

Java Path Relaxer: Extending JPF for JMM-Aware Model Checking

Huafeng Jin, Tuba Yavuz-Kahveci, and Beverly A. Sanders
 Dept. of Computer and Information Science and Engineering
 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 nevertheless 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 have developed an extension of Java Pathfinder called Java Path Relaxer (JPR) that generates an overapproximation of the executions that are legal according to the Java Memory Model and can thus be used to soundly reason about programs with data races. In this paper, we discuss some of the interesting implementation issues that arose when implementing the extension of JPF.

Index Terms—relaxed memory model, benign data races, Java memory model

I. 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) [19], 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. This implies that all threads see the writes in a consistent way. Sequential consistency is desirable because it corresponds with programmers’ intuition, and allows formal reasoning techniques and tools, most of which assume sequential consistency, to be used.

However, memory systems in most modern multi-core processors are not sequentially consistent. Also, a variety of compiler optimizations and transformations that would be correct in a sequential program may introduce sequentially inconsistent behavior into a multi-threaded program. For example, consider the following program fragment where `result` and `done` are visible to multiple threads.

```
result = computation();
done = true;
```

The variable `done` is initially false and not accessed by `computation()`, which updates `result`. Since the two statements are independent, the order could be reversed without affecting the sequential semantics. However, if this fragment occurs in a concurrent program executed under a relaxed memory model, this is not necessarily semantically neutral. If `done` is intended to be a flag to other threads that `computation()` is finished, then reversing the order

could result in another thread finding `done==true`, and then seeing a state reflecting an incomplete execution of `computation()`. This would probably be a catastrophic bug, and also one that would manifest itself only intermittently, with different frequencies on different systems. As a result, testing is problematic, and tool support welcome. Most model checkers, though, implicitly assume sequential consistency and would not exhibit the sequentially inconsistent behavior that would reveal the bug.

It is possible to prevent sequentially inconsistent behaviors. Architectures provide instructions such as memory fences that can be used to prevent reordering by the hardware and are typically inserted in the object code as a result of synchronization instructions in the program’s source. Compilers can refrain from certain optimizations that may cause sequentially inconsistent behavior. Traditionally, memory models have been defined for architectures, but more recently memory models have become part of a programming language’s semantics and memory models have been defined for languages including Java [11, Chapter 17], .net based languages such as C# [9, Partition I, section 12.6], C++ [4], and OpenMP [5]. Typically, programming languages guarantee sequential consistency only for programs that are *correctly synchronized*, i.e. *data race-free* (DRF) on all sequentially consistent executions.

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 the language’s *memory model*. In Java and C# (but not C++), all interthread accesses to volatile variables are considered ordered, as are all actions on the same thread. Thus marking `done` as volatile will eliminate the data races and guarantee that if any thread reads `result` after reading `done` and finding it true will see a value written not earlier than the latest write to `result` before setting `done`. There are a variety of other ways to introduce the synchronization orderings required to prevent data races; including locks and synchronized blocks, joins, barriers, operations on atomic objects, etc.

Many programming language memory models leave the semantics of programs with data races undefined so that a data race is always a bug. While the JMM guarantees sequential consistency only for programs that are data race free, it also constrains programs with data races in order to provide some weak security guarantees. If all of the legal executions, including those that are not sequentially consistent, of a racy

```

1 public final class String {
2     private final char value []; //final fields set in constructor
3     private final int offset;
4     private final int count;
5     private int hash;           //hash is not final, default value is 0
6     ...
7     public int hashCode(){
8         int h = hash;
9         int len = count;
10        if (h == 0 && len > 0){
11            int off = offset;
12            char val [] = value;
13            for(int i = 0; i < len; i++)
14                h = 31*h + val[off++];
15            hash = h;
16        }
17        return h;
18    }
19 }

```

Fig. 1: Java’s String class. The data race is benign.

program still satisfy the program’s specification, then we can consider the data race to be *benign*. While it is best to write data race-free programs (see [1] for a very strong opinion on the subject), occasionally, one may want to take advantage of the JMM semantics to allow races for better performance. Intentional data races can be found, for example, in the `java.lang.String` and `java.util.ConcurrentHashMap` classes.

A fragment of `java.lang.String` is shown in Fig. 1. The read of `hash` (line 8) and the write of `hash` (line 15) form a data race when two different threads invoke the method. However, in this context, the data race does not affect the correctness of `hashCode()`; in any execution it always returns the correct hash value. A more formal treatment will be given later; intuitively, the JMM guarantees that even with races, no “out-of-thin-air” values will be seen, thus any thread that reads `h` in line 8 will get 0, the initial value, or a value that has actually been assigned to it. The JMM also guarantees that correct values of final fields of properly constructed objects are always seen, thus multiple computations of `h` in lines 11-14 will always yield the same value. In a sequentially inconsistent execution, the value of the hashcode may be computed multiple times by different threads, but the correct value will always be returned. The result is that `String` objects can be treated as if immutable (and thus accessed by multiple threads without synchronization), even though the `hash` field is not marked final and is not set in the constructor. The lazy initialization provides significant performance benefit in Java programs which typically create many `String` objects that do not ever use the hashcode value.

Now consider a slightly modified version of the `String` class shown in Fig. 2. A redundant read has been added at line 17. This does not appear to change the semantics under SC. However, under the JMM, a read need not see the value of the most recent write, thus a thread calling `hashCode()` could see the non-zero value written concurrently by another thread on the first read (line 8), and the initial value on the second read (line 17) and return 0. This seemingly innocuous change has turned the benign race in 1 into a bug.

As can be seen from the above discussion, reasoning correctness about programs with data races is difficult, thus tool support is desirable. In this paper describe a JMM aware model checker, *Java PathRelaxer* (JPR), which is an extension of JPF

```

public final class String {
2     private final char value [];
3     private final int offset;
4     private final int count;
5     private int hash;           //hash is not final, default value is 0
6     ...
7     public int hashCode(){
8         int h = hash;
9         int len = count;
10        if (h == 0 && len > 0){
11            int off = offset;
12            char val [] = value;
13            for(int i = 0; i < len; i++)
14                h = 31*h + val[off++];
15            hash = h;
16        }
17        h = hash;
18        return h;
19    }
20 }

```

Fig. 2: A slightly different version of Java’s String class. A redundant assignment (line 17) is added.

that 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 [25] to be stricter than the designers intended. We have used an alternate approach where 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 by iteratively applying an extension of JPF. The set of paths generated by JPR is an overapproximation of the set of legal executions. Thus JPR can be used to verify properties of programs with races in the same way that JPF can be used to verify properties of sequentially consistent programs. JPR successfully identifies Fig. 1 being correct while Fig. 2 being incorrect. 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.

II. BACKGROUND

Below, we give a brief overview of the formal definition of the Java Memory Model. Our treatment follows that of [2], which is in turn based on the specification of the JMM given in [22], [11].

An *action* a is a memory-related operation that belongs to a thread, 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 t , detecting termination of thread t , and instantiating an object with a set of volatile fields *volatiles* and a set of non-volatile fields *fields*. All of the action kinds, with the exception of non-volatile read, non-volatile 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

Initially, $x == 0$

| Thread 1 | Thread 2 |
|---------------|---------------|
| A1: $x = 1$; | B1: $r = x$; |
| A2: $x = 2$; | |

Fig. 3: The read B2 could see either 0, 1, or 2.

- \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 $V(W(r))$.

It is not required that W returns the “most recent” write to the variable in question or that it is consistent for actions on different threads, thus allowing sequentially inconsistent behavior. In the execution sequence shown in Fig. 3, $W(B1)$ could either be the write of the initial value of x (past), A1 (most recent), or A2 (future), and hence $V(W(B1))$ could be either 0, 1, or 2.

The synchronizes-with relation, $<_{sw}$, 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 $<_{sw}$ pair as a *release* action, and the second as an *acquire*. 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 $<_{sw}$ and \leq_{po} . A well-formed execution satisfies some unsurprising constraints on the consistencies of the various partial and total orders and type safety which are omitted here for brevity. The two most important rules for our purposes are intra-thread consistency and happens-before consistency.

Definition 2 (Well-formed execution): See [2, 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 valid¹ 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} .

A *sequentially consistent* (SC) execution is one where there exists a total order, \leq_{sc} , on the actions consistent with \leq_{po}

¹Sequential validity essentially means that given the values obtained when a variable is read, each thread obeys the Java language semantics.

Initially, $x == y == 0$

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

Fig. 4: The rules for a well-formed execution admit traces with $r1 == r2 == \text{val}$, for any arbitrary “out-of-thin-air” value val of the correct type.

and \leq_{so} and where a read r of variable v sees the results of the most recent preceding write w , i.e. $w \leq_{sc} r$, and for all reads r of variable v : if $W(r) \leq_{sc} w \leq_{sc} r$ and w writes to v then $W(r) = w$.

A Java program is *correctly synchronized* if all SC executions are data race free. An important property of the JMM [11], [22] [2, Theorem 1], is that any legal execution of a well-formed correctly synchronized program is SC.

The JMM constrains programs with data races. The main goal was to provide a modicum of security guarantees even for incorrect programs with races while still allowing as many optimizations as possible. Desirable properties include type safety and no *out-of-thin-air* values. While “out-of-thin-air” value has not been precisely defined, the example in Fig. 4 [22] illustrates the idea and shows why well-formedness (Definition 2) does not suffice. In a sequentially consistent execution of the example in Fig. 4, the only values allowed are $r1 == r2 == 0$. However, letting $W(A1) = B2$, $W(B1) = A2$, and $V(A2) = \text{val}$, and $V(B2) = \text{val}$, for *any* value val of the correct type, we have a well-formed execution where $r1 == r2 == \text{val}$, and 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 causal loops that could lead to self-justifying speculative executions. 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.

Definition 3 (Legal Execution): [2, 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 $r \in A_i - C_{i-1}$, $W_i(r) \leq_{hb_i} r$
- 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. 4, 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. The difficulty of understanding and using this definition motivated the development of JPR.

III. JAVA PATHRELAXER

JPR extends JPF to generate non-SC paths (or traces). Assertions are checked at appropriate points during generation of paths. JPR maintains a function, $WriteSet$ that maps memory locations to sets of (write action, value) pairs. For a read action of variable x , instead of the standard JPF behavior where a read sees the value of the most recent write to x on the current path, the value from an element of $WriteSet(x)$ is chosen. By exploring all of the available pairs 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 writes that occurred during its execution.

An initial $WriteSet$ is obtained by running standard JPF, slightly modified to record writes. Then JPR is executed iteratively until the $WriteSet$ no longer changes. We can view an iteration of JPF as a monotonic function that takes a set of paths and a write set and generates a new set of paths and a write set. We have shown that this process converges and yields an overapproximation of the legal executions according to the JMM.[13, §4].

```

JMMAware(Program)
2  run standard JPF once and store write values in GlobalWriteSetold
   GlobalWriteSet ← GlobalWriteSetold
4  while not converged do
   GlobalWriteSet ← call
6     collectValuesJPF(GlobalWriteSetold, JMMListener)
   // Check for convergence
8   if GlobalWriteSet == GlobalWriteSetold then
   converged ← true
10  else
   GlobalWriteSetold ← GlobalWriteSet
12 endwhile

```

Fig. 5: **JMMAware**, the algorithm for computing the least fixed point from **collectValuesJPF** algorithm.

Now, we describe the algorithms used in JPR in more detail. JPF’s state representation is extended with the following metadata

- $WriteSet$: $MemLoc^2 \rightarrow 2^{ActionID \times Val}$ where each action is a *WRITE*.
- $ActionSet$: $2^{ActionID}$ $ActionSet$ is the set of actions that have been executed on the current path so far.
- $HBSet$: $2^{ActionID \times ActionID}$ in which $actionID_1 \leq_{hb} actionID_2$ for $(actionID_1, actionID_2)$. $HBSet$ records

²We use *memLoc* (memory location) to represent variables. The capitalized version means “the set of ...”.

```

collectValuesJPF(WriteSet, Listener l)
2  StatesVisited ← ∅
   Generate initial state  $s_0$ 
4  StatesVisited ← { $s_0$ }
   PathStack.push( $s_0$ )
6  Call  $l(SEARCH STARTS, WriteSet)$ 
   while PathStack is not empty do
8      $s_{curr}$  ← PathStack.peek()
     if there exists an enabled instruction  $actionID$  at  $s_{curr}$ 
       the next state is not visited yet then
10        call  $l(ADVANCING TO A NEW STATE, WriteSet)$ 
12        call  $l(EXECUTING ACTION, WriteSet)$ 
         // For read actions use Read(memLoc, actionID)
14        let  $s_{next}$  denote the next state after executing  $actionID$  on  $s_{curr}$ 
         StatesVisited ← StatesVisited ∪ { $s_{next}$ }
         PathStack.push( $s_{next}$ )
16         $s_{curr}$  ←  $s_{next}$ 
18     else
19        call  $l(BACKTRACKING, WriteSet)$ 
20        PathStack.pop()
   endwhile
22 return call  $l(SEARCH ENDS, WriteSet)$ 

```

Fig. 6: JPF’s modified DFS search algorithm that passes the write sets to the JMM listener.

```

JMMListener(SearchEvent, GlobalWriteSetold)
2  switch(SearchEvent)
   case SEARCH STARTS:
4     WriteSet ← HBSet ← ActionSet ← ImposeSet ← ∅
     Write ←  $\lambda x.undef$ 
6     Read ←  $\lambda x.undef$ 
     Stack.push(⟨∅, ∅, ∅, ∅, ∅, ∅⟩)
8     case SEARCH ENDS:
     return GlobalWriteSetold ∪ WriteSet
10    case ADVANCING TO A NEW STATE:
     Stack.push
11    (⟨WriteSet, ActionSet, ImposeSet, Write, Read, HBSet⟩)
     return
14    case BACKTRACKING:
     Stack.pop()
15    (⟨WriteSet, ActionSet, ImposeSet, Write, Read, HBSet⟩
16     ← Stack.peek())
18    if END OF PATH then
     GlobalWriteSet ← GlobalWriteSet ∪ {WriteSet}
20    if ImposeSet ⊆ ActionSet then
     report valuations
22    return
   continued with case EXECUTING ACTION in Fig. 8

```

Fig. 7: The algorithm for enforcing JMM’s semantics by keeping track of write sets and happens-before relation among the actions executed on this path.

the happens-before relation among the actions in $ActionSet$ and is updated at each step to reflect happens-before edges introduced due to program order and synchronization actions.

- $ImposeSet$: $2^{ActionID \times Val}$ where each action is a *WRITE*. $ImposeSet$ maintains information necessary to ensure that if a read r obtains a value from a future write w on some path, then w actually writes that value.
- $Read$: $ActionID \rightarrow ActionID \times Val^3$. $Read$ maps each read action to the write action it sees and the value it returns. We can get $W(r)$ and $V(W(r))$ from $Read$ where W and V are defined in Def.1.
- $Write$: $ActionID \rightarrow Val$. $Write$ maps each write action

³In Fig. 8, we also associate *memLoc* with $Read$ and $Write$ to locate the variable of the action faster.

```

24 continued from Fig. 7
case EXECUTING ACTION (actionID, actionType, memLoc, threadID):
26 ActionSet  $\leftarrow$  ActionSet  $\cup$  {actionID}
   HBSet  $\leftarrow$  HBSet  $\cup$  {(ThreadState(threadID).lastActionID, actionID)} //Add hb-edge due to  $\leq_{po}$ 
28 ThreadState(threadID).lastActionID  $\leftarrow$  actionID
   if actionType is a RELEASE then
30   if actionType is a VOLATILE WRITE then
     Write(memLoc, actionID)  $\leftarrow$  bytecodeStack.peek()
32   else if actionType is an ACQUIRE then
     for each release action releaseID that syncs with actionID do //Add hb-edges due to  $\leq_{so}$ 
34       HBSet  $\leftarrow$  HBSet  $\cup$  {(releaseID, actionID)}
       if actionType is a VOLATILE READ then
36         let latestWriteID denote last volatile write that syncs with actionID
           //action actionID reads from the past action latestWriteID
38         Read(memLoc, actionID)  $\leftarrow$  (latestWriteID, Write(memLoc, latestWriteID)) //Volatile read from most recent write
       else if actionType is a WRITE then
40         if (actionID, v)  $\in$  ImposeSet  $\wedge$  v  $\neq$  bytecodeStack.peek() then //Imposed value is not justified
           backtrack
42         else if (actionID, v)  $\in$  ImposeSet then
           if  $\exists aID \in$  ActionSet.(Read(memLoc, aID) == (actionID, ?)
44              $\wedge$  (aID, actionID)  $\in$  HBSet) //check for illegal future read
           then backtrack //hb-illegal future read
46         // no value imposed on this action or if done it proved to be consistent
           Write(memLoc, actionID)  $\leftarrow$  bytecodeStack.peek()
           WriteSet(memLoc)  $\leftarrow$  WriteSet(memLoc)  $\cup$  {(actionID, bytecodeStack.peek())}
48   else if actionType is a READ then
     non-deterministically choose (aID, v)  $\in$  GlobalWriteSetold(memLoc) do
50       if aID  $\in$  ActionSet then // past read from aID
52         if ((aID, actionID)  $\notin$  HBSet  $\vee$ 
           (aID, actionID)  $\in$  HBSet  $\wedge$ 
54           ( $\nexists$  write action waID to memLoc : waID  $\in$  ActionSet
            $\wedge$  ((aID, waID)  $\in$  HBSet  $\wedge$  (waID, actionID)  $\in$  HBSet))) //Legal past read
56         then
           Read(memLoc, actionID)  $\leftarrow$  (aID, Write(memLoc, aID))
58         //else hb-illegal past read, continue with next write set entry
       else if (aID, ?)  $\notin$  ImposeSet  $\vee$  (aID, v)  $\in$  ImposeSet then //Potential candidate for future read
60         ImposeSet  $\leftarrow$  ImposeSet  $\cup$  {(aID, v)}
           Read(memLoc, actionID)  $\leftarrow$  (aID, v)
62         // else illegal future read
   return

```

Fig. 8: Continued from Fig. 7. The algorithm for enforcing JMM’s semantics by keeping track of write sets and happens-before relation among the actions executed on this path. Synchronization actions *VOLATILE WRITE*, *UNLOCK*, etc. are *RELEASE* actions; *VOLATILE READ*, *LOCK* etc. are *ACQUIRE* actions.

to the value it writes. This is the same as $V(w)$ in Def.1

The **JMMAwareJPF** algorithm given in Fig. 5 represents JPR. The **collectValuesJPF** algorithm in Fig. 6 outlines the default depth-first search algorithm performed by standard JPF which can report various state space search events to a registered listener. The main addition made in JPR is to receive a *WriteSet* as input and pass it to the listener each time it is called. The listener returns a possibly expanded *WriteSet* used for the next iteration.

The **JMMListener** algorithm is given in Figs. 7 and 8. A non-volatile write action is finalized based on whether a value is imposed on it consistently. In other words, 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. *Write* and *WriteSet* are updated with the new write action and its value as long as this does not lead to inconsistencies with imposed

values. Additionally, it is required that if a read r sees write w , $r \not\leq_{hb} w$. Before executing a non-volatile read action, r , the algorithm finds past write actions to the memory location (those ordered before the read by \leq_{hb} as well as those that are not) and future write actions that are not ordered by \leq_{hb} . A write action w , that is in the past is eliminated if it would violate \leq_{hb} consistency due to the presence of an intervening write, i.e. if there is some other write to the location w' so that $w \leq_{hb} w' \leq_{hb} r$. For each of the remaining write action in the candidate set, the value of the write is stored in the *Read* function. If w is in the future, then the value read is stored as the imposed value in *ImposeSet*(w). If the chosen write action has already been imposed with a different value, it is eliminated from the set of candidate write actions.

One of the difficulties encountered when implementing JPR was the lack of a well-defined connection between the abstract notion of executions used to define the JMM and actual Java programs. In the development of JPR, this manifested itself in the representation of the actionID. Within a single execution,

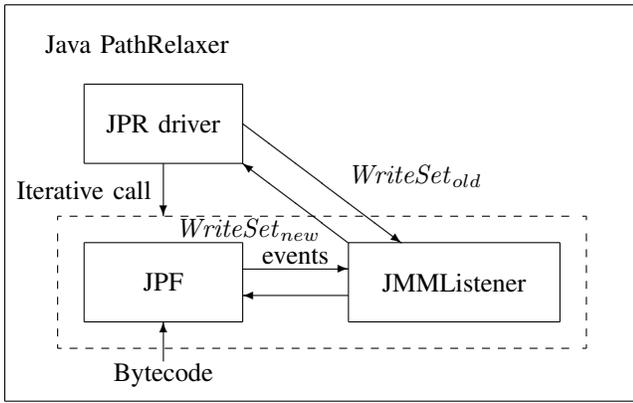


Fig. 9: The structure of Java PathRelaxer.

the basic requirement of the actionIDs is that they are unique. However, both the JMM definition of legal executions (Def. 3) and JPR require that the identity of actions be compared across different executions, i.e. we must be able to determine if, say, a read of x in one execution is the same action as a read of x in another by comparing their IDs. This becomes problematic for programs with branches. In this paper, we identify an action by $\langle k, t, x, n \rangle$ where n counts the occurrence of actions of type k from thread t on variable x . Thus, the n th occurrence of a certain type of action on a particular variable is always considered to be the same action, regardless of what happens in between, and whether or not the instructions occur in the same place in the source code. Other approaches are possible and are discussed [13], where different actionID schemes are presented and compared. However, this is still an open issue.

IV. IMPLEMENTATION

We realized the algorithm described in section III as an extension of JPF. The structure of JPR is shown in Fig. 9. JPR comprises three components; 1) JPR driver, 2) JPF core, and 3) **JMMListener**. JPR driver calls JPF (configured with **JMMListener**) iteratively as described in Fig. 5. Before each iteration, the JPR driver registers JPF with a new instance of **JMMListener** (Figs. 7 and 8) and passes it the $GlobalWriteSet_{old}$. At each event (executing an instruction or advancing/backtracking a state), JPF notifies **JMMListener** which updates the metadata (Sect. III), and in turn affects JPF when the data choice generators are set or backtracking is requested. At the end of each iteration, the JPR driver receives the $GlobalWriteSet_{new}$ from **JMMListener** and compares it with $GlobalWriteSet_{old}$. The iteration process stops when a fixed point is achieved.

A. JPR State Representation

The current implementation of JPR uses a separate stack to record the metadata together with JPF path exploration. An alternative approach would have been to extend JPF’s system state with metadata ($WriteSet$, $ActionSet$, $HBSet$, $ImposeSet$, $Read$, and $Write$). This would have simplified the control of JPR; when an ill-formed path is generated, simply requesting a backtrack would suffice. However, given the lack of an interface allowing the extension of JPF’s state

representation, following the alternative approach would have required modification of jpf-core, which was undesirable.

B. Checking Happens-before Consistency

For the JPR metadata, the basic data structures for $WriteSet$, $Write$, and $Read$ are hash tables; $ActionSet$ and $ImposeSet$ are simple sets of elements. There are many ways to implement HBSet. We used a direct implementation that constructs a directed acyclic graph (DAG) where actions are nodes and an edge between a_i and a_j implies that $a_i \leq_{hb} a_j$. When checking happens-before consistency (see item 9 of Definition 2) between a non-volatile read action r of variable var and a non-volatile write action w where $w = V(r)$, the graph is traversed to find possible paths between the two actions.

- i) The search stops when we find a path from r to w , which indicates a violation of “for all reads r of variable v , $r \not\leq_{hb} W(r)$ ”
- ii) The search stops when we find a path from w to r that contains another intermediate write action w' to the same variable, which indicates violation of “if $W(r) \leq_{hb} w \leq_{hb} r$ and w writes to v then $W(r) = w$.”

The time complexity depends on the path search algorithm used. If using depth-first search, the complexity would be $O(|A| + |E|)$ where $|A|$ represents the number of actions and $|E|$ represents the number of edges. Note that the graph is dynamic and changes as actions and edges are added to it. Happens-before consistency is frequently checked so the time complexity directly affects the performance of JPR.

C. Garbage Collection

JPF contains garbage collection which is used to recycle memory used by objects that will no longer be referenced. See the execution sequence shown in Fig. 10. Suppose Thread-1 first creates an instance of Helper at memory location $L1$ and assigns it to the shared reference h , then Thread-1 terminates. Thread-2 assigns another Helper instance to h at location $L2$, and access field x of that reference. According to the JMM, the read in Thread-2 could return either 3 or 5 (the field values of instances created at $L1$ and $L2$ respectively). However, because of the termination of Thread-1, the instance created in $L1$ is considered as “not referenced” and is automatically garbage collected by JPF. In order to simulate such JMM-legal executions, JPF garbage collection feature must be turned off for reference data types.

Suppose the constructor of Helper initializes a field x

| Helper h is a shared reference | |
|--------------------------------|-------------------------------------|
| Thread 1 | Thread 2 |
| h = new Helper(3); | |
| | h = new Helper(5); int r2 = h.x; |

Fig. 10: JPF Garbage Collection: After termination of Thread-1, the object created by Thread-1 will not be seen by Thread-2.

D. Reading Future Objects

Under the JMM, a read of a non-volatile variable may see any write to that variable, either in the past or future, as long as happens-before consistency is maintained. For reference data types, however, when reading from an object that will be created in the future, a null pointer exception will be thrown from JPF because no such object exists.

For example, see the execution sequence shown in Fig. 11. Suppose in the first iteration of JPR, Thread-1 creates a Helper instance at $L1$ and Thread-2 creates another instance at $L2$, so $WriteSet(h)$ contains two values $L1$ and $L2$. In the second iteration, given that execution sequence, the read in Thread-1 may see either the instance at $L1$ (previous write) or $L2$ (future write), but the instance at $L2$ has not been created at the time of read, so an exception will be thrown from JPF.

Suppose the constructor of Helper initializes a field x
 Helper h is a shared reference

| Thread 1 | Thread 2 |
|--|-----------------------------|
| $h = \text{new Helper}(3);$ $\text{int } r1 = h.x;$ | $h = \text{new Helper}(5);$ |

Fig. 11: Read ‘future’ object: Null pointer exception is thrown when Thread-1 reads the object that will be created in Thread-2 in the future.

To tackle this problem, we use lazy object initialization; JPR arbitrarily creates an instance at the specified location when reading a reference variable from future write if that instance has not been created.

E. Delayed Reporting of Assertion Violations

When checking program correctness, ordinary Java assertions are generally used. In standard JPF, assertion violations are caught by JPF’s generic `NoUncaughtExceptionProperty`; During model checking, JPF explores all possible interleavings and throws a `NoUncaughtException` immediately after an assertion violation occurs.

JPR on the other hand, does not report assertion errors immediately. Instead, it delays the reporting of the error until the end of each executing path. The reason is that a read may first see a future write and impose it with the value it sees, but the imposed value might not be justified when the write occurs (Fig. 8 line 40), and the path will be discarded. This means a read may return an invalid value initially and discard it later. In JPR, an assertion error will be detected when reading the invalid value, but not reported until the end of the executing path is successfully reached.

For example, see the execution sequence shown in Fig. 12. In the first iteration of JPR, Thread-2 writes 1 to x . In the second iteration, Thread-1 reads x as 1 and imposes the write of x in Thread-2 (underlined action) to write 1. Then the assertion in Thread-2 is violated. Now, we do not report the error here because the entire path will eventually be discarded later on because Thread-2 will not write 0 (not 1) to x in this case (i.e. imposed value is not justified).

| Initially, $x == y == 0$, x and y are non-volatile variables | |
|--|--|
| Thread 1 | Thread 2 |
| $r1 = x; \text{read } 1 \text{ (future), impose}$ $y = r1; \text{write } 1$ | $r2 = y; \text{read } 1 \text{ (previous)}$ $\text{assert}(r2 != 1);$ $\text{if}(r2 == 0)$ $\underline{x = 1};$ else $\underline{x = 0};$ |

Fig. 12: In the 2nd iteration, assertion is violated, but the path will also be discarded later, because the imposed value is not justified.

F. Working with Java Racefinder

The JMM guarantees that if a program is data race free on all of its SC executions (DRF), then all of its executions are SC. In this case, the original Java Pathfinder is sufficient and there is no need to apply JPR. However, is it necessary to verify that a program is DRF. Java Racefinder (JRF) [16], [17], [15] is an extension of JPF that detects data races according to the JMM precisely. If no data races are reported by JRF, the program is DRF. When checking program properties, we first run JRF to detect if the program is DRF. If not, in most cases, the next step would be to eliminate the data race. JPR is only applied to non-DRF programs where we want to determine whether an intentional race is benign.

JRF is based on JMM’s definition on DRF; “If all sequentially consistent executions of the program are free of data races” [22], it is a data race free program. During model checking for data races, JRF maintains a so-called h set that contains all the variables that are currently not involved in any data races in the sequentially consistent executions. At each read/write to non-volatile variables, JRF checks if the variable is within the h set or not. “When this condition holds for all non-volatile reads and writes in an (SC) execution, the execution is h -legal.” [16].

Because we run JRF before JPR, one might believe that we could maintain $WriteSet$ only for variables that are not in h set for any executions generated by JRF. However, h set is defined under the context of SC. A variable not involved in any sequentially consistent executions may be racy in a sequentially inconsistent one. See the example in Fig. 13. In any legal sequentially consistent executions, the write to x at line 3 will not execute, so x is not involved in data races. However, under the JMM, the read of y in line 5 may see the future write in line 4 to let the write of x happen. Therefore, if a program is non-DRF, we must maintain $WriteSet$ for all non-volatile variables in the program.

V. EXPERIENCE

To evaluate the performance, we ran JPR on three groups of test programs. All the testing was performed on 2.40 GHz Intel® Core™ 2 Duo E4600 CPU with 2MB cache, 4GB main memory, with Linux 2.6.32 operating system, JDK 1.6, and JPF version 6 (Revision 473).

| | Thread# | JPR | | | | | | | JPF | | |
|----------|---------|------|-------|------|------|----|--------|------|------|--------|------|
| | | iter | time | 1st | ave | WS | states | mem | time | states | mem |
| tc1 | 2 | 3 | 1.2s | 0.4s | 0.4s | 5 | 105 | 74M | 0.4s | 40 | 59M |
| tc2 | 2 | 3 | 1.3s | 0.4s | 0.4s | 5 | 186 | 74M | 0.3s | 49 | 59M |
| tc3 | 3 | 3 | 2.8s | 0.7s | 0.9s | 6 | 1410 | 105M | 0.4s | 309 | 59M |
| tc4 | 2 | 3 | 1.0s | 0.3s | 0.3s | 4 | 63 | 59M | 0.3s | 36 | 59M |
| tc5* | 4 | 3 | 4.5s | 1.2s | 1.5s | 10 | 4678 | 172M | 0.6s | 1169 | 74M |
| tc6 | 2 | 3 | 0.9s | 0.3s | 0.3s | 4 | 76 | 74M | 0.3s | 31 | 59M |
| tc7 | 2 | 4 | 1.9s | 0.4s | 0.5s | 8 | 291 | 105M | 0.4s | 60 | 59M |
| tc8 | 2 | 3 | 1.0s | 0.3s | 0.3s | 5 | 105 | 74M | 0.3s | 40 | 59M |
| tc9 | 3 | 3 | 2.4s | 0.6s | 0.8s | 8 | 1259 | 105M | 0.4s | 243 | 59M |
| tc9a | 4 | 3 | 1.8s | 0.5s | 0.6s | 6 | 589 | 105M | 0.4s | 229 | 74M |
| tc10* | 2 | 3 | 3.4s | 0.9s | 1.1s | 7 | 2399 | 171M | 0.4s | 477 | 59M |
| tc11 | 2 | 4 | 2.4s | 0.4s | 0.6s | 11 | 695 | 105M | 0.3s | 90 | 59M |
| tc12 | 2 | 3 | 1.2s | 0.4s | 0.4s | 8 | 102 | 59M | 0.4s | 58 | 59M |
| tc13 | 2 | 3 | 1.1s | 0.3s | 0.3s | 2 | 34 | 74M | 0.2s | 21 | 59M |
| tc16 | 2 | 3 | 1.1s | 0.3s | 0.3s | 3 | 147 | 74M | 0.3s | 42 | 59M |
| tc17 | 2 | 3 | 1.3s | 0.4s | 0.4s | 7 | 318 | 74M | 0.3s | 66 | 59M |
| tc18 | 2 | 3 | 1.4s | 0.4s | 0.4s | 7 | 318 | 74M | 0.4s | 66 | 59M |
| tc19 | 3 | 3 | 2.7s | 0.6s | 0.9s | 11 | 1419 | 108M | 0.4s | 381 | 74M |
| tc20 | 3 | 3 | 2.6s | 0.7s | 0.8s | 11 | 1419 | 108M | 0.3s | 381 | 74M |
| hash | 2 | 3 | 1.5s | 0.5s | 0.5s | 9 | 225 | 74M | 0.4s | 55 | 59M |
| hash | 4 | 3 | 17.7s | 4.2s | 5.9s | 15 | 12442 | 198M | 1.2s | 3720 | 108M |
| hash2 | 2 | 3 | 0.9s | 0.3s | 0.3s | 9 | 22 | 74M | 0.3s | 98 | 59M |
| isprime | 2 | 3 | 1.6s | 0.5s | 0.5s | 22 | 300 | 105M | 0.3s | 114 | 59M |
| dcl | 2 | 3 | 0.7s | 0.3s | 0.2s | 5 | 20 | 74M | 0.3s | 235 | 74M |
| peterson | 2 | 3 | 0.8s | 0.3s | 0.2s | 12 | 20 | 59M | 0.4s | 187 | 59M |
| dekker | 2 | 3 | 0.8s | 0.3s | 0.2s | 13 | 20 | 74M | 0.3s | 196 | 59M |

Fig. 14: Experimental results comparing the performance of JPR with standard JPF. * means executions are forbidden by JMM but generated by JPR.

| Initially, $x = y = z = 0$ | |
|----------------------------|-------------|
| Thread 1 | Thread 2 |
| 1 $r1 = z;$ | 5 $r2 = y;$ |
| 2 $\text{if}(r1 == 1)$ | 6 $z = r2;$ |
| 3 $x = 1;$ | 7 $r3 = x;$ |
| 4 $y = 1;$ | |

Fig. 13: A not racy variable under SC may be racy under non-SC.

A selection of experimental results is listed in Fig. 14. The columns contain the number of threads, and for JPR, the number of iterations of `collectValues`, JPF, the total time, the time of the 1st iteration, the average time per iteration, the `WriteSet` size, the number of states visited in the last iteration, and the maximum memory consumed, respectively. The last group of columns indicate the resource usage for standard JPF for comparison purposes; the running time, number of states explored, and memory usage.

The programs labeled (*tc1* through *tc20*, and *tc9a*) are derived from the JMM Causality Test Cases[14]⁴. These

⁴Omitted cases *tc14* and *tc15* are both programs that are correctly synchronized and thus data race free. Typically, one would analyze a program first to determine if there are data races, such as running JRF, and only employ JPR if data races are found.

| 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. 15: Test case *tc5* from [14]. Illegal behavior $r1 == r2 == 1$, and $r3 == 0$ is generated by JPR.

examples were specifically designed to illustrate the properties of the JMM and for these, we output all of the paths generated by JPR in order to compare with the legal executions according to the JMM. In all cases, JPR generated all allowed paths. For examples *tc5* and *tc10*, additional executions were generated.

tc5 is shown in Fig. 15, where 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 “out-of-thin-air” in any execution where $r3 == 0$. *tc10* is similar to *tc5*. 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.

The second group contains more realistic examples where assertions were applied to test whether the data races were benign. Examples *hash* and *hash2* are the two versions of the `hashCode` method discussed in Fig. 1 and Fig. 2 respectively. For *hash*, we also presented the two-thread and four-thread versions. In the *isprime* example [23, §2.6], data races are created 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 particular program, reading a stale value only affects performance but not program correctness; it always correctly identifies if the given integer is prime number or not, so the data race is therefore benign.

```

//Global variable
2 Foo foo = new Foo();
...
4 class Foo{
    private Helper helper = null;
6    public Helper getHelper() {
        if (helper == null){           //read helper
8            synchronized(this){
                if (helper == null){
10                 helper = new Helper(); //construct helper
12             }
14         }
        return helper;
16 }
    class Helper{
18     public int x;
        public Helper(){ x = 10; }
20 }
    class Thread0 extends Thread{
22     public void run(){
        Helper h1 = foo.getHelper ();
24     assert(h1.x != 0);
        }
26 }
    class Thread1 extends Thread{
28     public void run(){
        Helper h2 = foo.getHelper ();
30     assert(h2.x != 0);
        }
32 }

```

Fig. 16: Double checked locking

The examples in the third group are well-known synchronization problems. *dcl* is the infamous *double-checked locking* (DCL) idiom [3] 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 example given in Fig. 16, there are two threads calling `getHelper()` method of `Foo`. In `getHelper()`, the read of `helper` (line 7) is placed outside the synchronized block, while the construction of `helper` (line 10) is within the synchronized block. There is a data race between the two actions. Suppose at one time, `Thread0` is executing line 10 while `Thread1` is executing line 7 just before `Thread0` has finished construction of `helper`. Then `Thread1` detects that `helper` is not empty

and returns it immediately without entering the synchronized block. In this case, `Thread0` is actually returning a partially constructed object. To capture this bug, we inserted assertions to check if the reference returned from `getHelper()` is correctly constructed or not (line 24 and 30); if correctly constructed, the `x` field of the reference should not be 0 (initial value).

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. Peterson's algorithm is shown in Fig. 17. Under SC, line 10 in `Thread0` is mutually exclusive with line 19 in `Thread1`. After termination of the two threads, `x` should always be 2. Under the JMM, it is possible that `Thread1` writes `flag[1]` to true at first but `Thread0` later on still reads `flag[1]` as the old value false and hence skips the busy wait (line 9). Then both threads will be executing the mutually exclusive regions. In this case, the two `x++` will interfere with each other and the assertion (line 29) will fail. Dekker's algorithm uses a similar strategy as Peterson's algorithm. Assertions inserted to check non-interference in the critical sections in *peterson* and *dekker* failed as expected.

```

//Global variables
2 boolean flag [] = new boolean[]{false, false };
int turn, x = 0;
4 ...
class Thread0 extends Thread{
6     public void run(){
            flag [0] = true;
8         turn = 1;
            while( flag [1] == true && turn == 1){}
10        x++;
            flag [0] = false ;
12    }
}
14 class Thread1 extends Thread{
    public void run(){
16        flag [1] = true;
            turn = 0;
18        while( flag [0] == true && turn == 0){}
            x++;
20        flag [1] = false ;
22    }
}
//main thread
24 Thread0 t0 = new Thread0();
    Thread1 t1 = new Thread1();
26 t0.start (); t1.start ();
try{
28    t0.join (); t1.join ();
        assert(x == 2);
30 }catch(Exception e){}

```

Fig. 17: Peterson's algorithm: guarantees mutual exclusion under SC, but fails under JMM.

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. For these test cases, JPR took less time and explored fewer states than JPF because the assertion violations terminated JPR before the full path exploration was complete.

From Fig. 14, we find that JPR is able to detect synchronization problems under JMM, while JPF cannot. JPR could generate all the JMM-legal executions but yield an overesti-

mated result set ($tc5$ and $tc10$). Generally, the average time per iteration of JPR is larger than the time taken by JPF, because JPR generates more paths due to both thread and data non-determinisms, while standard JPF only has thread interleaving. The average time per iteration is generally larger than the time of the 1st iteration, that's because of the monotone expansion of the *WriteSet*; more paths are explored in the following iterations.

VI. RELATED WORK

There are many other relaxed memory models that are “weaker” than SC but have slightly “stronger” semantics than JMM. The store-buffer based memory models; *Partial Store Order* (PSO) and *Total Store Order* (TSO) are hardware memory models used in SPARC architecture and allow certain hardware optimizations. PSO maintains a set of “store buffers” (FIFO queues), each associated with a (process, variable) pair, while TSO maintains only one store buffer per process. In PSO, when process p_i writes value v to variable x , it writes v to the corresponding store buffer (p_i, x) ; when p_i reads from x , it first reads from its store buffer (p_i, x) , and if it is empty, reads from the shared memory. The oldest values in the store buffers are *flushed* to the memory at some non-deterministic point. A *fence* operation on variable x performed by process p_i forces the most recent value written by p_i to be written back to the memory. Due to delayed stores caused by non-deterministic flush operations, PSO is more relaxed than SC; a process may not always see the value of the most recent store. However, PSO has more inter-process restrictions than JMM; “A process p_i should not observe values written to shared variable x by process p_j in an order different from the order in which they were written.”[18] This means the read can only see the values that have been previously written by other processes, and the values written by one process should be viewed in order. [18] introduced an approach to model check programs under PSO. [20] used an automata-based approach to verify programs under TSO.

We have adapted JPR to reason about programs under PSO. Since only previously happened writes are visible to the reads, iteration is no longer needed. We only need to run **collectValuesJPF** once and the metadata *ActionSet* and *ImPOSESet* can be discarded. However, we have to enforce a total order over the *WriteSet* with respect to the process. Thus the set of $(actionID, value)$ tuples for a memory location should be replaced by a list of $(processID, value)$ tuples. A simple operational semantic for JPR under PSO (p : process, x : variable, v : value) follows:

- *Store* (p_i, x, v) : Put (p_i, v) to the end of *WriteSet* (x) .
- *Load* (p_i, x) : Get last (p_i, v) from *WriteSet* (x) . If there is no (p_i, v) , get (p_j, v') non-deterministically from *WriteSet* (x) , then delete all the (p_j, v'') tuples in front of it.
- *Fence* (p_i) : Suppose the last store action of p_i before the fence is *Store* (p_i, x, v) , then in *WriteSet* (x) delete all the (p_i, v') tuples in front of the last (p_i, v) .

Other work on verification of programs executing under hardware memory models include a SAT-based bounded verification method to check concurrent data types under relaxed

memory ordering models [6] and a monitor algorithm that could be implemented by model checkers to verify relaxed memory models due to store buffers[7]. Gopalakrishnan, Yang, and Sivaraj [10] used a SAT-based technique to evaluate tests in the context of the Itanium memory model.

In the area of memory models for programming languages, T. Q. Huynh et. al [12] proposed a bytecode level model checker for the relaxed C# memory model. The MemSAT system [24] system has a different goal than verifying programs: it accepts a test program containing assertions and an axiomatic specification of a memory model and then uses an 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 [2] 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 [21] 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 JPF 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.

VII. 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.

REFERENCES

- [1] S. Adve, “Data races are evil with no exceptions: technical perspective,” *Commun. ACM*, vol. 53, pp. 84–84, November 2010. [Online]. Available: <http://doi.acm.org/10.1145/1839676.1839697>
- [2] D. Aspinall and J. Ševčík, “Formalising Java’s data race free guarantee,” in *Proceedings of the 20th international conference on Theorem proving in higher order logics*, ser. TPHOLS’07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 22–37.
- [3] D. Bacon, J. Bloch, J. Bogda, C. Click, P. Haahr, D. Lea, T. May, J. Maessen, J. Manson, J. D. Mitchell, K. Nilsen, B. Pugh, and E. G. Sirer, “The “double-checked locking is broken” declaration.” [Online]. Available: <http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>

- [4] H.-J. Boehm and S. V. Adve, “Foundations of the c++ concurrency memory model,” *SIGPLAN Not.*, vol. 43, no. 6, pp. 68–78, 2008.
- [5] G. Bronevetsky and B. R. de Supinski, “Complete formal specification of the openmp memory model,” *Int. J. Parallel Program.*, vol. 35, no. 4, pp. 335–392, 2007.
- [6] S. Burckhardt, R. Alur, and M. M. K. Martin, “Bounded model checking of concurrent data types on relaxed memory models: A case study,” in *Proceedings of the 18th International Conference on Computer Aided Verification*, 2006.
- [7] S. Burckhardt and M. Musuvathi, “Effective program verification for relaxed memory models,” in *Proceedings of the 20th International Conference on Computer Aided Verification*, 2008.
- [8] A. De, A. Roychoudhury, and D. D’Souza, “Java Memory Model aware software validation,” in *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2008.
- [9] ECMA International, *Standard ECMA-335 - Common Language Infrastructure (CLI)*, 4th ed., June 2006. [Online]. Available: <http://www.ecma-international.org/publications/standards/Ecma-335.htm>
- [10] G. Gopalakrishnan, Y. Yang, and H. Sivaraj, “Qb or not qb: An efficient execution verification tool for memory orderings,” in *Proceedings of the 16th International Conference on Computer Aided Verification*, 2004.
- [11] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java Language Specification*, 3rd ed. Addison Wesley, 2005.
- [12] T. Q. Huynh and A. Roychoudhury, “Memory model sensitive bytecode verification,” *Formal Methods in System Design*, vol. 31, no. 3, December 2007.
- [13] H. Jin, T. Yavuz-Kahveci, and B. A. Sanders, “Java memory model-aware model checking,” Department of Computer and Information Science, University of Florida, Tech. Rep. REP-2011-516, 2011. [Online]. Available: <http://www.cise.ufl.edu/tr/REP-2011-516/>
- [14] “JMM causality test cases.” [Online]. Available: <http://www.cs.umd.edu/~pugh/java/memoryModel/unifiedProposal/testcases.html>
- [15] “Java Racefinder.” [Online]. Available: <http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-racefinder>
- [16] K. Kim, T. Yavuz-Kahveci, and B. A. Sanders, “Precise data race detection in a relaxed memory model using model checking,” University of Florida, Tech. Rep. REP-2009-480, 2009.
- [17] —, “JRF-E: Using model checking to give advice on eliminating memory model-related bugs,” in *Proceedings of the 25th ACM/IEEE Conference on Automated Software Engineering*, 2010.
- [18] M. Kuperstein, M. Vechev, and E. Yahav, “Partial-coherence abstractions for relaxed memory models,” in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI ’11. New York, NY, USA: ACM, 2011, pp. 187–198. [Online]. Available: <http://doi.acm.org/10.1145/1993498.1993521>
- [19] L. Lamport, “How to make a multiprocessor computer that correctly executes multiprocess programs,” *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, September 1979.
- [20] A. Linden and P. Wolper, “An automata-based symbolic approach for verifying programs on relaxed memory models,” in *Model Checking Software*, ser. Lecture Notes in Computer Science, J. van de Pol and M. Weber, Eds. Springer Berlin / Heidelberg, 2010, vol. 6349, pp. 212–226.
- [21] J. Manson and W. Pugh, “The Java Memory Model simulator,” in *Workshop on Formal Techniques for Java-like Programs, in association with ECOOP*, 2002.
- [22] J. Manson, W. Pugh, and S. V. Adve, “The Java memory model,” in *POPL ’05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM Press, 2005, pp. 378–391.
- [23] “Oracle thread analyzer’s user guide.” [Online]. Available: http://download.oracle.com/docs/cd/E18659_01/html/821-2124/gecqt.html
- [24] E. Torlak, M. Vaziri, and J. Dolby, “MemSAT: checking axiomatic specifications of memory models,” in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI ’10. New York, NY, USA: ACM, 2010, pp. 341–350.
- [25] J. Ševčík and D. Aspinall, “On validity of program transformations in the java memory model,” in *Proceedings of the 22nd European conference on Object-Oriented Programming*, ser. ECOOP ’08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 27–51. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-70592-5_3