Verification of the Java Causality Requirements

Sergey Polyakov and Assaf Schuster

Department of Computer Science
Technion - Israel Institute of Technology
Technion City, Haifa

Abstract. The Java Memory Model (JMM) formalizes the behavior of shared memory accesses in a multithreaded Java program. Dependencies between memory accesses are acyclic, as defined by the JMM causality requirements. We study the problem of post-mortem verification of these requirements and prove that the task is NP-complete. We then argue that in some cases the task may be simplified by tracing the actual execution order of `Read` actions in each thread. Our verification algorithm has two versions: a polynomial version, to be used when the aforementioned simplification is possible, and a nonpolynomial version – for short test sequences only – to be used in all other cases. Finally, we argue that the JMM causality requirements could benefit from some fine-tuning. Our examination of causality test case 6 (presented in the public discussion of the JMM) clearly shows that some useful compiler optimizations – which one would expect to be permissible – are in fact prohibited by the formal model.

Keywords: Concurrency, Complexity, Java, Memory Model, Multithreading, Verification, Shared Memory

1 Introduction

In the past, memory models where formalized mostly for hardware implementations or abstract hardware/software systems [16,13,4,11]. In recent years there has been an effort to provide memory models for popular programming languages [5,10,7,8]. One reason is that some difficulties arise when implementing multithreading only by means of libraries [9]. The revised version of the Java Memory Model (JMM) [3,15] is the result of long-term efforts and discussions[17]. It is now expected to be the prototype for the multithreaded C++ memory model [7].

This article studies the problem of post-mortem verification of the JMM causality requirements. Post-mortem methods check that memory behavior is correct for a given execution of the multithreaded program. They analyze *traces* – test prints produced either by running an instrumented code or by running some special debug version of the JVM or simulator.

The Java system consists of the compiler, which converts a source code to a *bytecode*, and the Java Virtual Machine (JVM), which executes the bytecode. The Java Memory Model considers the system as a whole: what it describes is the behavior of the *virtual bytecode* – an imaginary bytecode that may be provided by a non-optimizing compiler. Such bytecode is a straight mapping of the Java source code and as such it expresses the programmer's comprehension of the program. This certainly simplifies matters when it comes to analyzing multithreaded programs, but complicates the memory model: The JMM must regulate optimizations performed by the real Java compiler and run-time system (the JVM). It is important that these optimizations not produce *causal loops* – a behavior by which some action "causes itself."

A causal loop might occur if some speculatively executed action affects another action upon which it depends. This phenomenon may adversely affect the behavior even of a correctly synchronized program. Causal acyclicity is therefore implied not only for Java but for most weak memory models. In release consistency [13], for example, causal loops are prevented informally by "respecting the local dependencies within the same processor" [12]. An example of a formal definition of causal acyclicity is given in [6,12], where it is expressed through the "reach condition."

In weak memory models, causal acyclicity is necessary to guarantee that programs which are data-race free on sequentially consistent platforms will maintain sequentially consistent behavior on weaker platforms as well. In Java, causal acyclicity also provides important safety guarantees for badly synchronized programs: out-of-thin-air values [3] are prohibited, for example. Because the JMM requires that only minimal restrictions be imposed on implementations, it gives a relaxed formalization of causal acyclicity that makes allowances for most compiler optimizations.

The relaxed definition of the JMM is not for free, however. The model does not give any guidelines for the implementor – it is a specification of user demands. While there are relatively simple rules for implementing synchronization [2], no such rules are provided for causality requirements. Therefore, when an implementation uses aggressive optimization methods, it is hard to decide if it meets the specification. Thus, the JMM needs well-defined verification methods.

The JMM guarantees causal acyclicity by requiring that actions be *committed* in some partial order: an action cannot depend on an action that succeeds it. Thus, the problem of verifying the causality requirements may be thought of as a search for some valid commitment order. We found that the problem is NP-complete because there may be a non-polynomial number of partial orders that can be considered as candidates for the commitment order. To make the problem solvable, we propose that the actual execution order of `Read` actions be traced by each thread. This order can then be used by the verification program as an outline of the commitment order. The method can only be applied, however, for JVM implementations in which it is possible to trace the actual execution order of the `Read` actions, which must also be known to comply with all admissible commitment orders. If the commitment order cannot be traced, the problem is still NP-complete with respect to execution length, and only short traces can be verified.

The formal JMM causality requirements are illustrated by the causality test cases [1]. Our manual verification of these test cases have shown that JMM is more restrictive than expected: Test case 6 could not be verified although it must be allowed by the JMM. In our minds, this illustrates a discrepancy between the formal model and its intuitive implications. One of the contributions of this paper is a proposed fix for this problem.

The rest of the article is organized as follows. Section 2 describes our assumptions about test sequences. In section 3 we prove that verification of the JMM causality requirements is NP-hard in the general case and suggest a method for solving the problem. We then provide a verification algorithm (section 4). Finally, in section 5 we discuss some problems and present our conclusions.

2 The Problem of Testing the JMM Causality Requirements

We briefly recount here how a Java program is executed. Readers interested in the complete JMM definition are referred to Appendix A, as well as to [3,15].

The execution of a Java program is described by a tuple $\langle P, A, \overset{po}{\rightarrow}, \overset{so}{\rightarrow}, W, V, \overset{sw}{\rightarrow}, \overset{hb}{\rightarrow} \rangle$, where

- P is the program,
- A is the set of actions,
- $\overset{po}{\rightarrow}$ is the program order,
- $\overset{so}{\rightarrow}$ is the total order on synchronization actions,
- W is a write-seen function,
- V is a function that assigns a value to each `Write`,
- $\overset{sw}{\rightarrow}$ is a partial order that arranges pairs of actions with release-acquire semantics,
- $\overset{hb}{\rightarrow}$ is the happens-before order.

The values of shared variables that appear in the execution must be consistent with various requirements – synchronization, causality, and the like.

We investigate the simplest case of testing the JMM causality requirements. Consider an execution E that has the following properties. It is finite. There are no synchronization actions, actions on final fields, or external actions. There are no happens-before relations that result from executing finalizers. We assume that the threads are spawned at the very beginning of the program; therefore $\stackrel{so}{\rightarrow}$ and $\stackrel{sw}{\rightarrow}$ are irrelevant and $\stackrel{hb}{\rightarrow}$ complies with $\stackrel{po}{\rightarrow}$. Note, however, that initialization `Write`s may be considered to be performed by a special thread and precede all other actions.

The rest of the execution tuple $(P, A, \stackrel{po}{\rightarrow}, W, V)$ is represented by the program and traces; the program is represented by the virtual bytecode. For simplicity, we use pseudocode in our examples and proofs. We have a separate trace for each thread, and a trace of a thread is called a sequence. Each sequence consists of [GUID:](`Read`/`Write`)(x_i, t_i, g_i) actions, where GUID is a unique identifier for the action, x_i is some variable, and t_i is the value which is read / written by the operation. For `Read`, g_i is the GUID of the `Write` operation that provided the value that is read. (We assume that the trace provides the write-seen function.) For `Write`, g_i is its own GUID.

We provide a simplified version of the causality requirements that matches our special case. Execution $E = \langle P, A, \stackrel{po}{\rightarrow}, W, V \rangle$ is *well-formed* if it has the following properties:

1. Each `Read` of a variable x sees a `Write` to x.
2. Each thread sequence is identical to that which would be generated by the correct uniprocessor.

A well-formed execution E is consistent with the causality requirements if all actions in A can be iteratively *committed*. Each step i of the commitment process must correspond to the set of committed actions C_i ($C_0 = \emptyset$, $C_i \subset C_{i+1}$). For each C_i (starting from C_1) there must be a well-formed execution E_i that contains C_i. Series $C_0, \ldots, C_i, \ldots$ and $E_1, \ldots, E_i, \ldots$ must have the following properties (less complex than in the general case):

1. $C_i \subseteq A_i$;
2. committed `Write`s produce final values (the same values as in E);
3. for each E_i , all `Read`s in C_i see `Write`s that have just been committed in C_{i-1};
4. for each E_i , all `Read`s in C_{i-1} see final values;
5. for each E_i ($i > 1$), all `Read`s that are not in C_{i-1} see default values or values written previously by the same thread. ($C_0 = \emptyset$, E_1 has no `Read`s and C_1 contains all default `Write`s.)

When we say that `Read` action r *sees* `Write` action w in E_i, we imply that the write-seen function of E_i maps r to w. Please note that committing a `Write` action makes it visible for another thread (two steps later), but previously committed `Write`s of the same variable remain visible as well (excluding the case when the new committed `Write`, previously committed `Write` and the observing `Read` belong to the same thread). In addition, note that committing a `Read` action does not "overwrite" `Write`s of the same variable that were seen earlier by the thread (the values are permitted to appear anew).

We define the problem of testing of the JMM causality requirements as follows.

INSTANCE: Input in the form of the program and its corresponding thread sequences. The complexity parameter is the length (number of characters) of the input.

QUESTION: Do the given tuple of the program and the thread sequences correspond to some valid Java execution? We prove that the problem is NP-complete even under hard restrictions.

3 Complexity of the Problem

3.1 General case

Theorem 1. *The problem of verifying JMM causality requirements, restricted to instances in which there are at most two threads and at most one shared variable, is NP-complete.*

Proof. We use the reduction from 3-Satisfiability (3SAT). Consider a 3SAT instance $\mathcal{F}$ with n variables, $v_1,\ldots,v_n$, and m clauses, $C_1,\ldots,C_m$. The execution E is represented by the program P and the trace sequences T. The program P is shown in Table 1 (initially $a == 0$). Unique identifiers for the actions are given in square brackets. Only memory access actions have identifiers. For clarity, we denote shared variables as alphabetical characters other than r (a, b, ...) and the values on the JVM stack as the letter "r" with index (r_1, r_2, ...).

Thread 1	Thread 2
[0:] a = 1;	[1:] r_{a1} = a;
	[2:] r_{b1} = a;
	$r_1 = r_{a1} - r_{b1}$;
	...
	[2*n-1:] r_{an} = a;
	[2*n:] r_{bn} = a;
	$r_n = r_{an} - r_{bn}$;
	< ***comment*****:** *in the following* ***if*** *statements*
	$\boldsymbol{r}_{cji}$ *may be any of* $\boldsymbol{r}_1 \ldots \boldsymbol{r}_n$;
	$\boldsymbol{r}_{cji}$ *simulates an appearance of a variable in place* $\boldsymbol{i}$ *of clause* $\boldsymbol{j}$;
	$\boldsymbol{s_{ji}}$ *is* **1** *if the variable appears without negation, and* **-1** *otherwise* >
	if (!($s_{11} == r_{c11}$ || $s_{12} == r_{c12}$ || $s_{13} == r_{c13}$))
	$r_0 = 1$;
	if (!($s_{21} == r_{c21}$ || $s_{22} == r_{c22}$ || $s_{23} == r_{c23}$))
	$r_0 = 1$;
	...
	if (!($s_{m1} == r_{cm1}$ || $s_{m2} == r_{cm2}$ || $s_{m3} == r_{cm3}$))
	$r_0 = 1$;
[2*n+3:] r_{n+2} = a;	[2*n+1:] r_{n+1} = a;
if ($r_{n+2} = 2$)	if ($r_0 == 0$ || $r_{n+1} == 3$)
[2*n+4:] a = 3;	[2*n+2:] a = 2;

Table 1. The program (pseudocode)

The trace sequences are shown in Table 2. Components g_i of the operations are not presented in the table because each `Write` operation provides a unique value and there is no ambiguity.

Note that correspondence between the operations in the program and the actions in the trace is provided here by matching of the unique identifiers. There is a one-to-one correspondence because there are no loops. If loops are present, indexes may be added to GUIDs (here there is no need for this).

An assignment to v_i is simulated by the following components:

1. operation $a = 1$ by Thread 1 and two operations $r_{ai} = a$ and $r_{bi} = a$ by Thread 2;

Thread 1	Thread 2
[0:] Write (a, 1)	[1:] Read (a, 1)
	[2:] Read (a, 1)
	[3:] Read (a, 1)
	[4:] Read (a, 1)
	...
	[2*n-1:] Read (a, 1)
	[2*n:] Read (a, 1)
[2*n+3:] Read (a, 2)	[2*n+1:] Read (a, 3)
[2*n+4:] Write (a, 3)	[2*n+2:] Write (a, 2)

Table 2. The trace sequences

2. corresponding actions *[0:] Write (a, 1)* by Thread 1 and *[2*i-1:] Read (a, 1)*, *[2*i:] Read (a, 1)* by Thread 2;
3. commitment order between actions *[2*i-1:] Read (a, 1)* and *[2*i:] Read (a,1)* that belong to Thread 2.

Stack variable r_i may receive the values 1, -1 or 0. An assignment of the value *true* to v_i is simulated by assigning the value 1 to r_i, and an assignment of the value *false* to v_i is simulated by assigning the value -1 to r_i. If r_i receives the value 0, this means that v_i cannot be *true* or *false* in any clause. r_i may receive the value 1 or -1 only if the two actions `Read` (a, 1) by Thread 2 belong to different commitment sets. Please note: the JMM does not require that commitment order of the accesses to the same variable a comply with the program order.

An OR for the clause j is simulated by the program operations
if (!(s_{j1} == r_{cj1} // s_{j2} == r_{cj2} // s_{j3} == r_{cj3})) and r_0= *1;* by Thread 2;

AND is simulated by the operations *if* (r_0== *0 //* r_{n+1}== *3)* and *a = 2;* as well as by action *[2*n+2] Write (a, 2)* by Thread 2.

Lemma 1. *Let $\mathcal{F}$ be the instance of a 3SAT problem and let E be the execution constructed as described above. Then E complies with the JMM causality requirements iff $\mathcal{F}$ is satisfiable.*

Proof.

⇐ Suppose $\mathcal{F}$ is satisfiable. Then, there exists a satisfying assignment Å for $\mathcal{F}$. We construct the commitment order $\mathbb{C}$ for E in which for each i, r_{ai} = *1* directly precedes r_{bi} = *1* if v_i is *true* in Å and, conversely, r_{bi} = *1* directly precedes r_{ai} = *1* if v_i is *false* in Å. The process of commitment is described in Table 3.

⇒ Conversely, suppose E is a positive instance. Assume $\mathbb{C}$ is a valid commitment order for E. If $r_{ai} = 1$ precedes $r_{bi} = 1$ in $\mathbb{C}$, then the satisfying assignment for v_i is assumed *true*, and if $r_{bi} = 1$ precedes $r_{ai} = 1$ in $\mathbb{C}$, then the satisfying assignment for v_i is assumed *false*. If r_{ai} and r_{bi} commit in the same step, then the assignment for v_i is neither *true* nor *false*.

Suppose that some clause j is unsatisfied. Then, by our construction, r_0 is assigned value 1. In this case, *[2*n + 2:] Write (a, 2)* cannot be committed before *[2*n + 4:] Write (a, 3)*. Because *[2*n + 4:] Write (a, 3)* can commit only after *[2*n + 2:] Write (a, 2)*, we may conclude that E is not a positive instance – a contradiction to our assumption. □

Lemma 2. *The problem of verifying JMM causality requirements is in NP.*

Proof. Given a series of commitment sets $C_1,\ldots,C_k$ and validating executions $E_1,\ldots,E_k$, the JMM causality requirements may be checked in polynomial time by simulating each execution E_i

Action	Final Value	First Committed in	First Sees Final Value In	Comment
a = 1	1	C_1	E_1	
...	...	...	...	
r_{ai} = a	1	C_2	E_3	simulates an assignment
r_{bi} = a	1	C_4	E_5	of the value *true* to v_i
...	...	...	...	
r_{aj}= a	1	C_4	E_5	simulates an assignment
r_{bj} = a	1	C_2	E_3	of the value *false* to v_j
...	...	...	...	
a = 2	2	C_3	E_3	commits before "a = 3" because r_0== 0
r_{n+2} = a	2	C_3	E_4	
a = 3	3	C_4	E_4	
r_{n+1} = a	3	C_4	E_5	

Table 3. Commitment order

(when shared memory `Read`s are replaced by reading constants) and checking properties 1-5 on page 3. □

By lemmas 1, 2 and because the construction may be done in polynomial time, theorem 1 holds. □

3.2 Preset restrictions on the commitment order

The previous section gives rise to the intuition that the problem is NP-hard due to uncertainty of commitment order. Let us search for some additional restrictions on this order for the purpose of making the problem tractable. We will use a memory model that is stronger than the JMM but, possibly, close enough to it to be applicable for some implementations.

The first possibility is to restrict the commitment order of `Read` actions so that it must comply with the program order. We reject this restriction, however, because it prohibits reasonable Java executions (see, for example, causality test case 7 in [1]).

The second possibility is to restrict the commitment order so that it must comply with the program order for operations on the same variable. This restriction does not help, however, because the problem is still NP complete (see Appendix B). Therefore, we do not see any acceptable stronger memory model that may be used as a substitute for the JMM.

3.3 Tracing the commitment order of actions for each thread

Let us try another approach. Actions in the trace may be augmented with their prospective numbers in the thread's commitment order. These numbers can be determined as follows: Each thread must hold a local counter. The counter is initialized to zero. When the value of some memory access a is *decided*, the current value of the counter is assigned to a as its number and the counter is incremented. Action numbers may not comply with the program order because of compiler and run-time optimizations. The method is applicable if, in the particular architecture, the moment when the value is decided can be fixed.

Having the modified trace, we reformulate the problem: Do the given tuple of the program and the thread sequences correspond to some valid Java execution whose commitment order complies with the numbering of actions in the trace?

When the number of threads is bounded, the problem may be solved by the polynomial algorithm using the *frontier graph* approach. Frontier graphs were used by Gibbons and Korach [14] for

verifying sequential consistency. The problem of verifying sequential consistency is NP-complete in the general case, but some private cases (for example, when the number of threads is bounded and the write-seen function provided) may be solved in polynomial time using frontier graphs. Here we explain briefly the main concepts. Readers interested in a more in-depth description should refer to [14].

For the set of k thread sequences with n memory access operations, the *frontier* defines some set of k actions, each action belonging to a different thread. Each action in the frontier is considered the last action performed by the thread at some moment in the execution. The last action may be null if the thread has not performed any action yet. If the order of actions in any thread sequence is fixed, the frontier defines some prefix. (For sequential consistency, the program order is considered, and for the JMM, the commitment order is considered.) There are $O(n^k/k^k)$ frontiers.

The *frontier graph* is defined as follows. Each possible frontier is mapped as a vertex in the graph, and it is marked "legal" or "illegal," depending on some problem-dependent validation rules. Then, if some frontier F_j is a legal extension of some frontier F_i, the frontier graph must contain the edge (F_i, F_j). The graph has a polynomial number of vertexes and edges and can be traversed in polynomial time. The execution is considered consistent with the memory model if there is a directed path (of legal vertexes only) from the starting frontier (having k null operations) to the terminating frontier (consisting of the last operation in each thread sequence).

We can use this approach to provide a polynomial algorithm for the JMM. This algorithm would work for the case where commitment order for each thread is fixed and the number of threads is bounded. (Note that the write-seen function is provided by the execution trace as well.)

3.4 Tracing the commitment order of `Read` actions for each thread

The method in section 3.3 may be further improved. The idea is that the commitment order of `Write`s does not matter. For each execution E_i having a set Ψ of committed `Read`s, there exists the minimum set Ω of `Write`s that must be committed in E_{i-1}. Ω includes the set of `Write`s that `Read`s from Ψ see in the final execution E. In addition, Ω includes each `Write` that is the last preceding (in $\overset{po}{\rightarrow}$) `Write` of the same variable for some `Read` from Ψ. There is no reason to commit additional `Write`s in E_{i-1}, because this would only restrict the set of possible values that `Read`s in E_i may see. Therefore, the set of committed `Write`s is unambiguously defined in $E_{i-1}$with respect to E_i.

Let us evaluate the complexity of the method. Consider an execution that has k threads and n shared memory accesses, among them n_r `Read`s. There are $O(\left(\frac{n_r}{k}\right)^k)$ frontiers. For two frontiers, the complexity of deciding if the first is a legal extension of the second is $O(n^2)$. See Algorithm 1 in section 4. Since the number of frontier pairs is $O(\left(\frac{n_r}{k}\right)^{2k})$, the total complexity does not exceed $O(\ n^2 * \left(\frac{n_r}{k}\right)^{2k})$.

In closing, we add that an augmented trace may be easily produced when running the test on a simulator. Note that if the simulator can be programmed to generate the global commitment order, the verification problem is trivial.

3.5 Short test sequences

If the commitment order of `Read`s cannot be traced, the verification task is NP-complete with respect to the execution length. Therefore, only short test sequences may be used. Consider an execution that has n shared memory accesses, n_r `Read`s and n_w `Write`s. The execution may be verified by checking all possible commitment orders that have at most n steps (the complexity is $O(n^2 * \left(\frac{n+n-1}{n}\right))$. In section 4 we suggest a method that is of complexity $O(n^2 * 2^{2*\mathrm{nr}})$, and is preferable when n_r is not too close to n.

4 The Verification Algorithm

The verification algorithm does not differ much for the two cases we consider: 1) that in which the commitment order of `Read`s is traced; 2) that in which it is not traced. The algorithm we present is therefore unified. We do remark, in our explanations, on some of the differences between the two cases. In case (1) the algorithm is polynomial, provided that the number of threads is bounded (i.e. significantly less than the number of actions); if it is not polynomial, only short tests may be used.

Consider an execution tuple $E = < P, A, \stackrel{po}{\rightarrow}, W, V >$. In order to validate E, we perform the following:

1. We construct a group Γ of candidate execution *skeletons* that may be used for building a validating sequence. Each skeleton is characterized by the set f of `Read` actions that are in the second step of commitment. These are `Read`s that *must* see their final values (the values that they see in E) in any execution that has the skeleton. Only `Read`s in f may see non-default `Write`s belonging to other threads. Other `Read`s may see only the default `Write`s or `Write`s that precede them in the program order (these `Write`s may or may not write final values). For convenience, we divide the group of skeletons into subgroups *F(E, i)*, where $i = |f|$, $1 \leq i \leq n_r$. Γ differs in the two variations of the algorithm (when short test sequences are used and when commitment of `Read`s is traced). In the first case, it includes skeletons corresponding to all possible *f*. In the second case, `Read`s in each f must form a prefix defined by some frontier.
2. We associate a *simulated* execution *S(E, f)* with each skeleton defined by *f*. *S(E, f)* is obtained by running each thread separately in program order (all values are initialized to the default) while some actions are replaced as follows. Each `Read` action *rd* in f is replaced by the action of reading the constant value. This constant value is equal to the final value returned by *rd* in *E*. For example, an action $r_3 = b$ with a final value of 5 is replaced by the action $r_3 = 5$. All other `Read`s are left unchanged. This simulation gives us the set of `Write`s that may be committed (these are `Write`s that write final values in *S(E, f)*). *S(E, f)* is valid if each `Read` in f can see a `Write` that may be committed.
3. We check if it is possible to build, from the simulated executions, a chain that complies with the commitment rules. This chain must have the following properties:
   (a) each simulated execution in the chain must be valid;
   (b) `Write`s seen by `Read`s in f_{i+1} may be committed in all S_j in the chain where $i \leq j$.

In the pseudocode in Algorithm 1 we use **R** (the reaching set) as a set of valid simulated executions *S(E, f_i)* (found in the previous step) that have the following property: Execution E_i that is simulated by *S(E, f_i)* may "reach" E by committing its actions step-by-step in a series of executions E_i, E_{i+1}, ..., E. **N** (next) is a set of simulated executions (found in the current step) that will replace **R** in the next step of the algorithm. **P** (passed) is a set of simulated executions that were in **R** in previous steps of the algorithm. **I** is a set that contains an initial execution: one where no `Read` is required to see its final value. We start from the *final* execution S_f where all `Read`s see their final values and then attempt to reach an *initial* execution S_i.

We further prove that the algorithm works. The intuition of the proof is as follows. `Read`s are committed in two steps, and they can see non-default values written in other threads beginning with the second step after committing the corresponding `Write`. In our chain of simulated executions, however, `Read`s can see all `Write`s in the previous step. Therefore this chain is more "rare" than the chain of executions used to validate E. The proof provides a mapping between the two chains.

$\Leftarrow$ First, we prove that if any execution is valid with respect to the causality requirement, the algorithm confirms this. Suppose that for some execution E there exists a chain of sets of committed

Algorithm 1 Verification of the Java causality requirements

JCV(E)

▷ initialization
1 $\mathbf{N} \leftarrow \phi$
2 $\mathbf{P} \leftarrow \phi$
3 $\mathbf{R} \leftarrow \{S(E, f') \mid f' \in F(E, n_r)\}$ ▷ $\mathbf{R}$ contains the final execution
4 $\mathbf{I} \leftarrow \{S(E, \phi)\}$ ▷ $\mathbf{I}$ contains the initial execution

▷ iterations
5 **while** $\mathbf{R} \neq \phi$
6 **for each** $r = S(E, f) \in \mathbf{R}$
7 **for each** $e \in \{F(E,i) \mid i < |f|\} \setminus (\mathbf{N} \cup \mathbf{P} \cup \mathbf{R})$
8 **do if** EXTENDS(e, r)
9 **then do**
10 **if** $e \in \mathbf{I}$
11 **return** *true*
12 **else**
13 $\mathbf{N} \leftarrow \mathbf{N} \cup \{e\}$
14 $\mathbf{P} \leftarrow \mathbf{P} \cup \mathbf{R}$
15 $\mathbf{R} \leftarrow \mathbf{N}$
16 $\mathbf{N} \leftarrow \phi$
17 **return** *false*

EXTENDS(v, u) ▷ v is extended by u

▷ check that: (1) v is valid; (2) `Read`s committed in v stay committed in u;
▷ (3) `Write`s seen by other threads in u may be committed in v

▷ (4) for each `Read` r that sees a final value in u, the last preceding `Write`
▷ of the same variable by the same thread may be committed in v and u
1 **for each** `Read` r in v
2 **if** r sees the last preceding `Write` of the same variable by the same thread
3 **do** nothing
4 **else if** r sees a default `Write` **and** there is no preceding `Write`
of the same variable by the same thread
5 **do** nothing
6 **else if** r sees in v and u a non-default `Write` of a final value by some other thread
that may be committed in v and in u **and**
the last `Write` to the same variable that precedes r in program order,
if it exists, writes final value in v and in u
7 **do** nothing
8 **else return** *false*

9 **return** *true*

actions C_0, C_1, C_2, ... and a chain V of corresponding executions E_1, E_2, ... that are used for validating E. We build a sub-chain S of V that starts from E_1. For each execution E_i in S, the next execution in S is the first execution after E_i in V that has more `Read`s returning values written by other threads than E_i. S has the following properties: (1) all executions in S are well-formed; (2) if any `Read` sees the final non-default value in some execution E_i (i $>$ 1) in S, then (a) the corresponding `Write` writes the final value in E_{i-1} and (b) the last preceding (by program order) `Write` to the same variable (if one exists) writes the final value in E_{i-1} and E_i.

Therefore we can replace each execution in S with the simulated one and obtain a valid path from the initial to the final simulated execution. (A `Read` that returns a value written by another thread is considered to be a skeleton.) Our algorithm would find this path because it checks all possible simulated executions.

$\Rightarrow$ Conversely, if the algorithm finds that some execution is valid with respect to the causality requirement, this execution is actually valid. Suppose we have a set of our simulated executions such that there exists a path from the initial to the final execution through valid executions only. For each pair ($S(E, f_i)$, $S(E, f_{i+1})$) of subsequent simulated executions in the path, there exist commitment sets C_i, $C_i^{'}$, C_{i+1} and corresponding executions E_i, E_i, E_{i+1} (where E_i is simulated by $S(E, f_i)$ and E_{i+1} is simulated by $S(E, f_{i+1})$), such that the following conditions hold:

1. $C_i \subseteq C_i^{'} \subseteq C_{i+1}$ and $C_i \subseteq A_i$, $C_i^{'} \subseteq A_i$, $C_{i+1} \subseteq A_{i+1}$;
2. all `Read`s that see in E_{i+1} final non-default values written by other threads are committed in $C_i^{'}$.

We can consider ... , E_i, E_i, E_{i+1}, ... to be a chain of executions that may be used for validating E and ... , C_i, $C_i^{'}$, C_{i+1} , ... as a chain of commitment sets. □

4.1 Examples

In the following examples we illustrate the non-polynomial algorithm for short test sequences.

Consider the code in Figure 1 (taken from Figure 16 in [3]) as an example.

Initially, x == y == z == 0

Thread 1	Thread2
1: r3 = x;	5: r2 = y;
if (r3 == 0)	6: x = r2;
2: x = 42;	
3: r1 = x;	
4: y = r1;	

r1 == r2 == r3 == 42 is a legal behavior

Fig. 1. An example of a valid execution

In Figure 2 we depict simulated executions, one for each possible set of `Read` actions that are required to see final values. Each execution is described by a table. The table contains information about the `Read` and `Write` accesses, represented by their GUIDs. A `Read` access is marked *'f'* if it is required to return the final value produced by the same `Write` as in the final execution, and *'⋆'* otherwise. A `Write` access w is marked *'c'* if it produces its final value and therefore *may* be committed; in addition, it is marked *'n'* if it is needed to validate any `Read` that is required to see w in the execution. Any access is marked '-' if it is absent in the execution. We mark an execution *initial* if all its `Read`s are marked either *'⋆'* or '-'.

E_8	Reads	Writes
valid	1 : f	2 : -
final	3 : f	4 : c n
	5 : f	6 : c n

E_5	Reads	Writes
valid	1 : ⋆	2 : c
	3 : f	4 : c n
	5 : f	6 : c n

E_6	Reads	Writes
invalid	1 : f	2 : -
	3 : ⋆	4 : n
	5 : f	6 : c n

E_7	Reads	Writes
invalid	1 : f	2 : -
	3 : f	4 : c
	5 : ⋆	6 : n

E_2	Reads	Writes
valid	1 : ⋆	2 : c
	3 : ⋆	4 : c n
	5 : f	6 : c

E_3	Reads	Writes
invalid	1 : ⋆	2 : c
	3 : f	4 : c
	5 : ⋆	6 : n

E_4	Reads	Writes
invalid	1 : f	2 : -
	3 : ⋆	4 :
	5 : ⋆	6 : n

E_1	Reads	Writes
valid	1 : ⋆	2 : c
initial	3 : ⋆	4 : c
	5 : ⋆	6 :

Fig. 2. Running the example in Figure 1

If any execution contains a `Write` that is marked n but not marked c, this execution is invalid. We assume that execution E_j extends execution E_i *($E_i \rightarrow E_j$)* if the following hold:

1. all `Read`s that are marked *'f'* in E_i are marked *'f'* in E_j;
2. any `Write` that is marked *'n'* in E_j is marked *'c'* in E_i;
3. any `Write` that is marked *'n'* in E_i is marked *'n'* in E_j.

In our example, there is a path from the initial to the final execution through valid executions only: $E_1 \rightarrow E_2 \rightarrow E_8$. Therefore, the example is valid.

Now consider the example in Figure 3 (taken from Figure 17 in [3]).

Initially, x == y == z == 0

Thread 1	Thread2	Thread 3	Thread 4
1: r1 = x;	3: r2 = y;	5: z = 42;	6: r0 = z;
2: y = r1;	4: x = r2;		7: x = r0;

r0 == 0, r1 == r2 == 42 is not a legal behavior

Fig. 3. An example of a prohibited execution

The behavior is not allowed because either the final execution is invalid (in the case where the write-seen function for (1:) returns (7:)), or there is no valid path from the initial to the final execution. Figure 4 illustrates this example.

Finally, consider the example in Figure 5 (causality test case 4 from [1]). The behavior is prohibited because a valid path from execution E_1 to execution E_4 does not exist (see Figure 6).

E_8	Reads	Writes
invalid	1 : f	2 : c n
final	3 : f	4 : c n
	6 : f	5 : c
		7 : n

Fig. 4. Invalid final execution

Initially, x == y == 0

Thread 1	Thread2
1: r1 = x;	3: r2 = y;
2: y = r1;	4: x = r2;

r1 == r2 == 1 is not a legal behavior

Fig. 5. Another example of a prohibited execution

E_4	Reads	Writes
valid	1 : f	2 : c n
final	3 : f	4 : c n

E_2	Reads	Writes
invalid	1 : f	2 : c
	3 : ⋆	4 : n

E_3	Reads	Writes
invalid	1 : ⋆	2 : n
	3 : f	4 : c

E_1	Reads	Writes
valid	1 : ⋆	2 :
initial	3 : ⋆	4 :

Fig. 6. Running the example in Figure 5

5 Discussion and Conclusions

The JMM specification reduces the complex task of validating an execution for causal acyclicity to a series of relatively simple commitment steps. Each such step deals with some well-formed intermediate execution. Well-formed executions must obey intra-thread consistency: each thread must execute as if it runs in program order in isolation, while the memory model determines the values of shared variables. Furthermore, because the set of values in the intermediate execution is predefined in the previous commitment steps, it cannot be influenced by optimizations. If we take all of this into account, we may disregard optimizations and assume that each thread in the intermediate execution runs as if there is no reordering. This assumption makes it possible to check the causality requirement by a series of relatively simple simulations.

It is the simplicity of the simulations that make the model practical. Without it, it would be hard to tell which compiler optimizations are allowed for intermediate executions. Note that we may reason in terms of compiler optimizations (as in [1]) in order to explain a behavior of some execution but not to prove that the behavior is allowed. This does not mean that the JMM prohibits compiler optimizations (and Theorem 1 in [5] proves the converse). What it does mean is that, when the values transferred by the memory model between the different threads are predefined and the intra-thread semantics respected, the simulated execution is indistinguishable from the one in which each thread executes in program order. We conclude that the executions should be simulated in program order; otherwise, the JMM must be defined in terms of compiler transformations. This conclusion gives rise to some problems with causality test case 6 [1], which is shown below in Figure 7:

Initially, A == B == 0

Thread 1	Thread2
1: r1 = A;	3: r2 = B;
if (r1 == 1)	if (r2 == 1)
2: B = 1;	4: A = 1;
	if (r2 == 0)
	5: A = 1;

r1 == r2 == 1 is a legal behavior

Fig. 7. Causality test case 6

The test is allowed ([1]) even though our algorithm shows that it fails (see Figure 8).

E_4	Reads	Writes
valid	1 : f	2 : c n
final	3 : f	4 : c n
		5 : -

E_2	Reads	Writes
invalid	1 : f	2 : c
	3 : ⋆	4 : n -
		5 : c

E_3	Reads	Writes
invalid	1 : ⋆	2 : n
	3 : f	4 : c
		5 : -

E_1	Reads	Writes
valid	1 : ⋆	2 :
initial	3 : ⋆	4 : -
		5 : c

Fig. 8. Running causality test case 6. There is no path from E_1 to E_4.

Two problems preclude us from adopting $E_1 \rightarrow E_2 \rightarrow E_4$ as a valid path. First, `Write` action (5:) does not appear explicitly in the final execution. But we must commit it for the purpose of producing a value for `Read` (1:) in E_2. However, by definition, the final execution E contains all the committed actions:

$$A = \bigcup (C_0,\ C_1,\ C_2,\ \dots\ ).$$

The implication follows that E might include some "virtual" actions that don't really appear. If so, this should be stated explicitly in the JMM specification.

Furthermore, in order for causality test case 6 to succeed, the write-seen function for (1:) must be allowed to return (4:) in E_4, and it must be allowed to return (5:) in E_2. But the write-seen function has to be the same in the final execution E and in any execution E_i (that is used to validate E) for all `Read`s in the second commitment step. (Recall that these `Read`s are in C_{i-1}.) This step, however, is the only one in which `Read`s are allowed to see non-default `Write`s by other threads. Therefore, for executions E_4 and E_2, (4:) and (5:) must be allowed to appear in this step. This in turn rules out the aforementioned requirement – that the write-seen function be the same – for the case of "virtual" `Write`s. This case is not addressed in the JMM specification.

Perhaps there is another solution to this problem, one which does not require weakening the constraints on the commitment process. Let us see what happens if we consider different actions to be one and the same. In our example there are two actions, (4:) and (5:), which are `Write`s of the

same constant value, only one of which may appear in any execution. In this particular case, we would have good reason to merge the actions. Doing so, however, might not help in more general cases. Thus, we suggest a more complicated formalization: Different `Write`s of the same variable by the same thread that are not separated by synchronization actions may be considered to be the same action from that moment in the commitment process when their values are determined to be the same.

In this paper we have proven that verification of the Java causality requirements is NP-complete in the general case. We suggest a verification algorithm that may be used in its two versions for the following two cases: the common, non-polynomial version for short test sequences, and the special, polynomial version for the special case when the trace provides a conjectural commitment order of `Read`s for each thread. We assume that no synchronization actions or final variables are used. We further add that their use makes testing more complicated because each `Read` might see a number of hb-consistent `Write`s. We conclude that the JMM specification requires some additional clarifications and adjustments, which may in turn lead to changes in our algorithm.

References

1. Causality Test Cases. *http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html*.
2. The JSR-133 Cookbook for Compiler Writers. *http://gee.cs.oswego.edu/dl/jmm/cookbook.html*.
3. JSR-133: Java Memory Model and Thread Specification. *http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf*, August 2004.
4. S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE Trans. on Parallel and Distributed Systems*, 4(6):613–624, 1993.
5. Sarita V. Adve, Jeremy Manson, and William Pugh. The Java Memory Model. In *POPL'05*, 2005.
6. S.V. Adve, K. Gharachorloo, A. Gupta, J.L. Hennessy, and M.D. Hill. Sufficient System Requirements for Supporting the PLpc Memory Model. Technical Report #1200, University of Wisconsin-Madison, 1993.
7. Andrei Alexandrescu, Hans Boehm, Kelvin Henney, Doug Lea, and Bill Pugh. Memory model for multithreaded C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1680.pdf, 2004.
8. Hans Boehm, Doug Lea, and Bill Pugh. Implications of C++ Memory Model Discussions on the C Language. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1131.pdf, 2005.
9. Hans-J. Boehm. Threads cannot be implemented as a library. *SIGPLAN Not.*, 40(6):261–268, 2005.
10. William Kuchera Charles. The UPC Memory Model: Problems and Prospects. In *Proceedings of the International Parallel and Distributed Processing Symphosium*. IEEE, 2004.
11. K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy, and M. D. Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 15(4):399–407, 1992.
12. K. Gharachorloo, S.V. Adve, A. Gupta, J.L. Hennessy, and M.D. Hill. Specifying System Requirements for Memory Consistency Models. Technical Report #CSL-TR-93-594, Stanford University, 1993.
13. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *Proceedings of the 17th Intl. Symp. on Computer Architecture*, pages 15–26, May 1990.
14. Phillip B. Gibbons and Ephraim Korach. New results on the complexity of sequential consistency. Technical report, AT&T Bell Laboratories, Murray Hill NJ, September 1993.
15. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, third edition, 2005.
16. Prince Kohli, Gil Neiger, and Mustaque Ahamad. A Characterization of Scalable Shared Memories. Technical Report GIT-CC-93/04, College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332-0280, January 1993.
17. The Java Memory Model mailing list archive. *http://www.cs.umd.edu/~*pugh*/java/memoryModel/archive*.

A The JMM Definition

Java threads communicate through shared memory (heap) by the following *inter-thread actions* (here and below, actions and virtual bytecode commands will be printed in `typewriter` font):

- `Read` marks the moment that a value is accepted by a thread. It corresponds to `getfield, getstatic` or array load (`aaload, faload, iaload`, etc.) bytecode;
- `Write` marks the moment that a value is issued by a thread. It corresponds to `putfield, putstatic` or array store (`aastore, fastore, iastore`, etc.) bytecode;
- `Lock` corresponds to `monitorenter` bytecode or to acquiring a lock when leaving the wait state;
- `Unlock` corresponds to `monitorexit` bytecode or to releasing a lock when entering the wait state;

`Read` and `Write` actions may access normal, *volatile* or *final* variables (a variable is a field or an array element). *Synchronization* actions operate in the same way as release or acquire [13,4] actions. They include `Lock`s and `Unlock`s, `Read`s and `Write`s of volatile variables, as well as actions that start a thread or detect that one has stopped. Each execution has a total order over all its synchronization actions, called *synchronization order*. Final fields don't change after initialization except in special cases of reflection and deserialization.

The JMM contains the following assumptions about memory access granularity. References to objects and all variable types except for *double* and *long* are treated as atomic memory locations. Accesses to these locations do not interfere with one another. A variable of type double or long is treated as two distinct 32-bit locations. Each access for reading or writing a double or long variable is treated as two `Read` or `Write` actions, respectively.

An execution E of a Java program P is described by a tuple $\langle P, A, \overset{po}{\rightarrow}, \overset{so}{\rightarrow}, W, V, \overset{sw}{\rightarrow}, \overset{hb}{\rightarrow} \rangle$, where:

- P is the program (virtual bytecode);
- A is the set of actions, each one represented by the thread to which it belongs, its type, the variable or monitor it operates, and an arbitrary unique identifier;
- $\overset{po}{\rightarrow}$ is the program order (the order of actions in each thread as defined by the program)
- $\overset{so}{\rightarrow}$ is the synchronization order;
- W is a function that, for each `Read` action r, gives the `Write` action seen by r (it is called a write-seen function)*;*
- V is a function that assigns a value to each `Write`;
- $\overset{sw}{\rightarrow}$ is a synchronizes-with partial order. The synchronizes-with order arranges pairs of actions with release-acquire semantics (the order in each such pair complies with $\overset{so}{\rightarrow}$). The following actions are paired as release with acquire: an `Unlock` with each subsequent `Lock` of the same monitor; a `Write` to a volatile variable with each subsequent `Read` of the same variable; the write of the default value with the beginning of each thread; the initialization of a thread with the start of that thread; the interruption of a thread with its "realization" that it has been interrupted; the termination of a thread with the detection of this by another thread;
- $\overset{hb}{\rightarrow}$ is the happens-before order. For non-final variables it is similar to the happens-before-1 relation in the Data-Race-Free-1 [4] memory model. There is a happens-before edge from the end of the constructor of any object o to the beginning of the finalizer for o. In addition, for any two actions a and b, $a \overset{hb}{\rightarrow} b$ if one of the following holds: $a \overset{po}{\rightarrow} b$, $a \overset{so}{\rightarrow} b$, or there exists another action c such that $a \overset{hb}{\rightarrow} c$ and $c \overset{hb}{\rightarrow} b$.

The JMM requires that the execution E be *well-formed*. Each thread must execute as a correct uniprocessor; each `Read` of some variable must see a `Write` of this variable; `Lock` and `Unlock` actions must be correctly nested. Furthermore, $\overset{hb}{\rightarrow}$ must be a valid partial order. In addition, $\overset{hb}{\rightarrow}$ and $\overset{so}{\rightarrow}$ must be consistent with the write-seen function W: in $\overset{hb}{\rightarrow}$ or $\overset{so}{\rightarrow}$, no `Read` r may precede its corresponding `Write` w and no `Write` of the same variable may intervene between w and r.

JMM also requires that the execution E meet *causality* requirements in order to prevent causal loops. E must be iteratively built by *committing* all its actions. Each step i of the commitment process must correspond to the set of committed actions C_i ($C_0 = \emptyset$, $C_i \subset C_{i+1}$). For each C_i (starting from C_1) there must be a well-formed execution E_i that contains C_i. Series $C_0, \ldots, C_i, \ldots$ and $E_1, \ldots, E_i, \ldots$ must have the following properties:

- for each E_i, $\overset{hb}{\rightarrow}$ and $\overset{so}{\rightarrow}$ between committed actions must be the same as in E;
- for each E_i, committed `Write`s must be the same as in E, and the same holds true for `Read`s in the "second step" of the commitment process (those belonging to C_{i-1}); all other `Read`s must see `Write`s that happen-before them; a `Read` may be committed starting with the step following one in which its corresponding `Write` (in E) commits; `Read`s in the "first step" of commitment (they are in C_i - C_{i-1}) must see previously committed (those belonging to C_{i-1}) `Write`s;
- for each E_i, if there is a release-acquire pair from the *sufficient set of synchronization edges* that happens-before some new committed action, this release-acquire pair must be present in all subsequent E_j. (A sufficient set of synchronization edges is a unique minimal set of $\overset{so}{\rightarrow}$ edges that produces $\overset{hb}{\rightarrow}$ when transitively closed with $\overset{po}{\rightarrow}$);
- if some action is committed in C_i, all external actions that happen-before it must be committed in C_i. (An external action is one that may be observed outside of an execution: printing and so forth.)

The JMM considers two additional issues that we don't deal with here. These include value transfer in programs that don't terminate in a finite period of time and *final field semantics* (which may weaken or strengthen the happens-before relation for final field accesses). In addition, [3] and chapter 17 of [15] cover some other aspects of multithreading: waits, notification, interruption and finalization.

B A Special Case of the Verification Problem

In the following theorem, we prove that restricting the commitment order for operations on the same variable does not simplify the problem.

Theorem 2. *The problem of verifying JMM causality requirements, restricted to instances in which there are at most two threads and at most two shared variables, and the commitment order complies with the program order for actions on the same variable, is NP-complete.*

Proof. We use the reduction from 3-Satisfiability (3SAT). Consider a 3SAT instance $\mathcal{F}$ with n variables, $v_1,\ldots,v_n$, and m clauses, $C_1,\ldots,C_m$. The execution E is represented by the program P and the trace sequences T. The program P is shown in Table 4 (initially $a == b == 0$). Notation is the same as in the proof of theorem 1.

The trace sequences are shown in Table 5.

The operations in the program and the actions in the trace are made to correspond by matching their unique identifiers.

An assignment to v_i is simulated by the following components:

1. operations $a = 1$ and $b = 1$ by Thread 1 and two operations $r_{ai} = a$ and $r_{bi} = b$ by Thread 2;
2. corresponding actions *[0:] Write (a, 1)*, *[1:] Write (b, 1)* by Thread 1 and *[2*i:] Read (a, 1)*, *[2*i+1:] Read (b, 1)* by Thread 2;
3. commitment order between actions *[2*i:] Read (a, 1)* and *[2*i+1:] Read (b,1)* that belong to Thread 2;

Thread 1	Thread 2
[0:] a = 1;	[2:] r_{a1} = a;
[1:] b = 1;	[3:] r_{b1} = b;
	r_1 = r_{a1}-r_{b1};
	...
	[2*n:] r_{an} = a;
	[2*n+1:] r_{bn} = b;
	r_n= r_{an}-r_{bn};
	< ***comment*****:** *in the following* ***if*** *statements*
	$\boldsymbol{r}_{cji}$ *may be any of* $\boldsymbol{r}_1 \ldots \boldsymbol{r}_n$*;*
	$\boldsymbol{r}_{cji}$ *simulates an appearance of a variable in place* ***i*** *of clause* ***j***;
	$\boldsymbol{s}_{\boldsymbol{ji}}$ *is* ***1*** *if the variable appears without negation, and* ***-1*** *otherwise* >
	if (!(s_{11} == r_{c11} || s_{12} == r_{c12} || s_{13} == r_{c13}))
	r_0= 1;
	if (!(s_{21} == r_{c21} || s_{22} == r_{c22} || s_{23} == r_{c23}))
	r_0= 1;
	...
	if (!(s_{m1} == r_{cm1} || s_{m2} == r_{cm2} || s_{m3} == r_{cm3}))
	r_0= 1;
[2*n+4:] r_{n+2} = a;	[2*n+2:] r_{n+1}= a;
if (r_{n+2}= 2)	if (r_0== 0 || r_{n+1} == 3)
[2*n+5:] a = 3;	[2*n+3:] a = 2;

Table 4. The program with two variables (pseudocode)

Thread 1	Thread 2
[0:] Write (a, 1)	[2:] Read (a, 1)
[1:] Write (b, 1)	[3:] Read (b, 1)
	[4:] Read (a, 1)
	[5:] Read (b, 1)
	...
	[2*n:]Read (a, 1)
	[2*n+1:] Read (b, 1)
[2*n+4:] Read (a, 2)	[2*n+2:] Read (a, 3)
[2*n+5:] Write (a, 3)	[2*n+3:] Write (a, 2)

Table 5. The trace sequences of the program with two variables

Stack variable r_i may receive the values 1, -1 or 0. An assignment of the value *true* to v_i is simulated by assigning the value 1 to r_i, and an assignment of the value *false* to v_i is simulated by assigning the value -1 to r_i. If r_i receives the value 0, this means that v_i cannot be used as *true* or *false* in any clause. r_i may receive the value 1 or -1 only if the two actions `Read` (a, 1) by Thread 2 belong to different commitment sets. Please note: the JMM does not require that commitment order of the memory accesses comply with the program order.

An OR for the clause j is simulated by the program operations
if (!(s_{j1} == r_{cj1} // s_{j2} == r_{cj2} // s_{j3} == r_{cj3})) and r_0= *1;* by Thread 2;

AND is simulated by the operations *if* (r_0== *0* // r_{n+1}== *3)* and *a = 2;* as well as by action *[2*n+3] Write (a, 2)* by Thread 2.

Lemma 3. *Let $\mathcal{F}$ be the instance of a 3SAT problem and let E be the execution constructed as described above. Then E complies with the JMM causality requirements iff $\mathcal{F}$ is satisfiable.*

Proof.

⇐ Suppose $\mathcal{F}$ is satisfiable. Then, there exists a satisfying assignment Å for $\mathcal{F}$. We construct the commitment order $\mathbb{C}$ for E in which for each i, r_{ai} = *1* directly precedes r_{bi} = *1* if v_i is *true* in Å and, conversely, r_{bi} = *1* directly precedes r_{ai} = *1* if v_i is *false* in Å. The process of commitment is described in Table 6.

Action	Final Value	First Committed in	First Sees Final Value In	Comment
a = 1	1	C_1	E_1	
b = 1	1	C_1	E_1	
...	...	...	...	
r_{ai} = a	1	C_2	E_3	simulates an assignment
r_{bi} = b	1	C_4	E_5	of the value *true* to v_i
...	...	...	...	
r_{aj}= a	1	C_4	E_5	simulates an assignment
r_{bj} = b	1	C_2	E_3	of the value *false* to v_j
...	...	...	...	
a = 2	2	C_3	E_3	commits before "a = 3" because r_0== 0
r_{n+2} = a	2	C_3	E_4	
a = 3	3	C_4	E_4	
r_{n+1} = a	3	C_4	E_5	

Table 6. Commitment order

⇒ Conversely, suppose E is a positive instance. Assume $\mathbb{C}$ is a valid commitment order for E. If $r_{ai} = 1$ precedes $r_{bi} = 1$ in $\mathbb{C}$, then the satisfying assignment for v_i is assumed *true*, and if $r_{bi} = 1$ precedes $r_{ai} = 1$ in $\mathbb{C}$, then the satisfying assignment for v_i is assumed *false*. If r_{ai} and r_{bi} commit in the same step, then the assignment for v_i is neither *true* nor *false*.

Suppose that some clause j is unsatisfied. Then, by our construction, r_0 is assigned value 1. In this case, *[2*n + 3:] Write (a, 2)* cannot be committed before *[2*n + 5:] Write (a, 3)*. Because *[2*n + 5:] Write (a, 3)* can commit only after *[2*n + 3:] Write (a, 2)*, we may conclude that E is not a positive instance – a contradiction to our assumption. □

By lemmas 3, 2 and because the construction may be done in polynomial time, theorem 2 holds. □