Autom Softw Eng (2012) 19:491–530
DOI 10.1007/s10515-012-0109-4

JRF-E: using model checking to give advice on eliminating memory model-related bugs

Kyung Hee Kim · Tuba Yavuz-Kahveci · Beverly A. Sanders

Received: 16 August 2011 / Accepted: 15 May 2012 / Published online: 6 June 2012

Abstract According to modern relaxed memory models, programs that contain data races need not be sequentially consistent. Executions that are not sequentially consistent may exhibit surprising behavior such as operations on a thread occurring in a different order than indicated by the source code, or different threads having inconsistent views of updates of shared variables. Java Racefinder (JRF) is an extension of Java Pathfinder (JPF), a model checker for Java bytecode. JRF precisely detects data races as defined by the Java memory model and can thus be used to verify sequential consistency. We describe an extension to JRF, JRF-Eliminator (JRF-E), that analyzes information collected during model checking, specifically counterexample traces and acquiring histories, and provides advice to the programmer on how to eliminate detected data races from a program. Once data races have been eliminated, standard model checking and other verification techniques that implicitly assume sequential consistency can be soundly employed to verify additional properties.

Keywords Data race · Relaxed memory model · Counterexample

1 Introduction

Virtually all approaches for reasoning about the behavior of concurrent programs, both the informal reasoning practiced by programmers writing a concurrent program and formal methods and tools such as model checkers, start with an assumption of

K.H. Kim (✉) · T. Yavuz-Kahveci · B.A. Sanders
University of Florida, Gainesville, USA
e-mail: khkim@cise.ufl.edu

T. Yavuz-Kahveci
e-mail: tyavuz@cise.ufl.edu

B.A. Sanders
e-mail: sanders@cise.ufl.edu

sequential consistency (SC) (Lamport 1979). Under sequential consistency, a concurrent program behaves as if all of its atomic actions occur in some global order that is consistent with the program order on each thread and thus that all threads have a consistent view of the way the memory has been updated. In modern systems, however, optimizations by compilers and hardware that significantly speed up programs without affecting their sequential semantics, may allow sequentially inconsistent behavior.

As an example, consider the following program fragment.

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

The (non-volatile) variable `done` is initially false and not accessed by `computation()`, which updates other variables. Since the two statement are independent, the order could be reversed[1] without changing the sequential semantics. However, if this fragment occurs in a concurrent program and `done` is intended to be a signal to other threads that `computation()` is finished, then another thread finding `done` true may observe a state that reflects an incomplete execution of `computation()`. This scenario is legal according to Java semantics, but violates SC and the programmer's expectations.

Exactly how threads interact with memory and how the programmer can control this is defined by a *memory model*. Originally, memory models were defined for architectures, and program behavior with respect to the memory model is constrained using low level operations such as fences and memory barriers. More recently memory models have become part of a programming language's semantics. The Java memory model (JMM) (Manson et al. 2005; Gosling et al. 2005) is an important example. A situation that can lead to non-SC behavior is called a *data race*. Java semantics guarantee that a program that is free from data races will behave as if it is SC, thus corresponding to programmer intuition and the implicit assumptions of model checkers and other tools.

Java Racefinder (JRF) (Kim et al. 2009a, 2009b) is an extension to the Java Pathfinder (JPF) model checking tool (Visser et al. 2003; Java Pathfinder 2012) that detects data races, as defined by the JMM, precisely. Note that the meaning of the term "data race" depends on the context. In much of the literature on data race detection, a data race is a situation where conflicting accesses[2] to shared variables are not ordered by synchronization. Sequential consistency is implicitly assumed and data races are of interest because they often, but by no means always, indicate some sort of concurrency related bug, for example, a forgotten lock that might lead to an atomicity violation. In contrast, we are interested in the specific notion of a data race given in the Java Memory model where the notion of "ordered by synchronization" is formally defined and incorporates a richer set of synchronization constructs than locking.

[1] Among other possibilities, write buffers or values updated by `computation()` held in registers instead of writing to main memory could cause this effect

[2] Two operations by different threads conflict if they access the same memory location and at least one is a write.

Once data races have been detected and eliminated from a program, it will be sequentially consistent and standard model checking techniques and other verification methods that assume sequential consistency can be soundly employed to reason about the program. However, understanding a data race is a tedious task particularly when multiple races are involved. In this paper, we describe a further extension, JRF-Eliminator (JRF-E), that can explain detected races and provide advice to the programmer about how to modify the source code to eliminate them. JRF-E analyses use information collected during model checking, specifically counterexample traces and acquiring histories. Experiments indicate that JRF-E is a practical tool.

2 Background

In this section we give a brief and informal description of the JMM (Gosling et al. 2005, Chap. 17) and (Manson et al. 2005; Aspinall and Sevcik 2007). We also briefly describe Java Racefinder (JRF). A more complete treatment can be found in Kim et al. (2009b).

2.1 The Java memory model

An execution of a Java program is a set of memory model related actions (read and write, volatile read and write,[3] lock and unlock a monitor lock, start a thread, detect termination of a thread, etc.) along with an order, $\xrightarrow{po}$, which totally orders the actions on each thread, and a synchronization order, $\xrightarrow{so}$, that totally orders the synchronization actions. Additionally, we have the value written function, V, that assigns a value to each write, and the write-seen function W that assigns a write action to each read so that the value obtained by a read action r is $V(W(r))$. The synchronization order, $\xrightarrow{so}$, on an execution induces a partial order on synchronization actions called the synchronizes-with order, $\xrightarrow{sw}$, according to the following rules:

- An unlock action on a monitor lock m synchronizes-with all subsequent lock actions on m by any thread.
- A write to a volatile variable v synchronizes-with all subsequent reads of v.
- The action of starting a thread synchronizes-with the first action of the newly started thread.
- The final action in a thread synchronizes-with an action in any other thread (e.g. join, or invoking the isAlive() method) that detects the thread's termination.
- The writing of default values of every object field synchronizes-with the first access of the field.

In the descriptions above, "subsequent" is determined by the synchronization order. Finally, the *happens-before* order, $\xrightarrow{hb}$, is a transitive, partial order on the actions in an execution obtained by taking the transitive closure of the union of $\xrightarrow{sw}$ and $\xrightarrow{po}$.

[3] Since Java 1.5, the volatile keyword prevents reordering of memory accesses across accesses to the volatile variable. This is usually implemented using a memory barrier.

Well-formed executions satisfy some unsurprising requirements such as type correctness, correct behavior of locks, and consistency with the sequential semantics of the program. In addition, a well-formed execution satisfies *happens-before consistency* which requires that a read r of variable v is allowed to see the results of a write $w = W(r)$ provided that r is not ordered before w, i.e. $\neg(r \xrightarrow{hb} w)$, and there is no intervening write w' to v, i.e. $\neg\exists w' : w \xrightarrow{hb} w' \xrightarrow{hb} r$. Well-formedness still allows non-determinism since actions, unless they are synchronization actions, on different threads need not be ordered. Also, it is not required that the write-seen function, W returns the "most recent" write to the variable in question, nor is it required that W is "consistent" for reads on different threads, thus allowing sequentially inconsistent behavior. *Legal executions* are well-formed executions that satisfy additional causality constraints intended to provide certain safety guarantees (no "out-of-thin-air" values) for programs, even when not sequentially consistent. Since our goal is to eliminate data races, the causality conditions are not relevant.

In the JMM, two operations *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 $\xrightarrow{hb}$. A *sequentially consistent* (SC) execution is one where there is a total order, $\xrightarrow{sc}$, on the actions consistent with $\xrightarrow{po}$ and $\xrightarrow{so}$ and where a read r of variable v sees the results of the most recent preceding write w such that $w \xrightarrow{sc} r$ and there is no intervening write w' to v, i.e. $\neg\exists w' : w \xrightarrow{sc} w' \xrightarrow{sc} r$. A Java program is *correctly synchronized* if all sequentially consistent executions are data race free. It has been shown (Manson et al. 2005; Aspinall and Sevcik 2007) that any legal execution of a well-formed correctly synchronized program is sequentially consistent.[4] This result justifies using a model checker to detect data races.

2.2 Summarizing $\xrightarrow{hb}$ with h

In this section, we describe the function, h, which is used by JRF to detect data races. At each point in an sequentially consistent execution, h summarizes $\xrightarrow{hb}$, allowing data races to be detected as they occur. Let *Addr* be the set of memory locations representing non-volatile variables in the program, *SynchAddr* be the set of memory locations representing variables with volatile semantics and locks, and *Threads* be the set of threads. Then $h : \mathit{SynchAddr} \cup \mathit{Threads} \to 2^{\mathit{Addr}}$ maps threads and synchronization variables to sets of non-volatile variables so that $x \in h(t)$ means that thread t can read or write variable x without causing a data race.

For a finite sequentially consistent execution E of program P, a set of static non-volatile variables $\mathit{static}(P)$, let E_n be the prefix of E of length n, i.e. the sequence of actions $a_0, a_1, \ldots, a_{n-1}$, and h_n be the value of h after performing all of the actions in E_n. We assume that thread *main* is the single thread that initiates the program. Initially, $h_0 = \lambda z.\text{if } z = \mathit{main} \text{ then } \mathit{static}(P) \text{ else } \bot$.

The way that h_{n+1} is obtained from h_n depends on the action a_n. First, we define four auxiliary functions *release*, *acquire*, *invalidate*, and *new*. The function

[4] JRF actually uses a slightly weaker but still sound notion of correct synchronization.

release(t, x) takes h and yields a new summary function by updating $h(x)$ to include the value of $h(t)$. It is used with actions by thread t that correspond to the source of a $\xrightarrow{sw}$ edge.

$$release(t, x)h \mathrel{\widehat{=}} h\big[x \mapsto h(t) \cup h(x)\big] \tag{1}$$

The function *acquire*(t, x) takes h and yields a new function by updating $h(t)$ to include the value of $h(x)$. It is used in actions that form the destination of a $\xrightarrow{sw}$ edge.

$$acquire(t, x)h \mathrel{\widehat{=}} h\big[t \mapsto h(t) \cup h(x)\big] \tag{2}$$

The function *invalidate* yields a new function by removing x from $h(z)$ for all $z \neq t$. It is used in actions where thread t writes non-volatile x.

$$invalidate(t, x)h \mathrel{\widehat{=}} \lambda z.\text{if } (t = z) \quad \text{then } h(z) \text{ else } h(z)\backslash\{x\}$$

The function *new* yields a new summary function by adding the set *fields* to the value of $h(t)$ and initializing the previously undefined values of h for the new volatile variables.

$$\begin{aligned} &new(t, fields, volatiles)h \\ &\mathrel{\widehat{=}}\lambda z. \quad \text{if } (t = z) \text{ then } h(t) \cup fields \\ &\qquad\quad \text{else if } (z \in volatiles) \text{ then } \phi \text{ else } h(z) \end{aligned} \tag{3}$$

The definition of h_{n+1}, which depends on h_n and action a_n, is given in Fig. 1.

To extend the model checker, we maintain h, and check that

$$norace(x, t) = x \in h(t) \tag{4}$$

holds before reading or writing non-volatile x by thread t. When this condition holds for all non-volatile reads and writes in an execution, the execution is *h-legal*. We have shown elsewhere (Kim et al. 2009b) that if all SC executions of a well-formed program are h-legal all of its legal executions are SC.

a_n by thread t	h_{n+1}
write a volatile field v	*release*(t, v) h_n
read a volatile field v	*acquire*(t, v) h_n
lock the lock variable lck	*acquire*(t, lck) h_n
unlock the lock variable lck	*release*(t, lck) h_n
start thread t'	*release*(t, t') h_n
join thread t'	*acquire*(t, t') h_n
t'.isAlive()	if (t'.*isAlive*()) then (*acquire*(t, t') h_n) else h_n
write a non-volatile field x	*invalidate*(t, x) h_n
read a non-volatile field x	h_n
instantiate an object	*new*$(t, fields, volatiles)$ h_n

Fig. 1 Definition of h_{n+1}

2.3 Java RaceFinder

Java PathFinder (JPF) model checks Java byte code by reading Java class files and simulating their execution using its own virtual machine with on-the-fly verification of specified properties. A property violation is reported by JPF along with a counterexample, the execution path that led to the violation.

JPF provides a listener interface which was used to extend its functionality for JRF. The interface provides a set of callback functions allowing low level operations such as object creation, object locking and unlocking, the start of a new thread, and each execution of an instruction to be intercepted and augmented with user-supplied code. JRF maintains a representation of the summary function h described in Sect. 2.2; the listener code intercepts relevant instructions and updates the representation as described in Fig. 1. In addition, the norace property, which was defined in Eq. (4), is checked prior to all non-volatile reads and writes. Because everything, including threads and locks are objects in Java, threads, locks, and variables are handled uniformly as "memory locations".

JRF employs a variety of additional optimizations and features that enhance usability. These include:

- **Search heuristics**: Specialized heuristics can be employed to search paths that are more likely to contain a data race earlier. For example, the writes-first(WF) heuristic prioritizes write operations while the watch-written(WW) heuristic prioritizes operations on a memory location that has recently been written by a different thread. The avoid release/acquire(ARA) heuristic prioritizes operations on threads that do not have a recent acquire operation preceded by a matching release on the execution path. The acquire-first(AF) heuristic prioritizes acquire operations that do not have a matching release along the execution path. This situation often corresponds to unsafe publication[5] of an otherwise correctly synchronized object.
- **Search space pruning**: Java Pathfinder (JPF) prunes the search space by giving each state a state number and stopping the search along path when a state number that has already been seen has been encountered. This is unsound in JRF since states with the same state number with different histories may have different values of h. Nevertheless, under certain conditions, tested by JRF the search space can be safely pruned.
- **Untracked variables**: JRF offers the option to mark individual non-volatile locations as untracked, so that data races involving these locations are not detected, but data races involving tracked variables are still found. An entire class can be marked as trusted which will result in all of its private non-volatiles becoming untracked. A package can be marked trusted, which will result in all package-private non-volatile variables to become untracked. There are two different motivations for not tracking variables. The first allows JRF to ignore certain data races that are considered to be benign.[6] The second is to improve the scalability of JRF. Marking

[5] An object is published when its reference is made visible to other threads. Unsafe publication is a common error that can allow a partially initialized object to be seen by other threads.

[6] Occasionally data races will be deliberately allowed for performance reasons, and because the JMM constrains the values that can be seen through data races enough to avoid type errors and out-of-thin-

classes or packages as trusted is a convenient way to reduce the time and memory requirements by not checking their internal variables. JRF does, however, continue to track happens-before edges involving volatile fields and locks defined in trusted classes and the non-volatile fields accessible outside of trusted classes (or packages) in order to continue to detect data races in or caused by other classes. For example, a common way of safely publishing objects is to make them available to other threads by passing them through a data structure such as a queue whose implementation is provided in the `java.util.concurrent` package where the insertion of an object happens-before removal of the object. If this package is marked as trusted, the happens-before edges related to inserting and removing objects will be preserved, but no checking will be done on the internal non-volatile variables in the class unless they are visible outside the trusted code. By marking the classes in the standard Java release as trusted, a significant reduction in the time and space requirements to model check an application class can be achieved.

– **Lazy representation of array elements**: In the representation of h, every memory location requires an entry in the h table. This also applies to arrays, which require an entry for each array element. However, JRF uses a single h entry to abstractly represent the entire array until such time that an individual element is updated and obtains a different value for h. At that point, an additional entry in the h table for that element is allocated. The abstract location is still used for the remaining elements. The lazy representation of array elements saves acquire and release time as well as space. This is especially important in programs that manipulate String objects, which hold characters internally in an array.
– **Thread local optimization**: If we have knowledge of which memory locations are not shared, this can be used to reduce the overhead. Since a data race on a non-volatile variable by definition involves two threads, a non-static variable that is only accessed by the thread that instantiated it can never be involved in a data race. Static variables are accessed by a class loader thread during static initialization. They are then accessed by at most one application thread, they can also never be involved in a data race because the JMM guarantees that static initialization is guaranteed to happen-before the first access of a variable by any application thread. Applying the thread local optimization requires somehow determining which variables are shared. We typically do this by running a slightly modified version of Java Pathfinder that collects sharing information.

Additional information about the implementation, including the data structures used for efficient implementation of h and proofs of soundness of the optimizations listed above can be found in Kim et al. (2009a, 2009b). JRF currently can handle all Java language features related to the JMM except for finalizers. In particular, JRF correctly handles the memory-model related semantics of the classes in the `java.util.concurrent.atomic` package. This package contains classes that support atomic operations on variables such as atomic increment and compare and swap. These classes are heavily used in lock-free algorithms (see Herlihy and

air values, this is feasible in principle. A well-known example is a racy lazy initialization of the hash code value in the java.lang.String class. Generally, however, reasoning about programs with races is quite difficult and should be considered to be a job for experts only.

Shavit 2008 for a thorough treatment of lock-free algorithms). The ability of JRF to precisely deal with races in this class of concurrent programs sets it apart from the vast majority of tools that detect data races.[7]

3 Counterexample analysis

JRF inherits JPF's ability to provide the sequence of statements (the counterexample path) that leads to a data race. This is extremely valuable information, but it is a tedious job to parse the JRF output to determine the interleaving sequence of the threads and the reason why the data race has occurred. JRF-E adds an analysis phase that analyzes the counterexample path and some additional information gathered during model checking and provides the programmer with a concise diagnosis of the problem and suggestions for source code modifications to eliminate a race.

Since a data race is defined to be the lack of a happens-before edge, JRF-E can leverage the information in h to identify the statement containing the write involved in a data race, which we call the *source statement*, and the *manifest statement*, the read or write where the data race occurred, i.e. where the *norace* condition failed. This information is then used to provide suggestions for ways to eliminate the data race by creating a happens-before relationship between those statements.

In addition to maintaining the h function, JRF-E also maintains the *acquiring history*. The idea is to store the synchronization operation that enabled a thread t to access a memory location m in a data-race free way. If accessing m by another thread t' results in a data-race, JRF-E suggests that t' perform the same synchronization operation that thread t has used before.

In any run of JRF-E multiple races will be detected. Some of these may be manifestations of the same race (i.e. the race source and race manifest statements are the same) occurring on different paths. The analysis can be configured to stop when *threshold* number of data races have been detected (or the whole state space is explored). For each unique race, JRF-E generates a list of suggestions that will eliminate the data race on the corresponding execution paths. A higher threshold may allow better suggestions to be provided at the cost of longer execution time.

A simple example will illustrate the practical benefits provided by JRF-E. A Java implementation of the program shown in Fig. 2 was analyzed using JRF. Both `x` and `done` are involved in races. Part of the output, including the counterexample path for a data race involving `x` is shown in Fig. 3. The omitted output gives similar results for additional (8 more) detected races.

Fig. 2 Thread 1 notifies Thread 2 that `x` is set through flag `done`

```
int x;
boolean done=false;

   Thread 1          Thread 2
=============   ================
 s1: x=1;          t1: while (!done);
 s2: done=true;    t2: assert (x==1);
```

[7] Lock-based algorithms including the Race Detector tool in JPF cannot handle these lock-free algorithms and will report false-positives since they approximate the happens-before orders only for locks.

```
JRF results
===================================================== data race #1
jrf.hbset.util.HBDataRaceException
      at THREAD      (simple.SimpleRace$Thread2@301 from "Thread t1 = new Thread2();"
                  at simple/SimpleRace.java:55 in (main))
      to MEMORY      (simple.SimpleRace.x from INITIALIZER)
      in INSTRUCTION (getstatic)
      of SOURCE      ("assert (x==1);" at simple/SimpleRace.java:74)
_____________________________________________________ trace #1
------------------------------------------------------ transition #0 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main}
      [2688 insn w/o sources]
  simple/SimpleRace.java:45      : static       boolean done=false;
  simple/SimpleRace.java:43      : public class SimpleRace {
      [1 insn w/o sources]
  simple/SimpleRace.java:54      : Thread t0 = new Thread1();
  simple/SimpleRace.java:60      : static class Thread1 extends Thread
      [181 insn w/o sources]
  simple/SimpleRace.java:60      : static class Thread1 extends Thread
  simple/SimpleRace.java:54      : Thread t0 = new Thread1();
  simple/SimpleRace.java:55      : Thread t1 = new Thread2();
      [1 insn w/o sources]
  simple/SimpleRace.java:69      : static class Thread2 extends Thread
      [1 insn w/o sources]
  simple/SimpleRace.java:55      : Thread t1 = new Thread2();
  simple/SimpleRace.java:69      : static class Thread2 extends Thread
      [134 insn w/o sources]
  simple/SimpleRace.java:69      : static class Thread2 extends Thread
  simple/SimpleRace.java:55      : Thread t1 = new Thread2();
  simple/SimpleRace.java:56      : t0.start();
------------------------------------------------------ transition #1 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {main,>Thread-0}
  simple/SimpleRace.java:56      : t0.start();
  simple/SimpleRace.java:57      : t1.start();
------------------------------------------------------ transition #2 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {main,Thread-0,>Thread-1}
  simple/SimpleRace.java:57      : t1.start();
  simple/SimpleRace.java:58      : }
------------------------------------------------------ transition #3 thread: 1
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {Thread-0,>Thread-1}
  simple/SimpleRace.java:64      : x = 1;
------------------------------------------------------ transition #4 thread: 1
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {Thread-0,>Thread-1}
  simple/SimpleRace.java:64      : x = 1;
  simple/SimpleRace.java:65      : done = true;
------------------------------------------------------ transition #5 thread: 1
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {Thread-0,>Thread-1}
  simple/SimpleRace.java:65      : done = true;
  simple/SimpleRace.java:66      : }
------------------------------------------------------ transition #6 thread: 2
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>Thread-1}
  simple/SimpleRace.java:73      : while(!done) { /*spin*/ }
  simple/SimpleRace.java:74      : assert (x==1);
_____________________________________________________ snapshot #1
thread index=2,name=Thread-1,status=RUNNING,this=simple.SimpleRace$Thread2@301,
                          priority=5,lockCount=0,suspendCount=0
  call stack:
     at simple.SimpleRace$Thread2.run(SimpleRace.java:74)
_____________________________________________________ trace #2
 . . . (similar output for 8 additional traces omitted)
```

Fig. 3 Partial output from JRF for the program in Fig. 2. Eight similar traces have been omitted

To understand what caused the detected race, we need to decode the counterexample path given as `"trace #1"`. Clearly, this is a tedious exercise, even for this simple program where the length of the counterexample path is only six. The path length may be several hundreds in realistic examples. In contrast, Fig. 4 shows the output of the analysis produced by JRF-E. For each unique race found, the race source statement, the race manifest statement, and suggestions for code modifications that will eliminate that race are given. Note that the tool recognized that marking `done` as volatile is sufficient to also eliminate the race on `x`.

```
JRF-E RESULT
================================================== data race #1
jrf.hbset.util.HBDataRaceException . . .
__________________________________________________ analyze counter example
data race source statement : "putstatic" at simple/SimpleRace.java:64  : "x = 1;"
data race manifest statement : "getstatic" at simple/SimpleRace.java:74: "assert (x==1);"

Change the field "simple.SimpleRace.x from INITIALIZER" to volatile.
Change the field "simple.SimpleRace.done from INITIALIZER" to volatile.

__________________________________________________ advice from acquiring history
NONE
================================================== data race #2
jrf.hbset.util.HBDataRaceException . . .
__________________________________________________ analyze counter example
data race source statement : "putstatic" at simple/SimpleRace.java:65  : "done = true;"
data race manifest statement : "getstatic" at simple/SimpleRace.java:73: "while(!done) { /*spin*/ }"

Change the field "simple.SimpleRace.done from INITIALIZER" to volatile.

__________________________________________________ advice from acquiring history
NONE
__________________________________________________ frequency of advice
[1times] Change the field "simple.SimpleRace.x from INITIALIZER" to volatile.
[2times] Change the field "simple.SimpleRace.done from INITIALIZER" to volatile.
__________________________________________________ statistic
JRF takes 0:0:1 to find 2 equivalent races with 9 counterexample traces.
JRF-E takes 0:0:0 in 9 races analysis.
```

Fig. 4 JRF-E output which explains the source of the race and suggests how to eliminate it

3.1 Suggestion generation

When JRF detects a data race, it passes the following information to JRF-E: the counterexample path (*pathInstr*), the *h* information for the counterexample path (*pathHB*), the acquire history (*AcquireHis*), the position of the manifest statement (*raceManifestIndex*), and the position of the source statement (*raceSourceIndex*) on the counterexample path. The remainder of this section describes the algorithms used in generating the suggestions.

3.1.1 Change a (non-array element) variable to volatile or implement with an atomic class

Due to the semantics of volatile variables in Java, changing a variable involved in a race to volatile is always sufficient to eliminate a data race involving that variable. Since volatile variables inhibit compiler optimizations and accesses to volatiles incur runtime overhead, the trivial way of eliminating races by making everything volatile is undesirable.

Changing a variable to volatile likely to be the most appropriate in situations where this variable is being used for publication (i.e. making the reference to a new object instance visible to other threads). Unsafe publication (Goetz et al. 2006) is a common error in concurrent Java programs written by programmers without a good understanding of the JMM and can lead to a situation where another thread sees a partially initialized object.

Figure 5 illustrates the well-known double-checked-locking antipattern and its fix using volatile to publish shared reference `helper`. The scenario given in Fig. 6 is the case where double-checked-locking is broken due to the unsafe publication of `helper`. Figure 7 illustrates the search path with a race and suggestions to fix it.

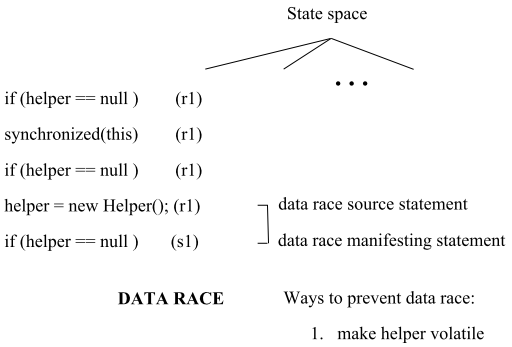
Fig. 5 Double-checked-locking antipattern with races on `helper` and `helper.data`

```
Helper helper=null;  /* adding volatile will fix the problem */
Helper getHelper() {                          class Helper {
   if (helper == null) {                           int data;
      synchronized(this) {                         public Helper()
         if (helper == null)                        { data = 2011; }
            helper = new Helper();          }
      }
   }
   return helper;
}
```

Fig. 6 Non-SC execution scenario where double-checked-locking is broken for the program in Fig. 5. Thread 2 failed to print correctly initialized value 2011

```
            Thread 1                                  Thread 2
=========================                  ======================
r1: print (getHelper().data);              s1: print (getHelper().data);
//  if (helper == null) {
//      synchronized(this) {
//        if (helper == null)
//          helper = new Helper();
                                             //  if (helper == null) {
                                             //  return helper;
                                             //  print (getHelper().data);
//                          data=2011;
//    }
// }
```

Fig. 7 Part of the state space for the program in Fig. 5 with a data race



Another guaranteed solution is to replace the variable with a final[8] reference to an instance of the atomic class corresponding to the variable's type in the `java.util.concurrent.atomic` package. For example, replace an `int` variable with an instance of the `AtomicInteger` class in `java.util.concurrent.atomic` package. These classes are less convenient than volatiles because they must be accessed with `get` and `set` methods and are the better choice only if the lock-free atomic update methods they provide are needed. These atomic update methods include `compareAndSet`, which is frequently used in lock-free

[8]Final fields must be set in the constructor, cannot be modified, and have special semantics in the JMM. Note that the value encapsulated in the atomic object can change, just not the object itself.

```
volatile int turn;
int shared;
volatile boolean[] flag = {false, false};

        Thread 1                                  Thread 2
==============================  ==============================
 r1: flag[0] = true;             s1: flag[1] = true;
 r2: turn = 1;                   s2: turn = 0;
 r3: while (flag[1] && turn==1) {/*spin*/}   s3: while (flag[0] && turn==0) {/*spin*/}
 r4: shared++;                   s4: shared++;
 r5: flag[0] = false;            s5: flag[1] = false;
```

Fig. 8 This implementation does not guarantee mutually exclusive access to `shared` since it has a race on `flag[0]` and `flag[1]`

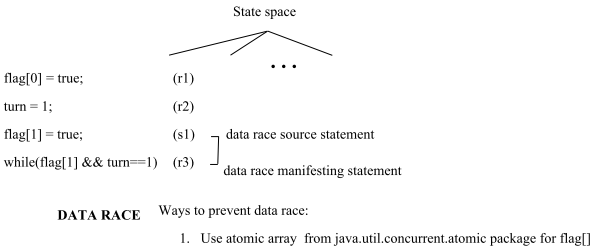


Fig. 9 Part of the state space with a race on `flag[1]`

Algorithm *makeChangeToVolatileSuggestions* (*raceManifestIndex*, *raceSourceIndex*)
`integer` *raceManifestIndex*, *raceSourceIndex*

let *v* `denote the variable accessed by instructions at` *raceManifestIndex* `and` *raceSourceIndex*
if *v* is an array element **then**
`print "Use atomic array ..."`
else
`print "Make` *v* `volatile"`

Fig. 10 Suggest change to volatile or atomic array

algorithms, and where appropriate for the type, methods such as `getAndAdd`, `addAndGet`, `getAndIncrement`, etc.

3.1.2 Change an array to an atomic array

Atomic arrays for various element types are provided in the `java.util.concurrent.atomic` package. They provide volatile semantics for array elements and thus this change is always sufficient to eliminate data races involving array elements. Arrays are objects in Java and a frequent error is to mark an array reference volatile without realizing that this does not provide volatile semantics for the accesses to the elements. Figure 8 is the example implementation of Peterson algorithm with a data race on non-volatile elements of volatile array and Fig. 9 illustrates one counterexample path with a race on `flag[1]`. Figure 10 implements these two suggestions.

3.1.3 Move source statement

Data races can sometimes be avoided by placing the source statement before a statement, $s1$, that is the source of a happens-before edge. Assume that the happens-before

```
Algorithm findHBEdges(pathInstr): Set of integer pairs
    Stack pathInstr
    Set of integer pairs HBEdges ←∅

    for indexDest from size(pathInstr) to 1 do
        if pathInstr(indexDest) is an acquire then
            for indexSource from indexDest − 1 to 1 do
                if pathInstr(indexSource) is a release matching
                    pathInstr(indexDest) then
                    HBEdges ←HBEdges ∪ (indexSource, indexDest)
                    break
    return HBEdges
```

Fig. 11 Find the set of happens-before edges through synchronization actions on path *pathInstr*

```
Algorithm makeMoveSourceInstructionSuggestions (pathInstr, raceManifestIndex, raceSourceIndex)
    Stack pathInstr
    integer raceManifestIndex, raceSourceIndex

    Set of integer pairs HBEdges ←findHBEdges(pathInstr)
    foreach pair p =(i1,i2) ∈ HBEdges do
        if i1 < raceSourceIndex
            AND
            (raceSourceIndex,raceManifestIndex) intersects p
            AND
            same thread executed pathInstr(i1) and pathInstr(raceSourceIndex)
            AND
            same thread executed pathInstr(i2) and pathInstr(raceManifestIndex) -
            then
                print "Move instruction at raceSourceIndex before i1"
```

Fig. 12 Suggest moving instruction

edge is between $s1$ and statement $s2$. As long as the source statement and $s1$ are executed by the same thread and hence ordered by happens-before due to the program order and similarly $s2$ and the manifest statement are ordered by happens-before, the move creates a happens-before edge between the source and the manifest statements due to transitivity of the happens-before relation.

The algorithm in Fig. 12 first calls the *findHBEdges* algorithm in Fig. 11 to compute all the happens-before edges that result from synchronization actions on the counterexample path.

A happens-before edge is a pair of instructions where the release instruction is the source vertex and the matching acquire instruction is the destination vertex. Instructions are identified by their positions on the counterexample path. After having all pairs representing the happens-before edges on the counterexample path, Fig. 12 compares the (source statement, manifest statement) pair, (*raceSourceIndex*, *raceManifestIndex*), with all other pairs from the set of happens-before edges. We suggest moving the source statement before an existing happens-before edge to get a transitive happens-before ordering through program orders and existing happens-before order. The candidate happens-before order should be between the thread executing source statements and the thread executing manifest statement.

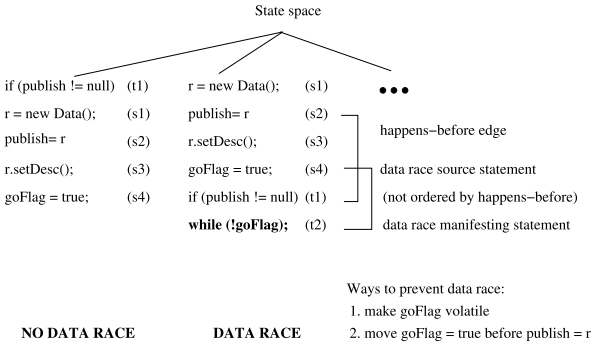
As an example, consider the program in Fig. 13 where `goFlag` and `publish` are shared variables. Since `publish` is a reference, the object to which it refers can also

Fig. 13 Via `goFlag` Thread 1 notifies Thread 2 when object `publish` is ready to be used

```
boolean goFlag;
volatile Data publish;

        Thread 1                          Thread 2
==================          ========================
s1: r = new Data();          t1: if (publish != null) {
s2: publish = r;             t2:   while (!goFlag);
s3: r.setDesc("e");          t3:   String s =  publish.getDesc();
s4: goFlag = true;           t4:   assert(s.equals("e");
                                  }
```

Fig. 14 Part of the state space of the showing a data race free path and a path with a data race



be accessed by both threads.[9] Thread 1 creates an object at line `s1` which is currently accessible only by itself. Then it publishes the object by storing the reference in a shared variable `publish` at line `s2`. The state of the object is updated at line `s3` and shared variable `goFlag` is set to true at line `s4` declaring that object descriptor has been set and can safely be read by other threads. Thread 2 checks whether `publish` is not null at line `t1` and if so, spins until the global flag becomes true at line `t2` and reads the object descriptor in line `t3`.

When the code is analyzed using standard JPF, no assertion violation is reported. However, an assertion failure at line `t4` is legal according to the Java semantics. The program contains a data race between writing the object's descriptor by Thread 1 and reading it by Thread 2, thus SC semantics is not ensured and it would be legal for lines `s3` and `s4` to be reordered. JRF correctly reports a data-race for this program.

Figure 14 shows part of the state space of the example. As long as JRF does not run out of memory or the user out of patience, it can explore all possible paths in the state space. The first path does not exhibit a data race. When JRF executes `t2` on the 2nd path that is explored, a data race is manifested (e.g., *assert norace* fails), the currently explored path is reported to the user as a counterexample, and JRF terminates. One way of eliminating this particular data race is making the global flag, `goFlag`, volatile, thus creating a happens-before edge between `s4` and `t2`. Another way is to move `s4` before `s2` and create a happens-before edge between `s4` and `t2`, which follows from the transitive property of the happens-before relationship: $\texttt{s4} \xrightarrow{hb} \texttt{s2}$, $\texttt{s2} \xrightarrow{hb} \texttt{t1}$, and $\texttt{t1} \xrightarrow{hb} \texttt{t2}$ implies $\texttt{s4} \xrightarrow{hb} \texttt{t2}$. Once this change is made, a

[9] Since JPF is working at the bytecode level, accessing a field potentially involves two bytecode instructions—one to get a reference to the object and one to access the field.

```
Algorithm makePutInSynchronizedBlockSuggestions (pathInstr, raceSourceIndex, raceManifestIndex)
  Stack pathInstr
  integer raceSourceIndex, raceManifestIndex
  Set of InstructionLocations syncLoc1 ←∅, syncLoc2 ←∅

  for index from raceManifestIndex − 1 to raceSourceIndex + 1 do
    if (pathInstr(index) is a MONITOREXIT instruction OR
     RETURN instruction of a synchronized method)
     AND
     same thread executed pathInstr(index) and pathInstr(raceSourceIndex) then
          let loc denote the source line for pathInstr(index)
          syncLoc1 ←syncLoc1 ∪ {loc}
    if (pathInstr(index) is a MONITORENTER instruction OR
     INVOKE instruction of a synchronized method)
     AND
     same thread executed pathInstr(index) and pathInstr(raceManifestIndex) then
          let loc denote the source line for pathInstr(index)
          syncLoc2 ←syncLoc2 ∪ {loc}

  foreach source line loc ∈ syncLoc1 do
    print "Put instruction  pathInstr(raceManifestIndex)
          in synchronized block as in line loc "
  foreach source line loc ∈ syncLoc2 do
    print "Put instruction  pathInstr(raceSourceIndex)
          in synchronized block as in line loc "
```

Fig. 15 Suggest a synchronized block

Fig. 16 Thread 2 need to synchronize on `lock` to access `data`

```
int data;
final Object lock=new Object();

          Thread 1                          Thread 2
==========================  ==============
s1: synchronized (lock) { /*lock*/    t1: print (data);
s2: data = 1;
s3: } /*unlock*/
```

new data race between the write at `s3` and the read at `t3` is exhibited. This can be eliminated by moving `s3` before `s2`. At this point the example is both correct (the assertion will never fail) and contains no data races.

3.1.4 Use a synchronized block

Using consistent locking is one way of creating happens-before edges between accesses to shared data. Using synchronized blocks is one way of implementing locking in Java. The algorithm in Fig. 15 finds all the locks that are released after the source statement and before the manifest statement and suggests protecting the manifest statement with these locks by referring to the specific source lines that performs the locking. It also suggests to protect the source statement with the locks acquired by the thread executing the manifest statement between the source and the manifest instruction.

Figure 16 shows an example in which Thread 1 acquires a lock before accessing the shared data `data` whereas Thread 2 does not acquire any lock before accessing `data`. Figure 17 shows the counterexample path that manifests the data race on `data`. At `s3` Thread 1 unlocks `lock` before the manifest statement `t1`. The data race can be eliminated by making Thread 2 acquire `lock` before `t1`.

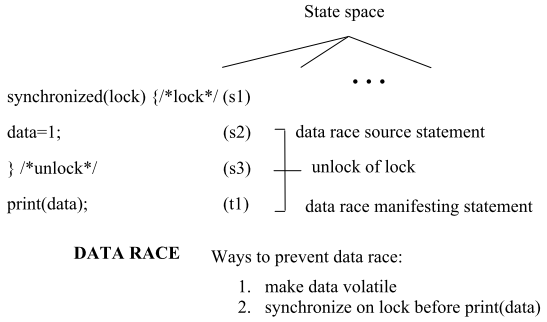


Fig. 17 Part of the state space with the unlock in between source and manifest statements

```
Algorithm isHappensBeforeOrdered (sourceIndex, destIndex, hbEdges) : boolean
    integer sourceIndex, destIndex
  Set of integer pairs hbEdges

  if sourceIndex and destIndex have same executing thread then
    return true;
  if (sourceIndex,destIndex) ∈ hbEdges then
    return true;
  foreach (s1, s2) ∈ hbEdges
     if isHappensBeforeOrdered(sourceIndex, s1, hbEdges)
       and isHappensBeforeOrdered(s2, destIndex, , hbEdges) then
           return true;
  return false;

Algorithm makeChangeOtherToVolatileSuggestions (manifestIndex, sourceIndex, hbEdges)
    integer manifestIndex, sourceIndex
  Set of integer pairs hbEdges

  for each write of v at s1 between sourceIndex and manifestIndex do
     if there exists a read of v at s2 between s1 and manifestIndex
       and isHappensBeforeOrdered(sourceIndex, manifestIndex, hbEdges∪(s1, s2)) then
           if v is an array element then
             print "Use atomic array for v..."
           else
             print "Make v volatile"
```

Fig. 18 Suggest changing a different memory locations to volatile

3.1.5 Change other memory locations to volatile or use atomic arrays

One way of creating a happens-before edge between the source and the manifest statement is to create a happens-before edge between a pair of statements $(s1, s2)$ that come between the source and the manifest statement in the execution sequence, i.e., $(source, \ldots, s1, s2, \ldots, manifest)$. For this to work we need $source \xrightarrow{hb} s1$ and $s2 \xrightarrow{hb} manifest$. If our program modifications establish $s1 \xrightarrow{hb} s2$, then by the transitivity of the happens-before relation, we will have $source \xrightarrow{hb} manifest$.

If $s1$ and $s2$ are the write and read of a variable v, respectively, then changing v to volatile creates a happen before edge between $s1$ and $s2$. Figure 18 shows the algorithm for checking the happens-before relation and the algorithm for this type of suggestion using it.

Figure 2 shows an example with two threads sharing two variables: `done` and `x`. If JRF-E is configured with *threshold* > 1, it is possible to find a counterexample that

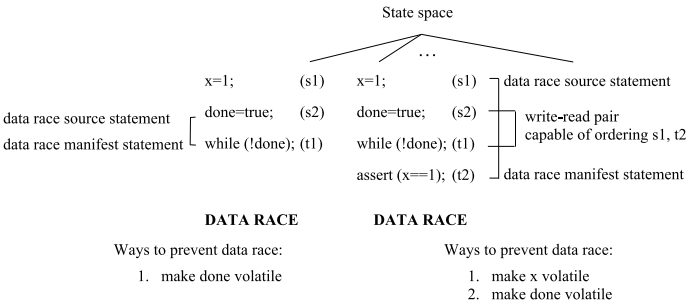


Fig. 19 Part of the state space that change `done` to volatile can eliminate a race on `x`

shows a data race manifested in statement `t2` as shown in Fig. 19. It turns out that between the source statement (`s1`) and the manifest statement (`t2`), there is a write of `done` followed by read of `done`. Since `s1` and `s2` are executed by Thread 1 and `t1` and `t2` are executed by Thread 2, changing `done` to volatile creates a happens-before edge between `s1` and `t2` and eliminates the data race.

In our experience, suggestions from this class are often the most appropriate solution in lock-free algorithms that exhibit data races on multiple variables.

3.1.6 Perform the same synchronization operation

JRF-E keeps track of the acquiring history to allow determination of how happens-before edges were created for non-racy accesses to a memory location. Formally, we define the acquiring history as a function: $AcquireHis : Addr \Rightarrow 2^{(SynchAddr \cup Threads) \times Threads}$. For a memory location m, $(v, t) \in AcquireHis(m)$ means that at some point in the computation so far, thread t performed an operation on v that resulted in m being added to $h(t)$. The actions by thread t that would result in (v, t) being added to $AcquireHis(m)$, for some m could be reading v, locking v, or joining v, where v is a volatile field, lock, or thread, respectively. In contrast to the summary function h, which only applies to a particular path, the *AcquireHis* is cumulative and contains information from all the explored paths.

The algorithm in Fig. 20 determines how previous accesses to the data race memory location (m) have been ordered by the happens-before relation and suggests performing the same acquire operation. Three possible acquire operation choices are `read`, `lock`, and `join` according to the type of memory location. If the memory location is a field, then it must be volatile and the corresponding acquiring operation is to read it. If the memory location is a lock, the acquire operation is to lock it. When the memory location is a reference to thread, joining it serves as an acquire.

The example in Fig. 21 motivates the use of the acquiring history. In execution sequence `(r1, r2, s1, s2, t1)` as shown in Fig. 22, there is a data race between `r1`, a write of `x` by Thread 1, and `t1`, the read of `x` by Thread 3. It should be noted that Thread 2 also performs a read of `x` but it does not result in a data race with Thread 1 and the reason is Thread 2 reads volatile `done` before reading `x` and this generates a happens-before edge between the write of `x` by Thread 1 and the read

```
Algorithm makePerformSameAcquireSuggestions (AcquireHis,m,h)
  Mapping of memoryLocation to Set of (ThreadId,agentLoc) AcquireHis
  MemoryLocation m, loc
  Mapping of memoryLocation to Set of memoryLocation h

  foreach (t, loc) ∈ acquireHistory[m] do
    if m ∈ h(loc) then
     if loc is reference to thread then
      print "join thread loc before manifest instruction"
     else if loc is a field then
      print "read field loc before manifest instruction"
     else
      print "lock the object loc before manifest instruction"
```

Fig. 20 Suggest performing an acquire operation that can add the data race memory location to h of the manifesting thread

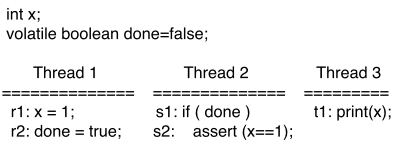


Fig. 21 Acquiring history of Thread 2 shows that Thread 3 will obtain race free access on x by reading done

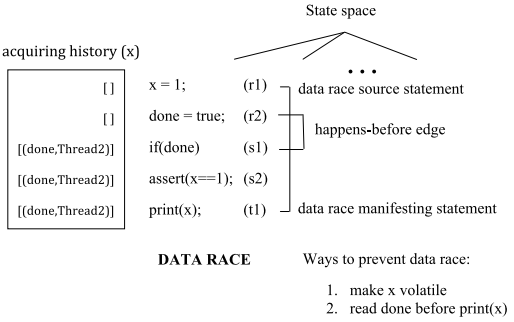


Fig. 22 Part of the state space with the acquiring history that guides how to eliminate the race

of x by Thread 2. The acquiring history stores this information and JRF-E uses it to suggest that Thread 3 reads volatile done before reading x to eliminate the data race.

The suggestions generated by the above algorithms are guaranteed to eliminate the data race on the path where the race was discovered. The "precision" of the suggestions are improved by filtering the set of suggestions to only include those that appear on all of the paths. More details are discussed in Sect. 3.3. Given the set of suggestions, the programmer determines the best solution and implements it. JRF-E should then be rerun.

3.2 Theoretical results

In this section, we prove that modifying the program according to the suggestions generated by our tool does not remove any happens-before edges that existed before the modification. Specifically, Theorems 1 and 2 show that this is the case for all execution paths considering changing a non-volatile to volatile and adding a new synchronization action via putting in a synchronized block or following an acquiring

history-based suggestion, respectively. Theorem 5 shows a similar result for the move suggestion only on the counter-example path under certain conditions.

One important thing to note is that though the code modifications we suggest never remove existing happens-before edges, they may have other side effects. For instance, changing non-volatile field to volatile would result in poor performance since it will disable some compiler optimization. It is clear that addition of a synchronization may result in a deadlock or livelock. It also changes search space by disallowing certain paths. When an instruction is moved, the programmer should make sure that it would not change the program semantics such as the control flow or an output value. However, in all these changes, it is guaranteed that no new race will be introduced as following theorems prove.

Theorem 1 *Changing a non-volatile variable to a volatile variable does not remove any of the existing happens-before edges that result from synchronization actions on any of the execution paths but it may introduce additional happens-before edges.*

Proof Accessing non-volatile variables are not synchronization actions so they cannot involve in the creation of happens-before edges resulting from synchronization actions.

Once a non-volatile variable is changed to a volatile variable, the write accesses will become release statements and the read accesses will become acquire statements and matching release and acquire pairs, if any, will create happens-before edges. □

Theorem 2 *Changing a program by adding a synchronization action* (*joining a thread*, *acquiring a lock*, *and reading a volatile variable*) *that involves an existing memory location*, *does not remove any of the existing happens-before edges that result from synchronization actions or the program order on any of the execution paths.*

Proof An existing happens-before edge that result from synchronization actions can be removed only by changing the source or the destination statement of the happens-before edge. Adding a synchronization action does not change such a statement. Also, an existing happens-before edge that result from program order does not change as a result of adding a synchronization action as happens-before is a transitive relation and all such existing happens-before edges would be preserved due to transitivity. □

Lemma 3 *Moving an instruction that accesses a non-volatile variable does not remove any of the existing happens-before edges that result from synchronization actions on any of the execution paths.*

Proof Same reasoning as in proof of Theorem 1. □

Let $tid(i)$ denote the id of the thread that executed instruction i on a given path.

Lemma 4 *Moving a data race source instruction*, `write x` *at step* h, *before an instruction that is the source of a happens-before edge*, *represented by* $[d, i]$ *where* $d < h < i$, *does not introduce a new data race that involves the moved statement on the same counter-example path if and only if the following conditions hold*:

Fig. 23 Via `goFlag` Thread 1 notifies Thread 2 when object `publish` is ready to be used. Thread 3 can also notify Thread 2 by checking a field of the object pointed by `publish`

```
int x;
boolean goFlag;
volatile Data publish;
Object o;

Thread 1                    Thread 2                      Thread 3
==========                  ==========                    ==========
s1: r = new Data();         t1: if (publish != null) {    u1: synchronized (o) {
s2: publish = r;            t2:   while (!goFlag);        u2:   if (publish.getDesc() != null)
s3: r.setDesc("e");         t3:   String s = publish.getDesc(); u3:   goFlag = true;
s4: synchronized (o) {      t4:   assert(s.equals("e"));  u4:   else
s5:   x=5;                        }                       u5:   goFlag = false;
s6: } // unlock o                                         u6: } // unlock o
s7: goFlag = true;
```

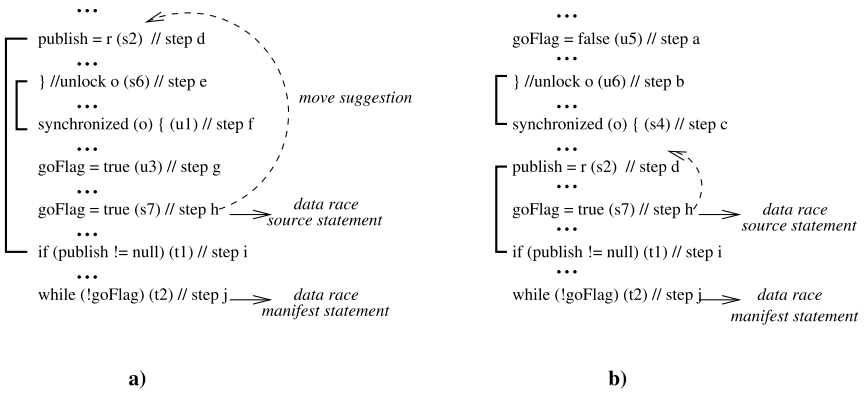


Fig. 24 (**a**) Instantiation of Condition 1 in Lemma 4 (**b**) instantiation of Condition 2 in Lemma 4 based on sample program in Fig. 23. Happens-before edges formed by synchronization actions are shown by lines connecting the matching release and acquire instructions

Condition 1: *The moved source instruction does not become a new data race source instruction. For any* `read/write x` *at step* g *s.t.* $d < g < h$, *there exists a happens-before edge represented by range* $[e, f]$ *s.t.* $d < e < f < g$ *and* $tid(h) = tid(e)$ *and* $tid(f) = tid(g)$.

Condition 2: *The moved source instruction does not become a new data race manifesting instruction. For the first* `read/write x` *at step* a *that precedes the instruction at* d, *there exists a happens-before edge* $[b, c]$ *s.t.* $a < b < c < d$ *and* $tid(a) = tid(b)$ *and* $tid(c) = tid(h)$.

Proof Condition 1: After the move, the old data source instruction (at step h) would be at step $d - 1$ and would be ordered with the instruction at step g by happens-before. As a concrete example consider the sample program in Fig. 23 and the corresponding counter-example path in Fig. 24a.

Condition 2: After the move, the old data source instruction (at step h) would be at step $d - 1$. The instruction at step a and the instruction at step $d - 1$ would be ordered by happens-before. As a concrete example consider the sample program in Fig. 23 and the corresponding counter-example path in Fig. 24b. □

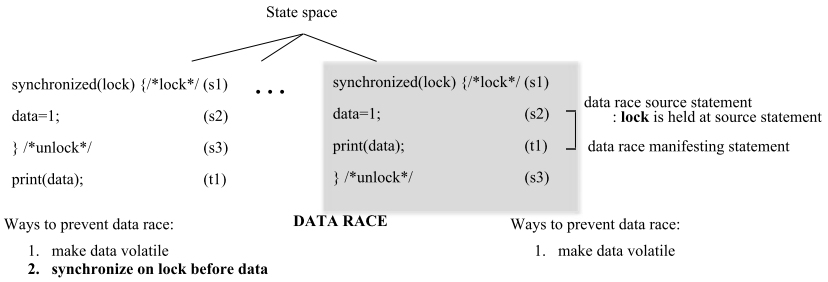


Fig. 25 Other counterexample path for the same race in Fig. 17

Theorem 5 *Moving a data race source instruction,* `write x` *at step h, before an instruction that is the source of a happens-before edge, represented by* $[d, i]$ *where* $d < h < i$*, does not involve any new data races if and only if Conditions*[10] 1 *and* 2 *in Lemma* 4 *hold.*

Proof Follows from Lemmas 3 and 4. □

3.3 Combining suggestions from multiple traces

As shown in Fig. 4, considering multiple races at the same time would give better choice among JRF-E suggestions. Even with one race,[11] it is worthwhile to consider the suggestions from multiple traces when selecting the most appropriate fix. It is obvious that one race is discovered in multiple traces with different set of suggestions. For example, the simple program in Fig. 2 has only two races on `x` and `done` but reports nine different traces. Since JRF-E suggestions are based on the happens-before relations among executions in the counterexample path, it is possible that a suggestion from one trace would not be applicable to other trace for the same race. As discussed in Sect. 4, JRF-E provides the ability to detect multiple race traces at one JRF-E run and the suggestions from this option are more precise than one race trace result.

The rule to combine suggestions for a race depends on the types of suggestions. Suggestions from Sects. 3.1.1 and 3.1.2 are common to all different traces for a race and always guarantee to eliminate the specific race. Suggestions from Sects. 3.1.3, 3.1.5 and 3.1.6 are specific to the counterexample path and cannot applicable to other traces with different happens-before relations. JRF-E produces combined suggestions for a race by intersecting those suggestion from different traces.

One exception is the suggestion in Sect. 3.1.4. In Fig. 25 which shows different trace for a race represented in Fig. 17, it is explained that this suggestion is only present in subset of all interleavings and the uncovered state spaces are excluded by the modification when the corresponding lock is held at the source or manifest

[10] Checking of the two conditions in Lemma 4 has been implemented in JRF and the suggestion is made only if the conditions hold. For brevity, the algorithm in Fig. 12 does not include checking these conditions.

[11] Two race traces report the same race when their source and manifest statements are the same.

```
Algorithm combineSuggestions (TraceInfo)
  Mapping of Traces to Set of (Suggestions,sSyncS,mSyncS,slocks,mlocks) TraceInfo
  integer t
  MemoryLocation l
  suggestion s
  Set of suggestion advices ←TraceInfo[1].Suggestions
  Set of (suggestion, MemoryLocation) sadvices ←TraceInfo[1].sSyncS
  Set of (suggestion, MemoryLocation) madvices ←TraceInfo[1].mSyncS

  for t from 2 to size(TraceInfo) do
    advices ←advices ∩ TraceInfo[t].Suggestions
    sadvices ←sadvices ∪ TraceInfo[t].sSyncS
    madvices ←madvices ∪ TraceInfo[t].mSyncS
  foreach (s, l) ∈ sadvices do
    for t from 1 to size(TraceInfo) do
      if l ∉ TraceInfo[t].slocks then
        sadvices ←sadvices \ (s, l)
        break
  foreach (s, l) ∈ madvices do
    for t from 1 to size(TraceInfo) do
      if l ∉ TraceInfo[t].mlocks then
        madvices ←madvices \ (s, l)
        break
  foreach s ∈ advices do
    print s
  foreach (s, l) ∈ sadvices do
    print s
  foreach (s, l) ∈ sadvices do
    print s
```

Fig. 26 Combine suggestions from multiple traces for the same race

statement depending on the reasons we suggested in Fig. 15. JRF-E only generates this type of suggestion when the corresponding lock is held at the source or manifest statement for all traces of a race.

Figure 26 summarizes the algorithm to combine suggestions from different traces. A trace information produced by JRF-E composed of suggestions from Sects. 3.1.1, 3.1.2, 3.1.3, 3.1.5 and 3.1.6 as `Suggestions`, suggestions from Sect. 3.1.4 divided into `sSyncS` and `mSyncS` according to the statement to put in synchronized block, and the set of locks held by the thread executing source or manifest statement as `slocks` and `mlocks`.

Combining suggestions for different races in a program can be ranked using the number of races it can eliminate. However, JRF-E provides all suggestions with their rank instead of reporting the highest rank one as shown as `frequency of advice` in Fig. 4. This is to let the programmer decide the most appropriate modification considering possible side-effects including a deadlock and overhead.

3.4 Separation of model checking and counterexample analysis

Clearly, adding the additional data structures and algorithms required for JRF-E increases the runtime and memory usage over that of JRF. Further, since more counterexample paths yield better suggestions, it is desirable to analyze multiple races and combine the suggestions. In our experience, one line of a racy code easily produces thousands of counterexample traces making it infeasible to store all of them without significantly reducing the applicability of the tool. We address this problem by providing an option to store the required information in a file and analyze it later with

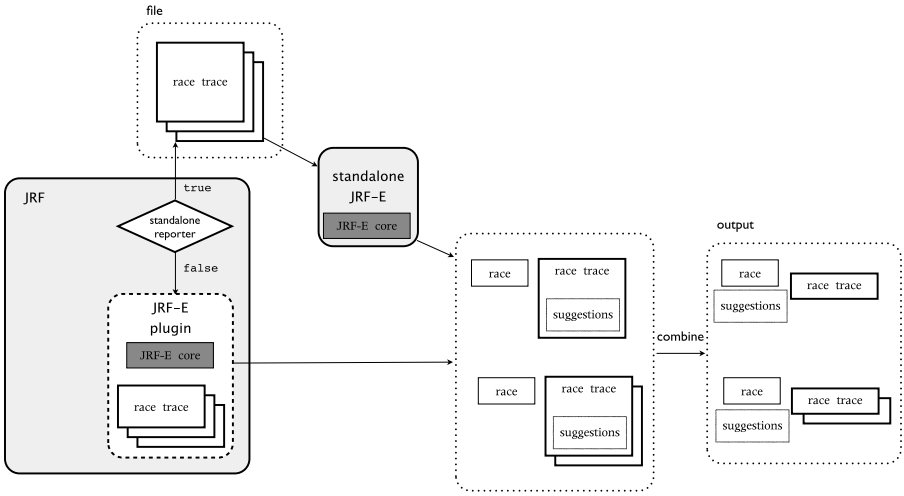


Fig. 27 The JRF-E execution model: plugin or standalone

a tool called standalone JRF-E as opposed to the combined tool with JRF as JRF-E plugin. Figure 27 shows the structure of JRF-E extension. Each counterexample trace is directed to a file without any analysis when JRF detects a race. JRF-E will analyze traces and categorize them into different races, bookkeeping the suggestions accordingly. Additional information, such as happens-before edges in the counterexample path and acquiring history, are available to better explain the race. This approach is close to the JPF extension *jpf-trace-server* that provides storing, querying, and analysis of execution trace. *jpf-trace-server* cannot be used directly because of the additional information required by JRF.

Standalone JRF-E also provides happens-before relations in each trace to help programmers understand the counterexample path. The computation of happens-before relations in a trace, *findHBEdges*, is the most time-consuming part of JRF-E analysis with the complexity of $O(n^2)$ where n is the length of the path.[12] This is stored in the file for reuse. The same mechanism is used for suggestions and allows standalone JRF-E to be incremental. This enhances the scalability of JRF-E.[13]

Figures 28 and 29 show the output using standalone JRF-E. The time and memory requirement for JRF standalone reporter usually took more than JRF but less than

[12]This assumes worst case scenario based on current JPF trace structure. We can enhance the algorithm by filtering out threadlocal instructions from the trace and managing trace data more wisely but this is left as future work.

[13]The `Filter` example from Herlihy and Shavit (2008) in Sect. 4 reported 6 races with 863 traces and was successfully analyzed using this incremental mechanism which ran out of memory in standalone mode when tried to analyze all at once. This is more important in user-friendly GUI interface as a future extension to standalone JRF-E, in that the number of traces loadable at the same time was the bottleneck in our experience and it is easy to selectively load minimum necessary information with this mechanism. In addition, it will be also possible to analyze races in parallel and merge the result. These two extension is left as future work.

```
khkim~/JRF/jpf-racefinder ]$ bin/jrf_standalone_reporter simple.SimpleRace
------------
## write to "jrf/simple.SimpleRace.trace1".
## write to "jrf/simple.SimpleRace.trace2".
## write to "jrf/simple.SimpleRace.trace3".
## write to "jrf/simple.SimpleRace.trace4".
## write to "jrf/simple.SimpleRace.trace5".
## write to "jrf/simple.SimpleRace.trace6".
## write to "jrf/simple.SimpleRace.trace7".
## write to "jrf/simple.SimpleRace.trace8".
## write to "jrf/simple.SimpleRace.trace9".
## write to "jrf/simple.SimpleRace.race" which has 2 different races with 9 counterexample traces.
------------
khkim~/JRF/jpf-racefinder ]$
```

Fig. 28 The JRF standalone reporter output for the program in Fig. 2. The race statistics is written to the file `jrf/simple.SimpleRace.race` and the traces are saved as `jrf/simple.SimpleRace.trace#`

JRF-E unless the program was very simple (such as the program used in Fig. 28), so that the overall disk access time outweighed the analysis time.

4 Experimental results

In order to evaluate the usefulness and determine good threshold values for number of data races to detect before stopping, JRF-E was used to analyze programs taken from a variety of sources including a textbook[14] on multiprocessor programming by Herlihy and Shavit (2008), the Amino Concurrent Building Blocks library (2012), barrier implementations from a Google concurrent data structures workshop (2012), the Java Grande Forum Benchmark Suite (2012), and undergraduate students assignments implementing a multithreaded web server simulator. JRF discovered a data race in 19 out of 65 examples from the textbook, 9 races in 20 Junit tests[15] and 10 examples in Amino[16], and 10 of 12 barrier implementations in the Google code. We also found data races in 6 out of 10 examples from the Java Grande Foum benchmarks and 7 out of 28 students projects. These results show that data races are a common error in Java programs, even when written by experts.

Figure 30 summarizes the statistics for all 51 tests with races. JRF-E was configured to stop at the first race found. Since the suggestions were counted when the analysis stopped at the first race detection, the suggestion might not be able to remove a similar race found in a different path. For example, the "move" suggestion in the table were the cases that the first race can be eliminated by moving the source instruction but it cannot fix the same race found in other path. The most appropriate suggestions are chosen according to the program semantic analysis and verified by manually testing the modified version using JRF-E again.

Most races were caused by not using volatile or atomic array. 31 races can be fixed by directly changing the field or array involved to volatile or atomic array. Four of

[14] Java implementations were obtained from the book's companion web site.

[15] 183 different functions are tested in 20 Amino Junit tests.

[16] These races involve features documented as not thread safe, thus indicate bugs in client code, not the Amino library.

```
khkim~/JRF/jpf-racefinder ]$ bin/jrfe_standalone simple.SimpleRace
------------
## read from "simple.SimpleRace.race" which has 2 different races with 9 counterexample traces.
        What do you want to do next?
                [1] read all traces.
                [2] read specific race.
                [3] read specific trace.
                [4] analyze all traces.
                [5] analyze specific race.
                [6] analyze specific trace.
                (choice)=1
target=simple.SimpleRace

=================================================== data race #1
...
trace=
===================================================  trace #1
...
---------------------------------------------------  HBedges

---------------------------------------------------  advice trace #1

===================================================  trace #2
...
===================================================  advice race #1

=================================================== data race #2
...
===================================================  advice race #2

___________________________________________________ frequency of advice

        What do you want to do next?
                [1] read all traces.
                [2] read specific race.
                [3] read specific trace.
                [4] analyze all traces.
                [5] analyze specific race.
                [6] analyze specific trace.
                (choice)=4
=================================================== data race #1
...
trace=
===================================================  trace #1
...
---------------------------------------------------  HBedges
[1]("monitorexit" at java/lang/ThreadGroup.java:870) ---> ("monitorenter" at java/lang/ThreadGroup.java:855)
[2]("monitorexit" at java/lang/ThreadGroup.java:870) ---> ("monitorenter" at java/lang/ThreadGroup.java:855)
---------------------------------------------------  advice trace #1
data race source statement : "putstatic" at simple/SimpleRace.java:64  : "x = 1;" by thread 1
data race manifest statement : "getstatic" at simple/SimpleRace.java:74: "assert (x==1);" by thread 2

Change the field "simple.SimpleRace.x from INITIALIZER" to volatile.
Change the field "simple.SimpleRace.done from INITIALIZER" to volatile.

===================================================  trace #2
...
===================================================  advice race #1
data race source statement : "putstatic" at simple/SimpleRace.java:64  : "x = 1;" by thread 1
data race manifest statement : "getstatic" at simple/SimpleRace.java:74: "assert (x==1);" by thread 2

Change the field "simple.SimpleRace.x from INITIALIZER" to volatile.
Change the field "simple.SimpleRace.done from INITIALIZER" to volatile.

=================================================== data race #2
...
===================================================  advice race #2
data race source statement : "putstatic" at simple/SimpleRace.java:65  : "done = true;" by thread 1
data race manifest statement : "getstatic" at simple/SimpleRace.java:73: "while(!done) { /*spin*/ }" by thread 2

Change the field "simple.SimpleRace.done from INITIALIZER" to volatile.

___________________________________________________ frequency of advice
[2times] Change the field "simple.SimpleRace.done from INITIALIZER" to volatile.
[1times] Change the field "simple.SimpleRace.x from INITIALIZER" to volatile.

        What do you want to do next?
                [1] read all traces.
                [2] read specific race.
                [3] read specific trace.
                [4] analyze all traces.
                [5] analyze specific race.
                [6] analyze specific trace.
                (choice)=0
## done.
------------
khkim~/JRF/jpf-racefinder ]$
```

Fig. 29 Standalone JRF-E output for the saved race traces in Fig. 28. Note the analysis results such as `HBedges` and `advice` are empty at the read traces phase but filled after the analysis phase. The only `HBedges` in this program are between the two threads starts and their first instructions

	Change to volatile or atomic	Move source statement	Use a synchronized block	Perform similar acquire	Change other to volatile or atomic
Herily-Shavit	19/9	3/0	0/0	9/5(1)*	6/4
Google	10/10	0/0	0/0	0/0	0/0
Amino	9/8	0/0	1/1	0/0	0/0
JGF	6/4	1/0	2/1**	3/2**	6/0
Students project	7/0	0/0	1/1(6)***	0/0	5/0
Total	**51/31**	**4/0**	**4/3(6)*** **	**12/7(1)***	**19/4**

* The correct solution is to perform similar acquire in acquiring history which happens later in the path.
** "Use a synchronized block" and "perform similar acquire" suggest to use same synchronization.
*** JRF-E failed to suggest correct ones since the share data require atomic accesses in addition to race freedom.

Fig. 30 The number of suggestions by JRF-E for the first races detected and the correct solutions to fix them for all test cases given as (JRF-E suggestions/correct solution). The correct solution for students projects cases in ()* are to ensure correct program semantics including atomicity in addition to eliminate the race

them depend on changing a different field to volatile or an atomic array. This is because many of the examples from the textbook, Amino, and Google barrier code were concurrent data structures that implement lock-free algorithms. Also, the programmers who implement the concurrent libraries are the experts who tend to make few basic concurrency mistakes caused by missing synchronization or inconsistent locking. JRF-E failed to suggest one appropriate solution of Herlihy and Shavit (2008) since the model checking was stopped at the first race trace and the acquiring history was incomplete at that moment.

In the students projects, on the other hand, the races were related to the fields accessed without any protection. Their code showed, in many cases, a lack of understanding of how to safely access shared data in a concurrent program. Since six of them didn't use any acquire/release operations, JRF-E only suggested the straightforward suggestions-to use volatile/atomic array or change others to volatile. Though all those suggestions by JRF-E can eliminate the races found, the correct program modification in those cases were using synchronization to protect shared data to ensure other program semantics such as atomicity. In this set of test cases, JRF-E failed to provide the most appropriate solutions in six out of seven. For inexperienced concurrent programmers, the best approach is to use plain old Java Pathfinder (which assumes sequential consistency) to find and correct those concurrency errors that are problems even in a sequentially consistent environment, and then use JRF-E to find and deal with races, possibly after using JRF to find the races and identify shared variables.

The Java Grande Forum examples are perhaps the most illustrative; they contained significant amounts of threading and synchronization, but were application programs rather than concurrency libraries. JRF-E successfully provided suggestions for all of them. Suggestions from Sects. 3.1.4 and 3.1.6 sometimes advise the same changes.

More in depth test results excluding students projects are shown in Table 1 to Table 8. Testing was performed on a i386/8 processor with 32 GB ram using Linux/2.6.32-24-generic OS, JPF version 5, and Sun Microsystems Inc./1.6.0_16 Java with 2 GB JVM heap memory. Tables 1, 3, 5 and 7 summarize the resources required to find 1, 10, and 100 race traces. The "states" column represents total state space visited and the "max length" column contains the number of transitions in the

Table 1 Experimental results for (Herlihy and Shavit 2008) examples containing races found by JRF-E. Results threshold as 1, 10, 100 are given

Example	Traces	States	Max length	JRF time (s)	JRF-E time (s)	Mem (MB)	Sync Addr[a]	Addr[b]
DisBarrier	1	108	108	8	0	70	36	300
	10	331	265	27	5	157	36	304
	100	1656	268	150	57	263	36	304
StaticTreeBarrier*	1	58	58	10	0	57	39	322
CoarseHashSet**	1	60	34	8	0	71	42	341
	10	177	38	37	2	157	69	374
LockFreeHashSet	1	55	55	7	0	68	44	380
	10	81	57	10	7	77	44	386
	100	138	61	18	26	107	44	406
RefinableHashSet	1	101	56	19	2	105	49	376
	10	251	61	66	23	223	63	383
	100	360	62	95	36	242	76	409
StripedHashSet**	1	60	34	9	0	63	46	351
	10	184	38	41	5	159	73	384
LazyList	1	64	64	10	0	86	54	353
	10	118	66	20	8	119	54	355
	100	489	77	108	81	241	68	362
OptimisticList	1	55	55	8	0	74	54	349
	10	118	59	22	8	123	54	351
	100	1645	59	597	24	412	102	407
Bakery	1	33	33	3	1	40	38	423
	10	36	34	4	16	39	38	430
	100	71	51	9	181	72	38	492
Filter	1	54	55	5	2	47	38	300
	10	72	70	6	32	55	38	304
	100	197	81	18	304	102	38	320
LockFreeQueue	1	25	25	1	0	29	29	236
	10	34	29	2	0	33	29	238
	100	254	38	11	5	76	29	238
Peterson	1	24	24	2	0	34	35	307
	10	29	26	2	2	33	35	319
	100	141	39	9	20	73	35	337

Table 1 (*Continued*)

Example	Traces	States	Max length	JRF time (s)	JRF-E time (s)	Mem (MB)	Sync Addr[a]	Addr[b]
ALock	1	22	22	2	0	31	35	245
	10	25	23	2	1	34	35	245
	100	161	36	9	9	72	35	245
CLHLock	1	19	19	1	0	32	33	250
	10	24	21	2	0	33	33	255
	100	265	31	18	3	80	33	255
MCSLock	1	17	17	1	0	32	33	253
	10	71	26	4	0	49	33	253
	100	312	30	18	3	94	33	279
CorrectedMCSLock*	1	17	17	1	0	29	35	251
DEQueue	1	26	26	2	0	32	29	245
	10	50	35	3	1	38	29	245
	100	274	37	13	11	80	29	246
UnboundedQueue	1	35	35	5	0	55	46	333
	10	40	37	6	5	52	46	335
	100	196	45	31	52	130	46	348

[a]Volatiles, locks, and threads

[b]Non-volatiles

*JRF-E ran out of memory before detecting 10 and 100 threshold races

**JRF-E ended by application assertion error before detecting 100 threshold races

longest counterexample path. The memory requirement and elapsed time for JRF and JRF-E are given in the next three columns. The overhead of counterexample analysis is the time and memory spent in managing additional data such as the acquiring history and path elements as well as applying the algorithm to generate suggestions. The final two columns show the field involved in a detected data race and the suggestions generated by JRF-E. Each row gives results generated using three different values for the number of races threshold. The suggestion in **bold** in Tables 2, 4, 6 and 8 corresponds to a solution chosen by a knowledgeable programmer as the most appropriate way to eliminate the data race on fields in the second column.

In the `DisBarrier` test, the `log[]` integer array is part of the test driver, not the barrier. The race on its elements could be eliminated trivially by changing the array to an atomic array. However, the suggestions show that changing the `flag[]` to an atomic array will also correct the races on `log[]`, and this is the best solution. `StaticTreeBarrier` and `CorrectMCSLock` marked with * ran out of memory before detecting 10 and 100 threshold race traces and used other optimization technique to exclude threadlocals to get JRF-E results. `TCuckooHashSetBarrier` ended with an application assertion failure before de-

Table 2 JRF-E suggestions for Herlihy-Shavit examples with a race summarized from 1/10/100 traces

Test suite	Race field of class	Analysis
DisBarrier	flag[] of Node	**use atomic array for flag[]**
	log[] of DisBarrier	**use atomic array for flag[]** or log[]
StaticTreeBarrier	sense of StaticTreeBarrie	**make sense volatile**
	log[] of StaticTreeBarrie	use atomic array for log[], **make sense volatile**
CoarseHashSet	size of CoarseHashSet	make size volatile, **lock the *lock***
LockFreeHashSet	bucket[] of LockFreeHashSet	**use atomic array for bucket[]**
	head of BucketList	make next volatile, **use atomic array for bucket[]**
	next of Node	make next or head of BucketList volatile, **use atomic array for bucket[]**
RefinableHashSet	size of RefinableHashSet	make size volatile,**lock the locks[]**
StripedHashSet	size of StripedHashSet	make size volatile, **lock the locks[]**
TCuckooHashSet	table[] of TCuckooHashSet	use atomic package for table[], **lock the locks[][]**
LazyListTest	next of Node	make next volatile, **lock the lock**
	marked of Node	make marked or next volatile, **lock the lock**
	key of Node	make key, marked, or next volatile, **lock the lock**
OptimisticList	next of Entry	make next volatile, **lock the lock**
	key of Entry	make key or next volatile, **lock the lock**
Bakery	label[] of Bakery	**use atomic array for label[] and flag[]**, make counter of Label volatile
	counter of Label	make counter or id of Label volatile, **use atomic array for label[]**
	flag[] of Bakery	**use atomic array for flag[]**
Filter	level[] of Filter	**use atomic array for victim[]** or level[]
	victim[] of Filter	**use atomic array for victim[]**
	counter of Filter	make counter volatile, **use atomic array for victim[]** or level[]
Peterson	victim of Peterson	make victim or counter volatile, **use atomic array for flag[]**
	flag[] of Peterson	**use atomic array for flag[]**
	counter of Peterson	make counter volatile, **use atomic array for flag[]**
LockFreeQueue	tail of LockFreeQueue	**make tail volatile**
	head of LockFreeQueue	**make tail volatile**
	items[] of LockFreeQueue	use atomic array for items[], **make tail volatile**
ALock	flag[] of ALock	**use atomic array for flag[]**, make value of Entry volatile
	counter of ALock	make counter or value of Entry volatile, **use atomic array for flag[]**

Table 2 (*Continued*)

Test suite	Race field of class	Analysis
CLHLock	locked of QNode	**make locked volatile**
	counter of CLHLock	**make locked of QNode** or counter **volatile**
MCSLock	next of QNode	**make next of QNode volatile**
	counter of MCSLock	**make locked of QNode** or counter **volatile**
	locked of QNode	**make locked of QNode volatile**
CorrectedMCSLock	counter of MCSLock	**make locked of QNode** or counter **volatile**
DEQueue	bottom of DEQueue	**make bottom volatile**
	map[] of DeQueue	**use atomic array for map[]**
UnboundedQueue	next of Node	make next volatile, **lock the enqLock**
	value of Node	make value or next volatile, **lock the enqLock**

*__bold__ entry indicates the most appropriate solution

tecting a race trace and other configuration of JRF-E revealed a race on `table[]`. `CoarseHashSet` uses a single lock, `StripedHashSet` uses a fixed-size array of locks, and `RefinableHashSet` uses a resizable array of locks to implement a closed-address hash set. All three cases have a race on `size` and are corrected using additional locking. `LockFreeHashSet` is implemented using AtomicIntegers and a `BucketList` which is a list of `Node`. The bucket list should be changed to atomic array to guarantee the "volatile" semantics for each element access. Even though the elements of a bucket list, are not protected at all, the thread-safe access of the BucketList provides the required ordering of accesses to the internal list elements. JRF-E verifies that using an atomic array for `bucket[]` will also eliminate the races on `Node`, `next` of `Node`, and `head` of `BucketList`. The race involving `next` is quite subtle. The next field is declared to be an `AtomicMarkableReference<Node>`. This means that the accesses of the objects referenced by the field can be safely accessed with volatile semantics. Since the data structure is a linked list, accesses to Nodes other than the first are safe. The next node of the head of the list however, exhibits data races. In `OptimisticList`, each list element is represented by an `Entry` object. Races on `next` and `key` fields are both removable by marking `next` as volatile. `Bakery`, `Filter`, and `Peterson` implemented mutex algorithms using arrays. All three had the common mistake of using an array reference declared as volatile without volatile semantics of element accesses. In `MCSLock` three fields of `QNode` are involved in races but one change, changing `next` to volatile, does not resolve the other two. From the suggestions, we can easily deduce that one more fix is needed, changing `locked` to volatile, which will also correct the data race involving `counter`.

The Amino open source software project (Amino concurrent building blocks 2012) implements concurrent building blocks in highly efficient and scalable codes, and aims to support a set of lockfree collection classes, parallel patterns, and scheduling algorithms. The `org.amino.ds.lockfree.LockFreeDeque` implements

Table 3 Experimental results for (Amino concurrent building blocks 2012) examples containing races found by JRF-E. Results threshold as 1, 10, 100 are given

Example	Traces	States	Max length	JRF time (s)	JRF-E time (s)	Mem (MB)	Sync Addr[a]	Addr[b]
IteratorTest (EBDeque)	1	26	26	33	0	139	91	564
	10	36	27	58	9	172	91	571
	100	47	28	81	14	228	91	582
IteratorTest (LockFreeDeque)	1	26	26	8	0	65	58	458
	10	36	27	14	7	80	58	465
	100	47	28	19	12	105	58	476
IteratorTest (LockFreeList)	1	38	38	10	0	66	52	446
	10	48	39	14	5	92	52	453
	100	59	40	18	8	95	52	464
IteratorTest (LockFreeOrderedList)	1	38	38	10	0	65	52	458
	10	48	39	15	5	99	52	465
	100	59	40	19	9	98	52	476
IteratorTest (LockFreePriorityQueue)	1	33	33	89	22	227	96	995
	10	47	34	137	159	248	93	978
	100	64	35	187	252	260	92	978
IteratorTest (LockFreeQueue)	1	61	61	20	0	102	57	518
	10	76	71	27	5	121	63	533
	100	126	71	76	58	238	63	533
QueueTest	1	14	14	2	0	39	44	314
	10	31	22	7	0	56	45	335
	100	204	47	43	6	130	47	355

[a]Volatiles, locks, and threads

[b]Non-volatiles

a double-ended queue using a Compare-And-Set based lock free algorithm. The lock free algorithm uses an `AnchorType` object with left and right pointers, a status field, and a number of elements in deque. One anchor is defined for each deque, and is immutable. It uses `java.util.atomic.AtomicIntegerFieldUpdater` to change the status field. In addition, an anchor for a deque is updated using a `java.util.atomic.AtomicReference` wrapper. The `org.amino.ds.lockfree.LockFreeQueue` is a lock free FIFO queue. It also uses two pointers, prev and next, instead of a standard singly linked list, stores head and tail of the queue in a volatile field, and is updated using `java.util.atomic.AtomicReferenceFieldUpdater`. `QueueItr` for this class is not thread safe and has a race on its `nextNode` field when the same iterator is used by multithreads. The 8 races found in this test suite were in the iterator which contained comments

Table 4 JRF-E suggestions from counterexample and acquiring history analysis for (Amino concurrent building blocks 2012) examples with races

Test suite	Race field of class	Analysis
Iterator(EBDeque)	cursor of DeqIterator	**make cursor volatile**
Iterator(LockFreeDeque)	cursor of DeqIterator	**make cursor volatile**
Iterator(LockFreeList)	next of ListItr	**make next volatile**
	cur of ListItr	**make** cur, **next volatile**
	prev of ListItr	**make** cur, **prev volatile**
Iterator(LockFreeOrderedList)	next of ListItr	**make next volatile**
	cur of ListItr	**make** cur, **next volatile**
	prev of ListItr	**make** cur, **prev volatile**
Iterator(LockFreePriorityQueue)	cursor of PQueueIterator	**make cursor volatile**
Iterator(LockFreeQueue)	nextNode of QueueItr	**make nextNode**, nextItem **volatile**
	nextItem of QueueItr	**make nextItem volatile**
	lastRef of QueueItr	**make** lastRef, **nextNode volatile**
Iterator(LockFreeSet)	next of CompositeStateHold	**make next volatile**
	cur of CompositeStateHold	**make** next, **cur volatile**
	prev of CompositeStateHold	**make prev volatile**
Queue	prev of Node	**make prev volatile**

*__bold__ entry indicates the most appropriate solution

indicating that it was not thread-safe. The race can be eliminated using volatile fields but cannot avoid concurrent modification within the iterator. Different configuration with threadlocal optimization allowed us to get the result for `LockFreeSet`. One of the example program, `OrderedListExample`, finds a benign[17] race in `java.lang.String` class.

The barrier implementations in Table 6 illustrates the limitations of JRF-E's approach. The race on `_value` of `CounterWithBarrier`, which is part of the test driver, not the barrier, can be eliminated by marking it volatile. However, the problem is that the implementation has multiple threads set the variable to the same value, with each update after the first manifesting a data race. A better solution is to avoid the multiple updates. This sort of semantic reasoning cannot be done by JRF-E. However, the suggestion given would eliminate the data race.

[17] Aspects of the JMM constrain the behavior of programs with races. Benign races are those where these constraints guarantee that overall behavior of the program is correct, even with a race. The hashcode variable in the `java.lang.String` class is the most important example. The hashcode is created lazily, when needed, and is never changed once initialized. The JMM semantics guarantee that the only values seen will be the correct hashcode value, or null, in which case the value will be (re) computed. Since the recomputed value will be the same, it doesn't affect correctness if the data race allows it to be done more than once.

Table 5 Experimental results for (Google concurrent data structures workshop barriers 2012) examples containing races found by JRF-E. Results threshold as 1, 10, 100 are given

Example	Traces	States	Max length	JRF time (s)	JRF-E time (s)	Mem (MB)	Sync Addr[a]	Addr[b]
LinearSenseBarrierVolatileTest	1	39	39	6	0	61	43	372
	10	53	54	8	5	66	43	372
	100	338	67	48	58	151	43	394
LinearSenseBarrier	1	40	40	6	0	48	44	400
	10	56	57	9	3	67	44	424
	100	366	72	60	37	176	44	514
SimpleBarrier[*]	1	37	29	5	0	44	38	335
SenseBarrier	1	48	48	7	0	57	41	362
	10	62	49	9	3	58	41	362
	100	498	50	71	32	206	41	380
SenseBarrierWithWaitTest	1	68	68	8	0	54	41	360
	10	90	69	10	6	69	41	360
	100	717	70	83	63	180	41	378
TreeBarrier	1	71	71	12	0	80	44	386
	10	88	72	14	2	74	44	386
	100	431	72	58	24	215	44	386
LockBarrier	1	34	34	6	0	65	58	406
	10	40	37	7	6	77	58	406
	100	217	43	60	67	176	62	407
CyclicBarrier	1	25	25	2	0	25	37	334
	10	41	31	3	1	58	37	334
	100	481	36	36	20	162	38	346
BinaryStaticTreeBarrier	1	60	60	10	0	66	47	391
	10	87	78	16	5	88	47	391
	100	1212	78	213	48	263	52	483
SplittedSenseBarrier	1	71	71	14	0	72	49	383
	10	94	72	18	8	82	49	383
	100	433	72	73	73	171	49	383

[a]Volatiles, locks, and threads

[b]Non-volatiles

[*]JRF-E ran out of Java heap space before threshold races and JPF failed to report the result

Four `jgf` examples have races on volatile arrays without volatile element access `IsDone[]` and `sync[]` and nonvolatile array `A[][]`. These widely available benchmarks were implemented before the current JMM, and before the issues relevant to the memory model were well known, and illustrate the useful-

Table 6 JRF-E suggestions from counterexample and acquiring history analysis for (Google concurrent data structures workshop barriers 2012) examples with races

Test suite	Race field of class	Analysis
LinearSenseBarrierVolatile	_value of CounterWithBarrier	**make _value volatile**, use atomic array for _threadDoneArray[]
	_threadDoneArray[] of BaseLinearSenseBarrierVolatile	**use atomic array for _threadDoneArray[]**, make _value of CounterWithBarrier volatile make value of Entry volatile
LinearSenseBarrier	_value of CounterWithBarrier	**make _value volatile**, use atomic array for _threadDoneArray[]
	_threadDoneArray[] of BaseLinearSenseBarrier	**use atomic array for _threadDoneArray[]**, make _value of CounterWithBarrier volatile make value of Entry volatile
SimpleBarrier	_value of Counter	**make _value volatile**
SenseBarrier	_value of Counter	**make _value volatile**
SenseBarrierWithWait	_value of Counter	**make _value volatile**
TreeBarrier	_value of Counter	**make _value volatile**
LockBarrier	_value of Counter	**make _value volatile**
CyclicBarrier	_value of Counter	**make _value volatile**
BinaryStaticTreeBarrier	_value of Counter	**make _value volatile**
SplittedSenseBarrier	_value of Counter	**make _value volatile**

*__bold__ entry indicates the most appropriate solution

Table 7 Experimental results for (The Java Grande Forum benchmark suite 2012) examples containing races found by JRF-E. Results threshold as 1, 10, 100 are given

Example	Traces	States	Max length	JRF time (s)	JRF-E time (s)	Mem (MB)	Sync Addr[a]	Addr[b]
BarrierBench	1	87	87	13	0	103	45	474
	10	128	102	27	2	137	45	535
	100	1667	102	637	22	284	45	556
SyncBench	1	111	103	18	0	115	48	507
	10	131	113	25	0	128	48	507
	100	666	113	229	5	273	48	507
lufact	1	34	34	4	0	39	35	274
	10	72	72	7	1	49	35	276
	100	331	331	29	64	156	35	300
sor	1	15	15	8	0	57	42	690
	10	77	77	43	13	130	42	691
	100	159	159	78	288	222	42	691
moldyn	1	2821	2821	638	32	723	37	521
	10	2861	2861	663	295	725	37	521
	100	3136	3136	697	3621	754	37	521
montecarlo	1	86	86	38	2	181	64	919
	10	91	88	50	19	190	64	984
	100	167	125	239	202	297	66	1103

[a]Volatiles, locks, and threads

[b]Non-volatiles

ness of a tool that can check for data races even in apparently correct code. One race found in `montecarlo` was a benign race caused by the redundant update on `UNIVERSAL_DEBUG` field.

The tests showed that in most cases, the tool gave the most appropriate suggestion. This is especially true for programs where the problem is lack of attention (or misunderstanding) of memory model issues, but the programs are basically correct. JRF-E allows programmers to confidently choose a solution where the volatility of one variable also guards other variables from data races. In addition, the results show that regardless of the number of memory locations involved in a race, the threshold plays important role to better suggestions. When considering only one trace to select a fix, there were more chances of discovering the same race in different path where the fix is not applicable. If the program contained races involving more than one memory location, then considering them together helps to find the most appropriate suggestions. Based on this fact, number of race traces determines the quality of suggestions and the optimization including separation of detection and analysis described in Sect. 3.4 is significant.

Table 8 JRF-E suggestions from counterexample and acquiring history analysis for (The Java Grande Forum benchmark suite 2012) examples with races

Test suite	Race field of class	Analysis
BarrierBench	IsDone[] of TournamentBarrier	**use atomic array for IsDone[]**
SynchBench	shared_count of CounterClass	make shared_count volatile, **synchronize on CounterClass**
lufact	IsDone[] of TournamentBarrier	**use atomic array for IsDone[]**
sor	sync[] of SOR	**use atomic array for sync[]**
	A[][] of RandomMatrix	**use atomic array for** A[][], **sync[]**
moldyn	IsDone[] of TournamentBarrier	**use atomic array for IsDone[]**
montecarlo	UNIVERSAL_DEBUG of Universal	make UNIVERSAL_DEBUG volatile**

*__bold__ entry indicates the most appropriate solution

**This is a benign race in redundant writes

Figure 31 compares the time and memory consumed in JPF, JRF, JRF-E respectively. JRF and JRF-E were configured to use a threshold trace of one and JPF was forced to stop at the same state number. The graph shows that in most cases JRF-E adds moderate overhead for both time and memory. (Interestingly, JRF-E in `Peterson` and `LinearSenseVolatileBarrier` outperforms JRF. One possible explanation would be the underlying java runtime environment behavior such as garbage collection had consumed more resources in JRF than JRF-E.) The standalone JRF was incomparable for one threshold configuration due to the disk access overhead. However, when the threshold grows, this performance gap reversed. We can conclude that the JRF-E is feasible to apply to a simple program without many races, but more complicated programs would be better to use with standalone mode to allow more traces and flexibility in analysis. This graph demonstrates the relative overhead JRF-E added on top of JRF and JPF. Scalability was always the problem in model checking and acceptable overhead is essential in this tool. From this point of view, the value added by JRF-E was valuable enough to compensate the time and memory spent for it.

5 Related work

Many, many, tools have been developed to detect data races, statically or dynamically, using a variety of definitions of a data race. As mentioned in the introduction, most of these tools use a slightly different notion of a data race that has nothing to do with a relaxed memory model.

The tool most closely related to JRF is Goldilocks (Elmas et al. 2007). Goldilocks is a dynamic analysis tool using an algorithm based on a relation that is very similar to the inverse of h. In other words, the Goldilocks algorithm maintains a function

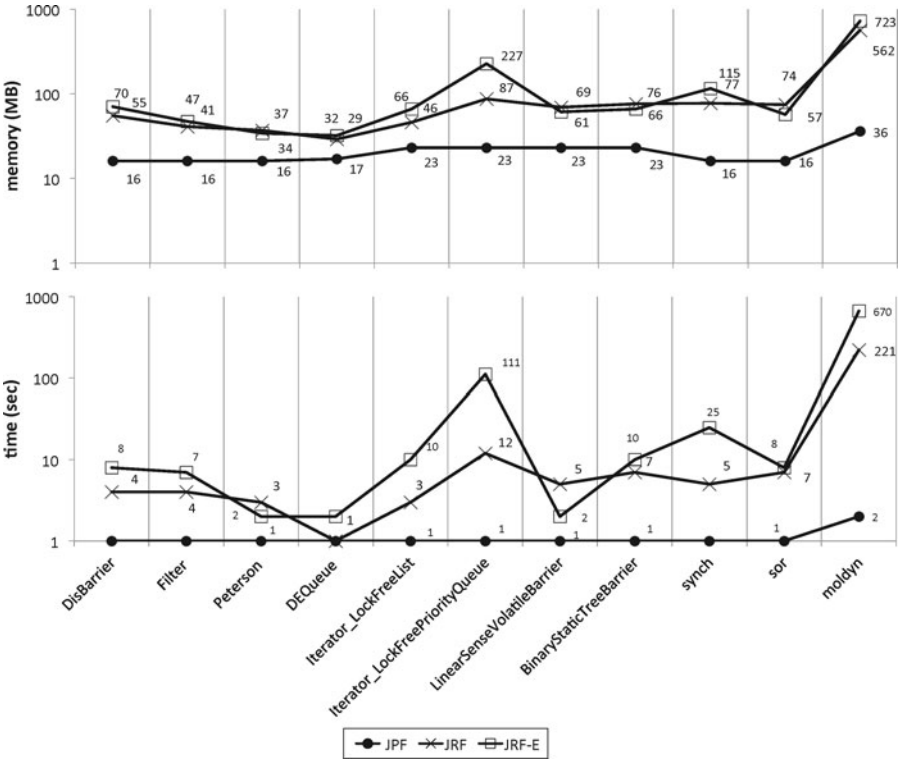


Fig. 31 The temporal and spatial performance comparison of JPF, JRF, JRF-E when JRF and JRF-E are configured to finish at one race trace and JPF is configured to stop at the same state

for each variable that indicates which threads can access the variable. As with all tools performing dynamic analysis, the required instrumentation of the program may change its behavior and the tool is limited to analyzing paths that happen to be tested. FastTrack (Flanagan and Freund 2009) is a more recent dynamic data race detector based on an optimized representation of vector clocks that is precise and claims to be more efficient than Goldilocks. Neither of these offer suggestions for fixing errors.

Recently, several studies have incorporated memory model awareness into model checking. For example, the Sobor tool (Burckhardt and Musuvathi 2008) considers programs assumed to be executing on hardware where the memory model is relaxed using store buffers. It can detect the presence of sequentially inconsistent executions and uses bounded model checking and a stateless model checker, CHESS (Musuvathi et al. 2007). The algorithm uses vector-clocks to capture the happen-before relation. Store buffers are much simpler than the JMM. The JMM is given as part of the programming language semantics constrains or allows compiler optimizations in addition to the low level reordering caused by store buffers. Also, since JPF is a state-based model checker, we can store the happen-before information for each state.

The tool described by Huynh and Roychoudhury (2007) considers the C# memory model and performs invariant checking at the bytecode-level using state-based model checker tailored for C#. The memory model is defined by specifying allowed

reorderings rather than based on happens-before consistency. The tool does not find data races, rather it performs the reorderings allowed by the memory model and verifies that a particular invariant holds, even on the sequentially inconsistent executions. Also, it provides simple program modifications in the form of inserting memory/barrier fences to eliminate data races. This type of suggestion is similar to our changing a non-volatile to a volatile. However, our tool provides a richer set of suggestions.

De et al. (2008) introduce a new memory model for Java called OpMM that allows some sequentially inconsistent executions, but is an underapproximation of the actual JMM. They have implemented a model checker that generates these sequentially inconsistent executions and checks that program properties are satisfied. Since the tool is an underapproximation of the JMM, it can only be used for bug-finding not verification. JRF can verify sequential consistency of a Java program without generating all such subsets but does not deal with sequentially inconsistent programs (except in the limited sense of allowing the user to specify that certain races that the use considers to be benign are ignored).

Several works compare a set of successful traces with a set of erroneous ones to localize the errors or to focus the debugging process on a relatively small part of the program. Groce and Visser (2003) considers both transition and invariant differences on successful traces and the counterexample paths. It provides feedback on how successful traces can be transformed into counterexample paths. However, our analysis provides feedback on how to transform a counterexample path to a possibly successful trace. Ball et al. (2003) generate multiple error traces having independent causes and for each error cause reports a single error trace. Brun and Ernst (2004) use dynamic analysis and machine learning to classify program properties as fault-revealing and non fault-revealing and report program invariants that are in the fault-revealing set. Basu et al. (2004), Flanagan et al. (2008) focus on error traces only. Basu et al. (2004) slice a counterexample path to find the statements that directly or indirectly affect the failure. Flanagan et al. (2008) compute the transactional happens-before edges on the dynamically generated execution traces to detect blocks that cannot preserve their atomicity and hence cannot be serialized.

6 Conclusion

Our data race detection tool, JRF is an extension of JPF that precisely detects data races, as defined by the Java Memory Model, in Java bytecode. This is important since standard JPF is unsound for programs that contain data races. Because it is based directly on the Java memory model, JRF can handle all concurrent programming idioms supported by Java, including lock-free algorithms. In this paper, we described JRF-E, an extension of JRF that analyses the counterexample path and acquiring history and provides suggestions for eliminating data races. The usefulness of the suggestion facility was evaluated by applying to a number of examples. Appropriate suggestions were provided in most cases, indicating that JRF-E can be a practical tool. Note that the suggestions provided by JRF-E may have side effects. For example, move source statement can change program semantics and results in other property violation in

new search path enabled by the change. New deadlock or livelock can be introduced when additional synchronization block is used. This would result in longer waiting time and even some existing interleaving would be disabled with additional locking. When a memory is changed to volatile or atomic array, certain compiler optimization would be prohibited and results in poorer performance. The validation and correctness of the suggestion is left to the programmer and it is highly recommended to use JPF again to evaluate the correctness and performance after the suggested modification. Even with this limitation, by explaining data races and providing a correct set of choices to fix them, JRF-E is a valuable addition to JRF.

References

Amino concurrent building blocks (2012). http://amino-cbbs.sourceforge.net/

Aspinall, D., Sevcik, J.: Formalising Java's data-race-free guarantee. In: TPHOLs 2007. LNCS, vol. 4732, pp. 22–37. Springer, Berlin (2007)

Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. In: Principles of Programming Languages, pp. 97–105 (2003)

Basu, S., Saha, D., Smolka, S.A.: Localizing program errors for simple debugging. In: FORTE, pp. 79–96 (2004)

Brun, Y., Ernst, M.D.: Finding latent code errors via machine learning over program executions. In: ICSE, pp. 480–490 (2004)

Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: CAV '08: Proceedings of the 20th International Conference on Computer Aided Verification, pp. 107–120. Springer, Berlin, Heidelberg (2008)

De, A., Roychoudhury, A., D'Souza, D.: Java memory model aware software validation. In: PASTE '08: Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp. 8–14. ACM, New York, NY, USA (2008)

Elmas, T., Qadeer, S., Tasiran, S.: Goldilocks: a race and transaction-aware Java runtime. In: PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, NY, USA, pp. 245–255. ACM Press, New York (2007)

Flanagan, C., Freund, S.N.: FastTrack: efficient and precise dynamic race detection. In: Programming Language Design and Implementation (2009)

Flanagan, C., Freund, S.N., Yi, J.: Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In: PLDI, pp. 293–303 (2008)

Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., Lea, D.: Java Concurrency in Practice. Addison-Wesley Professional, Reading (2006)

Google concurrent data structures workshop barriers (2012). http://code.google.com/p/concurrent-data-structures-workshop-barriers/

Gosling, J., Joy, B., Steele, G., Bracha, G.: Java Language Specification, 3rd edn. Addison-Wesley, Reading (2005)

Groce, A., Visser, W.: What went wrong: explaining counterexamples. In: SPIN Workshop on Model Checking of Software, pp. 121–135. Springer, Berlin (2003)

Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann, San Mateo (2008)

Huynh, T.Q., Roychoudhury, A.: Memory model sensitive bytecode verification. Form. Methods Syst. Des. **31**(3), 281–305 (2007)

Java Pathfinder (2012). http://babelfish.arc.nasa.gov/trac/jpf

Kim, K.H., Yavuz-Kahveci, T., Sanders, B.A.: Precise data race detection in a relaxed memory model using heuristic-based model checking. In: Proceedings of the 24th ACM/IEEE Conference on Automated Software Engineering (2009a)

Kim, K.H., Yavuz-Kahveci, T., Sanders, B.A.: Precise data race detection in a relaxed memory model using model checking. Technical report Rep-2009-480, University of Florida (2009b)

Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **C-28**(9), 690–691 (1979)

Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, NY, USA, pp. 378–391. ACM Press, New York (2005)

Musuvathi, M., Qadeer, S., Ball, T.: Chess: a systematic testing tool for concurrent software. Technical report MSR-TR-2007-149, Microsoft Research, (2007)

The Java Grande Forum benchmark suite (2012). http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html

Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. **10**(2), 203–232 (2003)