = marked or AcSet(k) = undef* **then**
23: *skip*
24: **else**
25: *ThSet(t)* $\leftarrow$ *AcSet(k)*
26: **end if**
27: **end if**
28: **end for**
29: *ws* $\leftarrow$ *WriteSet*
30: **return** *ws*

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 in the H_k for all threads k. A unique id is assigned to it. The local map for `ri` is updated to hold reference for 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

(Wr)

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

(Rd)

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

(Ul)

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

(L-1)

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

(L-2)

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

(V-Wr)

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

(V-Rd)

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

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

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 when a thread is interrupted and some other thread determines it later.

5. EXPRESSIVE POWER OF THE MODEL

In this section, we show that the memory model semantics proposed in Section 4.4 is an under-approximation of the Java Memory Model. We also show that OpMM is strictly weaker than the hardware memory model TSO.

5.1 Traces and Occurs-Before Relation

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

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

where Ω_0 is an initial state. The order of execution must obey the restrictions given in Section *4.4.1* and each transition $\langle c_j, \Omega_j\rangle \rightarrow \Omega_{j+1}$ must be allowed by the semantics given in Section *4.4.2*.

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 [13]) 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 $\xrightarrow{sw}$ is defined as in [13]. In particular, 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 v synchronizes-with all 'subsequent" (as per the synchronization order relation $\xrightarrow{so}$) reads of v performed by any thread.

- The happens-before $\xrightarrow{hb}$ is defined as in [13]. 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, and the associated Mask&Read algorithm (Alg. 1). These properties will be useful for establishing that OpMM is a strict under-approximation of the JMM, that is, any execution allowed by OpMM is also allowed by JMM.

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

PROOF. As happens-before relation is transitive closure of program-order and synchronizes-with relations, 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.*

PROOF. This can be proved by simple induction on the number of iterations of the main loop of the algorithm. □

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 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 the lemma 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), $ThSet(t') \neq undef$.*

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

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

5.4 Well-formedness of Executions

In this section, we show that the execution corresponding to a trace (as constructed in Section 5.2) is well-formed as per the JMM (see [13], Section 5.3).

Each read of a variable x sees a write to x. This is obvious as according to our model, a read statement returns a value from the write-list of the corresponding cell and the write-list is populated by write statements to the same cell.

The synchronization order has an order less than or equal to omega. This is again obvious as a synchronization action *occurs* only after finitely many steps.

Synchronization order is consistent with the program order. This is true because occurs-before is consistent with the program order (according to the construction in Section 5.2).

Lock operations are consistent with mutual exclusion. As no two instructions can *occur* at same time in the trace, at most one instruction can attempt to take the lock at one point of time. It can lock the monitor only if the monitor is not locked or it is locked by the same thread (Section *4.4.2*). This establishes the mutual exclusion.

The execution obeys intra-thread consistency. Instructions that operate on local data follow intra-thread semantics. Read operations immediately update the locals with the new value read so that the next instruction can see the updated local variables. Write operations do not change the local variables. Hence the execution obeys intra-thread semantics.

The execution obeys synchronization-order consistency. As operations on volatiles modify the global state, and they are totally ordered (by *so*), a volatile read always sees a write that immediately precedes it in synchronization order.

The execution obeys happens-before consistency. It can be shown by contradiction. Two cases can happen:

1. *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 $W(r)$. Hence contradiction.
2. *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 [13]). 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, 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 Bahavior

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.

5.7 Relationship with TSO

We conclude the section by showing that OpMM is weaker than traditional weak memory models such as SUN SPARC's *Total Store Order (TSO)* [6]. As TSO is a hardware memory model, to facilitate the comparison, we assume that the *lock* and *unlock* operations in TSO is comparable to those of OpMM. We also assume that there are only four types of actions: *read*/*write* of shared variables and *lock*/*unlock* of monitors.

An *execution* allowed by TSO is a tuple $\langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb} \rangle$, semantics of each component is same as Section 5.2. There must be a total order of the actions in A, consistent with *po* and *so*, but unlike Sequential Consistency, a read does not need to see the last write to the same location in that total order. A read is allowed to see any write before it in the total order which does not *occur before* the last write before a read or a lock action from the same thread. Moreover, if there is a write to the same location by the same thread in the trace, the read cannot see any write before that in that total order. Each thread must maintain intra-thread consistency. Mutual exclusion of synchronization actions are obeyed.

We now show that for each such execution in TSO, there exists a trace in OpMM producing the same execution (as per the construction given in Section 5.2). The trace can be taken to be same as the total order in the given TSO execution. To show that this trace $c_0, c_1, \ldots$ is allowed by OpMM, we need to show that there exists states $\Omega_0, \Omega_1, \ldots$ such that $\Omega_0 \xrightarrow{c_0} \Omega_1 \xrightarrow{c_1} \ldots$ under OpMM. If c_i is any action other than read, $\Omega_i \xrightarrow{c_i} \Omega_{i+1}$ is valid under OpMM as the trace is consistent with po and so, intra-thread consistent and obeys mutual exclusion. If c_i is a read, we need to show that it is allowed to see the same write as in the TSO execution.

The only way OpMM would prevent the read r to see the write w was if there is a write w' such that $w \xrightarrow{hb} w' \xrightarrow{hb} r$. If such is the case, then either w' is from the same thread as r or there is a lock action l by the reading thread such that $w \xrightarrow{ob} w' \xrightarrow{ob} l \xrightarrow{ob} r$. In both cases, r cannot see w under TSO too. Hence, the execution constructed from this trace will be same as the execution under TSO.

Algorithm 2 Mask&Read2

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**
      delete e from wl
    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
    if *ThSet(t) = marked* **then**
      delete e from wl
    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
    if *ThSet(t) = marked* **then**
      delete e from wl
    end if
  end if
end for
ws $\leftarrow$ *WriteSet*
return *ws*

---

6. IMPLEMENTATION ISSUES

In this section, we discuss implementation issues for implementing OpMM. In particular, we observe that in our proposed operational semantics, the write-lists can grow unboundedly and thus, the size of a state can also grow unboundedly. To solve this problem, we propose a technique to prune the write-lists. It should be noted that this technique might not be able to prune the write-lists in all cases. In particular, if in an execution, a thread keeps writing to a shared location inside a loop and there is no read to this location, the corresponding write-list may grow unboundedly.

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 is presented in Algorithm 2. To show that the proposed model is still an under-approximation of JMM, it is enough to show that by deleting an item, we do not *expose* a *masked write*. The following Theorem 9 formalizes this reasoning.

LEMMA 7. *If* $t_1 : a_1 \xrightarrow{hb} t_2 : a_2$ *and both of the actions are part of the same write-list in some execution allowed by OpMM (where the semantics of the read statement is given by Algorithm 2), then it cannot the case that* a_2 *gets deleted strictly before* a_1.

PROOF. The Lemma can be proved by contradiction. If the statement is not true, then there must be a read action such that by its invocation of Algorithm 2, a_2 is deleted but a_1 remains in the list.

As $a_1 \xrightarrow{hb} a_2$, hence $a_1 \xrightarrow{ob} a_2$ (by Theorem 1). By the proposed semantics, a_2 is encountered *before* a_1 during the particular invocation of Algorithm 2. As a_2 is deleted, hence *ThSet(t_2) = marked* after processing a_2. Using Lemma 2 and by induction on number of po or sw edges between a_1 and a_2, it can be shown that either a_1 is a write and t_1 is *marked* while encountering a_1, or a_1 is sync action and t_1 is *marked* after processing a_1. In both cases, the item gets deleted. Hence we reach a contradiction. □

LEMMA 8. *If* $w \xrightarrow{hb} r$ *but there is no intervening write, then* w *remains in the write-list after invocation of Algorithm 2 for read* r.

PROOF. This can be shown by induction on number of po or sw edges between w and r. □

THEOREM 9. *Corresponding invocations of Algorithm 1 and Algorithm 2 return same sets of write-items.*

PROOF. The proof of this theorem directly follows from Lemma 7 and Lemma 8. □

7. EXPERIMENTS

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

Our model checker is an explicit state on-the-fly model checker in the style of the Java Path Finder (JPF) [16]. 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.

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. We now describe the subject programs and the assertions checked for each of them.

The subject program `dcl` represents the original version of double-checked locking [15]. 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. 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` represents the version of double-checked locking where the singleton object reference is declared `volatile`. This version works correctly under JMM, and this is verified by our tool. In other words, we verify 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 verifies 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 the future, there exists scope for (a) optimizing the checker's implementation and (b) integrating well-known state space reduction methods like partial order reduction [4].

8. DISCUSSION

In this report, we develop OpMM, an operational description of the Java Memory Model. We formally prove that our description is an under-approximation, that is, any behaviors appearing in OpMM also appear in the JMM, but not vice-versa. We use OpMM for JMM-aware reasoning about programs, by building an OpMM-aware bytecode level invariant checker. Invariant violations found in such a checker can be used for finding subtle concurrency bugs

which may not surface in a standard program verifier assuming Sequential Consistency.

9. REFERENCES

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

[2] David 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] David L. Weaver and Tom Germond, Prentice Hall Publishers. *The SPARC Architecture Manual : Version 9*, 1994.

[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] James Gosling, Bill Joy, Guy Steele, and Gilad 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] Leslie Lamport. How to make a multiprocessor computer that correctlt executes multiprocess programs. *IEEE Transactions on Computers*, 28(9), 1979.

[12] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[13] 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.

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

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

[16] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *IEEE international conference on Automated software engineering (ASE)*, 2000.