Verification of Causality Requirements in Java Memory Model is Undecidable

Matko Botinčan[1], Paola Glavan[2], and Davor Runje[3]

[1] Department of Mathematics, University of Zagreb
`matko.botincan@math.hr`
[2] Faculty of Mechanical Engineering and Naval Architecture, University of Zagreb
`pglavan@fsb.hr`
[3] Faculty of Mechanical Engineering and Naval Architecture, University of Zagreb
`davor.runje@fsb.hr`

Abstract. The purpose of the Java memory model is to formalize the behavior of the shared memory in multithreaded Java programs. The subtlest points of its formalization are causality requirements that serve to provide safety and security guarantees for incorrectly synchronized Java programs. In this paper, we consider the problem of verifying whether an execution of a multithreaded Java program satisfies these causality requirements and show that this problem is undecidable.

Keywords: Java memory model, multithreading, verification.

1 Introduction

The Java language specification [GJSB05] and recent work on formalization of the Java memory model (JMM) [MPA,MPA05] attempt to give a precise specification of the behavior of the shared memory for multithreaded Java programs. The JMM has been designed having two goals in mind. The first one is to provide safety guarantees to programmers by:

– ensuring sequentially consistent behavior of correctly synchronized (data race free) programs, and
– promising that even for programs that are incorrectly synchronized with respect to JMM semantics (i.e., programs with data races) the values should not appear out of thin air.

The second one aims to guarantee compiler writers that common compiler optimization techniques are allowed as long as they do not violate these safety guarantees.

The original specification of the JMM was shown to have serious flaws, e.g., Theorem 1 in [MPA,MPA05] does not hold, infinite executions conflict with omega ordering of default initialization actions in the definition of the JMM, it is unclear how to handle dynamic allocation in this setting, etc. Although subsequent work on JMM [CKS07,AS07a,AS07b] managed to fix some of these problems, all variations of the JMM definition contain an inherent deficiency regarding decidability which we address in this paper.

The subtlest points of the JMM definition are causality requirements that serve to provide safety guarantees for incorrectly synchronized Java programs. The problem is that they are specified declaratively, and from this definition it is not evident how to effectively check them. In [PS06], authors deal with the problem of verifying the JMM causality requirements for a finite execution of a multithreaded Java program containing no synchronization actions, actions on final fields and external actions. They show that the problem is NP-complete, however, their result holds only under additional assumption (implicit from the proof) that all intermediate executions in the justification sequence are finite and polynomially bounded, which is generally not true for arbitrary multithreaded Java programs. In this paper, we consider what happens when this additional assumption is left out and show that the problem of verifying the JMM causality requirements for a finite execution of an arbitrary multithreaded Java program is undecidable.

The rest of the paper is structured as follows. The formal definition of the JMM is given in Section 3. Section 3 contains the main result of this paper. In Section 4, we give concluding remarks.

2 The Java Memory Model

Let us first introduce the concepts from [MPA05,MPA] that are needed for understanding the definition of the Java Memory Model (JMM).

We consider a multithreaded Java program that spawns a set of threads. The execution of each thread is represented as a sequence of actions. Formally, an *action* is a tuple $\langle t, k, v, u \rangle$, where t is the thread performing the action; k is the kind of the action: read, write, volatile read, volatile write, lock, unlock, thread create, thread join or an external action; v is the variable or monitor involved in the action; and u is an arbitrary unique identifier of the action (though, for readability, we do not write u explicitly). Non-volatile read and write actions are *non-synchronization* actions, the others are *synchronization* actions. In the rest of the text, we do not deal with thread create, thread join and external actions, however, we use the notion of initialization actions for setting up initial values of shared variables.

An *execution* is a tuple $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{swo}, \overset{hbo}{<} \rangle$,[4] where

- P is a Java program;
- A is a set of actions;
- $\xrightarrow{po}$ is the *program order* — a partial order over actions in A that is a total order over all actions preformed by the same thread;
- $\xrightarrow{so}$ is the *synchronization order* — a total order over all synchronization actions in A;

[4] Here we denote binary relations with $\xrightarrow{\alpha}$, for some label α. Transitive closure of relation $(\xrightarrow{\alpha})^+$ is denoted by $<_\alpha$, when it is a strict partial order.

- W is the *write-seen function* — a function assigning a write action $W(r)$ to each read action r in A;
- V is the *value-written function* — a function assigning a value $V(w)$ to each write action w in A;
- $\xrightarrow{swo}$ is the *synchronizes-with order* — the smallest relation over synchronization actions in A such that:
  - if a_1 is unlocking and a_2 is locking the same object, and $a_1 \xrightarrow{so} a_2$, then $a_1 \xrightarrow{swo} a_2$;
  - if a_1 is volatile writing to and a_2 is volatile reading from the same location, and $a_1 \xrightarrow{so} a_2$, then $a_1 \xrightarrow{swo} a_2$;.
- $\overset{hbo}{<}$ is the *happens-before order* — a strict partial order induced by the synchronizes-with order and the program order, i.e., $\overset{hbo}{<} = (\xrightarrow{po} \cup \xrightarrow{swo})^+$.

An execution E is *well-formed* if it obeys the Java intrathread semantics, i.e., if it satisfies the following conditions:

(1) **Each read of a variable x sees a write to x**. All reads and writes of volatile variables are volatile actions.
(2) **The synchronization order is at most an omega order**, i.e., for each synchronization action x, the set $\{y \mid y <_{so} x\}$ is finite.
(3) **Synchronization order is a strict total order consistent with program order**, i.e., $<_{po} |_{Dom(<_{so})} \subseteq <_{so}$.
(4) **Lock operations are consistent with mutual exclusion**, i.e., for all lock actions l on monitor m and all threads t (different from the thread of l) the number of locks of t before l in $<_{so}$ is the same as the number of unlocks of t before l in $<_{so}$.
(5) **The execution obeys intra-thread consistency**, i.e., for each thread t, the actions preformed by t in A are executed in the same order that would be generated if t is run as a single thread in isolation.
(6) **The execution obeys synchronization-order consistency**, i.e., for every volatile read $r \in A$, it is not the case that $r <_{so} W(r)$, and additionally, there must not exists a write w on the same variable v such that $W(r) <_{so} w <_{so} r$.
(7) **The execution obeys happens-before consistency**, i.e., for every read $r \in A$, it is not the case that $r <_{hbo} W(r)$, and additionally, there must not exist a write w on the same variable v such that $W(r) <_{hbo} w <_{hbo} r$.

A well-formed execution E is *JMM-consistent* if it satisfies the JMM causality requirements, i.e., if there exists a committing sequence of sets of actions $\emptyset = C_0 \subset C_1 \subset C_2 \subset \ldots$ such that $A = \bigcup_i C_i$, that get justified through a sequence of well-formed executions $E_1, E_2, \ldots$ of the program P. The sequences $(C_i)_i$ and $(E_i)_i$, where $E_i = \langle P, A_i, \xrightarrow{po_i}, \xrightarrow{so_i}, W_i, V_i, \xrightarrow{swo_i}, <_{hbo_i}\rangle$, are required to satisfy the following conditions:

1. $C_i \subset A_i$;
2. $<_{hbo_i} |_{C_i} = <_{hbo} |_{C_i}$;

3. $\xrightarrow{so_i}|_{C_i} = \xrightarrow{so}|_{C_i}$;
4. $V_i\ |_{C_i} = V\ |_{C_i}$;
5. $W_i\ |_{C_{i-1}} = W\ |_{C_{i-1}}$;
6. $\forall r \in A_i \backslash C_{i-1}, W_i(r) <_{hbo_i} r$;
7. $\forall r \in C_i \backslash C_{i-1}, W_i(r) \in C_{i-1}, W(r) \in C_{i-1}$.

3 Verification of the JMM Causality Requirements

We define the problem of verifying the JMM causality requirements as follows. The input to the problem is a finite well-formed execution E of a Java program P. The question we are interested in is whether E satisfies the JMM causality requirements, i.e., whether E is JMM-consistent.

Let S be an arbitrary sequential program containing no synchronization actions, no actions on final fields, no external actions, and no references to global variables. Let P be a multithreaded program with two threads T_a and T_b described as follows:

$$T_a: \quad \texttt{y = x;} \qquad T_b: \quad \begin{array}{l}\texttt{if (y == 0) \{ S \}}\\ \texttt{x = 1;}\end{array}$$

Assume that both `x` and `y` are initially set to `0` by initialization actions i_1 and i_2 in an initialization thread I, and they happen before any other action in the execution. We represent these facts by extending the program order. Let $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{swo}, \overset{hbo}{<} \rangle$ be a finite well-formed execution of P defined by the following components:

- $A = \{i_1 = \langle I, \texttt{write}, x\rangle, i_2 = \langle I, \texttt{write}, y\rangle, a_1 = \langle T_a, \texttt{read}, x\rangle,$ $a_2 = \langle T_a, \texttt{write}, y\rangle, b_1 = \langle T_b, \texttt{read}, y\rangle, b_2 = \langle T_b, \texttt{write}, x\rangle\}$;
- $\xrightarrow{po} = \{(a_1, a_2), (b_1, b_2), (i_1, i_2), (i_1, a_1), (i_2, a_1), (i_1, b_1), (i_2, b_1)\}$;
- $\xrightarrow{so} = \emptyset$;
- $W = \{(a_1, b_2), (b_1, a_2)\}$;
- $V = \{(i_1, 0), (i_2, 0), (a_2, 1), (b_2, 1)\}$;
- $\xrightarrow{swo} = \emptyset$;
- $\overset{hbo}{<} = \{(i_1, a_1), (i_2, a_1), (i_1, b_1), (i_2, b_1), (a_1, a_2), (b_1, b_2),$ $(i_1, a_2), (i_2, a_2), (i_1, b_2), (i_2, b_2)\}$.

Then the following lemmas hold.

Lemma 1. *If S terminates, i.e., if S run as a singlethreaded program has a finite execution, then E is JMM-consistent.*

Proof. Assume that S terminates. Then the following sequence of actions $(C_i)_i$ and executions $(E_i)_i$ satisfy the JMM causality requirements for E:

E_1: - $C_1 = \{i_1, i_2\}$;
- $W_1 = \{(a_1, i_1), (b_1, i_2)\}$;

- $V_1 = \{(i_1, 0), (i_2, 0), (a_2, 0), (b_2, 1)\}$;

(note that b_2 indeed can be executed "after" b_1 since S terminates);

E_2:
- $C_2 = C_1 \cup \{b_2\}$;
- $W_2 = \{(a_1, i_1), (b_1, i_2)\}$;
- $V_2 = \{(i_1, 0), (i_2, 0), (a_2, 0), (b_2, 1)\}$;

E_3:
- $C_3 = C_2 \cup \{a_1\}$;
- $W_3 = \{(a_1, b_2), (b_1, i_2)\}$;
- $V_3 = \{(i_1, 0), (i_2, 0), (a_2, 0), (b_2, 1)\}$;

E_4:
- $C_4 = C_3 \cup \{a_2\}$;
- $W_4 = \{(a_1, b_2), (b_1, i_2)\}$;
- $V_4 = \{(i_1, 0), (i_2, 0), (a_2, 1), (b_2, 1)\}$;

E_5:
- $C_5 = C_4 \cup \{b_1\}$;
- $W_5 = \{(a_1, b_2), (b_1, a_2)\}$;
- $V_5 = \{(i_1, 0), (i_2, 0), (a_2, 1), (b_2, 1)\}$;

Lemma 2. *If E is JMM-consistent then S terminates.*

Proof. Assume that E is JMM-consistent and S does not terminate. We claim that then the action b_2 cannot be committed through any sequence of actions $(C_i)_i$, and thus cannot take place in the final execution E, implying that E is not JMM-consistent. Namely, since C_1 contains only initializations actions, it cannot contain b_2. Assume that b_2 is not contained in some C_{i-1}. This means that in the execution E_i, the read of y in T_b (the action b_1) can only see either the initial write of 0 to y (the action i_2) or the write of 0 to y performed by T_a through the action a_2. Since in both cases the condition of the if-statement is satisfied, statements of the program S get executed infinitely, thus not allowing b_2 to get executed. Therefore, b_2 cannot be committed in C_i either.

Since determining whether a sequential program S terminates is undecidable, from Lemma 1 and Lemma 2 we conclude the following:

Theorem 1. *Verification of the JMM causality requirements is undecidable.*

4 Conclusions

In this paper, we considered the problem of verifying whether a finite execution of an arbitrary multithreaded Java program satisfies the causality requirements stemming from the Java memory model. It has been shown that this problem is undecidable.

We see this result as an important weakness of the JMM specification since it shows that one cannot have a dedicated verification algorithm in the general case. One can, however, employ the JMM definition in order to develop a simple model checker that solves the problem for some specific cases ("small" with respect to the number of program instructions, threads, and especially number of data races, see [Man04]).

The sequential consistency memory model has also been shown to be undecidable [AMP00]. This result, however, did not make a definite verdict on the practical aspect of the sequential consistency memory model verification [SG05]. Taking into account Java practitioners needs, we could also expect verifiable fragments of the JMM to appear in the future.

References

[AMP00] Rajeev Alur, Kenneth L. McMillan, and Doron Peled. Model-checking of correctness conditions for concurrent objects. *Inf. Comput.*, 160(1-2):167–188, 2000.

[AS07a] David Aspinall and Jaroslav Sevcik. Formalising Java's data-race-free guarantee. In *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2007)*, volume 4732 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2007.

[AS07b] David Aspinall and Jaroslav Sevcik. Java memory model examples: Good, bad and ugly. In *Proceedings of the 1st International Workshop on Verification and Analysis of Multi-threaded Java-like Programs (VAMP 2007)*, 2007.

[CKS07] Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *Proceedings of the 16th European Symposium on Programming (ESOP'07)*, 2007.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition.* Addison-Wesley, 2005.

[Man04] Jeremy Manson. *The Java memory model.* PhD thesis, University of Maryland, College Park, 2004.

[MPA] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model (expanded version). Submitted to ACM Transactions on Programming Languages and Systems.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, pages 378–391. ACM Press, 2005.

[PS06] Sergey Polyakov and Assaf Schuster. Verification of the Java causality requirements. In *Hardware and Software Verification and Testing, First International Haifa Verification Conference, Haifa, Israel, November 13-16, 2005, Revised Selected Papers*, volume 3875 of *Lecture Notes in Computer Science*, pages 224–246. Springer, 2006.

[SG05] Ali Sezgin and Ganesh Gopalakrishnan. On the decidability of shared memory consistency verification. In *Proceedings of the 3rd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2005)*, pages 199–208. IEEE, 2005.