Java Memory Model-Aware Model Checking

Huafeng Jin, Tuba Yavuz-Kahveci, and Beverly A. Sanders

University of Florida

Abstract. The Java memory model guarantees sequentially consistent behavior only for programs that are data race free. Legal executions of programs with data races may be sequentially inconsistent but are subject to constraints that ensure weak safety properties. Occasionally, one allows programs to contain data races for performance reasons and these constraints make it possible, in principle, to reason about their correctness. Because most model checking tools, including Java Pathfinder, only generate sequentially consistent executions, they are not sound for programs with data races. We give an alternative semantics for the JMM that characterizes the legal executions as a least fixed point and show that this is an overapproximation of the JMM. We have extended Java Pathfinder to generate these executions, yielding a tool that can be soundly used to reason about programs with data races.

Keywords: model checking, relaxed memory model, benign data races.

1 Introduction

The memory model of a programming language defines which values a thread can see when reading a variable from shared memory. If the memory model is *sequential consistency* (SC), then the program behaves as if all of its reads and writes occur in some order consistent with the program order on individual threads, and each read of a variable sees the most recent write to that variable in the order.

Memory systems in most modern multi-core processors are not sequentially consistent and in addition, a variety of compiler optimizations that would be correct in a sequential program may introduce sequentially inconsistent behavior into a multi-threaded one. For example, consider the program in Fig. 1. In any sequentially consistent execution, depending on how the threads interleave, the x field of the single object involved would change from 0 to 3 at some point and then remain 3 thereafter. A common compiler optimization which causes no problems in a single threaded program might, however, replace the last read r1.x in Thread 1 with an assignment, r5 = r2. This admits executions where it appears that the value of r1.x changes from 0 to 3 and then back to 0. Such an execution is not sequentially consistent.

Sequential consistency is desirable because it corresponds with programmers' intuition. Also, it allows formal reasoning techniques and tools, most of which assume sequential consistency, to be used. Most model checkers, for example, implicitly assume sequential consistency. If we used a model checker such as Java Pathfinder to check the scenario in Fig. 1, the legal, but the sequentially inconsistent execution described earlier would not be generated or checked.

Initially p == q, p.x == 0

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

Fig. 1. Execution trace with conflicting accesses to the same memory location. Variable names that begin with r are local variables of a thread. This example is from [11, §17.3].

Typical programming language memory models guarantee sequential consistency only for programs that are data race free. A *data race* is a pair of conflicting operations (i.e. the operations are performed by different threads, both access the same memory location and at least one is a write) that are not ordered by sufficient synchronization. Exactly what constitutes "sufficient synchronization" is defined by, and specific to, the memory model. The Java Memory Model (JMM), guarantees sequentially consistency only for programs that are data race free, but also constrains programs with data races in order to provide some weak security guarantees. If all of the legal executions, including the sequentially inconsistent ones, of a data racy program still satisfy the program's specification, then we can consider a data race to be benign. Occasionally, one may want to take advantage of this to improve performance. For example, intentional, benign data races can be found in the java.lang.String and java. util .ConcurrentHashMap classes.

The JMM is complicated and reasoning about programs with data races is difficult, thus tool support is desirable. We describe a JMM aware model checker, Java PathRelaxer (JPR) that is an extension of Java Pathfinder [22,15] and generates all of the legal executions of finite Java programs with data races so that their properties can be verified. The way the JMM defines legal executions in programs with data races does not lend itself to precise implementation with a model checker and has been shown [23] to be stricter than the designers intended. We use an alternate approach. Instead of defining a legal execution by the existence of a sequence of justifying executions as the JMM does, we compute a set of paths that is the least fixed point of a monotone function. We show that the set of paths generated by JPR is an overapproximation of the set of legal executions. Although the details of the formalization and implementation of JPR are specific for Java, the main ideas are applicable to other languages with a memory model based on the happens-before relation.

The main contributions of this paper are

– A new, fixed-point based, approach to the characterization of legal executions for relaxed memory models.
– A tool, JPR that generates all of the legal executions according to the fixed-point characterization.

– A proof that the fixed-point based approach is an overapproximation of the JMM, and thus JPR is sound for Java programs with data races.
– Insights into how the JMM maps (or does not map) into program constructs.

2 Background

Below, we give a brief overview of the formal definition of the Java Memory Model, including formal, JMM specific definitions of some concepts introduced previously. Our treatment follows that of [1], which is in turn based on the specification of the JMM given in [19,11].[1]

An *action a* is a memory-related operation with an arbitrary unique ID, *aid* that is performed by a thread *tid*, interacts with variable v or (monitor) lock m, and has a *kind*. The kind is one of the following: volatile read from v, volatile write to v, (non-volatile) read from v, (non-volatile) write to v, locking of lock m, unlocking of lock m, starting a thread, detecting termination of thread, and instantiating an object with a set of volatile fields *volatiles* and a set of non-volatile fields *fields* set to their default values. All of the action kinds, with the exception of read and write are *synchronization actions*.

Definition 1 (Execution). *An execution E is described by a tuple $\langle A, P, \leq_{po}, \leq_{so}, W, V\rangle$ where*

– *A is a finite set of actions*
– *P is a program*
– *$\leq_{po}$, the program order, is a partial order on A obtained by taking the union of total orders representing each thread's sequential semantics*
– *$\leq_{so}$, the synchronization order, is a total order over all of the synchronization actions in A*
– *V, the value written function, assigns a value to each write*
– *W, the write-seen function, assigns a write action to each read action so that the value obtained by a read action r is $V(W(r))$.*

A *sequentially consistent* (SC) execution is one where there exists a total order, $\leq_{sc}$, on the actions consistent with $\leq_{po}$ and $\leq_{so}$ and where a read r of variable v sees the results of the most recent preceding write w, i.e.

– $W(r) \leq_{sc} r$
– For all reads r of variable v: if $W(r) \leq_{sc} w \leq_{sc} r$ and w writes to v then $W(r) = w$.

[1] The most important differences between [19] and [1] are that the latter requires that the total order for SC executions be consistent with both the synchronization order and program order (as opposed to just the program order, correcting an apparent oversight in the JMM formulation), formulates the semantics in terms of finite executions, and ignores external actions.

The JMM relaxes SC because it is not required that W return the "most recent" write to the variable in question or that it is consistent for actions on different threads.

The synchronizes-with relation, $\leq_{sw}$, relates certain pairs of actions. For example, the action unlocking a monitor synchronizes-with any subsequent (according to $\leq_{so}$) unlock of the same monitor. Other pairs include writing a volatile variable and a subsequent read, the action of starting a thread and the first action of the newly started thread, etc. See [11, §17.4.4] for a complete list. We categorize the first action of a $\leq_{sw}$ pair as a *release* action, and the second as an *acquire* action. The *happens-before* order, $\leq_{hb}$, is a partial order on the actions in an execution obtained by taking the transitive closure of the union of $\leq_{sw}$ and $\leq_{po}$. A well-formed execution satisfies type safety and some unsurprising consistency requirements on the various partial and total orders. The two most important rules for our purposes are intra-thread consistency and happens-before consistency.

Definition 2 (Well-formed execution). *See [1, Definition 6] for the complete definition.*

7. *Program order is intra-thread consistent: for each thread t, the sequence of action kinds and values of actions performed by t in the program order $\leq_{po}$ is sequentially valid*[2] *with respect to P and t.*
9. *$\leq_{hb}$ is consistent with W: for all reads r of variable v, $r \not\leq_{hb} W(r)$ and there is no intervening write w to v, i.e. if $W(r) \leq_{hb} w \leq_{hb} r$ and w writes to v then $W(r) = w$.*

Two operations from different threads *conflict* if neither is a synchronization action, they access the same memory location and at least one is a write. A *data race* is defined to be a pair of conflicting operations *not* ordered by $\leq_{hb}$.[3]

A Java program is *correctly synchronized* if all of its SC executions are data race free. An important property of most programming language memory models, including the JMM [11,19] [1, Theorem 1], is that all legal executions of a well-formed correctly synchronized program behave as if they are sequentially consistent. This data race free guarantee (DRF) is important for programmers. Because "correctly synchronized" depends only on the sequentially consistent executions, detecting data races can be done with model checkers or other tools that assume sequential consistency. Java RaceFinder (JRF) [16,17], for example, extends Java Pathfinder to precisely detect data races in Java programs according to the memory model.

[2] Sequential validity essentially means that given the values obtained when a variable is read, each thread obeys the Java language semantics.

[3] Because reads and writes of volatile variables are synchronization actions, a volatile variable in Java can never be involved in a data race. Volatile variables can still be involved in non-deterministic behavior that is sometimes called a race condition. In this paper, we use the term race only in the context of data race as defined in the JMM.

While most Java programs should be data race free, the JMM attempts to define the semantics of programs with data races. The main goal was to provide a modicum of security guarantees even for incorrect programs with data races while still allowing as many optimizations as possible. Desirable properties include type safety and no *out-of-thin-air* values.

While the notion of out-of-thin-air value has not been precisely defined, the example in Fig. 2 [19] illustrates the idea and shows why well-formedness (Def. 2) and in particular happens-before consistency, does not suffice. In a sequentially consistent execution of the example in Fig. 2, the only values allowed are r1==r2==0. However, letting $W(A1) = B2$, $W(B1) = A2$, and $V(A2)$ =val, and $V(B2)$ =val, for *any* value val of the correct type, we have a well-formed execution where r1 == r2 == val, and in this situation, val is said to come out-of-thin-air.

To rule out such cases, the JMM requires *legal* executions to satisfy additional causality conditions intended to rule out so-called causal loops that could lead to self-justifying speculative executions. The idea is that a well-formed execution E is legal if there is (roughly speaking) a sequence of well-formed executions E_i with action sets A_i and a subset of actions C_i called the commit set where each committed read either sees a committed write or a write that happens-before it. It is required that $C_{i-1} \subseteq C_i$ and that the sequence eventually produces E with all of its actions committed.

Initially, x == y == 0

Thread 1	Thread 2
A1: r1 = x	B1: r2 = y
A2: y = r1	B2: x = r2

Fig. 2. The rules for a well-formed execution admit traces with r1 == r2 == val, for any arbitrary out-of-thin-air value val of the correct type

Definition 3 (Legal Execution). *[1, Definition 7] A well-formed execution* $E = \langle A, P, \leq_{po}, \leq_{so}, W, V\rangle$ *with happens-before order* $\leq_{hb}$ *is legal if there is a finite sequence of sets of actions* C_i *and well-formed executions* $E_i = \langle A_i, P, \leq_{po_i}, \leq_{so_i}, W_i, V_i\rangle$ *with happens-before order* $\leq_{hb_i}$ *such that* $C_0 = \phi$, $C_{i-1} \subseteq C_i$ *for all* $i > 0$, $\bigcup C_i = A$, *and for each* $i > 0$, *the following are satisfied:*

1. $C_i \subseteq A_i$
2. $\leq_{hb_i} |_{C_i} = \leq_{hb} |_{C_i}$
3. $\leq_{so_i} |_{C_i} = \leq_{so} |_{C_i}$
4. $V_i|_{C_i} = V|_{C_i}$
5. $W_i|_{C_{i-1}} = W|_{C_{i-1}}$
6. *For all reads* $r \in A_i - C_i, W_i(r) \leq_{hb_i} r$
7. *For all reads* $r \in C_i - C_{i-1}, W_i(r) \in C_{i-1}$ *and* $W(r) \in C_{i-1}$

For example, in Fig. 2, suppose that we want to commit the write action A2:y=r1;. Then $V(A2)$ is the value read in action A1:r1=x. The value of x must be obtained from a write that either happened-before $A1$ (the initialization action is the only option) or is already committed. In the former case, the value read is 0, in the latter case, it is the value written by $B2$. Similarly, the value written in $B2$ must be the value read in $B1$, which must be either committed or happen-before it. However, $A2$ was not committed, so the initialization action is the only option. Thus the only possible outcome is r1==r2==0. Clearly, understanding and using this definition is difficult for all but the most trivial programs.

3 Java PathRelaxer (JPR)

In order to check properties of a program with data races, we want to generate all the possible legal executions of the program under the JMM. To do this, we start with a set of legal executions, namely the sequentially consistent ones. Then, from those executions, we find which alternative writes could have been seen by a read, i.e. what are other possibilities for $W(r)$ that do not violate well-formedness, and use these to generate additional executions. The process is repeated until it converges. Completely out-of-thin-air values are avoided because each value seen by a read must have been written in some execution already generated. In the rest of this section, we describe how this process was implemented using model checking in JPR[4]. In Sect. 4, we formulate the process more formally as the computation of the least fixed-point of a monotone function and show that the set of executions generated is an overapproximation of the JMM.

JPR extends Java Pathfinder (JPF) [22,15]. JPF is an explicit-state model checker that analyzes Java bytecode. Its custom JVM provides an efficient representation of the explored state space and can potentially provide paths (or traces) corresponding to all possible interleaving of the threads. Assertions are checked at appropriate points during generation of paths. Generic properties such as deadlock freedom may also be checked. JPF has an extensible architecture via its Listener interface. While standard JPF explores paths corresponding only to sequentially consistent executions, JPR explores all paths allowed by the JMM.

The basic idea behind JPR is to maintain a map, *WriteSet*, that maps memory locations to sets of (write action, value written) pairs. For a read action of variable x, instead of the standard JPF behavior where the read sees the value of the most recent write to x on the current path (which also corresponds to sequentially consistent behavior), a value from an element of *WriteSet*(x) is chosen. By exploring all of the possible *WriteSet* entries at each point and discarding paths that do not correspond to a well-formed execution, an iteration of the JPR algorithm generates all of the well-formed paths consistent with a given *WriteSet*. It also returns a possibly expanded *WriteSet* containing all of the writes that occurred during its execution. By repeating the process until the *WriteSet* no longer changes, JPR generates a superset of the legal executions of the program.

The **JMMAwareJPF** algorithm given in Fig. 3 represents the overall structure of JPR. A JPR specific listener, **JMMListener** is registered with JPF, then JPF is invoked iteratively. **JMMListener** takes the *GlobalWriteSet* from

[4] A discussion of JPR focusing on more technical implementation issues related to extending JPF can be found in [13].

```
JMMAwareJPF(Program)
  $GlobalWriteSet_{old} \leftarrow GlobalWriteSet_{new} \leftarrow \emptyset$
  $converged \leftarrow false$
  while $\neg converged$ do
    Call JPF(JMMListener($GlobalWriteSet_{old}$))
    $GlobalWriteSet_{new} \leftarrow$ JMMListener.$GlobalWriteSet_{new}$
    if $GlobalWriteSet_{new} == GlobalWriteSet_{old}$ then
      $converged \leftarrow true$
    else      //not converged
      $GlobalWriteSet_{old} \leftarrow GlobalWriteSet_{new}$
  endwhile
```

Fig. 3. JMMAwareJPF, the top level algorithm in JPR

the previous iteration and returns a new, possibly extended *GlobalWriteSet*, terminating when *GlobalWriteSet* no longer changes. Initially, the *GlobalWriteSet* is empty.

JMMListener is described in Figs. 4 and 5. As various search related events in JPF occur (i.e. start search, advance state, backtrack, execute an instruction, as represented by the variable *searchEvent* in Fig. 4) occur, the corresponding code is executed. Σ is a representation of the current state and is pushed onto a stack when a search starts and when the state advances, and popped when the search backtracks. When the end of a path is reached, the path is tested to see if it is well-formed. If so, the *WriteSet* of the current path is unioned with the $GlobalWriteSet_{new}$, otherwise the current *WriteSet* and path are discarded.[5]

```
JMMListener($GlobalWriteSet_{old}$)
  $GlobalWriteSet_{new} \leftarrow \emptyset$                    //New global WriteSet
  $\Sigma : \langle WriteSet, ActionSet, HBSet, ImposeSet, Read, Write, ThreadLast \rangle$
                                                    //Current state metadata
  switch($searchEvent$)
    case SEARCH STARTS:
      $WriteSet \leftarrow GlobalWriteSet_{old}$
      $ActionSet \leftarrow HBSet \leftarrow ImposeSet \leftarrow \emptyset$
      $\forall loc : Read(loc) \leftarrow undef, Write(loc) \leftarrow undef$
      $\forall tid : ThreadLast(tid) \leftarrow undef$
      $Stack.push(\Sigma)$
    case STATE ADVANCES:
      $Stack.push(\Sigma)$
    case STATE BACKTRACKS:
      $\Sigma \leftarrow Stack.pop()$
      if END OF PATH then
        if path is well-formed then
          $GlobalWriteSet_{new} \leftarrow GlobalWriteSet_{new} \cup WriteSet$
        else ignore write set and discard path
    case INSTRUCTION EXECUTES:
        See Fig. 5
```

Fig. 4. JMMListener algorithm

In JPR, JPF's state representation is extended with the additional information given below, where *Aid* is the domain of action IDs, *Val* is the domain of values, *Loc* is the domain of memory locations, etc. Action was defined in Sect. 2.

[5] Although not shown in the algorithm, because paths may be discarded, assertion violations are not reported until the end of the path is reached. This is a departure from standard JPF behavior, which reports assertion violations when they occur.

```
case EXECUTING ACTION where $action = (aid, tid, kind, loc)$:
  $ActionSet \leftarrow ActionSet \cup \{action\}$ // add current action to action set
  $HBSet \leftarrow HBSet \cup \{(ThreadLast(tid), aid)\}$ //update $\leq_{hb}$ due to $\leq_{po}$
  $ThreadLast(tid) \leftarrow aid$
  if isRELEASE($kind$) then
    if $kind$ == VOLATILE WRITE writing $val$ then
      $Write(aid) \leftarrow val$
  else if isAQCUIRE($kind$) then
    // for each release action rel that syncs with action do
    for each $rel = (raid, rtid, rkind, rloc)$ s.t. isRELEASE($rel$)
              $\wedge (raid, aid) \in HBSet$ do
      $HBSet \leftarrow HBSet \cup \{(raid, aid)\}$ //update $\leq_{hb}$ due to $\leq_{so}$
      if $kind$ == VOLATILE READ then
        //let latest denote the most recent volatile write that syncs with action
        let $latest = (lid, ltid, lkind, lloc)$ s.t. $lkind$ == VOLATILE WRITE $\wedge$
            $(lid, aid) \in HBSet \wedge (\nexists a_k : a_k \in Aid \wedge (a_k, aid) \in HBSet \wedge Path(a_k) > Path(lid))$
        //Save the write action and value in Read. This is always a past write.
        $Read(aid) \leftarrow (lid, false, Write(lid))$
  else if $kind$ == WRITE of value $val$ then
    // if this write action is in the impose set, check for well-formedness
    if for some $val'$, $(aid, val') \in ImposeSet$ then
      if $val' \neq val$ then
        backtrack // value written is not the imposed value, abandon the path
      else //check for $\leq_{hb}$ consistency
        if $\exists r \in ActionSet : Read(r.aid) == (aid, true, *) \wedge r.aid \leq_{hb} aid$ then
          backtrack //not $\leq_{hb}$ consistent, abandon path
    //else path is still well-formed, save values and continue
    $Write(aid) \leftarrow val$
    $WriteSet(loc) \leftarrow WriteSet(loc) \cup \{(aid, val)\}$
  else if $kind$ == READ then
      non−deterministically choose $(w, val) \in WriteSet(loc)$ do
      if $w \in ActionSet|_{aid}$ then // this is a past read
        //check for $\leq_{hb}$ consistency
        if $(\nexists wa : wa \in ActionSet \wedge wa.kind == WRITE \wedge wa.loc == loc$
            $\wedge w \leq_{hb} wa.aid \wedge wa.aid \leq_{hb} aid)$ //$\leq_{hb}$ consistent past read
        then
            $Read(aid) \leftarrow (w, false, Write(w))$
        else //$\leq_{hb}$ inconsistent past read
            continue with next write set entry
      else // potential candidate for a future read
        if $(\nexists val' : val' \in Val \wedge (w, val') \in ImposeSet \wedge val' \neq val)$ then
            $ImposeSet \leftarrow ImposeSet \cup \{(w, val)\}$
            $Read(aid) \leftarrow (w, true, val)$ //true indicates future write
        else //illegal future read, was in impose set with inconsistent value
            continue with next write set entry
```

Fig. 5. Continued from Fig. 4. The algorithm for enforcing JMM's semantics by keeping track of write sets and happens-before relation among the actions executed on this path.

– ***Path***: Sequence of action ids that represent the current path of execution. For a given action id *aid*, *Path*(*aid*) represents the index of that action id, where *Path*(*aid*) is 1 for the id of the first executed action in *Path*.
– ***WriteSet***: $Loc \rightarrow 2^{Aid \times Val}$ maps a memory location to a set of action ID, value pairs, where each action is a *WRITE*.
– ***ActionSet***: 2^{Action} contains the actions that have been executed on the current path so far.
– ***HBSet***: $2^{Aid \times Aid}$ is a set of pairs of action IDs where $\langle aid_1, aid_2 \rangle \in$ ***HBSet**** if and only if both are in ***ActionSet*** and $aid_1 \leq_{hb} aid_2$ and where ***HBSet**** is the transitive closure of the relation represented by ***HBSet***.

- ***ImposeSet***: $2^{Aid \times Val}$ is a set of action ID, value pairs, where each action is a *WRITE*. In a well-formed path, if a read action r obtains a value *val* from write action w which may be executed in the future, w must occur at some point in any well-formed path containing r, and it must actually write *val*. Thus the *ImposeSet* maps write actions to values imposed on them by past reads.
- ***Read***: $Aid \rightarrow Aid \times boolean \times Val$ maps *READ* and *VOLATILE READ* action IDs to a triple containing the write action it sees, i.e. $W(rid)$ and the value it returns, $W(V(rid))$ for action id *rid*. The boolean value indicates whether the $W(rid)$ occured in the future on the current path.
- ***Write***: $Aid \rightarrow Val$ maps *WRITE* and *VOLATILE WRITE*. action IDs to the value written by the corresponding action, i.e. $V(wid)$.
- ***ThreadLast***: $Tid \rightarrow Aid$ maps a thread id to the latest action performed by the thread and is used to maintain the program order, $\leq_{po}$.

4 Properties of the JPR Algorithm

In this section, we discuss the properties of JPR and its basic algorithms. Most of the proofs and some lemmas are omitted for brevity but can be found in the companion technical report [12]. Executions are the abstraction used in the JMM and defined in Def. 1 while paths are the totally ordered sequences of actions generated by JPR. We say that path p corresponds to execution $E = \langle A, P, \leq_{po}, \leq_{so}, W, V \rangle$ where A is the set of actions that occur in p, P is prog, $\leq_{po}$ is the union over all threads of $\leq_{path}$ restricted to each thread, and $\leq_{so}$ is $\leq_{path}$ restricted to the synchronization actions in p. If a non-volatile read r uses WriteSet entry (w, val), then $W(r) = w$ and $V(w) = val$. $V(w)$ is well-defined since all reads of the same write action in a path must get the same value.

For a fixed program, *prog*, usually considered to be understood, and letting *WS* be the type of *WriteSet*, let $\mathrm{JPR}_{\mathrm{prog}} : WS \rightarrow WS * Paths$ be a function that takes a $ws \in WS$ and returns a new WS and a set of paths *paths*. $\mathrm{JPR}_{\mathrm{prog}}$ is a function represents an invocation of JPF seen in Fig. 3, where *Paths* is the set of paths searched by JPF. For $ws \in WS$ and path p, we say that $ws \overset{\mathrm{JPR}}{\rightarrow} p$ if $p \in \mathrm{JPR}_{\mathrm{prog}}(ws).paths$. We say that $ws \overset{\mathrm{JPR*}}{\rightarrow} p$ if $\exists i \geq 0 : p \in (\mathrm{JPR}^{i}_{\mathrm{prog}}(ws)).paths$[6]. For convenience, we overload $\overset{\mathrm{JPR}}{\rightarrow}$ and $\overset{\mathrm{JPR*}}{\rightarrow}$ and also say $ws \overset{\mathrm{JPR}}{\rightarrow} ws'$ or $ws \overset{\mathrm{JPR*}}{\rightarrow} ws'$ with the obvious meanings.

Lemma 1 (HBSet). *JPR accurately records $\leq_{hb}$ for any generated path p or prefix of a path. It is invariant that for $\forall a_i, a_j \in p : a_i \neq a_j : a_i \leq_{hb} a_j \equiv (a_i, a_j) \in HBSet \vee (\exists a_k : (a_i, a_k) \in HBSet \wedge (a_k, a_j) \in HBSet)$.*

Proposition 1 (Safety). *Let ws_{sc} be the set of (w, v) pairs seen in the sequentially consistent executions of prog. If $ws_{sc} \overset{JPR*}{\rightarrow} p$, then p corresponds to a well-formed execution of prog.*

[6] If $i = 0$, p must be empty.

Proposition 2 (Completeness). *$JPR_{prog}(ws)$ generates a path corresponding to every well-formed execution of prog satisfying ($\forall reads\ r \in A$: $(W(r), V(W(r))) \in ws$).*

Lemma 2 (Monotonicity of $\mathbf{JPR_{prog}}$). *JPR_{prog} is monotonic, i.e.*

- *$ws \subseteq ws'$ and $JPR_{prog}(ws) = (ws_1, paths)$ and $JPR_{prog}(ws') = (ws'_1, paths')$ then $ws_1 \subseteq ws'_1$, and $paths \subseteq paths'$.*
- *$ws \subseteq ws_1$.*

Theorem 1 (Convergence). *For finite state, terminating program prog, Suppose that JPR_{prog} is applied iteratively starting with ws_0. The process will reach a fixed point $ws*$ in a finite number of steps and the resulting $ws*$ will be the least fixed point of JPR_{prog} at least ws_0.*

Proof. Noting that the (finite) set of $(ws, paths)$ pairs with subset inclusion form a complete lattice, the result from the Knaster-Tarski fixed point theorem and lemma 2. □

Theorem 2 (Overapproximation). *Let ws_{sc} be the smallest WriteSet containing all of the values seen in the set of sequentially consistent executions of finite state, terminating program prog and $ws_{sc}*$ be the least fixed point of JPR_{prog} at least ws_{sc}. Let $JPR_{prog}(ws_{sc}*).paths$ be the set of paths generated by ws_{sc}. Let $JmmLegal_{prog}$ be the set of legal paths. Then $JmmLegal_{prog} \subseteq JPR_{prog}(ws_{sc}*).paths$.*

The above results show that the set of paths generated by JPR_{prog} is an overapproximation of the JMM. As a practical matter, this means that JPR is sound: if we show that a data race is benign by tesing with JPR then we can conclude that a precise tool (if one existed) would also find it benign. On the other hand, the overapproximation allows false alarms. Below, we discuss the source of the imprecision in JPR.

Initially, x = y = z = 0

Thread 1	Thread 2	Thread 3	Thread 4
A1: r1 = x	B1: r2 = y	C1: z = 1	D1: r3 = z
A2: y = r1	B2: x = r2		D2: x = r3

Fig. 6

In the example shown in Fig. 6, JPR generates a path with result r1 == r2 == 1, and r3 ==0. There is a valid path where action D2 writes 1, A1 reads D2, A2 writes 1, B1 reads A2, B2 writes 1. Then, on the next iteration, A1 reads B2 (and imposes 1), B1 reads A2, and then B2 successfully writes 1 as imposed by A1, while D1 reads the initialization action. However, this is not legal according to the JMM. In order for r1 == r2 == 1 to appear in a JMM-legal execution, D2 would need to be a committed action with $V(D2) == 1$. But then r3 must already be 1, so the execution is not legal. The value 1 is considered to come out-of-thin-air in any execution where r3 == 0. Note that this is the same program

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
y = r1;	if(r2 < 2)
	x = 3;
	x = 2;

(a) r1 == r2 == 2 is allowed by approach **scope** but forbidden by approach **occurrence**.

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
y = r1;	if(r2 == 2)
	x = 1;
	else
	x = 1;

(b) r1 == r2 == 1 is allowed by approach **occurrence** but forbidden by **scope**.

Fig. 7. ActionID examples

as Fig. 2 with the addition of Threads 3 and 4. In Fig. 2, JPR does not generate paths with out-of-thin-air values. Thus JPR may generate illegal paths with out-of-thin-air values only when the out-of-thin-air values actually do appear in some generated path. It does not generate completely arbitrary out-of-thin air values. JPR could be made more precise by tracking impose requirements across iterations and dependent actions at the cost of significantly increased time and space overhead.

5 Experience

One of the difficulties encountered when implementing JPR was the lack of a well-defined connection between the notion of executions used to define the JMM and actual Java programs. This manifested itself in the representation of the actionID. Within a single execution, the basic requirement of the actionIDs is uniqueness. However, both the JMM definition of legal executions (Def. 3) and JPR require that the identity of actions be compared across different executions and paths, i.e. we must be able to determine if, say, a read of x in one execution or path is the same action as a read of x in another by comparing their IDs. This becomes problematic for programs with branches.

We considered four approaches to identify actions. Let t be the thread, k be the kind, v be the variable, and val be the value read or written.

Occurrence. (k, t, v, n). n counts occurrences of k-actions by thread t on v.

Scope. (t, S, n). S refers to the lexical scope, repeated invocations of the same instruction, such as in a loop are differentiated by a sequence number n.

Value. (k, t, v, val). Actions with the same k,v, and t are distinguished by the value. This is the approach used in [7] is not adequate because actions are no longer uniquely identified if a thread writes the same value to a variable more than once.

Occurence-Val. (k, t, v, val, n). Adds an occurence count$n$ to **value**with the consequence that for a write w, $V(w)$ always maps to the same value, making legality rules 4 and 7 in Def. 3 redundant and inoperative, respectively.

	#thr	**scope**				**occurrence**				**occurrence-val**				**JPF**		
		iter	T	states	M	iter	T	states	M	iter	T	states	M	T	state	M
tc1	2	3	1.4	164	15	3	1.4	164	15	3	1.5	173	15	0.8	44	15
tc3	3	3	4.1	2315	25	3	4.1	2315	24	3	4.7	2582	25	0.9	349	15
tc5*	4	3	11.2	6326	26	3	12.3	6326	26	3	14.8	6877	26	1.2	1169	15
tc7	2	4	2.2	496	25	4	2.2	496	25	4	2.3	557	26	0.8	64	15
tc9	3	3	3.0	1737	15	3	3.0	1737	15	3	3.3	1929	15	1.0	279	15
tc9a	4	3	2.2	880	15	3	2.2	880	15	3	2.7	914	15	0.9	261	15
tc11	2	4	3.1	1147	26	4	3.2	1147	26	4	4.0	1452	25	0.9	95	15
tc13	2	3	1.2	32	15	3	1.2	32	15	3	1.2	32	15	0.8	24	15
tc17	2	3	1.9	565	15	3	1.9	565	15	3	1.9	641	15	0.8	72	15
tc19	3	3	5.2	2205	25	3	5.6	2205	25	3	5.5	2502	25	0.9	381	15
hash	2	3	1.5	237	15	3	1.5	237	15	3	1.5	237	15	0.7	60	15
hash	4	3	38.3	12442	33	3	38.2	12442	34	3	38.6	12442	34	1.7	3720	15
hash2	2	3	1.3	23	15	3	1.3	23	15	3	1.3	23	15	0.8	98	15
isprime	2	3	2.0	308	15	3	2.1	308	15	3	2.2	308	23	0.9	118	15
dcl	2	3	1.1	22	15	3	1.2	22	15	3	1.2	22	15	0.9	243	15
peterson	2	3	1.5	83	15	3	1.5	83	15	3	1.5	83	15	1.0	194	15
dekker	2	3	1.3	24	15	3	1.2	24	15	3	1.2	24	15	0.9	203	15

Fig. 8. Experimental results comparing the performance of JPR using ActionID approaches **scope**, **occurrence**, and **occurrence-val**, respectively. Column T represents the total time in seconds; column M represents the maximum memory consumption in megabytes. * means that JPR generates paths not allowed by JMM.

The different approaches yield different sets of legal executions. Consider Figs. 7b and 7a. Approach **occurrence** allows the outcome in Figs. 7b because both assignments to x are considered to be the same action; if committed, the assignments could be included in the justifying executions. However, it forbids the outcome in Fig. 7a since the assignment x = 2 in two different executions may have different actionIDs depending on whether or not the branch was taken. Approach **scope** allows the indicated outcome in Fig. 7a because regardless of the execution order, x = 2 is within the same lexical scope and can be committed and verified. It does not allow the outcome in Fig. 7b because the two x = 1 actions are within different scopes and if one is committed, it is impossible for the action to be included in subsequent verification executions. We have implemented **scope**, **occurrence**, and **occurrence-val** in JPR and compared these approaches for several examples. A thorough analysis of which ActionID approach would be more appropriate for JMM, however, is outside the scope of this paper; we limit our contribution to calling the issue to the research community's attention and implementing the three approaches in JPR.

We ran JPR on three groups of test programs. Representative results are listed in Fig. 8. The columns contain the number of threads, and for each action ID approach described above, the number of iterations of JPF required to converge, the total time, the number of states visited in the final iteration, and the maximum memory consumed, respectively. The final columns indicate the resource

```
   public final class String{
 2   private final char value[];   //final fields set in constructor
     private final int offset , count;
 4   private int hash;     //not final, default value is 0
     ...
 6   public int hashCode(){
       int h = hash;
 8     int len = count;
       if(h == 0&&len > 0){
10       int off = offset;
         char val[] = value;
12       for(int i = 0; i < len; i++){h = 31*h + val[off++];}
         hash = h;
14     }
       h = hash;   //redundant read
16     return h;
     }
18 }
```

Fig. 9. The data races are benign line 15 is removed from the program. Otherwise, the races are not benign

usage for standard JPF for comparison purposes. All testing was performed on a 2.27 GHz Intel(R) Core(TM) i5 CPU, 4 GB main memory, with 64-bit Windows 7 operating system, JDK 1.6, and JPF version 6.

The first group, labeled *tc*1 through *tc*20 are the test cases derived from the JMM Causality Test Cases [14], which were designed to illustrate the properties of the JMM (even numbered test cases are omitted for brevity, correctly synchronized test cases are not interesting). For these, we output the paths generated by JPR and compared them with the legal executions according to JMM. All legal executions were generated with *tc*5 and *tc*10 generating forbidden executions. *tc*5 is the example in Fig. 6 and discussed in section 4. *tc*10 is similar.

The second group contains more realistic examples. In *hash*, the hashCode method (Fig. 9 with line 15 deleted) contains a racy lazy initialization of its hash field; the read of hash (Line 7) and the write of hash (Line 13) may form a data race. This race is benign because in all legal executions, even the sequentially inconsistent ones, a call to the hashCode method will always return the correct hash code value. The assertions applied in both the 2-thread version and 4-thread version of *hash* confirm this finding.

*hash*2 on the other hand, calls a slightly different version of hashCode (Fig. 9) where the returned value is reread from hash (Line 15). This is correct under sequential consistency, but under the JMM, the race is not benign; a thread calling hashCode could get the initial value 0 instead of the correct hash code. The assertions failed in this case.

In *isprime* [20, §2.6], data races occur when multiple threads read and write elements of a shared array without synchronization. Because accesses to array elements in Java do not have volatile semantics, these accesses are racy and reads may see stale values. In this program, reading a stale value affects performance but not overall correctness; it always correctly identifies the prime numbers. The assertion succeeded in this test case.

```
class Foo{
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null){
                //read helper without synchronization, if not null, return value imme-
diately.
      synchronized(this){ //if helper was null, acquire monitor and read it again
        if(helper == null){ //if it is still null
          helper = new Helper();  //instantiate a Helper object
        }
      } //release the monitor lock by leaving synchronized block
    }
   return helper;
  }
}
```

Fig. 10. Double checked locking

The third group contains the well-known synchronization problems. *dcl* is the infamous *double-checked locking* (DCL) idiom [2] which attempts to reduce locking overhead by lazy initialization of an object, but fails to safely publish the object, allowing other threads to see a partially constructed object. In the test case, two threads call the getHelper() method of Foo shown in Fig. 10.

peterson and *dekker* are implementations of the classic mutual exclusion algorithms without using volatiles. They guarantee mutual exclusion under sequential consistency, but fail in relaxed memory models such as JMM. Assertions inserted to check non-interference in the critical sections in *peterson* and *dekker* failed as expected. The paths in which *dcl*, *peterson*, and *dekker* had assertion violations are legal according to JMM and therefore were detected by JPR but are not exhibited by sequentially consistent programs. Standard JPF cannot detect these problems.

6 Related Work

Ferrara [9] used a fixed point formulation to interpret the happens-before memory model. This work was done in the context of abstract interpretation, but was not implemented into a real tool. Botincan, et. al. [3] showed that the causality requirements of the JMM are undecidable.

Work has been done using various techniques to verify programs under relaxed hardware and programming language memory models. JUMBLE [10] is a dynamic analysis system that implements an adversarial memory by keeping track of a history of writes to racy variables. When a racy variable is read, the adversarial memory returns some past value that JMM allows and is likely to crash the program. Unlike JPR, this tool does not consider nonracy variables and cannot simulate reading from a future write, hence can only provide an under-approximation of JMM. RELAXER [6], a two-phase analysis tool, employs dynamic analysis in its first phase to detect races on SC executions and predicts potential happen-before cycles if run under one of TSO, PSO, or PSLO. In the second phase, it runs the tested program under the relaxed memory model with a controlled scheduler that realizes the one with happen-before cycle to check for program violations. JPR can be extended with a similar heuristic to prefer exploring paths that may end up with a happen-before cycles. We also mention that we have extended JPF to implement the TSO and PSO memory models. While not of significant practical interest, these could be implemented without requiring iteration, thus giving an illustration of the significant complexity of the JMM.

Burckhardt, Alur and Martin [4] applied a SAT-based bounded verification method to check concurrent data types under relaxed memory ordering models employed by multiprocessors while Burckhardt and Musuvathi [5] described a monitor algorithm that could be implemented by model checkers to verify relaxed memory models due to store buffers. The MemSAT system [21] system accepts a test program containing assertions and an axiomatic specification of a memory model and then uses a SAT solver to find a trace that satisfies the assertions and axioms, if there is one. Both the original JMM specification [11], and the modified version proposed by [1] were found to have surprising results when applied to the JMM Causality test cases. MemSAT is intended to be used with small "litmus test" programs to debug memory model specifications. In contrast, JPR is intended to reason about programs. It explores all possible paths according to the JMM and reports any assertion (program constrain violation) violations, which can help to decide whether the races are benign or not. JPR can be used with programs containing object instantiation, loops and other features that are not well supported in MemSAT. The authors of Java memory model developed a simple simulator for the JMM [18] which appears to be geared more towards understanding the memory model than serving as a tool for program analysis. De et al. [8] developed OpMM which uses a model checker similar to Java PathFinder for state exploration. In contrast to JPR, OpMM is an underapproximation of the JMM where read actions can see past writes that occur before it in a sequentially consistent execution. As an underapproximation, OpMM could be used for bug detection of racy programs, but not verification.

7 Conclusion

We have described JPR, an extension of JPF that generates an overapproximation of the JMM. With this extension, JPF can also be applied to the verification of Java programs with data races. Our approach runs the model checking algorithm in an iterative way to compute a least fixed point of a monotone function that can generate sequentially inconsistent executions.

Although, like any tool based on model checking, state-space explosion is a potential problem, we were able to successfully use the tool to show that data races in some examples are benign. We also demonstrated assertion violations in some programs, which are not detectable without awareness of the JMM.

Finally, we have shown that an operational semantics of JMM requires more precise definition of the action ID concept. We have proposed, implemented, and empirically compared three approaches. Although, drawing a conclusion on which of these approaches would be the most appropriate one is outside the scope of this paper, we hope to start a fruitful discussion on the topic.

References

1. Aspinall, D., Ševčík, J.: Formalising Java's Data Race Free Guarantee. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 22–37. Springer, Heidelberg (2007)
2. Bacon, D., Bloch, J., Bogda, J., Click, C., Haahr, P., Lea, D., May, T., Maessen, J., Manson, J., Mitchell, J.D., Nilsen, K., Pugh, B., Sirer, E.G.: The "double-checked locking is broken" declaration, `http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html`
3. Botinčan, M., Glavan, P., Runje, D.: Verification of Causality Requirements in Java Memory Model Is Undecidable. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009. LNCS, vol. 6068, pp. 62–67. Springer, Heidelberg (2010)
4. Burckhardt, S., Alur, R., Martin, M.M.K.: Bounded Model Checking of Concurrent Data Types on Relaxed Memory Models: A Case Study. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 489–502. Springer, Heidelberg (2006)
5. Burckhardt, S., Musuvathi, M.: Effective Program Verification for Relaxed Memory Models. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 107–120. Springer, Heidelberg (2008)
6. Burnim, J., Sen, K., Stergiou, C.: Testing concurrent programs on relaxed memory models. In: ISSTA (2011)
7. Cenciarelli, P., Knapp, A., Sibilio, E.: The Java Memory Model: Operationally, Denotationally, Axiomatically. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 331–346. Springer, Heidelberg (2007)
8. De, A., Roychoudhury, A., D'Souza, D.: Java Memory Model aware software validation. In: PASTE (2008)
9. Ferrara, P.: Static analysis via abstract interpretation of the happens-before memory model. In: Proceedings of the 2nd International Conference on Tests and Proofs (2008)
10. Flanagan, C., Freund, S.N.: Adversarial memory for detecting destructive races. In: PLDI, pp. 244–254 (2010)
11. Gosling, J., Joy, B., Steele, G., Bracha, G.: Java Language Specification, 3rd edn. Addison-Wesley (2005)
12. Jin, H., Yavuz-Kahveci, T., Sanders, B.A.: Java memory model-aware model checking. Tech. Rep. REP-2011-516, Department of Computer and Information Science, University of Florida (2011), `http://www.cise.ufl.edu/tr/REP-2011-516/`
13. Jin, H., Yavuz-Kahveci, T., Sanders, B.A.: Java Path Relaxer: Extending JPF for JMM-aware model checking. In: JPF Workshop 2011 (2011)
14. JMM causality test cases, `http://www.cs.umd.edu/ pugh/java/memoryModel/ unifiedProposal/testcases.html`
15. Java Pathfinder, `http://babelfish.arc.nasa.gov/trac/jpf`
16. Java Racefinder, `http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-racefinder`
17. Kim, K., Yavuz-Kahveci, T., Sanders, B.A.: JRF-E: Using model checking to give advice on eliminating memory model-related bugs. In: ASE (2010)
18. Manson, J., Pugh, W.: The Java Memory Model simulator. In: Workshop on Formal Techniques for Java-like Programs (2002)
19. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: POPL 2005 (2005)

20. Oracle thread analyzer's user guide, `http://download.oracle.com/docs/cd/E18659_01/html/821-2124/gecqt.html`
21. Torlak, E., Vaziri, M., Dolby, J.: MemSAT: checking axiomatic specifications of memory models. In: PLDI (2010)
22. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering Journal 10(2) (April 2003)
23. Ševčík, J., Aspinall, D.: On Validity of Program Transformations in the Java Memory Model. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 27–51. Springer, Heidelberg (2008)