JAVA MEMORY MODEL-AWARE MODEL CHECKING

By

HUAFENG JIN

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2012

I dedicate this to everyone that helped me in this dissertation.

ACKNOWLEDGMENTS

First of all, I would like to say thank you to my Ph.D advisor Dr. Beverly Sanders. It is my great honor to be her student. During the years of research, she gave me great instructions on how to look for research topics, how to read papers, how to solve problems, how to write good academic literatures, how to give presentations, and how to communicate with fellow researchers. I appreciate all her contributions in giving me research inspirations, and opportunities to attend academic confererences. Most importantly, she provided me an excellent example of rigorous computer scientist.

In regards to software model checking, I would like to thank Dr. Tuba Yavuz-Kahveci. She helped me a lot in understanding the model checking concepts. Moreover, she provided me many practical advices in the implementation details. I appreciate all the time she spent in my project.

I would also like to thank Dr. Kyunghee Kim, the former Ph.D student of Dr. Sanders. She shared her practical experience with me in Java Pathfinder without any reservation. Even after graduation, she continued answering my questions through email. I also appreciate her previous work on Java Racefinder (now jpf-racefinder). This tool is very helpful in determining whether the legal executions of a program are sequentially consistent or not, and hence may speed up the verification process. The open sourced tool provided me a good example on how to extend Java Pathfinder.

For the Java Pathfinder tool, I would like to thank all the developers from NASA, especially Dr. Neha Rungta. The tool provides an important way to verify real life concurrent Java programs, and gave me inspirations in my work. For the detailed Java Pathfinder usage issues, I am particularly indebted to Peter Mehlitz, who is in charge of Java Pathfinder projects. He answered my questions with great patience in the online JPF group.

I appreciate Dr. Stephen Thebaut's help very much. My basic understandings on software validation and verification was acquired from his graduate level course

on software testing/verification. He even gave me an opportunity to be his teaching assistant in this course, through which I learned even more on this subject.

For the dissertation, I would like to thank Dr. Randy Chow and Dr. Rick Smith. They provided me very insightful suggestions during my admission to candidacy. I also want to thank all the committee members in taking their time to read my dissertation.

I gratefully acknowledge the department of Computer and Information Science and Engineering. During my Ph.D studies, I was appointed as teaching assistant for 9 straight semesters and received full tuition waiver. The generous financial support from the department made my Ph.D work possible. Moreover, being a teaching TA, I learned how to express ideas to the audience and how to help the students.

I owe a lot to the department staffs, John Bowers, Joan Chrisman, Ernest Hall, Rachel Ngai, and formerly Keri Taylor, Crystal McJunkin, Matt Williams. They helped me a great deal in the paperworks, course registrations, software installations, as well as other everyday life aspects.

My graduate life at University of Florida was enjoyable and unforgetable mostly because of my friends and roommates, Xiaochun Xu, Xiao Li, Yang Jiao, Xiao Yu, Navya Kooram, Mariah Griner, and many others. I am grateful for the time spent with them.

Last but not least, I would like to say thank you to my parents for their continuous support and encouragement during my Ph.D studies. Most of all, I appreciate my wife Shan, without the love and support from whom it is difficult for me to achieve my goal.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

JAVA MEMORY MODEL-AWARE MODEL CHECKING

By

Huafeng Jin

May 2012

Chair: Dr. Beverly A. Sanders
Major: Computer and Information Science and Engineering

The Java memory model (JMM) determines whether an execution of a concurrent Java program is legal or not. For programs that are data race free, JMM guarantees that all the legal executions are sequentially consistent. For the programs with data races, the legal executions may be sequentially inconsistent, but are still subject to constraints that ensure weak safety properties. Occasionally, one allows programs to contain data races to improve performance. These constraints make it possible, in principle, to reason about the correctness of programs. If the data races do not affect the correctness of the program, we call them benign data races.

Model checking is generally applied to determine whether a program meets its specification. For example, Java Pathfinder (JPF) is a model checker for Java programs. However, most model checking tools, including JPF, only generate sequentially consistent executions, but not executions that are sequentially inconsistent. Therefore they are not sound to reason about programs with data races. But original JMM is not operationally defined and is difficult to be implemented in model checkers. We give an alternative semantics for the JMM that characterizes the legal executions as a least fixed point and show that this is an overapproximation of the JMM. We have extended Java Pathfinder to generate these executions, yielding a tool, Java PathRelaxer, that can be soundly used to reason about the correctness of programs with data races.

CHAPTER 1
INTRODUCTION

Most modern computer architectures allow programs with more than one concurrent thread. The threads communicate with each other by either message passing mechanisms or a single shared memory address space. The concurrency provides higher performance compared with uniprocess computers, but also raises many issues, both for programmers and for system designers, especially with shared memory architectures. In a shared memory architecture, multiple processes may access to the same address space simultaneously. A particular question then, is when a thread reads from a particular address, which value will it see? And once that is specified, what kinds of optimizations and transformations can be carried out by the underlying architecture (i.e. hardware, compiler) without violating this specification. To answer these questions, a concept called a *memory model* came into being.

A memory model defines how memory operations in a concurrent program may execute, or how processes interact with the shared memory. In other words, *it determines what values a process may see when reading from a shared memory location*. In a uniprocess program, the read action always returns the value of the latest write action in the order specified by the program (we call it program order). But it is more complicated in concurrent programs because of interference from other processes.

A memory model could be specified at the either the hardware or programming language levels. At the hardware level, Dubois et al. [28] discussed memory model on shared memory multiprocessors. Adve and Hill [1] proposed a weakly ordered hardware memory model. Gharachorloo [34] specifies a memory consistency model, process consistency (PC) model for multiprocessor architectures. The SPARC manuals [84, 85] define three memory models, total store order (TSO), partial store order (PSO), and relaxed memory order (RMO) for SPARC-V9 architectures. In programming language realms, Stärk and Börger [86] presented a .NET memory model for multithreaded C#

Initially, x == 0

Thread 1	Thread 2
x = 1;	
	x =2 ;
r = x;	
	x = 3;

Figure 1-1. Memory model defines which value a read action could see; In strict memory models, only 2 could be seen, but in some other memory models, either 1, 2, or 3 could be seen by the read.

applications. Boehm and Adve [12] described a concurrent C++ memory model. Batty et al. [11] established a mathematical semantics for C++ memory model. Cohen et al. [24] came up with an efficient memory model for C. Pugh et al. initially pointed out the fatal flaws in the original Java memory model in [80, 81], and Manson et al. [59, 60] proposed a new version of the Java memory model which has been included in Java Language Specification (JLS) [36, §17.4].

A memory model has important impacts on both programmers and system designers. From the programmers' point of view, the memory model tells them which executions are legal and which are not. They may interpolate the possible outcomes of the program based on the memory model, through which they could reason about the program correctness with regard to the specification. See the execution sequence of a concurrent program shown in Fig. 1-1, the memory model tells which value could the read action see. In certain strict memory models, only 2 could be seen; but in some other memory models, either 1, 2, or 3 could be seen by the read. A program may be correct under one memory model, but incorrect under another memory model. For example, the famous Peterson's algorithm [77] which guarantees mutual exclusion for concurrent programs under sequentially consistent memory model, fails to provide mutual exclusion under many relaxed memory models such as JMM, TSO, PSO, etc. The situation is similar to Dekker's algorithm [27, §2.1].

Initially, x == y == 0

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

Figure 1-2. Memory model may prohibit some compiler optimizations.

System designers apply numerous hardware or compiler optimizations or transformations to improve the efficiency of the system. Memory model tells them which optimizations or transformations can be carried out and which cannot. They have to keep the memory model in mind when designing the system. See the execution in Fig. 1-2, in a single-threaded program with only Thread 1, a redundant read elimination transformation could be applied by the compiler to improve the performance; the second read of x can be replaced by r2 = r1, so r2 = 0. But with the interference from Thread 2, this redundant read elimination is prohibited by some memory models, such as the sequentially consistent memory model, which only allows a read to return the value (1 in this case) of the most recent write in an execution.

Basically speaking, memory model serves as a bridge between programmers and system designers. It must be easy enough for the programmers to understand, and it must also be not difficult for system designers to comply with when designing underlying architectures. Memory models should strike the balance between these two aspects. Typically, “strict” memory models are easier to understand but difficult to implement, while “relaxed” memory models are just the opposite.

Among those, the simplest, and most commonly assumed memory model is *sequentially consistent (SC) memory model* [51], in which read actions can only return the value of the most recent write action along a certain execution path. SC memory model is easy to understand by programmers; only the “interleavings” of instructions are considered, otherwise just treat as a sequential program. SC memory model has

long been the implicit underlying assumption for most concurrent program analyzers, such as SPIN [73] and Java Pathfinder (JPF) [42]. However, SC memory model has many limitations. It restricts many very common optimizations and transformations that are carried out by modern hardware or compilers, as we saw in Fig. 1-2. Therefore, SC memory model is difficult to implement in reality.

To improve the execution performance, relaxed memory models are proposed. The PSO, TSO, partial store load order (PSLO) [5], properly labeled (PL) model [34], data-race-free (DRF) memory model [1], C# memory model, and JMM are relaxed memory models. Relaxed memory models allow compiler optimizations and transformations in certain degrees, and thus more behaviors are possible. Typically, if a memory model M_1 is "more relaxed" than M_2, then more behaviors are allowed by M_1 are more than allowed by M_2. A comparison between memory models can be found at [82].

The Java memory model is the first complete and widely accepted relaxed memory model for high-level programming languages. It is relaxed, which means it allows some optimizations and transformations from compilers. It also provides some constraints on the behavior of programs with data races. The JMM guarantees sequential consistency only if the program is data-race-free.

JMM is very comprehensive but it is still not perfect. Firstly, it still prohibits certain kinds of compiler transformations [92]. More importantly, it is declaratively and non-operationally defined, and is not straightforward to understand. To better understand JMM and reasoning about programs with data races under JMM, tool support is desirable. We describe a JMM-aware model checker, Java PathRelaxer (JPR), which is an extension of Java Pathfinder [42, 91] and generates all of the legal executions of finite Java programs with data races so that their properties can be verified. The way the JMM defines legal executions in programs with data races does not lend itself to precise implementation with a model checker and has been shown [92]

to be stricter than the designers intended. We use an alternate approach. Instead of defining a legal execution by the existence of a sequence of justifying executions as the JMM does, we compute a set of paths that is the least fixed point of a monotone function. We show that the set of paths generated by JPR is an overapproximation of the set of legal executions. Although the details of the formalization and implementation of JPR are specific for Java, the main ideas are applicable to other languages with a memory model based on the happens-before relation.

The main contributions of this work are

- A new, fixed-point based, approach to the characterization of legal executions for relaxed memory models.
- A tool, JPR that generates all of the legal executions according to the fixed-point characterization.
- A proof that the fixed-point based approach is an overapproximation of the JMM, and thus JPR is sound for Java programs with data races.
- Insights into how the JMM maps (or does not map) into program constructs.

The rest of the thesis is organized as follows: Chapter 2 introduces some useful theoretical backgrounds, some well-known memory models, the formal definition of JMM, the relationship between data race and program correctness, and model checking. Chapter 3 describes the core algorithm of JPR in detail. Chapter 4 formally proves that the algorithm generates an overapproximation of JMM. Chapter 5 presents the implementation issues related to JPR . Chapter 6 summarizes the experimental results and discusses the possible extension of the idea onto other relaxed memory models. Chapter 7 lists some of the related works. Finally, Chapter 8 gives a conclusion.

CHAPTER 2
BACKGROUND

This chapter introduces some theories and technologies used in the work. Before dipping into the details of Java memory model, we firstly introduce some of the well-known memory models; an easy but strict memory model, SC memory model, and two other relaxed memory models, PSO and TSO. After the formal description of Java memory model, we discuss the relationship between data race and program correctness. Finally, we talk about model checking and Java Pathfinder tool, which is basis for JPR.

2.1 Memory Models

As discussed in Chapter 1, a memory model defines how processes or threads interact with the shared memory. It determines which value does a given read action may see in an execution. In this section, we first introduce the widely known SC memory model; then talk about two relaxed store buffer-based memory models, PSO and TSO. With these memory models in mind, it would be easier to understand JMM.

2.1.1 Sequentially consistent memory model

Sequentially consistent memory model was first raised by Lamport in 1979. In SC memory model, “the result of an execution is the same as if the operations had been executed in the order specified by the program, and the operations of each individual processor appear in this sequence in the order specified by its program” [51]. This means that under SC memory model, the actions must appear one at a time, and in some total order which is consistent with the program order. In an execution sequence, a read action to a shared memory location only sees the value written by the most recent write action to the same memory location in that sequence.

Under SC memory model, programmers only need to consider instruction interleaving. Let’s see the execution sequence shown in Fig. 1-1, the actions appear one at a time, and the read action sees the most recent write, so the read of x in Thread 1

Initially, x == y == 0

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

Figure 2-1. SC memory model restricts the reordering of instructions 1 and 2, or 3 and 4, which are pairs of independent instructions within one thread. So r1 == 1 and r2 == 2 is prohibited.

can only see 2 because the write of 2 from Thread 2 is the most recent write action in the sequence. It cannot see either 1 or 3.

SC memory model is easy for programmers to understand. Given an execution sequence of a program, they can only get one outcome. Also, sometimes programmers don't need to use explicit synchronization mechanisms like locks to guarantee mutual exclusion, such as Peterson's algorithm.

Although SC memory model is an intuitive model, it restricts many common optimizations and transformations from both hardware and compiler. Reordering of memory operations is common for compilers. This may be a result of value caching, sub-expression elimination, etc. But SC memory model prohibits any kind of reordering of memory operations to shared locations, even if the operations have no control dependencies nor data dependencies. Consider the simple example shown in Fig.2-1 [60]. Lines 1 and 2 from Thread 1, as well as lines 3 and 4 from Thread 2, have no data nor control dependencies, so they might be switched by the compiler. In that case, r2 == 2 and r1 == 1 is a possible outcome. But in any sequentially consistent traces, this result is forbidden. We cannot find a total order of interleaved instructions that is consistent with program order to justify this result.

Another compiler optimization, redundant read elimination, which can be viewed as reordering, is also prohibited by SC memory model. Consider the example shown in Fig.2-2 [36, §17.3]. In any sequentially consistent trace, depending on how the threads interleave, the x field of the single object involved would change from 0 to 3 at some

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

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

Figure 2-2. SC memory model restricts redundant read elimination of replacing r5 = r1.x with r5 = r2.

Initially x == y == 0

Thread 1	Thread 2
x = 1;	if(x == 1){
if(y == 1)	x = 0;
print x;	y = 1;
	}

Figure 2-3. Under SC memory model, “1” cannot be printed out.

point and then remain 3 thereafter. If we apply a redundant read elimination on the program, replacing the last read r1.x in Thread 1 with r5 = r2. Then the value of r1.x would change from 0 to 3 and then back to 0. But such a trace is not sequentially consistent. See another example (Fig. 2-3 [92]), under SC memory model, the program can never print out “1”. Because if the print ever executes, the latest write to x is x = 0. However modern compilers may treat the read in the “print x” as a redundant read and replace it with “print 1”.

The restriction of common compiler optimizations/transformations is a major drawback of SC memory model. The implementation of SC memory model is very expensive. This significantly affects the performance of program execution. To overcome this drawback, relaxed memory models are proposed, such as weak ordering model in [1], release consistency model in [34], location consistency model in [33], partial store order (PSO) and total store order (TSO) [85]. Relaxed memory model allows

more compiler optimizations and transformations. The memory models for high level programming languages (Java, C, C++, C#) are all relaxed.

2.1.2 Partial store order and total store order

Partial store order (PSO) and *total store order* (TSO) are two of the three memory models for SPARC architectures [85] (TSO is also supported by X86 processors [75]). Both of them are based on *store buffers*. They allow more hardware optimizations than SC memory model, and are hence relaxed memory models. In store buffer based memory models, each process is associated with a list of first-in-first-out (FIFO) buffers (called "store buffers"). The write action does not write directly to the shared memory location, but instead writes to the corresponding store buffers associated with the process. This phase is called *store*. After some non-deterministic time, a separate *flush* phase commits the values in a store buffer to the main memory in an FIFO manner. The read action (called "load"), on the other hand, retrieves value from the store buffer before referring to the main memory.

The TSO memory model architecture is shown in Fig. 2-4 (derived from [84, §K.2]). Each process is associated with an FIFO store buffer. The store operation puts the value into the store buffer. The values in the store buffer are eventually flushed to the shared main memory in the same order as they were put in the buffer. The load operation gets the most recent value from the store buffer of the corresponding process. If the value doesn't exist, it then accesses the main memory to get the value. The PSO memory model is similar to TSO but "performance-enhanced". In PSO, each process maintains a set of FIFO store buffers, with each store buffer associated to a memory location.

If we use p_i to describe process, x to denote variables, and v to denote a value, then an informal operational semantics of PSO memory model is as follows:

- $store(p_i, x, v)$: put v to the store buffer associated with p_i and x
- $load(p_i, x)$: get the latest value from the store buffer associated with p_i and x, if it is empty then get the value of x from the main memory.

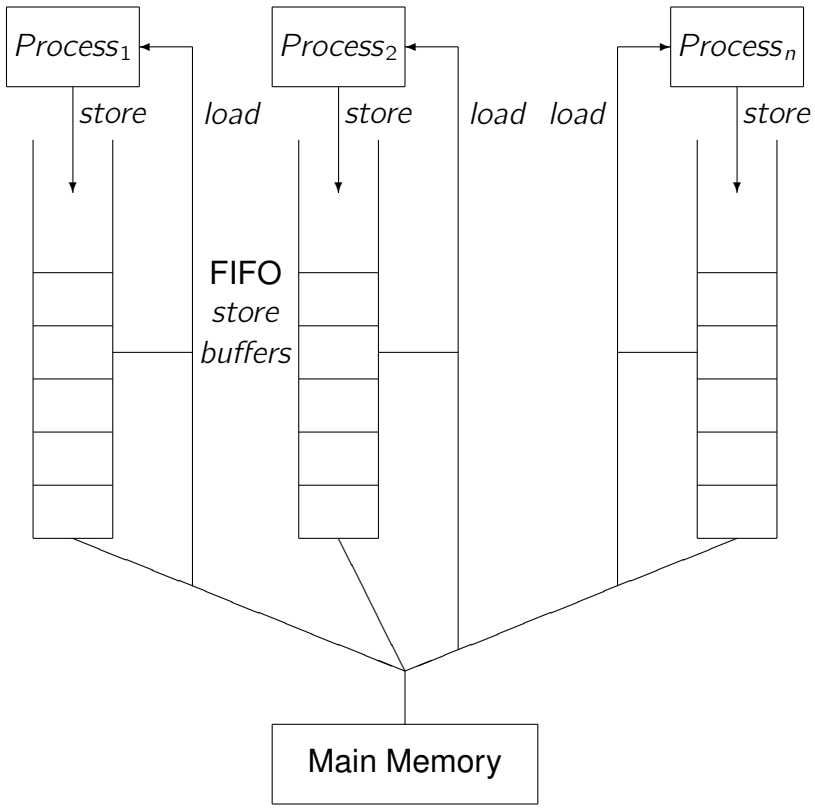

Figure 2-4. TSO memory model architecture.

- $flush(p_i, x)$: commit the oldest value of store buffer associated with p_i and x to the main memory and remove it from the store buffer.

Besides store, load, and flush, processors also provide a $fence$ instruction to allow programmers to enforce ordering of memory operations. The strongest fence can be viewed as:

- $fence(p_i)$: for each store buffer of p_i, if it is not empty, force flushing from store buffer to the main memory.

With the operational semantics, PSO memory model guarantees the following partial-coherence properties [50]:

- Intra-process coherence: A process should only see the most up-to-date value written by itself to a variable.
- Inter-process coherence: A process should see the values written by another process in the order they were written.

Process 0	Process 1

while(true){
store ent0 = true;
store turn = 1;
do{
load e = ent1;
load t = turn;
}while(e==true&&t==1);
//Critical Section
store ent0 = false;
}
	while(true){
	store ent1 = true;
	store turn = 0;
	do{
	load e = ent0;
	load t = turn;
	}while(e==true&&t==0);
	//Critical Section
	store ent1 = false;
	}

Figure 2-5. Peterson's algorithm doesn't guarantee mutual exclusion under PSO.

- Fence coherence: A fence writes the most up-to-date values written by the process to the main memory.

PSO and TSO memory models have less strict semantics than SC memory model. Because of the store buffers, the value written by a process may not be instantly visible to other processes. A read action may see an old value rather than the up-to-date value. So for Fig. 1-1, the read may see either 1 or 2. And the redundant read elimination in Fig. 2-3 is allowed.

Store buffer memory models allow more optimizations than SC memory model, but some programs that work fine under SC memory model now have problems. See the Peterson's Algorithm with explicit memory operations shown in Fig. 2-5[50], it guarantees mutual exclusion under SC memory model. When a process is entering the critical section, it loops over until the *ent* value of the other process is false, so that the two processes cannot access the critical section at the same time. But under PSO, the algorithm doesn't guarantee this property; in the presence of the store buffer, the load for *ent* of the other process may not return the most recent value, so both processes may enter at the same time. To make Peterson's Algorithm work correctly under PSO, certain fence operations (i.e. force flush of store buffers) should be inserted after appropriate position.

Initially, x == 0, done == false

Thread 1	Thread 2
x = 1;	while(!done){/*spin*/}
done = true;	r = x;

Figure 2-6. PSO allows more behaviors than SC memory model: r may read 0, not 1.

Also look at the example shown in Fig. 2-6, under SC memory model, if the read in Thread 2 ever executes, it can only read 1. But under PSO, the value 1 in Thread 1 may be written to the store buffer, not the memory, so Thread 2 may read the old value 0 instead.

Although PSO and TSO are more relaxed than SC memory model, some common optimizations are still restricted. The reordering mentioned in Fig. 2-1 is not permitted. In any execution sequence, we cannot get $r1 == 1 \&\& r2 == 2$. Also value 3 cannot be seen by the read in Fig. 1-1.

2.2 The Java Memory Model

Java memory model (JMM) serves as the core concept of this work. JMM is a relaxed memory model for Java. It is the first attempt to formalize a memory model for high level languages. JMM has encouraged other high level languages, such as C++ and C#, to design their own memory models. It also has great impact on hardware [64].

JMM allows many common optimizations and transformations. Sequential consistency is not always guaranteed in the executions. But JMM provides a sequential consistency guarantee for programs that are correctly synchronized, i.e. programs without data races (data-race-free programs, or *DRF* programs).

JMM is based on the definition of well-formed execution. To determine whether an execution is legal under JMM, we must first ensure that it is well-formed. The second step is to apply the non-operational causality requirement rules to justify the execution. If the execution can be justified by JMM's causality rules, then it is JMM legal. Well-formed execution guarantees basic intra- and inter-thread consistencies, and causality rules

are designed to rule out "*out-of-thin-air*" results. Here, *causal* means data and control dependency causes.

Below is a brief overview of the formal definition of the Java memory model. The detailed specification is given in [36, 60]. We will follow a brief version described in [7][1] .

An action in Java memory model is a memory-related operation that belongs to a thread. The formal definition of *action* is:

Definition 1 (Action)**.** *An action a is represented by a tuple $\langle t, k, v, u \rangle$, where*

- *t represents the thread that the action belongs to*
- *k represents the kind of the action*
- *v represents the memory location (variable or monitor) involved in the action*
- *u is an arbitrary unique identifier for the action*

Here the action kind could be either volatile[2] read, volatile write, non-volatile read, non-volatile write, lock, unlock, and special synchronization actions such as thread start, thread termination detection, etc. All the actions except for non-volatile read and write are called synchronization actions.

Definition 2 (Execution)**.** *An execution E is described by a tuple $\langle A, P, \leq_{po}, \leq_{so}, W, V \rangle$ where*

- *A is a finite set of actions.*
- *P is a program.*

[1] The most important differences between [60] and [7] are that the latter requires that the total order for SC executions be consistent with both the synchronization order and program order (as opposed to just the program order, correcting an apparent oversight in the JMM formulation), formulates the semantics in terms of finite executions, and ignores external actions.

[2] In Java, variables declared with *volatile* keyword [36, §8.3.1.4] are ensured for mutual exclusion. Writes to volatile variables guarantees *visibility*, i.e. a read action performed on volatile variables always see the value written by the most up-to-date write action.

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

The synchronizes-with order ($\leq_{sw}$) is a partial order that relates certain pairs of synchronization action in A. For synchronizes-with order $a_1 \leq_{sw} a_2$, we categorize a_1 (the source) as *release action*, and a_2 (destination) as *acquire action*. This is a partial list of synchronizes-with relations from [36, §17.4.4]:

- An unlock action on m synchronizes-with all subsequent lock actions on m. [3]
- A write to volatile variable v synchronizes-with all subsequent read actions on v.
- A thread start action synchronizes-with the first action of the started thread.
- The write to the default value (i.e. 0, false, null) to the variable synchronizes-with the first action of each thread.
- The final action in thread $T1$ synchronizes-with any action in another thread $T2$ that detects $T1$ has terminated.
- If thread $T1$ interrupts thread $T2$, the interrupt synchronizes-with any point where any other thread determines $T2$ has been interrupted.

The happens-before order $\leq_{hb}$ is a transitive closure of synchronizes-with order and program order. Formally it is: $\leq_{hb} = (\leq_{sw} \cup \leq_{po})^+$. Consider the execution sequence shown in Fig. 2-7, according to the order specified by the program, we get the program order: $a_1 \leq_{po} release(m)$ and $acquire(m) \leq_{po} a_2$; the synchronizes-with order defines

[3] Here, "subsequent" is defined according to the synchronization order.

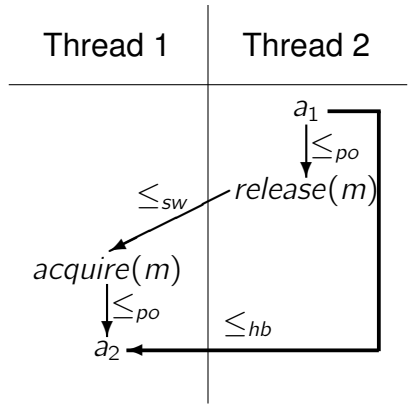

Figure 2-7. $\leq_{hb}$ is a transitive closure of $\leq_{sw}$ and $\leq_{po}$. We get $a_1 \leq_{hb} a_2$.

$release(m) \leq_{sw} acquire(m)$, so because of the transitive closure of happens-before, we get $a_1 \leq_{hb} a_2$.

2.2.1 Well-formed execution

JMM's well-formed execution satisfies type safety and some unsurprising consistency requirements on the various partial and total orders. The two most important rules for our purposes are intra-thread consistency and happens-before consistency.

Definition 3 (Well-formed execution)**.** *See [7, Definition 6] for the complete definition.*

1. *A is finite.*
2. *$\leq_{po}$ is a total order over actions in one thread.*
3. *$\leq_{so}$ is a total order over synchronizations in A.*
4. *$\leq_{so}$ is consistent with $\leq_{po}$.*
5. *W is properly typed.*
6. *Locking is proper: number of locks is the same as number of unlocks.*
7. *Intra-thread Consistency: Program order is intra-thread consistent. For each thread t, the sequence of action kinds and values of actions performed by t in the program order $\leq_{po}$ is sequentially valid[4] with respect to P and t.*

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

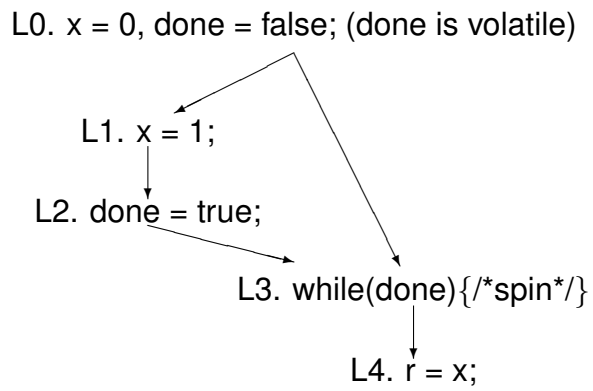

Figure 2-8. Under JMM, done == true && r == 0 is an impossible result.

8. *Synchronization Order Consistency:* $\leq_{so}$ *is consistent with* W*: For any volatile read action* r*,* $W(r) \leq_{so} r$ *and for any volatile write* w *such that* $w.v = r.v$*, either* $w \leq_{so} W(r)$ *or* $r \leq_{so} w$*.*

9. *Happens-before Consistency:* $\leq_{hb}$ *is consistent with* W*: for any read* r *of variable* v*,*

- $r \not\leq_{hb} W(r)$
- *there is no intervening write* w *to* v*, i.e. if* $W(r) \leq_{hb} w \leq_{hb} r$ *and* w *writes to* v *then* $W(r) = w$*.*

Among the well-formed execution rules, Rule 9, the happens-before consistency is the most important rule, others are obvious. It forbids a non-volatile read to see a write on the same variable that happens after it. And it also forbids a non-volatile read to see a write on the same variable that is happens-before it but with an interleaving write in between them. Let's look at the example shown in Fig. 2-6, suppose *done* is volatile, then if the read in Thread 2 ever executes, the execution sequence is Fig. 2-8. In the figure, happens-before edges are shown. According to the synchronizes-with relation rules listed in [36, §17.4.4], the writing to the default values happens-before the first action in each thread; and there is happens-before relation from the volatile write in Thread 1 to the volatile read in Thread 2. In this trace, if $W(L4) = L0$ (i.e. r == 0), then on the path $L0 \rightarrow L1 \rightarrow L2 \rightarrow L3 \rightarrow L4$, we have $W(L4) \leq_{hb} L1 \leq_{hb} L4$, where $L1$ is also

a write to the same variable x. This violates Rule 9 of Definition 3. So this execution is not well-formed.

2.2.2 Causality rules

In addition to the well-formed execution concept, JMM provides *Causality Requirements* or *Legality*. This is to rule out out-of-thin-air results. The idea is that a well-formed execution E is legal if there is (roughly speaking) a sequence of well-formed executions E_i with action sets A_i and a subset of actions C_i called the commit set where each committed read either sees a committed write or a write that happens-before it. It is required that $C_{i-1} \subseteq C_i$ and that the sequence eventually produces E with all of its actions committed.

Definition 4 (Legal Execution)**.** *[7, Definition 7]*[5] *of [60, §5.4]. A well-formed execution $E = \langle A, P, \leq_{po}, \leq_{so}, W, V \rangle$ with happens-before order $\leq_{hb}$ is legal if there is a finite sequence of sets of actions C_i and well-formed executions $E_i = \langle A_i, P, \leq_{po_i}, \leq_{so_i}, W_i, V_i \rangle$ with happens-before order $\leq_{hb_i}$ such that $C_0 = \phi$, $C_{i-1} \subseteq C_i$ for all $i > 0$, $\bigcup C_i = A$, and for each $i > 0$, the following are satisfied:*

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

Rules 6 and 7 are the most important rules. Rule 6 says, all the uncommitted read actions ($r \in A_i - C_i$) only see the writes that happens-before them. Rule 7 says, the

[5] There are two other rules, 8 and 9 in [60], but are omitted in [7] for brievity.

Initially, x == y == 0

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

Figure 2-9. r1 == r2 == 42 is an out-of-thin-air result, and is disallowed by JMM.

to-be-committed read actions ($r \in C_i - C_{i-1}$) must see writes that have already been committed in both E_i and E, but may see a different write in E_i from the one it sees in E. [60, Figure 8] shows an example of justifying JMM legal execution by applying causality rules listed above. Other than these two rules, the $\leq_{hb}$, $\leq_{so}$, and V in each justifying execution must be the same as the execution being justified.

The causality requirements are used to rule out "out-of-thin-air" values. A precise definition of out-of-thin-air values is complicated, but we can get the idea through looking at an example. Consider the example shown in Fig. 2-9 [60, §2.2], no matter what optimizations are applied, there is no way to bring the value 42 into the executions, so $r1 == r2 == 42$ is an out-of-thin-air result and should be forbidden. But "in a future aggressive system, Thread 1 could speculatively write 42 to y"[60], and then propagates 42 to x. However, this execution is well-formed according to Definition 3. There are no violations of any of the consistencies. Let's apply the causality rules to see this result is illegal under JMM. Suppose that we want to commit the write action $A2 : y = r1;$. Then $V(A2)$ is the value read in action $r1 = x$. The value of x must be obtained from a write that either happened-before $A1$ (the initialization action is the only option) or is already committed. In the former case, the value read is 0, in the latter case, it is the value written by $B2$. Similarly, the value written in $B2$ must be the value read in $B1$, which must be either committed or happen-before it. However, $A2$ was not committed, so the initialization action is the only option. Thus the only possible outcome is $r1 == r2 == 0$.

Sometimes, out-of-thin-air values are not as trivial as Fig. 2-9. Consider the example shown in Fig. 2-10 [41]. The result $r1 == r2 == 1, r3 == 0$ is a well-formed

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

Thread 1	Thread 2	Thread 3	Thread 4
r1 = x; y = r1;	r2 = y; x = r2;	z = 1;	r3 = z; x = r3;

Figure 2-10. r1 == r2 == 1, r3 == 0 is an out-of-thin-air result, and is disallowed by JMM.

Initially, x == y == 0

Thread 1	Thread 2

r1 = x;
r2 = x;
if (r1 == r2)
  y = 1;
	r3 = y;
	x = r3;

Figure 2-11. Under JMM, r1 == r2 == r3 == 1 is allowed.

result, but it is out-of-thin-air and should also be forbidden. In this example, the only way to bring value 1 to r1 and r2 is through the read of z in Thread 4. So in order to get $r1 == r2 == 1, r3$ must be 1. By applying the causality rules, there is no way to commit $r1 = x$ (read 1) without committing $r3 = z$ (read 1) first.

2.2.3 Evaluation of Java memory model

The Java memory model has a much more relaxed semantics than SC memory model. It allows more hardware or compiler optimizations and transformations, and more behaviors are allowed. Based on well-formed execution definition (Definition 3), a non-volatile read may see any value provided that the source write to that value is happens-before consistent with the read (Rule 9). A read may see either a value written by a write that happened before it (we call it *previous* write) or a value that to be written after it (we call it *future* write). For the execution sequence shown in Fig. 1-1, the read of x may see either 1 or 2 (previous write), or 3 (future write), and all the outcomes are JMM legal. Note that value 3 cannot be seen by the read under PSO. Also the redundant read elimination is allowed in Fig. 2-2 and Fig. 2-3. More interestingly, $r1 == 1 \&\& r2 == 2$ is allowed in the program shown in Fig. 2-1. The

Initially, x == y == 0

Thread 1	Thread 2

r1 = x;
y = r1;
	r2 = y;
	if (r2 == 1){
	  r3 = y;
	  x = r3;
	}
	else{ x = 1; }

Figure 2-12. Sometimes, the redundant read elimination is forbidden by JMM.

result can be proved by Definition 3 and Definition 4; the reads may see 1 and 2 after committing the writes. Note that this is restricted by both SC memory model and store buffer-based memory models. Even another example is shown in Fig. 2-11[8, 41], $r1 == r2 == r3 == 1$ is allowed by JMM, but prohibited by SC memory model. We may use the causality rules to justify it by first committing the write $y = 1$ in C_1, then committing the write $x = r3$ in C_2, and finally committing the two reads $r1 = x$ and $r2 = x$ at the same time. The complete justification sequence can be found at [41].

On the other hand, JMM still forbids many hardware and compiler optimizations and transformations though it is a well-known relaxed memory model. Ševčík and Aspinall identified some of these optimizations/transformations [92]. An interesting case is shown in Fig. 2-12[92]. In any JMM-legal executions, we cannot get $r2 == 1$; we cannot include $r3 = y$ in the justifying execution sequence. However if the compiler applies a redundant read elimination by replacing $r3 = y$ with $r3 = r2$, then we can get the result.

Although JMM forbids certain optimizations and transformations, it is currently a most widely recognized memory model for Java. The specification is already included in Java Language Specification [36, §17.4]. JMM also serves as a beacon in the formalization and construction of high level programming language memory models. Most notably [12] learned from JMM when designing the new C++ memory model, the C++0x.

2.3 Data Race and Program Correctness

Concurrent programs are complicated, and sometime difficult to debug. To help programmers to write better concurrent programs, various techniques are proposed. In software engineering, verification means using methods to check whether a program satisfies some requirements. If a program satisfies its specification, it is considered as a "*correct*" program.

To verify the correctness of sequential programs, we may use the *Hoare Logic* [39], in which the program specification is abstracted in terms of preconditions (P), postconditions (Q), and invariants (I). Suppose S is a statement, then we have the notation of $\{P\}\ S\ \{Q\}$. Programmer may formally verify the program by applying the axioms and inference rules provided by Hoare logic. However for concurrent programs, it is difficult to apply hoare logic to verify the program. The best known attempt is [76], where a *non-interference rule*[6] was proposed. But the rule is based on a strict *interleaving model*, in which all the actions from different processes are executed in an arbitrary sequential order.

Aside from the difficulty in verification of the correctness of concurrent programs, research in concurrency has long been focusing in another aspect of concurrent programs, the detection of *data race*. Program that contains data races is often erroneous.

2.3.1 Data race

What is data race? A data race, informally speaking, is a condition where two accesses from different threads accessing the same shared memory location, with at least one of them being a write. And there is no explicit mechanism to prevent the accesses from being simultaneous [83]. Data race is very common in multithreaded

[6] This rule is also summarized in [6, §2.3]

programs. In many cases, program errors are generated from data races, so data race is usually considered to be a symptom of bug.

For a long time, the concurrent program analysis, both dynamically and statically, has been focusing on the detection of data races. Savage et al. [83] introduced a tool called Eraser to dynamically detect data races in concurrent programs. Choi et al. [18] raised a dynamic data race detecting approach for multithreaded object-oriented programs. Naik et al. [66] proposed a static approach in detecting data races in concurrent Java programs. Flanagan and Freund [30] presented a static data race analysis for Java programs based on a type system. Pratikakis et al. [79] proposed a tool called LOCKSMITH for reasoning data races in C programs. O'Callahan and Choi [72] presented a dynamic method that combines lockset-based detection and happens-before-based detection. Chrisiaens et al. [19] introduced a tool called TRaDe that uses topological approach. It can detect data races in Java programs. Kahlon et al. [46] proposed a context-sensitive analysis to detect data races.

Despite the active research in data race for decades, the data race itself is a vague concept. There is lack of precise definition of data race. Many papers used their own definition. Unlike other memory models, Java memory model has its formal and precise definition for data race [60]. The JMM data race is based on two concepts: 1) *conflicting accesses* and 2) happens-before order. The conflicting accesses are accesses to the same shared memory location with at least one access is a write.

Definition 5 (Data Race)**.** *Two accesses x and y form a data race in an execution of a program if*

- *they are from different threads*
- *they conflict*
- *they are not ordered by happens-before partial orders*

Based on Definition 5, the execution shown in Fig. 2-8 is free of data races. For the two shared variables, x and $done$, any pairs of conflicting accesses are clearly ordered

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
if (r1 != 0)	if (r2 != 0)
y = 42;	x = 42;

Figure 2-13. Correctly synchronized (DRF) program, $r1 == r2 == 0$ is the only possible outcome.

by happens-before ($L0 \leq_{hb} L2 \leq_{hb} L3$ and $L0 \leq_{hb} L1 \leq_{hb} L4$). But if *done* is not volatile, then the happens-before from $L2$ to $L3$ is missing, and the execution contains data races.

From Definition 5, we see that JMM's definition on data race is based on execution, not program. In addition, JMM also provides a definition for data-race-free program:

Definition 6 (Data-Race-Free (DRF) Program)**.** *[2, 3, 60] A program is said to be correctly synchronized or data-race-free if and only if all sequentially consistent executions of the program are free of data races.*

This definition tells us if we can enumerate all the SC executions of a program, and any pairs of conflicting accesses are ordered by happens-before, then the program is DRF program. Fig. 2-6 is DRF if *done* is volatile. Also the program in Fig. 2-13[60] is DRF; there are no synchronizations in the program, but in any SC executions, the two *if* statements are not executed, so there are no data races in the SC executions. On the contrary, program in Fig. 2-1 is not DRF.

Definition 6 provides a nice guideline for data race detection. JRF [48] is an attempt to use model checking method to detect data races under JMM.

In real life Java programs, data race can be very subtle. A problem called *safe publication* is a practical case of data race. *Publishing* an object means "making it available to code outside of its scope." [35] When instantiating a Java object, if the reference is visible by a thread other than the thread that creating it, and the thread sees a partially constructed object, then this publication is an unsafe publication. The reason

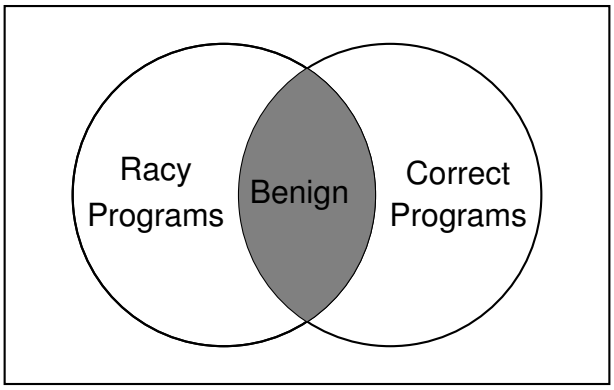

Figure 2-14. The relationship between racy programs, correct programs, and programs with benign data races.

for reading partially constructed object is the lack of happens-before ordering between the object creation and the read of the reference. The famous *double-checked locking* idiom [9] contains data race and suffers unsafe publication problem.

2.3.2 Program correctness and benign data race

What properties does a DRF program have? Aspinall and Ševčík [7] proved that any legal execution E of a well-formed execution of a DRF program is sequentially consistent.

Theorem 2.1 (DRF Guarantee)**.** *Any legal execution E of a well-formed data race free program is sequentially consistent.*

This theorem implies that JMM guarantees sequential consistency for programs that are data race free. Sequential consistency can be understood by programmers easily, and sequentially inconsistent programs are often erroneous, such as the Peterson's algorithm in Fig. 2-5.

Although data race is very likely to lead to unintened errors, data race isn't equal to program incorrectness, and data race free doesn't necessarily imply program correctness. Under JMM, data race free only guarantees sequential consistency, not correctness. We say a program is correct if and only if the results meet its specification. Some data races are actually "*benign*"; the presence of data race doesn't affect the correctness of the program. We call the data races that may lead to errors are *harmful*

balance is shared, and is volatile

Thread 1	Thread 2

r1 = balance;
if (r1 > 1000){
  r1 = r1 - 1000;
  balance = r1;
}
	r2 = balance;
	if (r2 > 1000){
	  r2 = r2 - 1000;
	  balance = r2;
	}

Figure 2-15. Sometimes DRF program is erroneous.

data races, while the races that won't affect the correctness are *benign data races*. The relationship of racy programs, correct programs, and programs with benign data races is shown in Fig. 2-14, where the shaded part between racy programs and correct programs contains programs with benign data races. They are correct but contain data races.

On the other hand, some DRF programs are incorrect. See Fig. 2-15, suppose *balance* represents a bank account balance, two threads are trying to withdraw from the same bank account. In this case, even if *balance* is volatile, two threads may read it at the same time and only one 1000 is deducted from *balance*. This program is correctly synchronized, but suffers from atomicity problem.

Benign data race is very common. An example of benign data race is Fig. 2-9. The program is not DRF; in any SC executions the write of x and the read of x are not ordered by happens-before. But if the specification is "in any execution, we can get $r1 == r2 == 0$", then the program is correct, because $r1 == r2 == 0$ is the only possible outcome of this program.

Also see the example shown in Fig. 2-16. This is the source code of Java's String class. The fields of *value*, *offset*, and *count* are declared as final[7] , so no

[7] In Java, a final field may only be given a value in the initializer and classes with all of their fields final are considered immutable. Final fields also have special semantics with respect to the memory model: roughly speaking, provided that the "this" reference does

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

Figure 2-16. Benign data race example: Java's String class. No matter how many threads are running hashCode() method, the correct hash code will always be returned.

```
public class test{                    // 3105 is the desired hash code for string "ab"
        static String str = "ab"
        public static void main(String args[]){
                new Thread(new Runnable(){
                        public void run()
                        {int r1 = str. hashCode(); assert(r1 == 3105); }
                }). start ();
                new Thread(new Runnable(){
                        public void run()
                        {int r2 = str. hashCode(); assert(r2 == 3105); }
                }). start ();
                ...
        }
}
```

Figure 2-17. Multiple threads are concurrently calling hashCode(). Despite the existence of a data race, the assertion never fails.

not escape the constructor, there is a happens-before edge from the initializing write to every read of the field [36, §17.5]

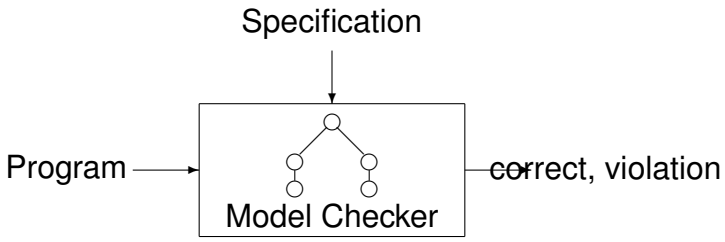

Figure 2-18. Model Checking Structure.

synchronizations are needed. But *hash* is not final, and if the multiple threads are concurrently calling hashCode() method as shown in Fig. 2-17, there would be a data race involving *hash*; one thread is writing *hash* in line 16 while another thread is reading *hash* in line 8. However, this is a benign data race, as hashCode() will always return the correct hash code no matter how many threads are running it.

Benign data race is extremely difficult to identify and there are not many studies about benign races. Narayanasamy et al. [67] proposed a dynamic approach to classify benign and harmful data races, but it is inaccurate.

2.4 Model Checking

When reasoning about the correctness of a single threaded program, formal mathematical reasoning techniques such as Hoare logic [39] are widely applied. But under multithreaded context, because there are so many nondeterminisms, a technique called *model checking* is generally used.

Model checking, defined by [44], is an "automatic technique for verifying finite state concurrent systems". Formally, model checking is defined as follows:

Definition 7 (Model Checking [22])**.** *Let M be a Kripke structure (i.e., state-transition graph). Let f be a formula of temporal logic (i.e., the specification). Find all states s of M such that $M, s \models f$.*

Informally speaking, we rely on model checking tools to check whether the properties in the given specifition are satisfied by automatically generating all the possible executing paths. The structure is shown in Fig. 2-18, where the program and the specification are given as an input, and the after exploration of all the possible states

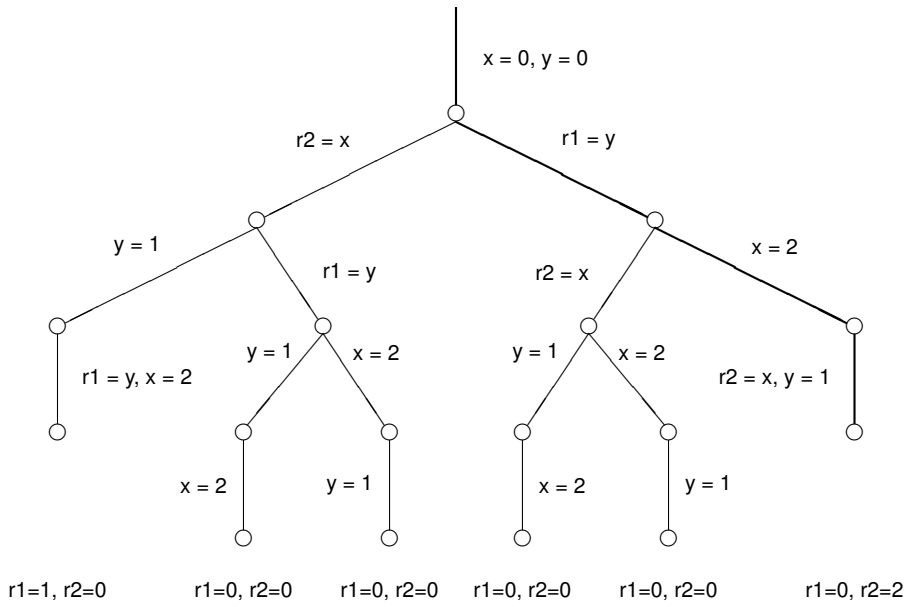

Figure 2-19. Model checking program in Fig. 2-1 under SC memory model.

the model checker answers whether the program meets its specification or not. The difference between model checking and software testing is that testing only execute one particular path, while model checking explores all the paths. Model checking has been used in both concurrent hardware and software systems.

The key procedure of model checking is state exploration. Fig. 2-19 shows the explicit-state exploration structure of Fig. 2-1 under SC memory model. Each circle represents a *state*. The model checker starts from the write of the default values and explores all the possible interleaving of instructions from different threads in a depth-first search (DFS) manner. When a new state is chosen to explore, the model checker *advances* to that new state. If there are no more interleaving choices, the model checker *backtracks* to the "parent" state and making other choices. The route from the first state to one of the end states forms a path. It at last explores all the paths and gets all the possible outcomes under SC memory model. After model checking, we find that $r1 = 1 \wedge r2 = 2$ is not valid under SC memory model.

We can see that even for the simple two-threaded-two-line program, model checker generates so many states, then for complex programs model checker may generate

astronomical number of states. This is called *state space explosion problem*. State exploration problem is a major limitation of model checking. Because of the state explosion problem, the model checker may run out of memory eventually. To tackle this problem, many mechanmisms are designed to reduce the number of states, such as partial order reduction [90], abstractions [23, 25, 56], symbolic model checking [61], and symmetric reduction [87]. Other mechanisms to alleviate state explosion problem can be found at [63].

2.4.1 Model checking tools

There are numbers of model checking tools avaiable. SPIN [73] is a model checker to verify properties specified by Linear Temporal Logic (LTL) [78]; TVLA [54] checks reachability properties based on shape analysis, abstraction, and 3-valued logic; Action Language Verifier (ALV) [93] is capable of checking properties given in Computation Tree Logic (CTL) [21]; SLAM [10] is an on-going Microsoft project that is aimed at model checking safety properties in C programs using predicate abstraction; BLAST [38] is a C model checker that uses software abstraction; F-Soft [40] is another C program model checker that applies abstractions. Other model checkers include NuSMV [71], MRMC [65], LTSA [57], Bandera tool set [37], Java Pathfinder (JPF) [42], etc. These model checkers are either *explicit state*, where the program states are explicitly explored; or *symbolic*, where states are summarized into formulas or *binary decision diagrams* (BDDs) [4].

2.4.2 Java Pathfinder

Java Pathfinder (JPF) [42, 91] is a software model checking tool for concurrent Java programs developed by NASA. JPF is explicit-state, Java bytecode based. It can be viewed as a virtual machine (VM) for Java. JPF takes Java class files as input and explores all the possible execution paths of the program. The verification result is returned after verification.

When checking program correctness, JPF can automatically detect nonfunctional properties like deadlocks or uncaught exceptions caused by Java's assert statements[8] . Other functional properties can be customly defined. The checking of these properties are done with JPF listeners; at certain point of an execution, JPF awakes an event handler which checks some properties.

JPF provides a mechanism called ChoiceGenerator to handle the uncertainties when making a choice. Thread interleaving is automatically handled by the built-in *ChoiceGenerator*. For data uncertainties, e.g. which value to choose when reading a variable, JPF also provides *BooleanChoiceGenerator*, *IntChoiceGenerator*, *Double-ChoiceGenerator*, etc. to handle the data uncertainties of some data type. Other choices can also be specified by extending ChoiceGenerator class.

JPF is highly extensible. It can be easily extended for many purposes. For example, Java Racefinder (JRF) [48, 49], now jpf-racefinder, is an extension of JPF to precisely detect and eliminate data races under JMM definition; Zhang and Breugel [94] associates probabilities into JPF to model check the randomized algorithms; Nguyen et al. [69] extends JPF to check if a Java program is correct with regard to UML sequence diagram specification; jpf-ltl [68] enables JPF to verify LTL properties for sequential and concurrent Java programs; Kebrt and Šerý [47] makes JPF to run JUnit test cases; Leungwattanakit et al. [53] enables JPF to model check networked applications (distributed system); jpf-awt [62] is a recent extension that enables JPF to model check AWT (Abstract Window Toolkit) programs.

JPF implicitly assumes sequential consistency as the underlying memory model which means only executions as shown in Fig. 1-1 can be generated by JPF, i.e. read only sees the value of the most recent write, other previous writes or future writes are

[8] An assert statement contains a boolean expression. It serves like a predicate inside the program: an error will be reported if the expression is evaluated to false.

invisible. Under JMM, JPF is only sound for programs without data race, because only DRF programs are sequentially consistent. For those programs with data races, JPF cannot generate possible sequentially inconsistent executions.

We will talk more about JPF's details in the JPR implementation section (Chapter 5).

CHAPTER 3
THE ALGORITHM

This chapter presents the main algorithm of JPR. The algorithm is aimed at model checking concurrent programs under JMM. The input is a target program with *assert* statements to describe the specification, and the output is true or false (i.e. whether the program meets its specification or not). The basic idea behind the algorithm is to maintain a map, $WriteSet$, that maps memory locations to sets of (write action, value written) pairs. For a read action of variable x, instead of the standard JPF behavior where the read sees the value of the most recent write to x on the current path (which also corresponds to sequentially consistent behavior), a value from an element of $WriteSet(x)$ is chosen. The algorithm is in *fixed-point* style; It loops over to expand $WriteSet$ and terminates when a fixed-point is reached, i.e. $WriteSet$ doesn't change. Through this process, completely out-of-thin-air values are avoided because each value seen by a read must have been written in some execution already generated. This algorithm is described in the context of JPF, but can be applied to any similar explicit state model checking tools with a listener style interface.

In this chapter, we first give an overview of this algorithm, then introduce some metadata used in the algorithm before describing the algorithm in detail. Finally we give an example to show how the algorithm works.

3.1 Algorithm Overview

Traditional model checking tools assume SC memory model by default, so they only explore all the interleaving of threads. Fig. 2-19 shows the exploration structure of these model checkers. Each read only sees the value written by the most recent write action. At each state, the model checker only has to determine which thread to choose from, then simply selects the first instruction that hasn't been executed from that thread. Therefore, if the execution shown in Fig. 1-1 is generated by these model checkers, the

Initially, $WriteSet(x) = \{(0,0)\}$, $WriteSet(y) = \{(0,0)\}$

1 r2 = x (0) 2 y = 1 3 r1 = y (0, 1) 4 x = 2	1 r2 = x (0) 3 r1 = y (0) 2 y = 1 4 x = 2	1 r2 = x (0) 3 r1 = y (0) 4 x = 2 2 y = 1	3 r1 = y (0) 1 r2 = x (0) 2 y = 1 4 x = 2	3 r1 = y (0) 1 r2 = x (0) 4 x = 2 2 y = 1	3 r1 = y (0) 4 x = 2 1 r2 = x (0, 2) 2 y = 1
$\langle 0,0\rangle$, $\langle 1,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$, $\langle 0,2\rangle$

After 1st run, $WriteSet(x) = \{(0,0),(4,2)\}$, $WriteSet(y) = \{(0,0),(2,1)\}$

Figure 3-1. The executions of 1st run of the extended model checker.

Initially, $WriteSet(x) = \{(0,0),(4,2)\}$, $WriteSet(y) = \{(0,0),(2,1)\}$

1 r2 = x (0, 2) 2 y = 1 3 r1 = y (0, 1) 4 x = 2	1 r2 = x (0, 2) 3 r1 = y (0, 1) 2 y = 1 4 x = 2	1 r2 = x (0, 2) 3 r1 = y (0, 1) 4 x = 2 2 y = 1	3 r1 = y (0, 1) 1 r2 = x (0, 2) 2 y = 1 4 x = 2	3 r1 = y (0, 1) 1 r2 = x (0, 2) 4 x = 2 2 y = 1	3 r1 = y (0, 1) 4 x = 2 1 r2 = x (0, 2) 2 y = 1
$\langle 0,0\rangle$, $\langle 1,0\rangle$ $\langle 0,2\rangle$, $\langle 1,2\rangle$	$\langle 0,0\rangle$, $\langle 1,0\rangle$ $\langle 0,2\rangle$, $\langle 1,2\rangle$	$\langle 0,0\rangle$, $\langle 1,0\rangle$ $\langle 0,2\rangle$, $\langle 1,2\rangle$	$\langle 0,0\rangle$, $\langle 1,0\rangle$ $\langle 0,2\rangle$, $\langle 1,2\rangle$	$\langle 0,0\rangle$, $\langle 1,0\rangle$ $\langle 0,2\rangle$, $\langle 1,2\rangle$	$\langle 0,0\rangle$, $\langle 1,0\rangle$ $\langle 0,2\rangle$, $\langle 1,2\rangle$

After 2nd run, $WriteSet(x) = \{(0,0),(4,2)\}$, $WriteSet(y) = \{(0,0),(2,1)\}$

Figure 3-2. The executions of 2nd run of the extended model checker.

underlined read can only see 2, but not 1 (the *previous write*), and 3 (the *future write*). But both $r = 2$ and $r = 3$ are legal results under JMM.

To let reads see other previous writes, one intuition is to keep a data structure that maintains a history of all the writes with respect to the memory locations. Then at the time of read, instead of reading the most recent write, we choose values from the history of the corresponding memory location. This idea has been expressed by [26], the data structure in which is called *WriteSet*. We may view *WriteSet* as a mapping from memory location to a pair of (write action, value written). With *WriteSet*, the model checker needs not only the nondeterminism of threads, but also *data nondeterminism* when performing a read.

Things become complicated when it comes to the future writes. This is not that straightforward to model. Because at the time of the read, we don't know what will happen in the future, so we cannot keep a history. Our idea is to run the model checker iteratively so that the read may see the values that will be generated in the future. In the

L0: Initially, 0 x == y == 0

Thread 1	Thread 2
L1: r1 = x;	L3: r2 = y;
L2: y = r1 + 1;	L4: x = r2;

Figure 3-3. If read from future write, that write must write the same value as the value read.

first run, the read may only see previous writes. After the run, we get a *WriteSet* from the generated executions. The *WriteSet* is then passed to the second run. In this run, the read still chooses value from the *WriteSet*, but may see more values, and hence the *WriteSet* might also be expanded. Let's take a look at the program in Fig. 2-1. Initially, the *WriteSet* only contains the write of default values (i.e. (x, 0), (y, 0)). In the first run, we get the outcomes of Fig. 3-1. The read see the value of previous writes. Note the results are the same as what we see in Fig. 2-19. The *WriteSet* is expanded at the end of the exploration. In the second run of the model checker, we get the results in Fig. 3-2. Now the reads are able to see the future writes. and the result $r1 = 1 \wedge r2 = 2$ can be generated.

However, we cannot let reads see any writes nondiscriminately. See the example shown in Fig. 3-3, in the first run, we get $WriteSet(x) = \{(L0, 0), (L4, 1)\}$ and $WriteSet(y) = \{(L0, 0), (L2, 1)\}$. If a read may see any writes in the *WriteSet*, then we shall get $WriteSet(x) = \{(L0, 0), (L4, 1), (L4, 2)\}$ and $WriteSet(y) = \{(L0, 0), (L2, 1), (L2, 2)\}$. But according to JMM's causality rules, if we commit L2, then y must write 1, so 2 is not a legal value. Based on this observation, if a read sees a future write, then that write must actually write the same value as the read sees. We call this write being "*imposed*" by the read. Then if we apply this rule in Fig. 3-3, we will not generate value 2. This avoids those completely out-of-thin-air values.

In order to capture the future writes in *WriteSet*, we call model checker in an iterative way, but it cannot loop forever. There is a termination condition in the algorithm. Note in Fig. 3-2, the *WriteSet* after the 2nd run is the same as 1st run. If the *WriteSet*

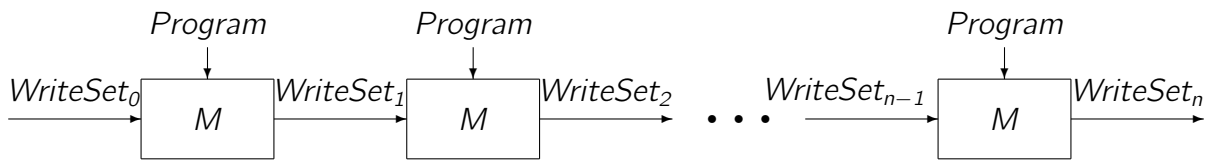

Figure 3-4. Algorithm Structure. After some n, $WriteSet_{n-1} = WriteSet_n$

after a run is the same as the *WriteSet* in the previous run, then the iteration terminates. If we view the model checker as a function f, and *WriteSet* as an argument to f, then the last run before termination can be viewed as $f(WriteSet) = WriteSet$. This condition is a *fixed point*. In *lattice theory*, fixed point is defined as:

Definition 8 (Fixed Point)**.** *[70, §4.2] Given a monotone function $f : L \to L$ on a complete lattice $L = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$, a fixed point of f is an element $l \in L$ such that $f(l) = l$ and we write $Fix(f) = \{l | f(l) = l\}$*

In Chapter 4, we will formally prove that our algorithm can be viewed as a monotone function and we can get to a *least fixed point* (LFP), which is the termination condition for finite-state programs.

The fixed-point style structure of the algorithm is shown in Fig. 3-4, where M is the model checker, $Program$ is the program being verified. Initially, we pass the $WriteSet_0 = \varnothing$ to the first run of model checking, and get a possibly expanded $WriteSet_1$ which is passed to the next run. After some run n, we get $WriteSet_n$ which is the same as $WriteSet_{n-1}$, and terminate the iteration. During this procedure, for any run i, we have $WriteSet_i \subseteq WriteSet_{i+1}$. We will explain this in more detail in Chapter 4.

Besides previous writes and future writes, we must also take care of the rules of well-formed execution (Definition 3), especially the happens-before consistency requirements (Rule 9). In our algorithm, we keep a data structure *HBSet* to record the happens-before relations ($\leq_{hb}$) in an execution. Different executions have different happens-before relations, so *HBSet* is not passed between runs. *HBSet* can be viewed as a set that contains (action1, action2) pairs. The set is expanded during the

exploration procedure. It contains all the program orders ($\leq_{po}$) and synchronizes-with orders ($\leq_{sw}$).

The first rule of happens-before consistency is the "no interleaving write" (i.e. $\nexists w : W(r) \leq_{hb} w \leq_{hb} r$). This rule is to justify legal previous writes; When the model checker is performing a read on variable x, it nondeterministically selects a pair from a set of (action, value written) from $WriteSet(x)$. If the write action of the selected pair is a previous write, then it checks the $HBSet$. The value is chosen only if the write satisfies this condition (i.e. no interleaving write), otherwise this value is discarded, and the model checker selects another value.

The second rule, $r \not\leq_{hb} W(r)$, is to justify legal future writes. The model checker doesn't check this rule when performing read. All the writes that are not executed at the time of read are considered to be "potential candidates" of future write. When a write is being executed, the model checker loops over the read actions that had previously read from the write and checks this rule by referring to $HBSet$. If this rule is violated by some reads, the current entire path is discarded, and the model checker is backtracked to the parent state.

Basically, our algorithm allows read actions to see any previous writes by introducing *WriteSet*. It lets reads see future writes by running model checker iteratively while it also has restrictions to rule out completely out-of-thin-air values and executions that violates JMM's well-formed execution rules. The algorithm can be summarized as:

- *Read from any previous writes*: Uses *WriteSet* to record the write history. Read chooses value from *WriteSet*.
- *Read from any future writes*: Run model checker iteratively to collect future writes into *WriteSet*.
- *Rule out out-of-thin-air values*: Rules out some completely out-of-thin-air values by imposing future writes.
- *Ensure well-formed executions*: Uses *HBSet* to record happens-before relations. Well-formed execution rules are checked when performing read or write.

The next section introduces the metadata that will be used in the algorithm.

3.2 Metadata

Our algorithm uses *WriteSet* to keep a history of write actions. This data structure is passed between different runs of model checker. Besides *WriteSet*, we also keep several other information that are execution specific, such as the *HBSet*. These information (metadata) is extended into the model checker's state representation.

In the following metadata list, Aid is the domain of action IDs. An action ID is an arbitrary unique identifier for the action. We will talk about different action ID schemes in Chapter 5. Val is the domain of values. Here value is a general concept, it could be either int, long, float, double, char, reference, or whatever kind of data type. Loc is the domain of memory locations. In the program memory locations are represented by variables. $Action$ is represented by a tuple of $\langle t, k, v, u \rangle$. It is formally defined in Definition 1.

- ***Path***: Sequence of action ids that represent the current path of execution. For a given action id aid, $Path(aid)$ represents the index of that action id, where $Path(aid)$ is 1 for the id of the first executed action in $Path$.
- ***WriteSet***: $Loc \rightarrow 2^{Aid \times Val}$ maps a memory location to a set of action ID, value pairs, where each action is a $WRITE$.
- ***ActionSet***: 2^{Action} contains the actions that have been executed on the current path so far.
- ***HBSet***: $2^{Aid \times Aid}$ is a set of pairs of action IDs where $\langle aid_1, aid_2 \rangle \in$ ***HBSet**** if and only if both are in ***ActionSet*** and $aid_1 \leq_{hb} aid_2$ and where ***HBSet**** is the transitive closure of the relation represented by ***HBSet***.
- ***ImposeSet***: $2^{Aid \times Val}$ is a set of action ID, value pairs, where each action is a $WRITE$. In a well-formed path, if a read action r obtains a value val from write action w which may be executed in the future, w must occur at some point in any well-formed path containing r, and it must actually write val. Thus the $ImposeSet$ maps write actions to values imposed on them by past reads.
- ***Read***: $Aid \rightarrow Aid \times boolean \times Val$ maps $READ$ and $VOLATILE\ READ$ action IDs to a triple containing the write action it sees, i.e. $W(rid)$ and the value it returns,

$W(V(rid))$ for action id rid. The boolean value indicates whether the $W(rid)$ occured in the future on the current path.

- ***Write***: $Aid \rightarrow Val$ maps $WRITE$ and $VOLATILE\ WRITE$. action IDs to the value written by the corresponding action, i.e. $V(wid)$.
- ***ThreadLast***: $Tid \rightarrow Aid$ maps a thread id to the latest action performed by the thread and is used to maintain the program order, $\leq_{po}$.

Note that there is also a *WriteSet* in the metadata, but this is not to be confused with the *WriteSet* that is passed between runs. This one is a local copy to a state. To distinguish between the two *WriteSet*s, we call the *WriteSet* which is passed between runs the *GlobalWriteSet*. When a run of model checking begins, the *GlobalWriteSet* of the last run is copied to the *WriteSet* of the first state. As the model checker makes advancements, the *WriteSet* of the current state is copied to the new states, and new pairs are appended into the *WriteSet* according to the memory operations involved in the advancement. At the end state of each path (i.e. the state has no choice to make, and the model checker will backtrack to the parent state), we take a union of the *WriteSet* of this state with the *GlobalWriteSet*.

Here the *ActionSet* records the actions executed so far. It is expanded when a new memory operation is executed. We may determine whether a write action w is previous write or future write by checking if $w \in ActionSet$ is true or not. The *HBSet* is the set that records the happens-before relations between the actions that have been executed so far. The *ImposeSet* keeps a history of the imposed write actions. As we mentioned in the last section, this is used to rule out some completely out-of-thin-air results. The *Write* can be viewed as the value-written function (V). The *Read* records the value a read sees, the source write action, and whether it is a previous write or a future write. The *ThreadLast* is used to construct the program order within each thread.

These metadata is carried along with state. Each state has a separate copy of the metadata. They are not passed between runs like the *GlobalWriteSet*. We can formally describe a *state* as Σ:

$\Sigma = \langle Path, WriteSet, ActionSet, HBSet, ImposeSet, Read, Write, ThreadLast \rangle$.

3.3 Formal Description

This section formally describes the algorithm. The algorithm is presented in a pseudocode, and is listener styled. Before describing the algorithm in detail, we will firstly introduce listener style and JPF's state stack structure.

Listener Style, also called *Observer Pattern*, or *Publish-Subscribe Pattern* [32, §5], is one of the behavioral software design patterns. Under listener style, the system has one object (called *publisher*) and one or more dependents (called *subscriber* or *listener*) registered to the publisher. When there is an *event* (the state of the publisher is changed), the subscribers are notified and take actions accordingly. While on the other hand, the subscribers may also change the state of the publisher. This is a one-to-many dependency relationship. Listener style is widely used in Java, where Swing is a good example. Listener is typically an interface with each event a separate method. Programmers implement the interface to let it perform according to the events. The listeners in Java are all subinterfaces of EventListener.

JPF is also in listener style. The ease of extension of JPF is largely due to the usage of listener style. Before running JPF, one or more listeners are registered to it. Upon receiving of events from JPF, the listeners may respond accordingly. The events of JPF vary from search events and VM events. The search events include all the state space search events, such as state advanced, state backtracked, state restored, search finished, etc. The VM events include JPF virtual machine-based events, such as instruction executed, thread started, thread blocked, object created, choice generator advanced, etc. The main function of our algorithm is written in JPF listener style. It takes actions according to the JPF events; modify the metadata, expand the *GlobalWriteSet*, and may also affect the search process of JPF (adding more choices to a state and force backtracking).

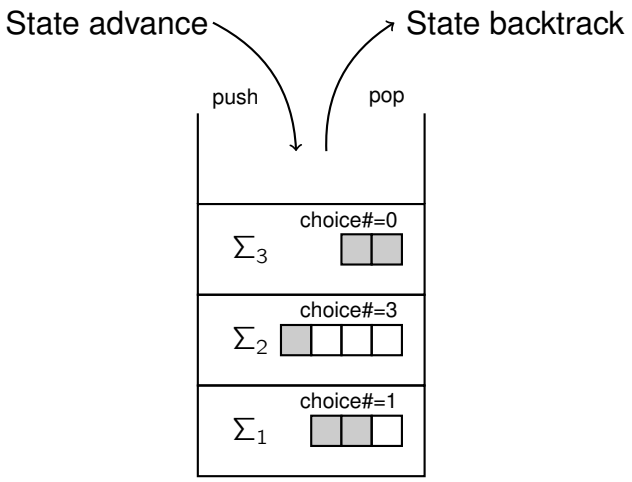

Figure 3-5. The stack structure of JPF state exploration. The shaded blocks represents choices that have already been selected; the empty blocks represents the current available choices.

The state exploration of JPF is in a stack structure (see Fig. 3-5). A number of *choices* are attached to each state. The choice can be either a scheduling choice (i.e. which thread to choose in the next step), or data choice (i.e. which data to choose from in the next step). At a state, JPF traverses its unselected choices, makes a selection, and advances to a new state accordingly (i.e. a new state is pushed onto the top of the stack). If the state has no more unselected choices, JPF performs a state backtrack action (i.e. pop the state on the top of the stack). In Fig. 3-5, there are no more availble choices for the state on top of the stack, so JPF pops Σ_3. Now the top state becomes Σ_2 which has three unselected choices. JPF then chooses one choice from them, mark it as selected, performs accordingly, and then pushes a new state onto the stack. If a state doesn't have any choices when it is pushed on the stack, we say that JPF is reaching "the end of a *path*". JPF stops when there are no more states on the stack.

Basically, the algorithm of JPR is comprised of two components:

- **JMMAwareJPF** Driver of JPR. It calls JPF iteratively and passes *GlobalWriteSet*.
- **JMMListener** The listener styled algorithm that is registered to JPF.

```
JMMAwareJPF(Program)
  GlobalWriteSet_old ← GlobalWriteSet_new ← ∅
  converged ← false
  while ¬converged do
    Call JPF(JMMListener(GlobalWriteSet_old))
    GlobalWriteSet_new ← JMMListener.GlobalWriteSet_new
    if GlobalWriteSet_new == GlobalWriteSet_old then
      converged ← true
    else        //not converged
      GlobalWriteSet_old ← GlobalWriteSet_new
  endwhile
```

Figure 3-6. **JMMAwareJPF**, the top level algorithm in JPR.

The **JMMAwareJPF** algorithm given in Fig. 3-6 serves as JPR driver. The input of the algorithm is the program being verified. Initially, the *GlobalWriteSet* is empty, and the converge condition is set to false. The $GlobalWriteSet_{old}$ is the *GlobalWriteSet* of the last iteration, and $GlobalWriteSet_{new}$ is the *GlobalWriteSet* of the current iteration. After initialization, the algorithm calls JPF iteratively. In each iteration, the JPR specific listener, **JMMListener**, is registered to JPF with $GlobalWriteSet_{old}$ passing to it. After execution of JPF, the **JMMListener** returns $GlobalWriteSet_{new}$, which is a new and non-decreasing *GlobalWriteSet* collected from the current iteration. We compare the $GlobalWriteSet_{new}$ with $GlobalWriteSet_{old}$. If they are equal, then the iteration terminates.

JMMListener is described in Fig. 3-7 and continued in Fig. 3-8. It is the listener that is registered to JPF. As various events in JPF (i.e. start search, advance state, backtrack, execute an instruction, as represented by the variable *searchEvent* in Fig. 3-7) occur, a corresponding code segment is executed. The JPF search events are listed in Fig. 3-7, and the VM event *INSTRUCTION EXECUTES* is listed in Fig. 3-8.[1]

[1] Other VM events, including thread divergence events (i.e. thread start, thread join, etc.) and object creation event introduce special synchronizes-with orders, and they will be discussed in more detail in Chapter 5.

```
1  JMMListener(GlobalWriteSet_old)
     GlobalWriteSet_new ← ∅                                   // New global WriteSet
3    Σ : ⟨Path, WriteSet, ActionSet, HBSet, ImposeSet, Read, Write, ThreadLast⟩
                                                              // Current state metadata
5    switch(searchEvent)
       case SEARCH STARTS:
7        WriteSet ← GlobalWriteSet_old
         ActionSet ← HBSet ← ImposeSet ← ∅
9        ∀loc : Read(loc) ← undef, Write(loc) ← undef
         ∀tid : ThreadLast(tid) ← undef
11       Stack.push(Σ)
       case STATE ADVANCES:
13       Stack.push(Σ)
       case STATE BACKTRACKS:
15       Σ ← Stack.pop()
         if END OF PATH then
17         if path is well-formed then
             GlobalWriteSet_new ← GlobalWriteSet_new ∪ WriteSet
19         else ignore write set and discard path
       case INSTRUCTION EXECUTES:
21         See Fig. 3-8
```

Figure 3-7. **JMMListener** algorithm

JMMListener takes $GlobalWriteSet_{old}$, the $GlobalWriteSet$ of the last iteration of JPF, as an input, and calculates $GlobalWriteSet_{new}$, the $GlobalWriteSet$ of the current iteration of JPF. Initially, the $GlobalWriteSet_{new}$ is empty. Σ is a representation of the current state of JPF. It is a tuple that contains all the metadata introduced in §3.2. When search starts, we copy $GlobalWriteSet_{old}$ to the $WriteSet$ of Σ, and initialize all other metadata to empty sets or undefined mappings. Then we push the initialized Σ onto the top of the state stack (Fig. 3-5). Σ is pushed onto the stack when JPF advances to a new state. When JPF backtracks, the state on the top of stack is popped and copied to the current state Σ. At the end of a search path, the path is tested to see if it is well-formed, i.e. all the writes that the reads have seen were actually executed in this iteration. If so, the $WriteSet$ of the last state on the path is unioned with the $GlobalWriteSet_{new}$, otherwise

the $WriteSet$ of the current state, as well as the entire path are discarded.[2] Then JPF performs state backtrack operation, and search for other paths.

Now we explain Fig. 3-8 in detail. When a memory related action is executed by JPF, an action tuple $action = (aid, tid, kind, loc)$ is formed. The aid is calculated by one of the action ID schemes introduced in Chapter 5. The action is then appended to $ActionSet$. The program order is formed by appending a $\leq_{hb}$ relation from the ID of the last action in the current thread ($ThreadLast(tid)$) to aid (Line 24), and the $ThreadLast(tid)$ is updated. The *isRELEASE* and *isACQUIRE* functions determine whether the action is a release (i.e. unlock, volatile write) or acquire (i.e. lock, volatile read). For release actions, if it is a volatile write, we update the $Write$ function of Σ. For acquire actions, we loop over the release actions on the same loc in $ActionSet$ and add the happens-before relations to $HBSet$. If the acquire action is a volatile read, we assign the value of the most up-to-date volatile write of loc to $Read(aid)$. This is according to the definition of volatile keyword in Java.

If the action is a write to a non-volatile variable, then we have to determine whether it is a future write by looking into $ImposeSet$. $ImposeSet$ records the writes that futurely read (i.e. being read before actual execution) by some read actions. If it is not a future write, then update $Write(aid)$ with the value it is written, and a new pair is appended to $WriteSet(loc)$. If it is a future write, then we have to check two things: 1) whether the value written by the write is the same as the value being read by previously by a read; and 2) whether it satisfies the 1st rule of happens-before consistency (i.e. $r \not\leq_{hb} W(r)$) (see Definition 3). 1) is straightforward because $ImposeSet$ records the value information. For 2), we loop over all the reads r to loc from $ActionSet$, and check

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

```
22 case EXECUTING ACTION where action = (aid, tid, kind, loc):
     ActionSet ← ActionSet ∪ {action} // add current action to action set
24   HBSet ← HBSet ∪ {(ThreadLast(tid), aid)}   //update ≤_hb due to ≤_po
     ThreadLast(tid) ← aid
26   if isRELEASE(kind) then
       if kind == VOLATILE WRITE writing val then
28       Write(aid) ← val
     else if isACQUIRE(kind) then
30     // for each release action rel that syncs with action do
       for each rel = (raid, rtid, rkind, rloc) s.t. isRELEASE(rel) ∧ (raid, aid) ∈ HBSet do
32       HBSet ← HBSet ∪ {(raid, aid)}  //update ≤_hb due to ≤_so
         if kind == VOLATILE READ then
34         //let latest denote the most recent volatile write that syncs with action
           let latest = (lid, ltid, lkind, lloc) s.t. lkind == VOLATILE WRITE ∧
               (lid, aid) ∈ HBSet ∧ ∄((a_k, aid) ∈ HBSet ∧ Path(a_k) > Path(lid))
           //Save the write action and value in Read. This is always a past write.
38         Read(aid) ← (lid, false, Write(lid))
     else if kind == WRITE of value val then
40     // if this write action is in the impose set, check for well-formedness
       if for some val′, (aid, val′) ∈ ImposeSet then
42         if val′ ≠ val then
             backtrack // value written is not the imposed value, abandon the path
44         else //check for ≤_hb consistency
               if ∃r ∈ ActionSet : Read(r.aid) == (aid, true, ∗) ∧ r.aid ≤_hb aid then
46               backtrack   //not ≤_hb consistent, abandon path
       //else path is still well-formed, save values and continue
48     Write(aid) ← val
       WriteSet(loc) ← WriteSet(loc) ∪ {(aid, val)}
50   else if kind == READ then
         non−deterministically choose (w, val) ∈ WriteSet(loc) do
52       if w ∈ ActionSet|_aid then // this is a past read
             //check for ≤_hb consistency
54           if (∄wa : wa ∈ ActionSet ∧ wa.kind == WRITE ∧ wa.loc == loc
                 ∧ w ≤_hb wa.aid ∧ wa.aid ≤_hb aid)  //≤_hb consistent past read
56           then Read(aid) ← (w, false, Write(w))
             else   //≤_hb inconsistent past read
58               continue with next write set entry
         else   // potential candidate for a future read
60           if ∄((w, val′) ∈ ImposeSet ∧ val′ ≠ val) then
                 ImposeSet ← ImposeSet ∪ {(w, val)}
62               Read(aid) ← (w, true, val) //true indicates future write
             else  //illegal future read, was in impose set with inconsistent value
64               continue with next write set entry
```

Figure 3-8. **JMMListener** algorithm continued from Fig. 3-7.

the happens-before relation from $HBSet$ between r and the the current write. If the action violates either 1) or 2), then JPF is forced to do a state backtrack operation (Line 42 and 46), otherwise update $Write(aid)$ with the value it is written, and a new pair is appended to $WriteSet(loc)$ like previous writes.

If the action is a read of a non-volatile variable, then we choose a (w, val) pair from $WriteSet(loc)$ non-deterministically (Line 51). Here "non-deterministically" actually means by adding data choices for the current state. Then we determine whether w is a previous write or a future write by checking if w is included in the $ActionSet$. Note that only actions that have already been executed can be added to $ActionSet$. If w is a previous write, then we have to check the 2nd rule of happens-before consistency, i.e. $\nexists w'$ to loc, s.t. $W(r) \leq_{hb} w' \leq_{hb} r$ by referring to $HBSet$ (Line 55). If there are no interleaving writes to loc, then update the $Read(loc)$, otherwise ignore this (w, val) and select the next pair from $WriteSet(loc)$. If w hasn't been executed so far, then we will check $ImposeSet$ to see if w is imposed by other reads that executed previously. If no other reads have imposed w, or w is imposed by other reads but the value they imposed is the same as val, then we regard w as a "potential" candidate of future write and update $Read(aid)$ accordingly, otherwise this is an illegal future read, and select the next pair from $WriteSet(loc)$. Here the potential future write might be invalidated at the time of that write if the value written is different from val or it violates happens-before consistency.

In this algorithm, there are three places where entire path would be abandoned because of the illegal future write. The first place is line 19 of Fig. 3-7, where the end of a path is reached. The algorithm loops over $ImposeSet$ to see whether all the imposed write actions are actually executed on this path. The path will be discarded if the condition fails. The second place is line 42 of Fig. 3-8. When a non-volatile write is imposed, then we check whether the imposed value is actually the value being written by the write. The third place is just a couple of lines below (line 46 of Fig. 3-8). If the

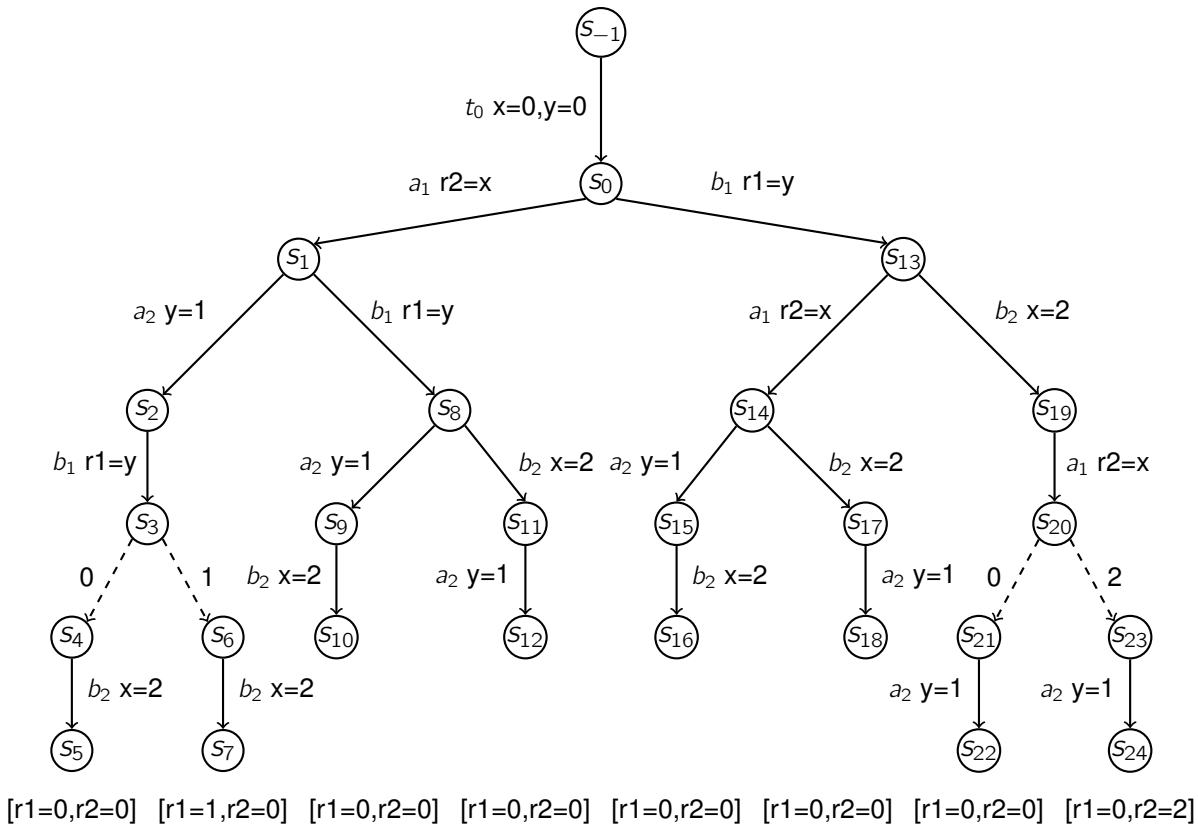

Figure 3-9. 1st iteration of JPR on the program shown in Fig. 2-1. Here the dashed arrows represent data choices and solid arrows represent thread choices.

non-volatile is imposed and it violates happens-before consistency, then we force JPF to backtrack. In all these places, a potential future write may cause violation of the well-formedness of a path. So when a read is reading from a write that hasn't been executed, the algorithm "potentially" allows it, and may discard it later on. An assertion violation on a path may not reflect the incorrectness because the path would later be discarded. Because of this feature of future write, the report of assertion error should be delayed. In JPF, as soon as an assertion is violated, it throws an exception and terminates. But in JPR, the assertion error should be reported at the end of the path. We will talk more on this in Chapter 5.

3.4 An Example

In this section, we present a simple example to illustrate how the algorithm works.

State	*ActionSet*	*WriteSet*(x)	*WriteSet*(y)	*HBSet*	*Read*
s_{-1}	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\lambda x : undef$
s_0	t_0	$(t_0, 0)$	$(t_0, 0)$	$\emptyset$	$\lambda x : undef$
s_1	t_0, a_1	$(t_0, 0)$	$(t_0, 0)$	(t_0, a_1)	$a_1 : (t_0, 0)$
s_2	t_0, a_1, a_2	$(t_0, 0)$	$(t_0, 0), (a_2, 1)$	$(t_0, a_1), (a_1, a_2)$	$a_1 : (t_0, 0)$
s_3	t_0, a_1, a_2, b_1	$(t_0, 0)$	$(t_0, 0), (a_2, 1)$	$(t_0, a_1), (a_1, a_2), (t_0, b_1)$	$a_1 : (t_0, 0)$
s_4	t_0, a_1, a_2, b_1	$(t_0, 0)$	$(t_0, 0), (a_2, 1)$	$(t_0, a_1), (a_1, a_2), (t_0, b_1)$	$a_1 : (t_0, 0), b_1 : (t_0, 0)$
$\underline{s_5}$	t_0, a_1, a_2, b_1, b_2	$(t_0, 0), (b_2, 2)$	$(t_0, 0), (a_2, 1)$	$(t_0, a_1), (a_1, a_2), (t_0, b_1, (b_1, b_2))$	$a_1 : (t_0, 0), b_1 : (t_0, 0)$
s_6	t_0, a_1, a_2, b_1	$(t_0, 0)$	$(t_0, 0), (a_2, 1)$	$(t_0, a_1), (a_1, a_2), (t_0, b_1)$	$a_1 : (t_0, 0), b_1 : (a_2, 1)$
$\underline{s_7}$	t_0, a_1, a_2, b_1, b_2	$(t_0, 0), (b_2, 2)$	$(t_0, 0), (a_2, 1)$	$(t_0, a_1), (a_1, a_2), (t_0, b_1, (b_1, b_2))$	$a_1 : (t_0, 0), b_1 : (a_2, 1)$
s_8	t_0, a_1, b_1	$(t_0, 0)$	$(t_0, 0)$	$(t_0, a_1), (t_0, b_1)$	$a_1 : (t_0, 0), b_1 : (t_0, 0)$
s_9	t_0, a_1, b_1, a_2	$(t_0, 0)$	$(t_0, 0), (a_2, 1)$	$(t_0, a_1), (t_0, b_1), (a_1, a_2)$	$a_1 : (t_0, 0), b_1 : (t_0, 0)$
$\underline{s_{10}}$	t_0, a_1, b_1, a_2, b_2	$(t_0, 0), (b_2, 2)$	$(t_0, 0), (a_2, 1)$	$(t_0, a_1), (t_0, b_1), (a_1, a_2), (b_1, b_2)$	$a_1 : (t_0, 0), b_1 : (t_0, 0)$
s_{11}	t_0, a_1, b_1, b_2	$(t_0, 0), (b_2, 2)$	$(t_0, 0)$	$(t_0, a_1), (t_0, b_1), (b_1, b_2)$	$a_1 : (t_0, 0), b_1 : (t_0, 0)$
$\underline{s_{12}}$	t_0, a_1, b_1, b_2, a_2	$(t_0, 0), (b_2, 2)$	$(t_0, 0), (a_2, 1)$	$(t_0, a_1), (t_0, b_1), (b_1, b_2), (a_1, a_2)$	$a_1 : (t_0, 0), b_1 : (t_0, 0)$
s_{13}	t_0, b_1	$(t_0, 0)$	$(t_0, 0)$	(t_0, b_1)	$b_1 : (t_0, 0)$
s_{14}	t_0, b_1, a_1	$(t_0, 0)$	$(t_0, 0)$	$(t_0, b_1), (t_0, a_1)$	$b_1 : (t_0, 0), a_1 : (t_0, 0)$
s_{15}	t_0, b_1, a_1, a_2	$(t_0, 0)$	$(t_0, 0), (a_2, 1)$	$(t_0, b_1), (t_0, a_1), (a_1, a_2)$	$b_1 : (t_0, 0), a_1 : (t_0, 0)$
$\underline{s_{16}}$	t_0, b_1, a_1, a_2, b_2	$(t_0, 0), (b_2, 2)$	$(t_0, 0), (a_2, 1)$	$(t_0, b_1), (t_0, a_1), (a_1, a_2), (b_1, b_2)$	$b_1 : (t_0, 0), a_1 : (t_0, 0)$
s_{17}	t_0, b_1, a_1, b_2	$(t_0, 0), (b_2, 2)$	$(t_0, 0)$	$(t_0, b_1), (t_0, a_1), (b_1, b_2)$	$b_1 : (t_0, 0), a_1 : (t_0, 0)$
$\underline{s_{18}}$	t_0, b_1, a_1, b_2, a_2	$(t_0, 0), (b_2, 2)$	$(t_0, 0), (a_2, 1)$	$(t_0, b_1), (t_0, a_1), (b_1, b_2), (a_1, a_2)$	$b_1 : (t_0, 0), a_1 : (t_0, 0)$
s_{19}	t_0, b_1, b_2	$(t_0, 0), (b_2, 2)$	$(t_0, 0)$	$(t_0, b_1), (b_1, b_2)$	$b_1 : (t_0, 0)$
s_{20}	t_0, b_1, b_2, a_1	$(t_0, 0), (b_2, 2)$	$(t_0, 0)$	$(t_0, b_1), (b_1, b_2), (t_0, a_1)$	$b_1 : (t_0, 0)$
s_{21}	t_0, b_1, b_2, a_1	$(t_0, 0), (b_2, 2)$	$(t_0, 0)$	$(t_0, b_1), (b_1, b_2), (t_0, a_1)$	$b_1 : (t_0, 0), a_1 : (t_0, 0)$
$\underline{s_{22}}$	t_0, b_1, b_2, a_1, a_2	$(t_0, 0), (b_2, 2)$	$(t_0, 0), (a_2, 1)$	$(t_0, b_1), (b_1, b_2), (t_0, a_1), (a_1, a_2)$	$b_1 : (t_0, 0), a_1 : (t_0, 0)$
s_{23}	t_0, b_1, b_2, a_1	$(t_0, 0), (b_2, 2)$	$(t_0, 0)$	$(t_0, b_1), (b_1, b_2), (t_0, a_1)$	$b_1 : (t_0, 0), a_1 : (b_2, 2)$
$\underline{s_{24}}$	t_0, b_1, b_2, a_1, a_2	$(t_0, 0), (b_2, 2)$	$(t_0, 0), (a_2, 1)$	$(t_0, b_1), (b_1, b_2), (t_0, a_1), (a_1, a_2)$	$b_1 : (t_0, 0), a_1 : (b_2, 2)$

Figure 3-10. The metadata of the states in the 1st iteration. The state number is corresponding to Fig. 3-9.

Let's apply the algorithm to the program shown in Fig. 2-1. Suppose we label the default write as t_0, the actions of Thread 1 as a_0 and a_1, and the actions of Thread2 as b_0 and b_1, then the state exploration of the 1st iteration of the algorithm is Fig. 3-9. The values of $r1$ and $r2$ are listed at the end of each path. The state is numbered according to the depth-first search order by JPF. The exploration is similar to Fig. 2-19, except the dashed transitions. The dashed transitions are generated due to data choices; A read may see not only the most recent write, but any previous writes, while the SC memory model-based model checkers only have scheduling choices.

The states' metadata are shown in Fig. 3-10. *Path* is not shown in the table but is reflected by the action order of *ActionSet*. The *ImposeSet* is empty for all the states because in the 1st iteration, reads cannot see future writes, so it is not listed. Also *ThreadLast* and *Write* are trivial and are omitted. To save space, the futurely read signal in *Read* is ignored. The *HBSet* column contains only the direct happens-before relation. When checking the happens-before consistency, we must calculate the transitive closure of it. s_{-1} is the initial state of JPF. Initially the *WriteSet* is empty. There are a couple of places where a read may see a previously written, but not up-to-date values, namely s_4 and s_{21}. In s_4, b_1 is reading from t_0. From the transitive closure of s_4's *HBSet*, there is no write w to y such that $t_0 \leq_{hb} w \leq_{hb} b_1$, so this is a legal read. The underlined states are the last states on a path. Before backtracking from these states, the *WriteSet* is unioned with $GlobalWriteSet_{new}$. After the 1st iteration, we get: $GlobalWriteSet_{new}(x) = \{(t_0, 0), (b_2, 2)\}$ and $GlobalWriteSet_{new}(y) = \{(t_0, 0), (a_2, 1)\}$. This $GlobalWriteSet_{new}$ is then passed to the 2nd iteration.

With the expanded *GlobalWriteSet*, the search space of the 2nd iteration is greatly expanded. For simplicity, we only show a particular path in Fig. 3-11. There are a lot more data choices (dashed arrows) in the 2nd iteration. This enables us to get $r1 == 1 \&\& r2 == 2$ via path $w_{-1} \rightarrow w_0 \rightarrow w_{20} \rightarrow w_{21} \rightarrow w_{22} \rightarrow w_{25} \rightarrow w_{26}$.

The states' metadata of path $w_{-1} \rightarrow w_0 \rightarrow w_{20} \rightarrow w_{21} \rightarrow w_{22} \rightarrow w_{25} \rightarrow w_{26}$ is shown in Fig. 3-12. Other states' metadata is not listed for brevity. Initially, the *GlobalWriteSet* of the last iteration is passed to the *WriteSet* of w_{-1}. When JPF is executing a_1, there are two data choices from *WriteSet*; 0 (previously written by t_0) and 2 (future write by b_2). If we choose 2 as the value seen by the read, then we should impose b_2 to write 2 by adding $(b_2, 2)$ to the *ImposeSet*. Then at state w_{26} when b_2 is executed, we must determine whether the imposed value is actually written. In this case, b_2 writes 2 which justifies the *ImposeSet*. Furthermore, we must also check the happens-before consistency by referring to the transitive closure of *HBSet*. In this case, a_1 is reading

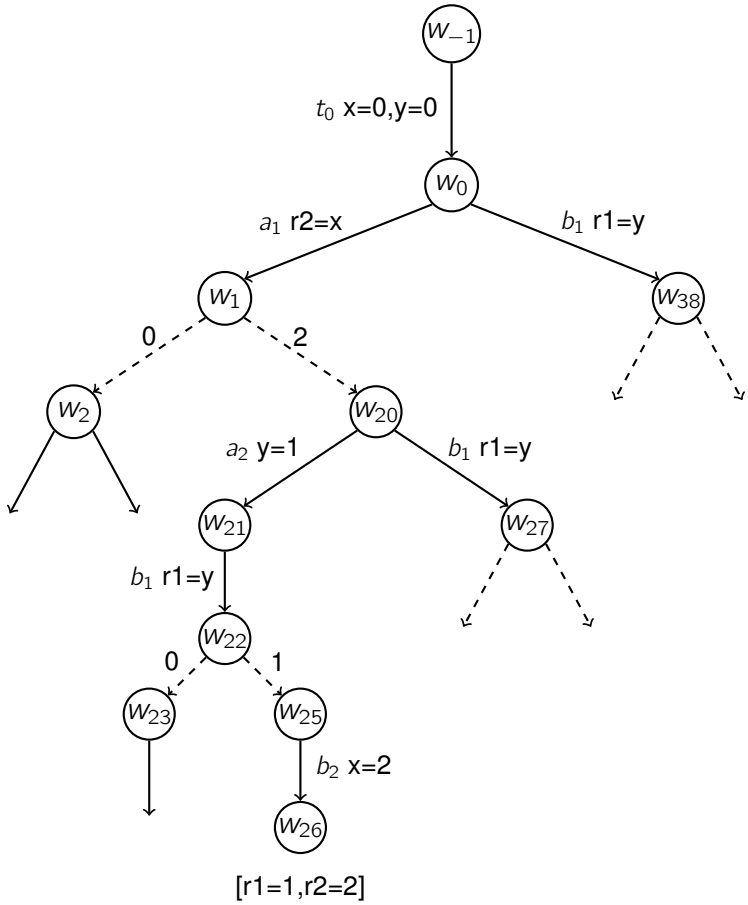

Figure 3-11. 2nd iteration of JPR on the program shown in Fig. 2-1. Here the dashed arrows represent data choices and solid arrows represent thread choices.

State	*ActionSet*	*WriteSet*(x)	*WriteSet*(y)	*HBSet*	*ImposeSet*	*Read*
w_{-1}	$\emptyset$	$(t_0, 0), (b_2, 2)$	$(t_0, 0), (a_2, 1)$	$\emptyset$	$\lambda x : undef$	$\lambda x : undef$
w_0	t_0	$(t_0, 0), (b_2, 2)$	$(t_0, 0), (a_2, 1)$	$\emptyset$	$\lambda x : undef$	$\lambda x : undef$
w_1	t_0, a_1	$(t_0, 0), (b_2, 2)$	$(t_0, 0), (a_2, 1)$	(t_0, a_1)	$\lambda x : undef$	$\lambda x : undef$
w_{20}	t_0, a_1	$(t_0, 0), (b_2, 2)$	$(t_0, 0), (a_2, 1)$	(t_0, a_1)	$(b_2, 2)$	$a_1 : (b_2, 2)$
w_{21}	t_0, a_1, a_2	$(t_0, 0), (b_2, 2)$	$(t_0, 0), (a_2, 1)$	$(t_0, a_1), (a_1, a_2)$	$(b_2, 2)$	$a_1 : (b_2, 2)$
w_{22}	t_0, a_1, a_2, b_1	$(t_0, 0), (b_2, 2)$	$(t_0, 0), (a_2, 1)$	$(t_0, a_1), (a_1, a_2), (t_0, b_1)$	$(b_2, 2)$	$a_1 : (b_2, 2)$
w_{25}	t_0, a_1, a_2, b_1	$(t_0, 0), (b_2, 2)$	$(t_0, 0), (a_2, 1)$	$(t_0, a_1), (a_1, a_2), (t_0, b_1)$	$(b_2, 2)$	$a_1 : (b_2, 2), b_1 : (a_2, 1)$
w_{26}	t_0, a_1, a_2, b_1, b_2	$(t_0, 0), (b_2, 2)$	$(t_0, 0), (a_2, 1)$	$(t_0, a_1), (a_1, a_2), (t_0, b_1), (b_1, b_2)$	$(b_2, 2)$	$a_1 : (b_2, 2), b_1 : (a_2, 1)$

Figure 3-12. The metadata of the states in the 2nd iteration. The state number is corresponding to Fig. 3-11.

from b_2, and $\nexists a_1 \leq_{hb} b_2$, so it satisfies the happens-before consistency. From this path, we get $r1 == 1 \&\& r2 == 2$. The same as the 1st iteration, at the end of each path, the $WriteSet$ is unioned with $GlobalWriteSet$. After the 2nd iteration of JPF, the $GlobalWriteSet$ is the same as the 1nd, so the iteration terminates.

CHAPTER 4
ALGORITHM PROPERTIES

In this chapter, we discuss the properties of JPR and its basic algorithms (Figs. 3-6, 3-7, and 3-8). The main results are that JPR only generates paths corresponding to well-formed executions and that the set of paths generated is an overapproximation of the JMM. Executions are the abstraction used in the JMM and defined in Definition 2 while paths are the totally ordered sequences of actions generated by JPR. We say that path p corresponds to execution $E = \langle A, P, \leq_{po}, \leq_{so}, W, V \rangle$ where A is the set of actions that occur in p, P is $prog$, $\leq_{po}$ is the union over all threads of $\leq_{path}$ restricted to each thread, and $\leq_{so}$ is $\leq_{path}$ restricted to the synchronization actions in p. If a non-volatile read r uses $WriteSet$ entry (w, val), then $W(r) = w$ and $V(w) = val$. $V(w)$ is well-defined since all reads of the same write action in a path must get the same value.

For a fixed program, $prog$, usually considered to be understood, and letting WS be the type of $WriteSet$, let $\text{JPR}_{\text{prog}} : WS \rightarrow WS * Paths$ be a function that takes a $ws \in WS$ and returns a new WS and a set of paths $paths$. JPR_{prog} is a function represents an invocation of JPF seen in Fig. 3-6, where $Paths$ is the set of paths searched by JPF. For $ws \in WS$ and path p, we say that $ws \overset{\text{JPR}}{\rightarrow} p$ if $p \in \text{JPR}_{\text{prog}}(ws).paths$. We say that $ws \overset{\text{JPR}*}{\rightarrow} p$ if $\exists i \geq 0 : p \in (\text{JPR}^{i}_{\text{prog}}(ws)).paths$[1] . For convenience, we overload $\overset{\text{JPR}}{\rightarrow}$ and $\overset{\text{JPR}*}{\rightarrow}$ and also say $ws \overset{\text{JPR}}{\rightarrow} ws'$ or $ws \overset{\text{JPR}*}{\rightarrow} ws'$ with the obvious meanings.

4.1 Safety, Completeness, and Convergence

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

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

Proof. The proof is straightforward by induction on the length of a path. New elements of $\leq_{hb}$ can be created due to program order on a thread, or when an acquire operation is performed. The former is handled in line 24 in Fig. 3-8 while the latter is handled in line 32. □

Proposition 4.1 (Safety). *Let ws_{sc} be the set of (w, v) pairs seen in the sequentially consistent executions of $prog$. If $ws_{sc} \overset{JPR*}{\rightarrow} p$, then p corresponds to a well-formed execution of $prog$.*

Proof. Since paths are generated from a sound model checker executing a properly compiled Java program, most of the rules for a well-formed execution hold by construction. Rule 9 holds because of Lemma 1 and the test in (Fig. 3-8, line 55). □

Proposition 4.2 (Completeness). *$JPR_{prog}(ws)$ generates a path corresponding to every well-formed execution of prog satisfying $(\forall reads\ r \in A : (W(r), V(W(r))) \in ws)$.*

Proof. This proposition depends on the behavior of the underlying model checker, namely that it will explore all of its choices. Suppose there is such a well-formed path p that is not generated by $\text{JPR}_{\text{prog}}(ws)$ and let p' be the longest prefix of p that has been searched. There are two possibilities. One is that the last action in p' caused it to be discarded. This will happen in four circumstances in lines 42, 46, 57, and 63 of Fig. 3-8. However, all of these indicate at violation of well-formedness. The other possibility is that the first action a in $p - p'$ is never taken. If a is a $READ$, then this would mean that the model checker failed to take an available choice at line 51. If a is not a read, then the model checker's search strategy failed to explore a valid transition. Either case violates our basic assumption about the completeness of the model checker[2] . □

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

[2] Because we have not studied the interaction of JPF's partial order reduction with the listener described in Figs. 3-7 and 3-8, we use JPF without this feature.

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

- *$ws \subseteq ws_1$.*

Proof. Follows from the fact that elements are never removed from the WriteSet. □

Theorem 4.1 (Convergence)**.** *For finite state, terminating program $prog$, Suppose that JPR_{prog} is applied iteratively starting with ws_0. The process will reach a fixed point $ws*$ in a finite number of steps and the resulting $ws*$ will be the least fixed point of JPR_{prog} at least ws_0.*

Proof. Noting that the (finite) set of $(ws, paths)$ pairs with subset inclusion form a complete lattice, the result from the Knaster-Tarski fixed point theorem [88] and Lemma 2. In Knaster-Tarski's theorem, if L is a complete lattice, and if $f : L \rightarrow L$ is a monotone function, then the set of fixed points of f is also a complete lattice. This implies that the least fixed point of f is the bottom of the complete lattice so it can be achieved by performing finite number of steps. □

4.2 Overapproximation

In this section, we formally describe the most important property of JPR, the overapproximation of JMM. We firstly present the theorem and proofs, then give two examples to show the overapproximation. Finally, we state the relationship of the executions generated by JPR with legal executions of other memory models.

Lemma 3 (Paths with only past reads are generated)**.** *Let ws_{sc} be an initial write set formed by collecting values written in the sequentially consistent executions of prog. Then for each path p corresponding to well-formed execution E that does not read future values $ws_{sc} \overset{JPR*}{\rightarrow} p$.*

Proof. The proof is by contradiction. Suppose there is a path p corresponding to well-formed execution E that does not read future values and it is not the case that $ws_{sc} \overset{\text{JPR*}}{\rightarrow} p$. Let pre be the maximal prefix of p such that pre is a prefix of some p' where

for some write set $ws_{p'}$, $ws_{sc} \overset{\text{JPR}*}{\rightarrow} ws_{p'} \overset{\text{JPR}}{\rightarrow} p'$. Note that pre includes at least the initialization actions and is thus not empty. Now, consider the next action a after pre in p. We argue that $ws_{sc} \overset{\text{JPR}*}{\rightarrow} ws_{p'} \overset{\text{JPR}*}{\rightarrow} pre@a$. This action is not in p'. By assumption, if a is any operation other than a read, it will be generated by $\text{JPR}_{\text{prog}}(ws_{sc})$, thus a must be a read such that $W(a)$ is not an any WriteSet generated by $ws_{p'}$. But since a is a past read $W(a)$ is in pre, and thus in $CV_{\text{prog}}(ws_{p'}).WS$. □

Lemma 4 (Paths with reads in past or in WriteSet are generated)**.** *Let ws_i be a write set where $ws_{sc} \subseteq ws_i$. Then for each path p corresponding to well-formed execution E where each r reads a past write or a write in ws_i, $ws_i \overset{\text{JPR}*}{\rightarrow} p$.*

Proof. Follows from Prop 4.2 and Lemma . □

The JMM defines a legal execution as one that can be obtained via a sequence of so-called committing executions where the executions in the sequence are related to each other by a set of constraints. According to Theorem 4.2, if we generate all the paths corresponding to some committing execution, then we will also generate all the paths in any execution that could possibly come next in the committing sequence.

Theorem 4.2 (Overapproximation)**.** *Let ws_{sc} be the smallest WriteSet containing all of the values seen in the set of sequentially consistent executions of finite state, terminating program $prog$ and $ws_{sc}*$ be the least fixed point of JPR_{prog} at least ws_{sc}. Let $JPR_{prog}(ws_{sc}*).paths$ be the set of paths generated by ws_{sc}. Let $JmmLegal_{prog}$ be the set of legal paths. Then $JmmLegal_{prog} \subseteq JPR_{prog}(ws_{sc}*).paths$.*

Proof. An execution E of prog is legal if there is a sequence of justifying executions E_0,E_1, ..., E_n satisfying the requirements for legal executions in Definition 4. Since we are only considering finite state, terminating programs, $E_n = E$. We will prove by induction that a path corresponding to every E_i for $1 \leq i$ in a valid justifying sequence is generated by $\text{JPR}_{\text{prog}}(ws_{sc}*)$. Consider execution E_i with a commit set C_i. Without loss of generality, we assume a minimal E where A_i be the minimal set of actions such

Initially, x = y = z = 0

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

Figure 4-1. A labeled version of Fig. 2-10. JPR_{prog} generates a path with r1 == r2 == 1&&r3 == 0. This is not legal according to JMM's causality rules.

that $C_i \subseteq A_i$ and E_i is well-formed. Let ws_i be the WriteSet that generates a path corresponding to E_i.

Base case: There is a path p_1 corresponding to E_1 such that $ws_{sc} \overset{\text{JPR*}}{\rightarrow} p_1$. Because $C_0 = \phi$, Definition 4, rule 6 requires that for all reads r in A_1, $W(r) \leq_{hb} r$ and are thus in the past. The result follows from lemma 4.2.

Induction step: Assume that a path corresponding to E_i with commit set C_i has been generated by $\text{JPR}_{\text{prog}}(ws_i)$. Now consider execution E_{i+1} with commit set C_{i+1}. From proposition 4.2, it suffices that all writes $w \in A_{i+1}$ are in ws_{i+1} where $ws_i \overset{\text{JPR*}}{\rightarrow} ws_{i+1}$.

From Definition 4, rule 4, for all write actions w in C_{i+1}, the same value are written in E_i, E_{i+1}, and E, i.e. $V_i(w) = V_{i+1}(w) = V(w)$. From rule 5, for all the read actions r in C_i, the same writes are seen in E_i, E_{i+1}, and E, i.e. $W_i(r) = W_{i+1}(r) = W(r)$. Further, for each $r \in C_i$, $W(r) \in ws_i$. Thus we are only concerned with the writes in $A_{i+1} - C_i$. From rule 7, for any read $r \in C_{i+1} - C_i$, $W_{i+1}(r) \in C_i$, and thus in ws_i. From rule 6, for any read $r \in A_{i+1} - C_{i+1}$, $W_{i+1}(r) \leq_{hb_{i+1}} r$. Thus all all reads in A_{i+1} are either in ws_i or are past reads. From lemma 4, for some path p_{i+1} corresponding to E_{i+1}, $ws_i \overset{\text{JPR*}}{\rightarrow} p_{i+1}$. □

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

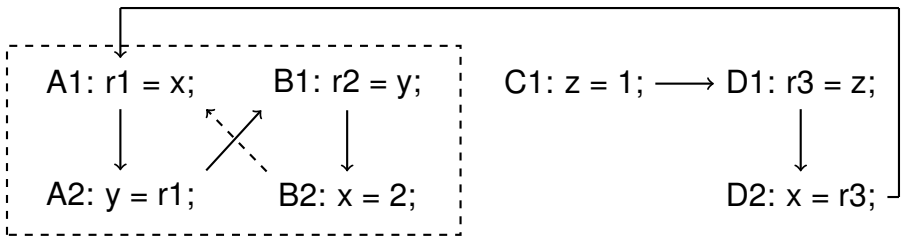

Figure 4-2. Value propagation of Fig. 4-1.

In the example shown in Fig. 4-1, JPR generates a path with result $r1 == r2 == 1 \&\& r3 == 0$. There is a valid path where action D2 writes 1, A1 reads D2, A2 writes 1, B1 reads A2, B2 writes 1. Then, on the next iteration, A1 reads from B2 (and imposes 1 on B2), B1 reads from A2, and then B2 successfully writes 1 as imposed by A1, while D1 reads from the default write action (value 0). However, this is not legal according to JMM. In order for $r1 == r2 == 1$ to appear in a JMM-legal execution, D2 would need to be a committed action with $V(D2) == 1$. But then $r3$ must already be 1, so the execution is not legal. The value 1 is considered to come out-of-thin-air in any execution where $r3 == 0$.

Fig. 4-2 shows how the value 1 is propagated to the fragment in the dashed box. In the first iteration, "1" is passed along C1 $\rightarrow$ D1 $\rightarrow$ D2 $\rightarrow$ A1 $\rightarrow$ A2 $\rightarrow$ B1 $\rightarrow$ B2. In the second iteration, "1" can be passed from B2 to A1, then A1 $\rightarrow$ A2 $\rightarrow$ B1 $\rightarrow$ B2 $\rightarrow$ A1 forms a loop on data dependencies. Note that this program is the same program as Fig. 2-9 with the addition of two other threads, Thread 3 and 4 which introduce an out-of-thin-air value to the execution. In Fig. 2-9, JPR does not generate paths with out-of-thin-air values.

Yet another example to show JPR generates an overapproximation of JMM is Fig. 4-3. JPR could generate $r1 == 1 \&\& r2 == 1 \&\& r3 == 2$. In the first iteration, to let $z = 1$ execute, the statements in the *else* should be executed. Then in the second iteration, $r1$ sees the future write of z of 1. $z = 1$ is imposed and then Thread 1 enters the *if*. Thread 2 then get $r2 == 1$ and $r3 == 2$, and writes 1 to z. This write justifies

Initially, x = y = z = 0

Thread 1	Thread 2
r1 = z;	r2 = x;
if(r1 == 1){	r3 = y;
x = 1;	if(r2 + r3 == 3)
y = 2;	z = 1;
}	

Figure 4-3. r1 == 1&&r2 == 1&&r3 == 2 is illegal result by JMM, but generated by JPR.

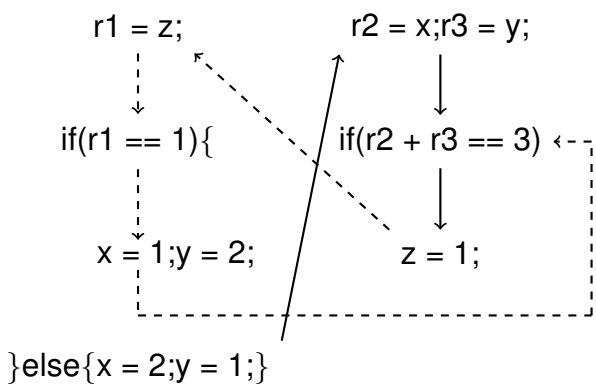

Figure 4-4. Data and control dependencies of Fig. 4-3. Here the solid arrows show the dependencies in the 1st iteration; the dashed arrows show a dependency loop formed in the 2nd iteration.

the imposed value and we get $r1 == 1 \&\& r2 == 1 \&\& r3 == 2$. But this result is prohibited by JMM. Applying the causality rules, in E_0, all the reads only see the writes that happens-before them. So only the *else* is executed. Then in order to commit $z = 1$ we must first commit the actions in *else*. So there is no way to commit the actions in *if*.

Fig. 4-4 shows the data and control dependencies of Fig. 4-3. In the first iteration, actions in the *else* make $z = 1$ to happen. In the second iteration, $r1$ sees the future write of z so the *if* branch executes. Interestingly both *if* and *else* let the condition in Thread 2 be true. So a dependency loop is formed (the dashed arrows) from *if*. This loop causes the illegal result $r1 == 1 \&\& r2 == 1 \&\& r3 == 2$ to generate. Such loop is called "*causal cycle*" in [60], but the concept is not formally defined. Early execution of an action does not result in a causal cycle if "its occurrence is not dependent on a read

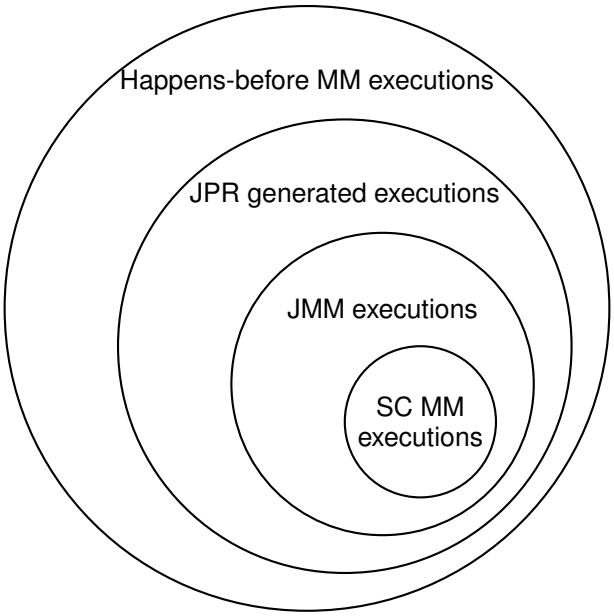

Figure 4-5. Relationship between the executions generated by JPR and legal executions of SC memory model, JMM, and Happens-before memory model.

returning a value from a data race"[60]. In this example, the early execution of $z = 1$ is depended on data races with x and y involved. JMM's causality rules are aimed at detecting such "causal cycles", but JPR's algorithm doesn't check this because checking it is very expensive.

From the two examples, we see that JPR may generate some illegal paths with out-of-thin-air values only when the out-of-thin-air values actually do appear in some generated path. It does not generate completely arbitrary out-of-thin air values. JPR could be made more precise by tracking impose requirements across iterations and dependent actions at the cost of significantly increased time and space overhead.

The relationship between the executions generated by JPR and legal executions of SC memory model, JMM, and Happens-before memory model is shown in Fig. 4-5. Happens-before memory model is a simpler memory model than JMM. Basically it requires an execution to satisfy synchronization order consistency and happens-before consistency, but no causality rules is required. SC memory model has the smallest execution space. JPR generates an overapproximation of JMM, but also rules out

certain kinds of out-of-thin-air results so the execution space is smaller than that of happens-before memory model.

CHAPTER 5
IMPLEMENTATION

This chapter describes the implementation issues involved in developing JPR. We identify an ambiguation in JMM on the action ID definition (element u in Definition 1). Without a proper definition on action ID, it is difficult to relate actions between different iterations. In this chapter, we firstly state the action ID ambiguation problem, and propose four action ID schemes, of which three schemes are actually implemented in JPR. Then we describe the overall structure of the tool. Finally, we list some selected major implementation issues and provide the solutions. The implementation issues are grouped into JPF-related issues and non-JPF related issues.

5.1 JMM Disambiguation

One of the difficulties encountered when implementing JPR was the lack of a well-defined connection between the notion of executions used to define the JMM and actual Java programs. This manifested itself in the representation of the action ID. In Definition 1, JMM only specifies that an arbitrary unique identifier u is associated with an action, but doesn't explain how to ensure the uniqueness, neither does it explain how to obtain the identifier. Within a single execution, the basic requirement of the action IDs is uniqueness. However, both the JMM definition of legal executions (Definition 4) and JPR require that the identity of actions be compared across different executions and paths, i.e. we must be able to determine if, say, a write to x in one execution or path is the same action as a write to x in another by comparing their IDs. Recall that in Definition 4, $C_{i-1} \subseteq C_i$. It requires that all the actions that have already been committed in E_{i-1}must also be committed in E_i. But JMM doesn't tell us how to relate the actions in E_{i-1} and E_i. This becomes problematic for programs with branches.

We considered four approaches to identify actions. Let t be the thread that the action belongs to, k be the action kind[1] , v be the memory location (i.e. variable), and val be the value read from or written to.

Occurrence (k, t, v, n). n counts occurrences of actions of kind k by thread t on variable v. With this approach, the nth read (or write) of a particular variable is always considered to be the same action, regardless of what happens in between, and whether or not the instructions occur in the same place in the source code.

Scope (t, S, n). S refers to the lexical scope of t, repeated invocations of the same instruction, such as in a loop are differentiated by a sequence number n. Here, actions are distinguished by their location in the source code, with repeated invocations of the same instruction, such as in a loop, differentiated by their sequence numbers.

Value (k, t, v, val). Actions with the same k,v, and t are distinguished by the value. This is the approach used in [17] but it is not adequate because actions are no longer uniquely identified if a thread writes the same value to a variable more than once. For this reason, we have not pursued this approach.

Occurence-Val (k, t, v, val, n). Adds an occurence count (n) to **value** with the consequence that for a write w, $V(w)$ always maps to the same value. This is an attempt to rescue the **value** approach by distinguishing different actions of the same kind that operate on the same variable with the same value with a counter. This tangles the notion of an action set with the value-written function V so that for a write w, $V(w)$ always maps to the same value, making legality rules 4 and 7 in Definition. 4 redundant and inoperative, respectively.

[1] For brevity, we only restrict attention to read and write. Other synchronization actions like lock and unlock don't associate with values, so we may just count the occurrence with regard to the monitors.

Initially, x == y == 0

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

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

Initially, x == y == 0

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

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

Figure 5-1. ActionID examples I. Comparison between **scope** and **occurrence**.

The different approaches yield different sets of legal executions. Consider Figs. 5-1b and 5-1a. Approach **occurrence** allows the outcome in Figs. 5-1b because both assignments to x are considered to be the same action; if committed, the assignments could be included in the justifying executions. However, it forbids the outcome in Fig. 5-1a since the assignment $x = 2$ in two different executions may have different actionIDs depending on whether or not the branch was taken. If the branch is taken, then it is the second write to x, otherwise is the first write to x. Approach **scope** allows the indicated outcome in Fig. 5-1a because regardless of the execution order, $x = 2$ is within the same lexical scope and can be committed and verified. It does not allow the outcome in Fig. 5-1b because the two $x = 1$ actions are within different scopes and if one is committed, it is impossible for the action to be included in subsequent verification executions.

Fig. 5-2 shows the different interpretations of JMM between **occurrence** and **occurrence-val**. The result $r1 == r2 == 1$ is allowed by **occurrence-val**, but forbidden by **occurrence**. Using **occurrence-val**, we may firstly commit $x = 1$ in E_1. This action is the first write to x with value 1 in Thread 2. Then we can commit $r1 = x$ (1), $y = r1$ (1), and $r2 = y$ (1) in the subsequent justifying executions. Now the branch is taken, and we get $x = 2$ and $x = r2$ (1) to execute. Here $x = r2$ is also the first write to x with value

Initially, x == y == 0

Thread 1	Thread 2

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

Figure 5-2. ActionID examples II. r1 == r2 == 1 is allowed by **occurrence-val**, but forbidden by **occurrence**.

1 in Thread 2, the same value is written as $x = 1$ in E_1, so we can commit it to justify $C_{i-1} \subset C_i$. However, using **occurrence**, after committing $x = 1$ (the first write to x in Thread 2, it writes 1) in E_1, and $r1 = x$ (1), $y = r1$ (1), and $r2 = y$ (1) in the following justifying executions, we cannot go further, because now the first write to x in Thread 2 becomes $x = 2$ which writes value 2, not 1. This violates $W_i|_{C_{i-1}} = W|_{C_i}$.

Based upon the observations, we see that different definition on the action ID may lead to complete different interpretations of JMM, but JMM makes this issue open without given clarifications. In order to compare the action ID schemes, we have implemented **scope**, **occurrence**, and **occurrence-val** in JPR. An analysis of different schemes based on experiments is included in Chapter 6. A conclusion on the schemes is desirable, however it is actually very hard to tell which one is better than the other. Our finding is a reminder to the JMM designers to give a much clearer definition on this.

5.2 JPR Structure

Typically, extensions to JPF are realized by listeners. A “standard” JPF extension usually registers project-specific listeners to JPF and runs JPF only once. All the properties can be collected through the listeners. Projects such as jpf-racefinder[48] and jpf-awt[62] are standard JPF extensions. JPR is not a standard extension, because it calls JPF iteratively.

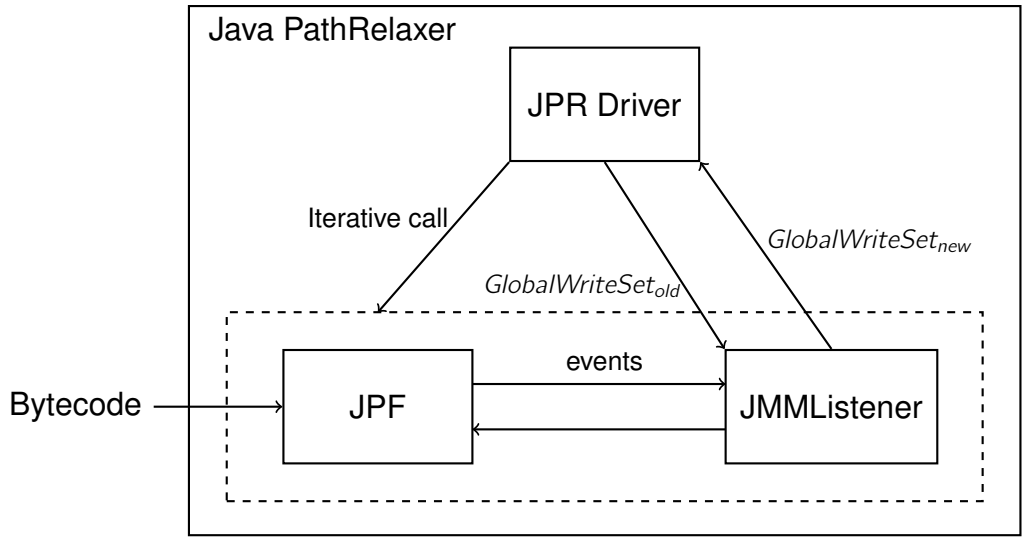

Figure 5-3. The overall structure of Java PathRelaxer (JPR).

We realized the algorithm described in §3.3 as a standalone project on top of JPF. The structure of the implemented JPR is shown in Fig. 5-3. Basically, there are three components: 1) JPR driver, 2) JPF core, and 3) **JMMListener**. JPR driver realizes **JMMAwareJPF** algorithm (Fig. 3-6). It iteratively calls JPF, which is registered with **JMMListener**. **JMMListener** realizes the algorithm shown in Fig. 3-7 and Fig. 3-8. The program being verified is in Java bytecode (.class file), the compiled code for Java virtual machine to execute. The specification is written in terms of assert statements.

Initially, $GlobalWriteSet_{old}$ that is passed between iterations is empty. Before each iteration, JPR driver passes the $GlobalWriteSet_{old}$ of to **JMMListener** and registers JPF with **JMMListener**. JPF takes the Java bytecode of the target program and does model checking. At each event (scheduling event or VM event), JPF notifies **JMMListener** which accordingly takes operations on the metadata (§3.2). On the other hand, **JMMListener** may also influence the state exploration procedure of JPF by adding more data choices, or by forcing JPF to do state backtrack operations. At the end of each iteration, JPR driver gets the $GlobalWriteSet_{new}$ from **JMMListener** and compares it with $GlobalWriteSet_{old}$. The iteration process stops when a fixed point is achieved (i.e. $GlobalWriteSet_{new} = GlobalWriteSet_{old}$). If the assertions in the target program

are satisfied during this process, then the program must be correct under JMM. If an assertion is violated at then end of a path, JPR stops and reported an exception. Remember JPR overapproximates JMM (§4.2). It generates more executions than JMM, so it is possible that a JMM-legal program fails in JPR.

5.3 JPF-related Implementation Issues

In this section, we discuss some JPF-related implementation issues and present our solutions.

Heap Structure. JPF uses a *heap* structure to maintain the objects and arrays. Each object or array occupies an element (called *ElementInfo*) on the heap. The element is uniquely identified by an index. So the heap can be viewed as a collection of *ElementInfo*s. The fields and array members are stored inside the element as a *cell*. An object field is represented by *FieldInfo*. In §3.2, we used Loc (the memory location) as the *key* for $WriteSet$, $Read$, $Write$, etc.. In JPF, the key is represented by *ElementInfo*, and *FieldInfo* or array element index:

- Field access (obj.fi): (class name)@(object index).(field name)
- Array access (arr[i]): (array type)@(array index).(element number)

Action ID. As stated in §5.1, JMM is ambiguous on the action ID. We explained 4 schemes for action ID. When implementing JPR, we implemented **scope**, **occurrence**, and **occurrence-val**. **value** was not realized because it suffers non-unique problem. When encountering an action, an actionID is retrieved from different implementations of $getID()$ method.

5.3.1 Bytecode-action translation

JPF is based on Java bytecode; The program being verified is specified bytecode, but JMM is defined on top of memory related actions. Bytecode is a set of instructions designed for Java Virtual Machine (JVM). It is in very low level and is stack-based; The bytecode instructions operates on one operand stack. The full list of bytecode can be found at [55]. In JMM, an action is represented by $\langle t, k, v, u \rangle$, where k can be

non-volatile read/write, volatile read/write, lock, unlock, and other special synchronization actions such as thread start, write to the default values, etc. It is defined in higher level than Java bytecode. Before implementing JPR, we need to form a mapping from Java bytecode to JMM actions.

Table 5-1. Java bytecode-JMM action mapping.

Java Bytecode[a]	JMM Action
getfield, *getstatic*	non-volatile read or volatile read
putfield, *putstatic*	non-volatile write or volatile write
aaload, *iaload*, *faload*, *baload*, *caload*	non-volatile read
aastore, *iastore*, *fastore*, *bastore*, *castore*	non-volatile write
monitorenter	lock
monitorexit	unlock
new, *newarray*	write to the default values
invokevirtual	thread start, thread join

[a] Only memory related bytecodes are listed.

Table 5-1 summarizes a bytecode to action mapping. *getfield* and *getstatic* retrieve value from a static[2] or non-static field, then push the value onto the operand stack. These two instructions can be treated as JMM read actions. Whether the action is volatile or non-volatile can be determined by referring to the field declaration. *putfield* and *putstatic* set a field with the value on top of the operand stack. Like *getfield* and *getstatic*, they correspond to JMM write actions. *getfield*, *getstatic*, *putfield* and *putstatic* are grouped by JPF as *FieldInstructions*.

aaload, *iaload*, *faload*, *baload*, and *caload* are array reading instructions. They retrieve an entry value from an array and place the value on the operand stack. The type of the array is distinguished by their first letter. For example, *i* means integer array, and *a* means reference array. Similarly, *aastore*, *iastore*, *fastore*, *bastore*, and *castore* are array

[2] Static fields are also called class variables. They are special fields that are associated with the class, not a specific object.

writing instructions. They get a value from the top of the stack and store it to an array entry. The array load/store instructions can also be viewed as JMM read/write actions. However, none of them can be volatile actions. Although an array can be declared as volatile, it only guarantees reads to the reference of the array see the most up-to-date value, but there is no guarantee for the individual entries. In JPF, the array instructions are categorized as *ArrayLoadInstructions* and *ArrayStoreInstructions*.

monitorenter and *monitorexit* are monitor instructions. *monitorenter* gives the executing thread the ownership of an monitor if there are no other threads owning that monitor. *monitorexit* releases the monitor from the executing thread. These two instructions are mapped to JMM's lock and unlock respectively. JPF group them as *LockInstructions*.

new and *newarray* allocate memory space for an object and an array respectively. The object fields and array entries are initialized to the default values. They can be mapped into JMM's write to default values. Based on [36, §17.4.4], the write of default value to each variable synchronizes-with the first action in every thread. This is not specified in the algorithm because it is special. We will talk more on this in a following subsection.

The thread start and thread join are handled by virtual methods start() and join() in Java. In bytecode, *invokevirtual* dispatches to a virtual method. In JPF, thread start is also captured by *threadStarted* event.

5.3.2 JPF state representation

JPF's state is represented by *gov.nasa.jpf.jvm.SystemState* class. It mainly captures the choices (called *ChoiceGenerator*) associated with the state. In §3.2, we mentioned that in JPR, JPF's state representation is expanded with metadata $\Sigma = (Path, WriteSet, ActionSet, HBSet, ImposeSet, Read, Write, ThreadLast)$. However in reality, we use a separate stack (called *state stack*) to record the metadata in the current implementation of JPR. This stack operates together with JPF path exploration, and is maintained by

JMMListener. When a *stateAdvanced* event is captured by the listener, the state stack pushes the current state Σ onto the stack. Similarly, at *stateBacktracked* event, the top of state stack is removed and copied to Σ. An intuitive alternative approach would have been to extend JPF's *SystemState* with the metadata. This would have simplified the control of JPR; when an ill-formed path is generated, simply requesting a backtrack would suffice. However, given the lack of an interface allowing the extension of JPFs state representation, following the alternative approach would have required modification of jpf-core. As an extension, it is not desirable to modify the kernel of JPF.

5.3.3 Garbage collection

Garbage collection (GC) is a memory management mechanism used by programming languages (Java, C++, C#, Lisp, etc.) to recycle memory spaces that are no longer in use. A *garbage* in Java can be viewed as an object that is not referenced. GC is an important mechanism because it allows programmer to reuse memory space.

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

Thread 1	Thread 2
h = new Helper(3);	
	h = new Helper(5); int r2 = h.x;

Figure 5-4. JPF Garbage Collection: After termination of Thread 1, the object created by Thread 1 will not be seen by Thread 2.

JPF also has GC features. Typically when a thread terminates, all the object created in this thread will be garbage collected. See the execution sequence shown in Fig. 5-4. Suppose Thread 1 first creates an instance of *Helper* at memory location $L1$ and assigns it to the shared reference h, then Thread 1 terminates. Thread 2 creates another *Helper* instance at location $L2$ and assigns it to h, and access field x of that reference. According to the JMM, the read in Thread 2 could return either 3 or 5 (the field values of *Helper* instances created at $L1$ and $L2$ respectively). However, because of

the termination of Thread 1, the instance created in $L1$ is considered as "not referenced" and is automatically garbaged collected by JPF. In order to allow such results, JPF garbage collection feature should be turned off for shared references. In JPF, we may tell the heap memory to stop garbage collection at some reference by calling *gov.nasa.jpf.jvm.Heap.registerPinDown()* method.

5.3.4 Reading future objects

Under JMM, a non-volatile read may see any write, either in the past or in the future, to that variable, as long as happens-before consistency is maintained. There is no problem when reading from a future write to variables of primitive data types (i.e. int, float, double etc); We simply retrieve the value from $WriteSet$ and put it on the operand stack to let JPF continue. However, reading a reference from future write becomes problematic because the object at that reference is not yet created at the time of read. When JPF is trying to access the reference, a null pointer exception will be thrown.

As an example, see the execution sequence shown in Fig. 5-5. Suppose in the first iteration of JPR, Thread 1 creates a $Helper$ instance at $L1$ and Thread 2 creates another instance at $L2$, so in the end, $WriteSet(h)$ contains two pairs $(a_1, L1)$ and $(a_3, L2)$. In the second iteration, given that execution sequence, the read at a_2 may see either the instance at $L1$ (previous write) or $L2$ (future write), but the instance at $L2$ has not been created at that time, so an exception will be thrown from JPF if reading from that reference.

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

Thread 1	Thread 2
a_1 h = new Helper(3);	
a_2 int r1 = h.x;	
	a_3 h = new Helper(5);

Figure 5-5. Read 'future' object: Null pointer exception is thrown when Thread 1 reads the object that has not been created by Thread 2.

To solve this problem, we apply lazy object initialization strategy; When reading from a future write to a reference type and the object is not yet created, then JPR arbitrarily creates an object at the specific location on the heap. This will let JPF go without breaking at an exception. When the future write (i.e. object creation) actually happens, it can detect that the object has already been created.

5.3.5 Checking program properties

Assertion. When checking program correctness, ordinary Java assertions are generally used. In standard JPF, assertion violations are caught by JPF's generic *NoUncaughtExceptionProperty*; During model checking, JPF explores all possible interleaving of instructions and throws a *NoUncaughtException* immediately after an assertion violation occurs. JPF stops when an exception is thrown.

JPR on the other hand, does not report assertion errors immediately. Instead, it delays the reporting of the error until the end of each executing path. The reason behind it is that a read may first see a future write and impose it with the value it sees, but the imposed value might not be justified when the write occurs (Fig. 3-8, line 42), or the write doesn't execute at all (Fig. 3-7, line 19). In both cases, the path will be discarded. This means in a path, a read may initially see an invalid value and the whole path will be discarded later on. In JPR, an assertion error will be detected when reading the invalid value, but not reported until the end of the executing path is successfully reached.

For example, see the execution sequence shown in Fig. 5-6. In the 1st iteration of JPR, Thread 2 writes 1 to x. In the 2nd iteration, Thread 1 reads x as 1 and imposes the write of x in Thread 2 (underlined action) to write 1. Then the assertion in Thread 2 is violated. Now, JPR does not report the error here because the entire path will eventually

Initially, x == y == 0, x and y are non-volatile variables

Thread 1	Thread 2
r1 = x; read 1 (future), *impose* y = r1; write 1	

	r2 = y; read 1 (previous)
	assert(r2 != 1);
	if(r2 == 0)
	  x = 1;
	else
	  x = 0;

Figure 5-6. In the 2nd iteration, the assertion is violated, but the path will also be discarded later, because the imposed value is not justified.

be discarded later because Thread 2 will now write 0 (not 1) to x in this case[3] (i.e. imposed value is not justified).

Report Scheme. Java assert statement can only check *universally* held properties, but it is insufficient to check properties such as checking the *existence* of some behaviors which is intensively used in [41] in reasoning about JMM-legal behaviors. See the example shown in Fig. 2-1, assert statement can check properties as "$r1 == r2 == 42$ is prohibited in any of the executions" by adding "$assert(!(r1 == 42 \&\& r2 == 42))$" at some point of the program. Using *Computation Tree Logic* (CTL) [20] formula, it can be written as $AG(\neg(r1 == r2 == 42))$. Here $AG\phi$ means along all paths, ϕ holds on the entire path. But assertions cannot check properties like "$r1 == 1$ and $r2 == 2$ is allowed in *some* executions", or $EF(r1 == r2 == 42)$ in CTL. Here $EF\phi$ means there exists at least one path that ϕ eventually holds.

To overcome this, we implemented a reporting scheme: at the end of each legal executing path, a list of records that correspond to all the read actions will be written to a report file. Each record contains the value it reads and the source write action's line

[3] Note that in this case, the write will not be justified no matter what kind of action ID scheme is applied (**scope**, **occurrence**, or **occurrence-val**).

number. It is in this format: ⟨ [read line number], [thread ID], read(field) = [value] from [line number of source write] [thread ID of source write] ⟩ Here is an example,

9 Thread−0 read(ttt1@50.x) = 1 from 22 Thread−1.

The properties that cannot be specified by assert statements could be checked by manually analyzing the generated report file.

5.4 Non-JPF Implementation Issues

In this section, we discuss some non-JPF implementation issues.

5.4.1 Data types

In §3.2, *Val* is used by *WriteSet*, *ImposeSet*, *Read*, and *Write*. We mentioned that *Val* is the domain of general value. The data type could be either primitive types (int, long, float, double, boolean) or reference. In JPR, *Val* is implemented as the interface *Value*. The value of each data type is a class that implements *Value*. The class diagram is shown in Fig. 5-7[4] .

TYPE is an enumeration with each element corresponds to a data type. In JPF, the reference of an instance is the index on the heap, so *RefValue* can be viewed as a special type of int. Each data type has a default value *DEFAULT*. The default values of the data types shown in the figure are listed in Table 5-2[5] . A complete list can be found in [36, §4.5.5]. When a new instance or array is created, the fields or the array elements will be initialized to the default values of the data type. We will explain the instance/array creation in §5.4.2.

5.4.2 Object and array creation

Among the synchronizes-with rules in [36, §17.4.4], there is an interesting rule about the write to the default values. It says "The write of the default value (zero, false or null) to each variable synchronizes-with the first action in every thread." The rule implies

[4] Other data types are not shown in the figure for brevity.

[5] In JPF, null is represented by integer -1.

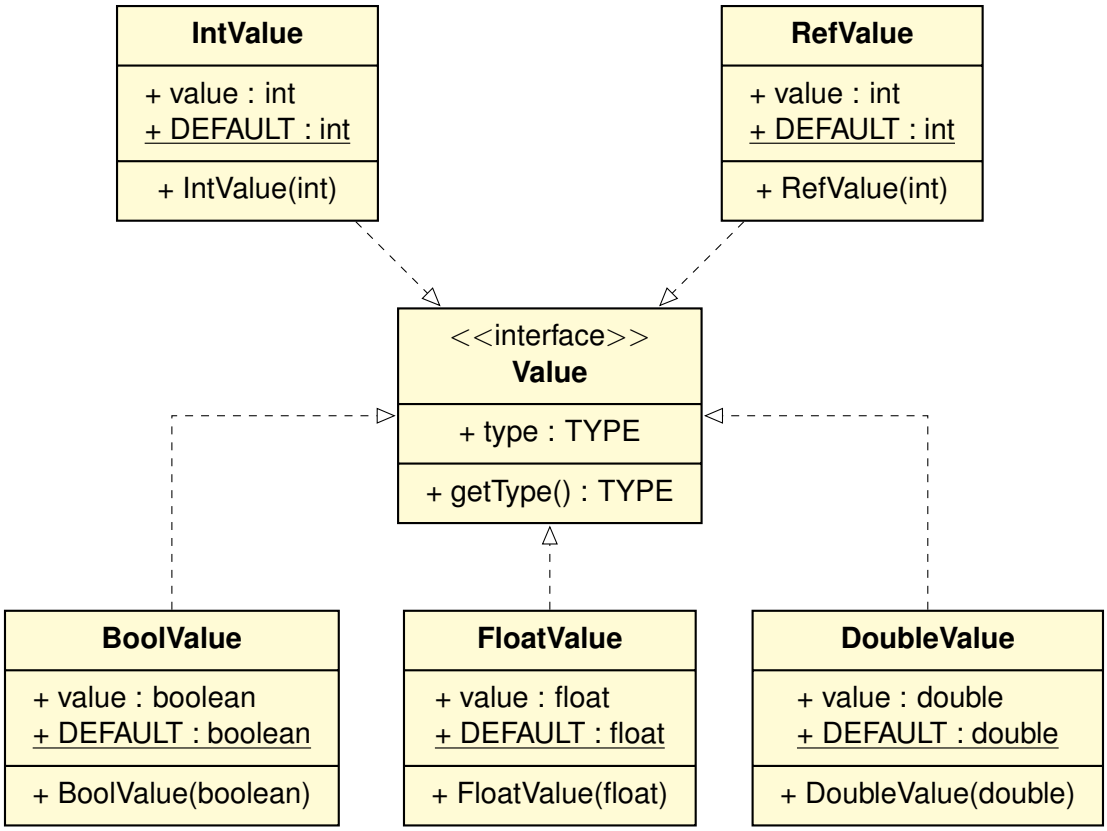

Figure 5-7. Class Diagram of Data Types.

Data Type	Default Value
boolean	false
int	0
float	0.0f
double	0.0d
reference	null

Table 5-2. Default values in Java.

that before the object containing the variable is allocated, there should be a write to the default value. Conceptually, at the start of the program, every object is created with default value written to it. However, this is not practical for JPF to capture; before actual execution of the program, we don't know exactly which object will be created. This rule requires special treatment and therefore is not included in **JMMListener** algorithm (Figs. 3-7 and 3-8).

```
case OBJECT/ARRAY CREATED:
    for each field of the object (or the array element) fld do
        Create naction = (naid, init_thread, WRITE, fld)
        ActionSet ← ActionSet ∪ naction //add naction to action set
        let def be the default value of fld's data type
        //update Write and WriteSet with default value
        Write(naid) ← def
        WriteSet(fld) ← WriteSet(fld) ∪ {(naid, def)}
        //update ≤hb due to default value rule of ≤sw
        for each thread id ttid do
            let first be first.tid == ttid ∧ first.kind == THREAD_START
            HBSet ← HBSet ∪ {(naid, first.aid)}
case THREAD STARTS:
    let the started thread id be tid and id of the thread that starts it be ptid
    Create action = (aid, tid, THREAD_START, undef) //we don't care the loc
    ActionSet ← ActionSet ∪ {action} //add action to action set
    HBSet ← {(ThreadLast(ptid), aid)} //update ≤hb due to thread start rule of ≤sw
    ThreadLast(tid) = aid
    //update ≤hb due to default value rule of ≤sw
    for each action naction s.t. naction ∈ ActionSet ∧ naction.tid == init_thread do
        HBSet ← HBSet ∪ {(naction.aid, aid)}
```

Figure 5-8. Algorithm that handles object/array creations, an extension from Fig. 3-8

In our implementation of JPR, we maintain a special thread called *init_thread*. The actions of *init_thread* are all writes to the default vallues. Unlike other threads, *init_thread* is dynamically constructed; The set of actions is not fixed, but keeps on expanding when an object or array is created by some thread. The detailed algorithm is shown in Fig. 5-8. Here the object and array instantiation are represented by Java's bytecode *new* and *newarray* respectively. JPF can capture the execution of the two. Upon object/array creation, we loop over the fields of the object or the array elements, and create a write action for each of them. The value that associated with the write is the default value of the corresponding data type. The *ActionSet*, *Write*, and *WriteSet* are updated because of the default write. The *HBSet* is also updated by looping over all the started threads and adding an happens-before edge from the default write to the THREAD_START action of that thread. Here THREAD_START is a special action kind for thread start actions. It is also a pair of (t, k, v, u), but the memory location v is not defined.

Besides the operations at object/array creation event, we also need to take care of thread start event. When a thread is started, we must loop over the actions of *init_thread* and add $\leq_{hb}$ edge from them to the THREAD_START action. Also, we need to update the $ActionSet$, $ThreadLast$, as well as the $HBSet$ due to the thread start rule of synchronizes-with order.

5.4.3 Checking happens-before consistency

For the JPR metadata, the data structures used for $WriteSet$, $Write$, and $Read$ are hash tables; $ActionSet$ and $ImposeSet$ are simple sets of elements. For $HBSet$, there are many ways to implement. The difference between $HBSet$ and other metadata is that $HBSet$ is expanded by direct $\leq_{hb}$ relations, but checked by a transitive closure. An direct way of implementing $HBSet$ is to maintain the transitive closure of it. We may use the kleene closure $HBSet^*$ to denote the transitive closure. This implementation facilitates the look up performance, but consumes large memory spaces.

In JPR, we construct a *directed acyclic graph* (DAG) where actions are nodes and an directed edge between two nodes a_i and a_j implies that $a_i \leq_{hb} a_j$. When checking happens-before consistency (see item 9 of Definition 3) between a non-volatile read action r of variable var and a non-volatile write action w where $w = V(r)$, the graph is traversed to find possible paths between the two actions.

i) The search stops when we find a path from r to w, which indicates a violation of "for all reads r of variable v, $r \not\leq_{hb} W(r)$"

ii) The search stops when we find a path from w to r that contains another intermediate write action w' to the same variable, which indicates violation of "if $W(r) \leq_{hb} w \leq_{hb} r$ and w writes to v then $W(r) = w$."

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

5.4.4 Working with Java Racefinder

The JMM guarantees that if a program is free of data races on all of its sequentially consistent executions, then all of its executions are sequentially consistent. Such program is called DRF program. In any execution of DRF programs (Definition 6), a read only sees the value of the most up-to-date write, but not other writes (other past writes and future writes). This guarantee is proved by [7] and is listed in Theorem 2.1. For DRF programs, because its executions are sequentially consistent, so the standard JPF is sufficient to carry out model checking. Standard JPF doesn't have iterations, and requires less memory space than JPR for not maintaining metadata (§3.2). Therefore, if we know that a program is DRF, then we may improve the performance by simply running standard JPF.

Java Racefinder (JRF, now called jpf-racefinder) [45, 48, 49] is a tool that precisely identifies DRF programs. If no data races are reported by JRF, then the program is DRF. JRF is based on JMM's definition on DRF; "If all sequentially consistent executions of the program are free of data races" [60], it is a DRF program. JRF is a standard extension to JPF. During its path exploration, JRF maintains a so-called h set that contains all the variables that are not involved in any data races in the current SC execution so far. h set is expanded or shrunken according to JRF operational semantics. At each non-volatile read/write action, JRF checks whether the target variable is included in the h set or not. "When this condition holds for all non-volatile reads and writes in an (SC) execution, the execution is *h-legal*." [48]. It has been proved that *h-legal* executions are free of data races. JRF is precise. It detects all the data races without false alarms.

With the presence of JRF, we have the calling structure shown in Fig. 5-9. We firstly run JRF on the target program to check whether the program contains data race or not. If the program is DRF, we simply run standard JPF; if not, then we run JPR.

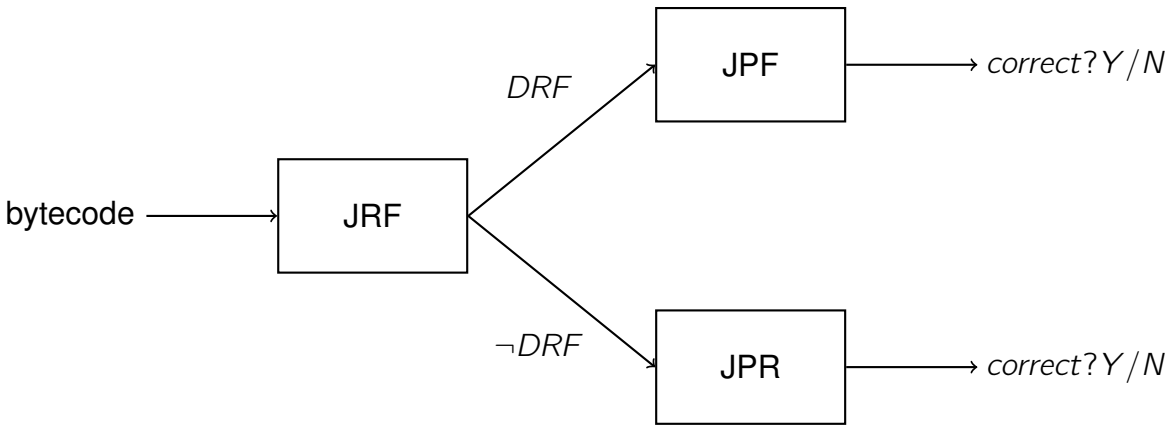

Figure 5-9. Working with JRF.

The following two issues are about the h set in JRF and its potential improvements on JPR. We will show that they cannot be used in JPR.

h **set and** $WriteSet$. In JRF, we check data race when executing non-volatile read or write on variable x in thread t with $norace$ rule: $norace(x, t) = x \in h(t)$. Basically, the h set contains the variables that are not involved in the data race so far. Because we run JRF before JPR, one might believe that we could only maintain $WriteSet$ only for variables that are not in h set for any executions generated by JRF, and treat the variables within h set as volatile variables (i.e. read the most recent write). However, h set is defined under the context of SC memory model. A variable not involved in any sequentially consistent executions may still be racy in some sequentially inconsistent executions.

Initially, x = y = z = 0

Thread 1	Thread 2
1 r1 = z;	5 r2 = y;
2 if(r1 == 1)	6 z = r2;
3 x = 1;	7 r3 = x;
4 y = 1;	

Figure 5-10. A not racy variable under SC may be racy under non-SC.

See the example in Fig. 5-10. This program is not DRF. Using JRF, we may get two data races involving y and z, but x is not reported. In any legal sequentially consistent executions, the write to x at line 3 will not execute, so x is not involved in data races. However, under JMM, the read of y in line 5 may see the future write in line 4 to let the write of x execute. Then x could still be involved in a data race. So x cannot be treated as volatile variable. Therefore, if a program is detected as non-DRF by JRF, we must maintain $WriteSet$ for *all* the non-volatile variables in the program.

h **set and** $HBSet$. Based on JMM's data race definition (Definition 5), we should know all the happens-before relations in the current execution when checking data race. Typically the $\leq_{hb}$ relations form a DAG as discussed in §5.4.3. JRF however, gets around the expensive construction and searching of the graph by h set. In JRF, each synchronization address (volatile variable or monitor) and thread has an h set. Formally, h set is a mapping of $h : SynchAddr \cup Threads \rightarrow 2^{Addr}$. Variables are added to h set by object instantiation, and removed from h set by non-volatile write. h set is copied between threads and variables by release and acquire actions;

One could believe that h set could be used in JPR also. However, h set answers "whether two actions are ordered by $\leq_{hb}$ or not", which is a yes or no question. h set doesn't care the exact $\leq_{hb}$ order between actions. JPR however need to know the exact $\leq_{hb}$ order between actions in order to check happens-before consistency (rule 9 of Definition 3), so a simple set that contains "non-racy" variables is not enough.

CHAPTER 6
EXPERIENCE AND EVALUATION

In this chapter we present the experience and evaluation of JPR and its algorithm. We firstly present some benchmark examples. There examples are used to show that

- JPR can generate all the executions that are allowed by JMM and can rule out forbidden executions to a certain degree.
- JPR could be soundly used to identify benign data races.

In the second section, we list the experiment results of the examples. We compare different actionID schemes, and point out common benign data race patterns.

Finally, we show that the idea of JPR is not restricted to JMM, but can be further extended to other relaxed memory models. We present the revised algorithm for PSO, which is a hardware memory model used by SPARC systems, and explain how it works. We also present a similar algorithm for TSO in Appendix B.

6.1 Test Suites

To evaluate JPR, we ran it on three groups of test programs. The first group, labeled $tc1$ through $tc20$ are the test cases derived from the JMM Causality Test Cases [41] (also listed in the Appendix), which were designed to illustrate the properties of the JMM. For these, we output the paths generated by JPR and compare them with the legal executions according to JMM. All legal executions were generated by JPR with $tc5$ and $tc10$ generating forbidden executions. $tc5$ is the example shown in Fig. 4-1 and discussed in §4.2. $tc10$ is similar to $tc5$ but with some branch conditions. $tc14$ and $tc15$ are not tested because they are DRF programs which are identified by JRF, and can be analyzed by standard JPF instead of JPR.

The Java source code of $tc10$ is shown in Fig. 6-1. [41] claims that $r1 == r2 == 1 \wedge r3 == 0$ is forbidden under JMM because of the data and control dependencies; $r1$ and $r2$ cannot be 1 unless $r3$ is 1. This test case has a similar effect as of the program without branch conditions. The explanation of the overapproximation can be found at Fig. 4-2.

```
public class tc10{
    static int x = 0, y = 0, z = 0;        // shared variables
    public static void main(String args[]){
        new Thread(new runnable(){
            public void run(){
                int r1 = x;
                if (r1 == 1)    y = r1;
            }
        }). start ();
        new Thread(new runnable(){
            public void run(){
                int r2 = y;
                if (r2 == 1)    x = r2;
            }
        }). start ();
        new Thread(new runnable(){
            public void run(){
                z = 1;
            }
        }). start ();
        new Thread(new runnable(){
            public void run(){
                int r3 = z;
                if (r3 == 1)    x = r3;
            }
        }). start ();
    }
}
```

Figure 6-1. Java code of test case 10 from [41]

The second group contains more realistic examples where assertions were applied to test whether the data races were benign. These include *hash* (with 2- and 4-thread versions), *hash2*, *isprime*, *lazy-fib*, and *badbit*.

hash is derived from Java's String class. In *hash*, the hashCode method (Fig. 6-2 with line 15 deleted) contains a racy lazy initialization of its hash field; the read of hash (Line 7) and the write of hash (Line 13) may form a data race. This race is benign because in all legal executions, even the sequentially inconsistent ones, a call to the hashCode method will always return the correct hash code value. The assertions applied in both the 2-thread version and 4-thread version of *hash* confirm this finding.

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

Figure 6-2. Recap of Fig. 2-16. The driver class is shown in Fig. 2-17. The data races are benign if line 15 is removed from the program. Otherwise, the races are not benign.

The result is that the String class can be treated as if it were immutable[1] , even though the hash field is not marked final and is not set in the constructor, and its methods may be safely invoked without synchronization.

hash2 on the other hand, calls a slightly different version of hashCode (Fig. 6-2) where the returned value is reread from hash (Line 15). This is still correct under SC memory model, but under the JMM, the race is not benign; a thread calling hashCode could get the initial value 0 (past, not up-to-date value) instead of the correct hash code.

The Java version of *isprime* [74, §2.6] is shown in Fig. 6-3. The *pflag* array records the already identified prime numbers and is lazily initialized; when a prime number is identified by some thread, the corresponding flag is changed to false. In *isprime*,

[1] In Java, immutable object means the object's state cannot be modified after instantiation[35, §3.4]. Immutable objects are always thread-safe.

```
public class Prime{
    private boolean pflag[] = new boolean[N]{true, true, ...}          // N is an integer
    ...
    public boolean isprime(int v){
        int bound = (int)Math.floor(Math.sqrt((double)v)) + 1;
        for(int i = 2; i < bound; i ++){
            if (!pflag[i])
                continue;
            if (v % i == 0){
                pflag[v] = false;
                return false;
            }
        }
        return true;
    }
}
```

Figure 6-3. Detect prime numbers by lazy initialization of *pflag* array.

data race occurs at lines 7 and 10, when multiple threads read (line 7) and write (line 10) elements of the shared *pflag* array without synchronization. These accesses are racy and reads may see stale, but not up-to-date values. Note that accesses to array elements in Java do not have volatile semantics, so even if the array is marked as volatile, we cannot guarantee visibility of most recent value for the reads. In this program, however, reading a stale value only affects performance but not overall correctness; The *pflag* only accelerates the identification process. When a stale value is read by a thread, the loop will be executed more times, but it always correctly identifies the prime numbers.

lazy-fib is shown in Fig. 6-4. Given an integer, the calculateFib method calculates the fibonacci number lazily. The shared *fib* array is used to record the fibonacci numbers that have been calculated so far. Initially the elements in *fib* are 0 (the default int value is 0). If calculateFib method is called by multiple threads, then there is a data race between line 5 (read) and line 11 (write). It is possible that the read in some thread sees 0 instead of the up-to-date fibonacci number calculated by other threads, and calculates the number again. But this doesn't affect the correctness of the program; the correct

```
public class Fibonacci{
  private int fib [] = new int[20];
  ...
  public int calculateFib(int n){
      int fib = fib[n];
      if (fib == 0){
          if (n == 1 || n == 2)
              fib = 1;
          else
              fib = calculateFib(n − 1) + calculateFib(n − 2);
          fib[n] = fib;
      }
      return fib;
  }
}
```

Figure 6-4. Calculating fibonacci number by lazy initialization.

fibonacci number is guaranteed to be returned to the caller. Applying assert statements, JPR doesn't detect violations.

badbit is derived from [74, §2.6]. The Java version of the program is listed in Fig. 6-5. The class BadBit has a shared variable *isbad* and a shared array *dataArray*. Two worker threads are calling checkBadArray method at the same time with each thread checking a different section of *dataArray*. The checkBadArray method loops over the elements in the specified section of *dataArray*. In each loop, it checks *isbad* field to see if it is set to 1 by other threads. If not, it checks the elements in the section and returns as soon as an element is bad (i.e. 1) and assigns *isbad* to 1. After the termination of the two threads, the main thread checks *isbad* to see if there are any bad bits identified. Because *isbad* is not volatile, there is a data race between line 17 and line 20. However, this data race is benign. Suppose one thread doesn't see the updated value of *isbad*, the only effect would be more iterations, but in the end, the main thread always gets the correct *isbad* value after joining of the worker threads.

The third group contains some well-known synchronization problems. They are all correct under SC memory model, but fail under JMM. This group includes *dcl*, *peterson*,

```
public class BadBit{
    private int isbad = 0;
    private static int [] dataArray = new int[]{0, 0, 0, 0, 0, 0, 0, 1, 0, 0};
    public static void main(String args[]){
        Thread t1 = new Thread(new Runnable(){
            public void run(){ BadBit.checkBadArray(0, 4);
        });
        Thread t2 = new Thread(new Runnable(){
            public void run(){ BadBit.checkBadArray(5, 9);
        });
        t1.start (); t2.start ();
        try{ t1.join (); t2.join (); }catch(Exception e){}
        assert(isbad == 1);
    }
    public int checkBadArray(int start, int end){
        for(int i=start; i<=end; i++){
            if (isbad == 1) return;
            else{
                if (dataArray[i] == 1){
                    isbad = 1;
                    return;
                }
            }
        }
    }
}
```

Figure 6-5. Program checks if there is a bad bit in an array.

and *dekker*. Although JPR generates more behaviors than JMM, which means it has false alarms in identifying harmful data races, we show that JPR's identifications on these test cases are correct.

dcl is the infamous *double-checked locking* (DCL) idiom [9] which attempts to reduce locking overhead by lazy initialization of an object. In the test case, two threads call the getHelper method of Foo shown in Fig. 6-6. The read of *helper* (line 7) is placed outside the synchronized block, while the construction of *helper* (line 10) is placed within the synchronized block. There is a data race between the two actions. Suppose at one time, Thread0 is executing line 10 while Thread1 is executing line 7 just before Thread0 has finished construction of *helper*. Then Thread1 detects that *helper* is not

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

Figure 6-6. Double checked locking

empty and returns it immediately without entering the synchronized block. In this case, Thread0 is actually returning a partially constructed object, allowing other threads to see a partially constructed object. This is the unsafe publication problem. To capture this bug, we inserted assertions to check if the reference returned from getHelper() is correctly constructed or not (line 24 and 30); if correctly constructed, the x field of the reference should not be 0 (the initial value). To solve the DCL problem, *helper* should be declared as volatile.

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

Figure 6-7. Peterson's algorithm: guarantees mutual exclusion under SC, but fails under JMM.

peterson (Peterson's algorithm) and *dekker* (Dekker's algorithm) are implementations of the classic mutual exclusion algorithms without using volatiles. They guarantee mutual exclusion under sequential consistency, but fail in relaxed memory models such as JMM. Peterson's algorithm is shown in Fig. 6-7. Under SC, line 29 in Thread0 is mutually exclusive with line 19 in Thread1. After termination of the two threads, x should always be 2. Under the JMM, it is possible that Thread1 writes $flag[1]$ to true at first but Thread0 later on still reads $flag[1]$ as the old value false and hence skips the busy wait

(line 9). Then both threads will be executing the mutually exclusive regions. In this case, the two x++ will interfere with each other and the assertion (line 29) will fail. Assertions inserted to check non-interference in the critical sections in *peterson* failed as expected.

Dekker's algorithm was proposed by Th. J. Dekker, and is presented in [27, §2.1]. It is one of the famous solutions to guarantee mutual exclusion of two threads executing on a critial section under sequential consistency. It applies a similar but more complicated logic to Peterson's algorithm (Fig. 6-7). Dekker's algorithm is proved to be more efficient than Peterson's algorithm, but cannot be generalized to programs with more than two threads. The Java version of Dekker's algorithm with assert statements is shown in Fig. 6-8. In this algorithm, *flag* array and *turn* are shared variables that are used to guarantee mutual exclusion. The increment of x within the two threads are critical sections. Before entering the critical sections, each thread performs a check on *flag* and *turn* to see if it is its turn to enter. Different from Peterson's algorithm, Dekker's algorithm busy waits on *flag* and *turn* in inner and outer loops. After the execution of the critical section, the right of entrance is handed to the other thread. Same as Peterson's algorithm, Dekker's algorithm guarantees mutual exclusion under SC memory model, but fails under relaxed memory models such as JMM and PSO. In Dekker's algorithm, operations on *flag* array and *turn* are not synchronized. There are no happens-before relations between the read and write of them. So it is possible that a thread reads stale values and skips the checks in lines 8 and 9, or 22 and 23 and both threads will be executing the critical section at the same time. Using JPR, the assert statement in line 39 failed as expected.

Besides Dekker's algorithm, a similar approach is Lamport's bakery algorithm [52], which also guarantees mutual exclusion by busy wait on shared arrays *choosing* and *number*. Lamport's bakery algorithm doesn't have a limit on the number of threads. However, it still fails under JMM because of the lack of happens-before relations.

```
// Global variables
boolean flag[] = new boolean[]{false, false};
int turn, x = 0;
...
class Thread0 extends Thread{
    public void run(){
        flag[0] = true;
        while(flag[1] = 1){
            if (turn != 0){
                flag[0] = false;
                while(turn != 0){}      // busy wait
                flag[0] = true;
            }
        }
        x++;                            // critical section
        turn = 1; flag[0] = false;
    }
}
class Thread1 extends Thread{
    public void run(){
        flag[1] = true;
        while(flag[0] = 1){
            if (turn != 0){
                flag[1] = false;
                while(turn != 1){}      // busy wait
                flag[1] = true;
            }
        }
        x++;                            // critical section
        turn = 0; flag[1] = false;
    }
}
// main thread
Thread0 t0 = new Thread0();
Thread1 t1 = new Thread1();
t0.start(); t1.start();
try{
    t0.join(); t1.join();
    assert(x == 2);
}catch(Exception e){}
```

Figure 6-8. Dekker's algorithm: guarantees mutual exclusion under SC, but fails under JMM.

The paths in which the test cases in the third group had assertion violations are legal according to JMM and therefore were detected by JPR but are not exhibited by

sequentially consistent programs. Standard JPF cannot detect these problems. For these test cases, JPR took less time and explored fewer states than JPF because the assertion violations terminated JPR before the full state space exploration was complete.

6.2 Performance and Evaluation

	#th	scope				occurrence				occurrence-val				JPF		
		iter	time	states	mem	iter	time	states	mem	iter	time	states	mem	time	state	mem
tc1	2	3	1.4s	164	15M	3	1.4s	164	15M	3	1.5s	173	15M	0.8s	44	15M
tc2	2	3	1.6s	320	15M	3	1.6s	320	15M	3	1.6s	377	15M	0.8s	54	15M
tc3	3	3	4.1s	2315	25M	3	4.1s	2315	24M	3	4.7s	2582	25M	0.9s	349	15M
tc4	2	3	1.3s	94	15M	3	1.3s	94	15M	3	1.4s	94	15M	0.8s	40	15M
tc5*	4	3	11.2s	6326	26M	3	12.3s	6326	26M	3	14.8s	6877	26M	1.2s	1169	15M
tc6	2	4	1.6s	161	25M	3	1.4s	125	15M	3	1.4s	125	15M	0.8s	34	15M
tc7	2	4	2.2s	496	25M	4	2.2s	496	25M	4	2.3s	557	26M	0.8s	64	15M
tc8	2	3	1.6s	148	15M	3	1.4s	148	15M	3	1.4s	156	15M	0.8s	44	15M
tc9	3	3	3.0s	1737	15M	3	3.0s	1737	15M	3	3.3s	1929	15M	1.0s	279	15M
tc9a	4	3	2.2s	880	15M	3	2.2s	880	15M	3	2.7s	914	15M	0.9s	261	15M
tc10*	2	3	5.7s	3233	25M	3	5.8s	3233	25M	3	5.8s	3233	25M	1.0s	477	15M
tc11	2	4	3.1s	1147	26M	4	3.2s	1147	26M	4	4.0s	1452	25M	0.9s	95	15M
tc12	2	3	1.5s	175	15M	3	1.5s	175	15M	3	1.5s	175	15M	0.8s	63	15M
tc13	2	3	1.2s	32	15M	3	1.2s	32	15M	3	1.2s	32	15M	0.8s	24	15M
tc16	2	3	1.4s	197	15M	3	1.4s	197	15M	3	1.5s	197	15M	1.0s	46	15M
tc17	2	3	1.9s	565	15M	3	1.9s	565	15M	3	1.9s	641	15M	0.8s	72	15M
tc18	2	3	1.9s	565	15M	3	1.8s	565	15M	3	2.0s	641	15M	0.8s	72	15M
tc19	3	3	5.2s	2205	25M	3	5.6s	2205	25M	3	5.5s	2502	25M	0.9s	381	15M
tc20	3	3	5.1s	2205	25M	3	4.9s	2205	25M	3	6.0s	2502	25M	0.9s	381	15M
hash	2	3	1.5s	237	15M	3	1.5s	237	15M	3	1.5s	237	15M	0.7s	60	15M
hash	4	3	38.3s	12442	33M	3	38.2s	12442	34M	3	38.6s	12442	34M	1.7s	3720	15M
hash2	2	3	1.3s	23	15M	3	1.3s	23	15M	3	1.3s	23	15M	0.8s	98	15M
isprime	2	3	2.0s	308	15M	3	2.1s	308	15M	3	2.2s	308	23M	0.9s	118	15M
lazy-fib	2	3	3.1s	280	15M	3	3.1s	280	15M	3	3.2s	280	15M	0.8s	86	15M
badbit	2	3	5.6s	1143	26M	3	5.2s	1143	26M	3	5.8s	1143	26M	0.8s	430	15M
dcl	2	3	1.1s	22	15M	3	1.2s	22	15M	3	1.2s	22	15M	0.9s	243	15M
peterson	2	3	1.5s	83	15M	3	1.5s	83	15M	3	1.5s	83	15M	1.0s	194	15M
dekker	2	3	1.3s	24	15M	3	1.2s	24	15M	3	1.2s	24	15M	0.9s	203	15M

Figure 6-9. Experimental results comparing the performance of JPR using ActionID approaches **scope**, **occurrence**, and **occurrence-val**, respectively. * means that JPR generates paths not allowed by JMM.

Representative results are listed in Fig. 6-9. The columns contain the number of threads, and for each action ID approach described above, the number of iterations of JPF required to converge, the total time, the number of states visited in the final iteration, and the maximum memory consumed, respectively. The final columns indicate the resource usage for standard JPF for comparison purposes. All testing was performed on a 2.27 GHz Intel(R) Core(TM) i5 CPU, 4 GB main memory, with 64-bit Windows 7 operating system, JDK 1.6, and JPF version 6.

From Fig. 6-9, we can see that JPR is able to reason about concurrent programs under JMM, while standard JPF cannot. JPR generates an overestimation of JMM-legal executions (additional behaviors of $tc5$ and $tc10$). Except *hash2*, *dcl*, *peterson*, and *dekker* where JPR caught assertion errors and terminated before complete exploration, JPR generally takes longer time and generates more states than standard JPF as expected due to JPR's iterative nature, and its exploration of more paths due to data non-determinsms. Even the 1st iteration takes longer time than standard JPF. This is shown in the experiment table in [43][2] . Also the average time per iteration is generally larger than the time of the 1st iteration, that's because of the monotone expansion of the *WriteSet*; more paths are explored in the following iterations.

The experiments reflect the factors that affect the JPR running time. The same as standard JPF, the number of threads is a main factor. The more the number of threads, the more scheduling choices to make when doing state advancement. A good example is the 2-thread *hash* and 4-thread hash. The algorithm in both test cases are the same except for the number of threads. The increase of threads may result in an exponential increase of time. Another factor is the number of shared non-volatile variables. The more the number of shared variables, the more data choices may be processed by JPF.

[2] The table would be overcrowded if the time consumption in the 1st iteration of JPR is included, so this information is not shown in Fig. 6-9.

For example, $tc11$ has 4 shared variables. Although it has only 2 threads, it takes much longer time than $tc1$ and $tc2$ which have 2 shared variables and 2 threads.

The Java source code of *tc11* is listed in Fig. 6-10. It has two threads and 4 shared integer variables w, x, y, and z with initial value 0.

```
public class tc11{
    static int w = 0, x = 0, y = 0, z = 0;                    // shared variables
    public static void main(String args[]){
        new Thread(new runnable(){
            public void run(){
                int r1 = z;
                w = r1;
                int r2 = x;
                y = r2;
            }
        }). start ();
        new Thread(new runnable(){
            public void run(){
                int r4 = w;
                int r3 = y;
                z = r3;
                x = 1;
            }
        }). start ();
    }
}
```

Figure 6-10. Java code of test case 11 from [41]

JMM causality test cases web page [41] claims that $r1 == r2 == r3 == r4 == 1$ is a legal behavior for *tc11* under JMM. The value 1 can be propagated to all the shared variables. Because there is no branch conditions, so the JMM interpretations are the same for **scope**, **occurrence**, and **occurrence-val**. This behavior is an existential condition, so we cannot use assert statement to verify it. All the possible outcomes of local variables $r1$, $r2$, $r3$, and $r4$ after execution are listed in Table 6-1. This table is a direct translation of JPR's report scheme. The values returned by the reads, as well as the write action they see are listed. $r1 == r2 == r3 == r4 == 1$ is the last row in the table.

Table 6-1. List of all the possible outcomes of local variables $r1$, $r2$, $r3$, and $r4$ after execution. Translated from the report scheme of JPR.

	r1	r1 read from	r2	r2 read from	r3	r3 read from	r4	r4 read from
1	0	line 2	0	line 2	0	line 2	0	line 2
2	0	line 2	0	line 2	0	line 9	0	line 2
3	0	line 2	0	line 2	0	line 2	0	line 7
4	0	line 2	0	line 2	0	line 9	0	line 7
5	0	line 2	1	line 17	0	line 2	0	line 2
6	0	line 16	0	line 2	0	line 2	0	line 2
7	0	line 16	1	line 17	0	line 2	0	line 2
8	0	line 2	1	line 17	1	line 9	0	line 2
9	0	line 2	1	line 17	0	line 2	0	line 7
10	0	line 2	1	line 17	1	line 9	0	line 7
11	0	line 16	0	line 2	0	line 9	0	line 2
12	0	line 16	0	line 2	0	line 2	0	line 7
13	0	line 16	0	line 2	0	line 9	0	line 7
14	0	line 16	1	line 17	0	line 2	0	line 7
15	1	line 16	1	line 17	1	line 9	0	line 2
16	1	line 16	1	line 17	1	line 9	1	line 7

Let's analyze the iterations of JPR. In the 1st iteration of JPR, only $r2$ may see value 1; In the 2nd iteration, $r1$, $r2$, and $r3$ may see 1; All the local variables may see 1 in the 3rd iteration, and the last iteration has the same result as 3rd iteration and JPR converges.

tc11 has a data dependencies of (in line number) $17 \rightarrow 8 \rightarrow 9 \rightarrow 15 \rightarrow 16 \rightarrow 6 \rightarrow 7 \rightarrow 14$. So from Table 6-1, we find that $r1$, $r3$, and $r4$ cannot be 1 unless $r2$ is 1. Also $r1$ can $r4$ cannot be 1 unless $r3$ is 1.

Initially, A == B == 0

Thread 1	Thread 2
r1 = A;	r2 = B;
if (r1 == 1)	if (r2 == 1)
B = 1;	A = 1;
	if (r2 == 0)
	A = 1;

Figure 6-11. *tc6*: "r1 == r2 == 1 is allowed" by JMM according to [41].

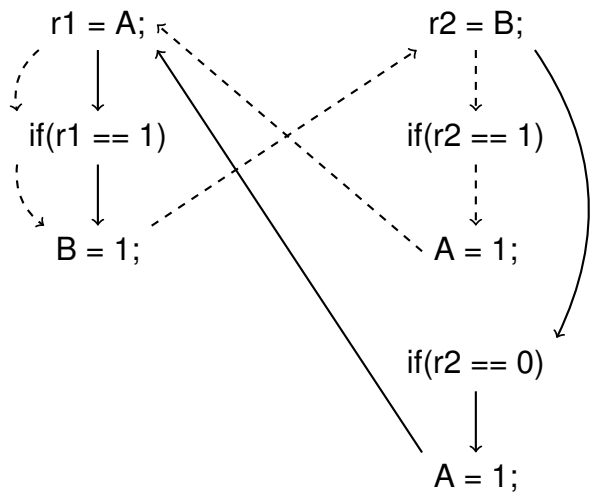

Figure 6-12. Data and control dependencies of Fig. 6-11. r1 == r2 == 1 can be generated by JPR if **scope** actionID scheme is applied.

The experiment results also reflect the different interpretations between action ID schemes of **scope**, **occurrence**, **occurrence-val**. Fig. 6-11 shows $tc6$. In [41], it says "r1 == r2 == 1" is allowed by JMM. This statement is true if **occurrence** or **occurrence-val** schemes are applied, but is false on **scope**. Using **scope**, the two $A = 1$s are different actions, so if $r2$ to get 1, then we cannot have the 2nd $A = 1$ committed. While the two $A = 1$s are the same action if **occurrence** and **occurrence-val** schemes are applied.

More interestingly, although $r1 == r2 == 1$ is forbidden by JMM if using **scope**, JPR can still generate this result as an overapproximation. Fig. 6-12 explains the reason. The solid arrows reflect the flow of value 1 in the 1st iteration of JPR. We see that 1 can be propagated into the execution via the second if of Thread 2. In the 2nd iteration (dashed arrows), $r2$ futurely read 1 from $B = 1$, and the first if executes. Finally, we still get $B = 1$ executed. Via this path, we can get $r1 == r2 == 1$ without forcing the second if to execute. The problem behind this is the lack of relation of impose values between iterations. JPR uses $ImposeSet$ to enforce read from future writes, but $ImposeSet$ is not passed between iterations. Passing $ImposeSet$ between iterations requires a more complicated algorithm.

```
@NotThreadSafe
public class UnsafeLazyInitialization{
    private static Resource resource;
    public static Resource getInstance(){
        if (resource == null)
            resource = new Resource();   //unsafe publication
        return resource;
    }
}
```

Figure 6-13. Unsafe lazy initialization.

Compared between actionID schemes; **scope**, **occurrence**, and **occurrence-val**, we find that their performance in JPR is similar with regard to time, the number of states, and memory consumed. **occurrence-val** slightly generates more states than **scope** and **occurrence** because it distinguishes the values of write actions. Although it is still difficult to answer which actionID scheme is better than the others, we recommend **occurrence**. **scope** has a good performance, but forbids too many behaviors such as Figs. 5-1a and 6-11. **occurrence-val** allows many behaviors, but generates more states, and makes JMM causality rules 4 and 7 redundant. **occurrence** on the other hand, is a more natural scheme. It allows most of the executions that **occurrence-val** allows, and has a good performance in JPR.

Using JPR, we found a common benign data race pattern: the lazy initialization. Test cases *hash*, *isprime*, *lazy-fib*, and *badbit* contain benign data races, and all applied lazy initialization. Lazy initialization "defers initializing an object until it is actually needed while at the same time ensuring that it is initialized only once."[35, §2.2] Lazy initialization follows a *check-then-act* idiom; The program first checks whether a field is initialized or not, if not then initialize it. Lazy initialization works correctly in single-threaded programs, but in multithreaded Java programs, it may suffer unsafe publication problem, which is a data race-related error. See the example shown in Fig. 6-13[35], suppose two threads T1 and T2 are calling getInstance method. T1 checks that *resource* is null and initialize it; T2 checks *resource* is not null and skip the *if*. Because the lack of happens-before

```
public class ConcurrentSkipListMap<K,V> extends AbstractMap<K,V>
    implements ConcurrentNavigableMap<K,V>, Cloneable, java.io.Serializable {
  private static final Random seedGenerator = new Random();
  private transient int randomSeed;
   ...
  private int randomLevel() {
        int x = randomSeed;
        x ^= x << 13;
        x ^= x >>> 17;
        randomSeed = x ^= x << 5;
        if ((x & 0x80000001) != 0)
            return 0;
        int level = 1;
        while (((x >>>= 1) & 1) != 0) ++level;
        return level;
    }
}
```

Figure 6-14. java.util.concurrent.ConcurrentSkipListMap

relation between the initialization of *resource* in T1 and the read of it in T2, a data race is formed. T2 may not get the up-to-date states of *resource*. DCL (test case *dcl*) uses lazy initialization but has the same problem. Typically, in order to solve this problem, the lazy initialization method should be synchronized.

However, if the lazy initialization is applied on variables of primitive types, the data race might be benign. No synchronization mechanisms are needed for these caess. In *hash* (Fig. 6-2), hashCode always calculates the same hash value no matter how many threads are calling it. In *isprime* (Fig. 6-3), the *pflag* array is used to record the already identified prime numbers. It is lazily initialized, but the method always returns the correct answer. The same as *lazy-fib* (Fig. 6-4), the *fib* array records the already calculated fibonacci numbers. The correct result is guaranteed to be returned. Also in *badbit* (Fig. 6-5), the *isbad* field is lazily initialized to record the already identified bad bits. The four test cases follow the same patthern; the lazily initialized variables are used to improve the performance by avoiding recalculations, but are not affecting the correctness of the program.

Besides lazy initialization on primitive typed shared variables, another type of benign data race arises from the generation of random number. The benign data race may let the random number less random. One example is ConcurrentSkipListMap class in java.util.concurrent package. The code snippet of ConcurrentSkipListMap is shown in Fig. 6-14. This class has a non-volatile field *randomSeed*. Method randomLevel performs a read from the old *randomSeed* and assigns *randomSeed* a new value. If randomLevel method is called by more than one thread, there would be a data race between line 7 and line 10. However, this data race doesn't affect the correctness of ConcurrentSkipListMap operations such as get, put, and remove, but only affects the time cost of these operations. The reason is that ConcurrentSkipList implements a tree-like structure to store the list. The expected time cost of its operations is $O(logN)$. Given a new element, randomLevel calculates a proper random level for it. If the threads don't see updates for *randomSeed*, then more than two elements will be mapped into the same level. In the worst case, all the elements would be mapped to one level and this structure would be degraded to a list, but with operations still performing correctly. But in reality, the probability with all the threads seeing the default value of *randomSeed* is very small.

Table 6-2. Latency comparison on *lazy-fib* between no explicit synchronization, AtomicLong array, and fully synchronized method.

	No synchronization	Atomic array	Explicit sync
1	1.178ms	1.220ms	1.462ms
2	1.185ms	1.211ms	1.464ms
3	1.182ms	1.224ms	1.462ms
Average	1.182ms	1.218ms	1.463ms

Table 6-2 shows the latency comparison on *lazy-fib* between no explicit synchronization, atomic integer/long array[3] , and fully synchronized method. Note that volatile keyword cannot guarantee DRF for individual array elements, so it is not included in the table. Each synchronization scheme creates 10 threads with each thread calculating fibonacci number 500. We run each scheme three times to compare the average latency. The experiments are carried out on SPARC Enterprise T5220 server with 60GB memory, SunOS 5.10 OS, and JDK 1.6. It's easy to see that the program without any synchronization mechanisms runs the fastest, and the fully synchronized method mechanism has the largest latency. This phenomenon is even more obvious for larger programs. From this table, we can see that the identification of benign data race is very important to improve the performance of the programs.

6.3 Model Checking Under PSO

In this section we show that the idea of keeping history of writes can also be applied to model check programs under other relaxed memory models, such as PSO. *Partial Store Order* (PSO) is decribed in §2.1.2. It is a relaxed memory model used by SPARC systems. The architecture of PSO is similar to *Total Store Order* (TSO) (Fig. 2-4) except that each process maintains a set of store buffers with each store buffer associated to a memory location. PSO is relaxed; Because of the delayed write back from store buffer to the main memory, a read may see an old, but not up-to-date value. The major difference between JMM and PSO is that, in PSO, a read cannot see a write that hasn't been executed, but JMM allows this. In this sense, PSO is "simpler" to model than JMM.

With JPR, we extended the $WriteSet$ idea to model checking PSO. JPR's fixed point style algorithm is used to collect values that would be written in the future. Because

[3] In Java, the operations of atomic variables (AtomicInteger, AtomicLong, AtomicReference, etc.) are all atomic. Atomic variables also guarantees $\leq_{hb}$ like volatile variables.

PSO doesn't allow reading from a future write, we don't need the iterative running of JPF. Instead, the PSO model checker could be a standard JPF extension project which only runs JPF once. Also, PSO doesn't have happens-before relations and imposing future writes, so $HBSet$, $ImposeSet$, and $ThreadLast$ can be removed from the metadata.

The listener styled PSO model checking algorithm is listed in Fig. 6-15. Here all the metadata specified in §3.2 are removed except $WriteSet$ and $Read$:

- ***WriteSet***: $Loc \rightarrow 2^{Aid \times Proc \times Val \times Flag}$
- ***Read***: $Aid \rightarrow Aid \times Val$

Different from JPR, the $WriteSet$ for PSO is is a mapping from memory location loc to pairs of $(aid, proc, val, flag)$, where aid represents the actionID, $proc$ is the process id, val is the value it writes to, and $flag$ is a boolean variable which indicates whether the current value is in the main memory or not. Another difference is that the $WriteSet$ is not a simple set, but maintains a proper order of the write actions. The $WriteSet$ has a property: at any time, there is at most one pair in $WriteSet(loc)$ with $flag = true$. This means for a variable, there should be at most one value placed in the main memory. Because there is no future writes, the boolean signal that indicates future write is removed from $Read$.

The three basic operations in PSO are *store*, *load*, and *fence*. The *store* can be mapped to write action in Java, and *load* can be mapped to read action. There is no *fence* in Java, but we can randomly pick a seldomly used statement to represent it. Fig. 6-15 only lists the instruction executing event. It deals with other events the same as Fig. 3-7 except that the state $\Sigma = \langle WriteSet, Read \rangle$, and there is no $GlobalWriteSet$.

For *store* action, the algorithm simply appends $WriteSet(loc)$ with a new pair whose $flag$ field is false. This is corresponding to the placement in the store buffer associated with the process and the variable. For *load* action, we non-deterministically choose a pair from $WriteSet(loc)$. The non-determinism is handled by data choice generator. Here a read may see a value val written by either the same process or some other process. If

reading from the same process, only the most up-to-date value can be seen. This value might be still in the store buffer.

Reading from another process $proc_j$ implies that 1) the value is already placed in the main memory; 2) the store buffer of the read process is empty; and 3) all the values written by the writes that executed before the write of val in $proc_j$ are removed from the store buffer. Based on the above observations, if read loc from $WriteSet$ pair $(taid, proc_j, val, flag)$, the $flag$ field is set to true, meaning that the value is currently on the memory. For $proc_j$, all its pairs of loc before the write of the value from $WriteSet(loc)$ are removed. For $proc_i$, the process of the read, all its pairs on loc are removed from $WriteSet(loc)$. This means the store buffer of read process on loc is empty. For all other processes, the pairs on loc with $flag$ = true are removed. This is to ensure that at any time, there should only be one pair on loc with $flag$ = true.

The fence operation ensures that the most recent values of the calling process are written to the main memory. For each variable, the algorithm loops over its pairs in $WriteSet$. All the pairs from the calling process are removed except the latest write. The $piHasWrite$ signal is set to true if the calling process has a write to that variable. All the pairs from other processes with $flag$ = true are removed if $piHasWrite$ is true.

In **PSOListener**, we use $WriteSet$ to collect the read candidates just like **JMMListener**. However, the $WriteSet$ is not non-decreasing. Some pairs may be removed from it at LOAD and FENCE. Except for data collection, the $WriteSet$ in **PSOListener** also simulates the store buffer. This is a major difference from JPR.

We implemented the algorithm shown in Fig. 6-15 as a standard JPF extension project and tested several examples on it. The peterson's and dekker's algorithms failed as expected. And it doesn't generate the result shown in Fig. 2-1. This tool took shorter time than JPR generally because it doesn't have iterations and has fewer data choices. Similar to PSO, the TSO can also be model checked using this idea, but it requires more restrictions on LOAD.

```
1  PSOListener(searchEvent) {
     switch(searchEvent){
3          ......     // other events
         case EXECUTING ACTION:
5            Let action = (aid, proc_i, kind, loc)
             switch(kind){
7                case STORE (proc_i, loc, val):
                     WriteSet(loc) ← WriteSet(loc) ∪ (aid, proc_i, val, false);
9                    break;
                 case LOAD (proc_i, loc):
11                   non−deterministically choose pair T: (taid, proc_j, val, flag) from WriteSet(loc)
                     // Read from same process, only the most recent value
13                   if proc_i = proc_j then
                         if T is the latest write action in proc_i then
15                           Read(aid) ← (taid, val)
                         // else ignore
17                   // Read from different process
                     else if proc_i ≠ proc_j then
19                       Read(aid) ← (taid, val)
                         (taid, proc_j, val, flag) → (taid, proc_j, val, true);
21                       // Delete all the pairs in front of T in proc_j
                         for each T′: (aid′, proc_j, val′, flag′) in front of T in WriteSet(loc) do
23                           WriteSet(loc) ← WriteSet(loc) \ T′
                         // Delete all the pairs of proc_i
25                       for each T″: (aid″, proc_i, val″, flag″) in WriteSet(loc) do
                             WriteSet(loc) ← WriteSet(loc) \ T″
27                       // Delete the pairs in main memory
                         for each T‴ :(aid‴, proc_k, v‴, true) (k ≠ i ∧ k ≠ j) in WriteSet(loc) do
29                           WriteSet(loc) ← WriteSet(loc) \ T‴
                     break;
                 case FENCE (proc_i):
                     for each location loc in WriteSet do
33                       bool piHasWrite = false;
                         for each pair T: (taid, proc_j, val, flag) from WriteSet(loc) do
35                           // Current process: delete all the pairs except the latest write
                             if proc_i = proc_j then
37                               if T is the latest write action on loc then
                                     (taid, proc_j, val, flag) → (taid, proc_j, v, true)
39                                   piHasWrite = true;
                                 else
41                                   WriteSet(loc) ← WriteSet(loc) \ T
                             // Other processes: delete the pairs in main memory
43                           else if proc_i ≠ proc_j then
                                 if flag = true ∧ piHasWrite = true then
45                                   WriteSet(loc) ← WriteSet(loc) \ T
                     break;
47           }
             break;
49   }
   }
```

Figure 6-15. Listener-styled PSO algorithm.

CHAPTER 7
RELATED WORK

This chapter presents some related works contributed by other researchers, and compares them with our approach.

Ferrara [29] used a fixed point formulation to interpret the happens-before memory model. This work was done in the context of abstract interpretation, but was not implemented into a real tool. Botincan, et al. [13] showed that the causality requirements of the JMM are undecidable.

Work has been done using various techniques to verify programs under relaxed hardware and programming language memory models. JUMBLE [31] is a dynamic analysis system that implements an adversarial memory by keeping track of a history of writes to racy variables. When a racy variable is read, the adversarial memory returns some past value that JMM allows and is likely to crash the program. Unlike JPR, this tool does not consider nonracy variables and cannot simulate reading from a future write, hence can only provide an under-approximation of JMM. RELAXER [16], a two-phase analysis tool, employs dynamic analysis in its first phase to detect races on SC executions and predicts potential happen-before cycles if run under one of TSO, PSO, or PSLO. In the second phase, it runs the tested program under the relaxed memory model with a controlled scheduler that realizes the one with happen-before cycle to check for program violations. JPR can be extended with a similar heuristic to prefer exploring paths that may end up with a happen-before cycles. We also mention that we have extended JPF to implement the TSO and PSO memory models. While not of significant practical interest, these could be implemented without requiring iteration, thus giving an illustration of the significant complexity of the JMM.

Burckhardt, Alur and Martin [14] applied a SAT-based bounded verification method to check concurrent data types under relaxed memory ordering models employed by multiprocessors while Burckhardt and Musuvathi [15] described a monitor algorithm that

could be implemented by model checkers to verify relaxed memory models due to store buffers. The MemSAT system [89] system accepts a test program containing assertions and an axiomatic specification of a memory model and then uses a SAT solver to find a trace that satisfies the assertions and axioms, if there is one. Both the original JMM specification [36], and the modified version proposed by [7] were found to have surprising results when applied to the JMM Causality test cases. MemSAT is intended to be used with small “litmus test” programs to debug memory model specifications. In contrast, JPR is intended to reason about programs. it explores all possible paths according to the JMM and reports any assertion (program constrain violation) violations, which can help to decide whether the races are benign or not. JPR can be used with programs containing object instantiation, loops and other features that are not well supported in MemSAT. The authors of Java memory model developed a simple simulator for the JMM [58] which appears to be geared more towards understanding the memory model than serving as a tool for program analysis. De et al. [26] developed OpMM which uses a model checker similar to Java PathFinder for state exploration. In contrast to JPR, OpMM is an underapproximation of the JMM where read actions can see past writes that occur before it in a sequentially consistent execution. As an underapproximation, OpMM could be used for bug detection of racy programs, but not verification.

CHAPTER 8
CONCLUSION

In this thesis, we have described a simple memory model, SC memory model, in which a read only sees the value of the most recent write. SC restricts most hardware or compiler optimizations and transformations. JMM on the other hand, is a relaxed memory model. It allows a read to see more writes so that many optimizations and transformations are allowed. However, most modern model checkers are based on SC memory model, so that under JMM, they can only be used to reason about data-race-free programs (guarantees sequential consistency), but not programs that contain data races. Based on JMM's declarative rules, we presented a new fixed-point semantics that overapproximates JMM. This approach runs the model checking algorithm in an iterative way to compute a least fixed point of a monotone function that can also generate sequentially inconsistent executions.

We also implemented the semantics into a tool, JPR, which is built on top of JPF. With this extension, JPF can also be applied to the verification of Java programs with data races. We ran JPR on three groups of test cases; JMM causality test cases, programs that contain benign data races, and programs that contain harmful data races. We found that JPR can generate all the allowed behaviors but can also generate some forbidden behaviors. Because of this overapproximation, JPR can be soundly used to identify benign data races. From the performance perspective, JPR generally runs longer time and generates more states than original JPF because of the data non-determinisms and iterations. Although, like any tool based on model checking, state-space explosion is a potential problem, we were able to successfully use the tool to show that data races in some examples are benign. We also demonstrated assertion violations in some programs, which are not detectable without awareness of the JMM.

When implementing JPR, we found that an operational semantics of JMM requires more precise definition of the action ID concept. We have proposed, implemented,

and empirically compared three approaches (**scope**, **occurrence**, and **occurrence-val**). Although, drawing a conclusion on which of these approaches would be the most appropriate one is outside the scope of this thesis, we hope to start a fruitful discussion on the topic.

Although our approach is presented in the context of JMM, the idea of the approach is not only restricted to JMM, but can be generalized to other simpler relaxed memory models such as PSO and TSO. We presented the algorithm for PSO in Chapter 6. The difference between PSO and JMM is that PSO only allows read to see past writes, so that iteration is not needed in the algorithm.

There are definitely many future works to be done. One direction is the identification of paths that violate the assertions. This information is very helpful to programmers to understand the potential program bugs. Another direction is identification of more benign data race patterns, and the automatic categorization of patterns by JPR. Also, to help alleviating the state exploration problem, we may apply heuristics to reach paths with assertion violations faster. Moreover, it would be interesting to study other relaxed memory models and apply the idea to them. This may lead to better understanding of those memory models.

APPENDIX A
JMM CAUSALITY TEST CASES

In this appendix, we present the JMM causality test cases listed in [41]. These test cases are used to reason about the performance of JPR in Chapter 6.

tc1: $r1 == r2 == 1$ is an allowed behavior.

Initially, x == y == 0

Thread 1	Thread 2

r1 = x;
if(r1 ≥ 0)
y = 1;
	r2 = y;
	x = r2;

tc2: $r1 == r2 == r3 == 1$ is an allowed behavior.

Initially, x == y == 0

Thread 1	Thread 2

r1 = x;
r2 = x;
if(r1 == r2)
y = 1;
	r3 = y;
	x = r3;

tc3: $r1 == r2 == r3 == 1$ is an allowed behavior.

Initially, x == y == 0

Thread 1	Thread 2	Thread 3

r1 = x;
r2 = x;
if(r1 == r2)
y = 1;
	r3 = y;
	x = r3;
		x = 2;

tc4: $r1 == r2 == 1$ is a prohibited behavior.

Initially, x == y == 0

Thread 1	Thread 2
r1 = x; y = r1;	r2 = y; x = r2;

tc5: $r1 == r2 == 1 \wedge r3 == 0$ is a prohibited behavior.

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

Thread 1	Thread 2	Thread 3	Thread 4
r1 = x; y = r1;	r2 = y; x = r2;	z = 1;	r3 = z; x = r3;

tc6: $r1 == r2 == 1$ is an allowed behavior.

Initially, x == y == 0

Thread 1	Thread 2

r1 = x;
if(r1 == 1)
  y = 1;
	r2 = x;
	if(r2 == 1)
	  x = 1;
	if(r2 == 0)
	  x = 1;

tc7: $r1 == r2 == r3 == 1$ is an allowed behavior.

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

Thread 1	Thread 2

r1 = z;
r2 = x;
y = r2;
	r3 = y;
	z = r3;
	x = 1;

tc8: $r1 == r2 == 1$ is an allowed behavior.

Initially, x == y == 0

Thread 1	Thread 2

r1 = x;
r2 = 1 + r1 * r1 - r1;
y = r2;
	r3 = y;
	x = r3;

tc9: $r1 == r2 == 1$ is an allowed behavior.

Initially, x == y == 0

Thread 1	Thread 2	Thread 3

r1 = x;
r2 = 1 + r1 * r1 - r1;
y = r2;
	r3 = y;
	x = r3;
		x = 2;

tc9a: $r1 == r2 == 1$ is an allowed behavior.

Initially, x == 2, y == 0

Thread 1	Thread 2	Thread 3

r1 = x;
r2 = 1 + r1 * r1 - r1;
y = r2;
	r3 = y;
	x = r3;
		x = 0;

tc10: $r1 == r2 == 1 \wedge r3 == 0$ is a prohibited behavior.

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

Thread 1	Thread 2	Thread 3	Thread 4

r1 = x;
if(r1 == 1)
y = 1;
	r2 = y;
	if(r2 == 1)
	x = 1;
		z = 1;	r3 = z;
			if(r3 == 1)
			x = 1;

tc11: $r1 == r2 == r3 == r4 == 1$ is an allowed behavior.

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

Thread 1	Thread 2

r1 = z;
w = r1;
r2 = x;
y = r2;
	r4 = w;
	r3 = y;
	z = r3;
	x = 1;

tc12: $r1 == r2 == r3 == 1$ is a prohibited behavior.

Initially, x == y == 0, a[0] == 1, a[1] == 2

Thread 1	Thread 2

r1 = x;
a[r1] = 0;
r2 = a[0];
y = r2;
	r3 = y;
	x = r3;

tc13: $r1 == r2 == 1$ is a prohibited behavior.

Initially, x == y == 0

Thread 1	Thread 2

r1 = x;
if(r1 == 1)
  y = 1;
	r2 = y;
	if(r2 == 1)
	  x = 1;

tc14: $r1 == r3 == 1 \wedge r2 == 0$ is a prohibited behavior.

Initially, a == b == y == 0, y is volatile

Thread 1	Thread 2

r1 = a;
if(r1 == 0)
  y = 1;
else
  b = 1;
	do{
	  r2 = y;
	  r3 = b;
	}while(r2 + r3 == 0);
	a = 1;

tc15: $r0 == r1 == r3 == 1 \wedge r2 == 0$ is a prohibited behavior.

Initially, a == b == x == y == 0; x, y are volatile

Thread 1	Thread 2	Thread 3

r0 = x;
if(r0 == 1)
  r1 = a;
else
  r1 = 0;
if(r1 == 0)
  y = 1;
else
  b = 1;
	do{
	  r2 = y;
	  r3 = b;
	}while(r2 + r3 == 0);
	a = 1;
		x = 1;

tc16: $r1 == 2 \wedge r2 == 1$ is an allowed behavior.

Initially, x == 0

Thread 1	Thread 2
r1 = x; x = 1;	r2 = x; x = 2;

tc17: $r1 == r2 == r3 == 42$ is an allowed behavior.

Initially, x == y == 0

Thread 1	Thread 2

r3 = x;
if(r3 != 42)
  x = 42;
r1 = x;
y = r1;
	r2 = y;
	x = r2;

tc18: $r1 == r2 == r3 == 42$ is an allowed behavior.

Initially, x == y == 0

Thread 1	Thread 2

r3 = x;
if(r3 == 0)
  x = 42;
r1 = x;
y = r1;
	r2 = y;
	x = r2;

tc19: $r1 == r2 == r3 == 42$ is an allowed behavior.

Initially, x == y == 0

Thread 1	Thread 2	Thread 3

join Thread 3
r1 = x;
y = r1;
	r2 = y;
	x = r2;
		r3 = x;
		if(r3 != 42)
		  x = 42;

tc20: $r1 == r2 == r3 == 42$ is an allowed behavior.

Initially, x == y == 0

Thread 1	Thread 2	Thread 3

join Thread 3
r1 = x;
y = r1;
	r2 = y;
	x = r2;
		r3 = x;
		if(r3 == 0)
		  x = 42;

APPENDIX B
MODEL CHECKING UNDER TSO

```
TSOListener(searchEvent)}{
    switch(searchEvent){
        case EXECUTING ACTION:                //other events not listed
            Let action = (aid, proc_i, kind, loc)
            switch(kind){
                case STORE (proc_i, loc, val):
                    WriteSet ← WriteSet∪ (aid, proc_i, val, false);
                    break;
                case LOAD (proc_i, loc):
                    non−deterministically choose pair T = (taid, proc_j, val, flag) from WriteSet(loc)
                    if proc_j = proc_i then //Read from same process, only the most recent value
                        if T is the latest write action in P_i then
                            Read(aid) ← val
                    else if proc_j ≠ proc_i then //Read from different process
                        Read(aid) ← val
                        (taid, proc_j, val, flag) → (taid, proc_j, val true);
                        //i) proc_i, other variables:
                        //Delete all pairs before latest write on loc except most recent on each variable
                        Let T_{i_loc} = (tilaid, proc_i, u, flag) be the most recent write in WriteSet(loc) by proc_i
                        for each variable loc′ (loc′ ≠ loc) in WriteSet do
                            Let T_{i_loc′} = (tilpaid, proc_i, w, flag)
                            be the most recent write on loc′ by proc_i just before T_{i_loc} in WriteSet
                            (tilpaid, proc_i, w, flag) → (tilpaid, proc_i, w, unknown)
                            for each pair T_loc with proc = proc_i before T_{i_loc′} in WriteSet(loc′) do
                                WriteSet(loc′) ← WriteSet(loc′) \ T_loc
                        //ii) proc_i, loc: Delete all the pairs on loc
                        for each pair T_loc with proc = proc_i in WriteSet(loc) do
                            WriteSet(loc) ← WriteSet(loc) \ T_loc
                        //iii) proc_j, other var: Delete all pairs before T except latest one on each var
                        for each variable loc′ (loc′ ≠ loc) in WriteSet do
                            Let T_{j_loc′} = (tjlpaid, proc_j, w, flag)
                            be the most recent write on loc′ by proc_j just before T in WriteSet
                            (tjlpaid, proc_j, w, flag) ← (tjlpaid, proc_j, w, unknown)
                            for each pair T_loc′ with proc = proc_j before T_{j_loc′} in WriteSet(loc′) do
                                WriteSet(loc′) ← WriteSet(loc′) \ T_loc′
                        //iv) proc_j, loc: Delete all the pairs on loc before T
                        for each pair T_loc with proc = proc_j in front of T in WriteSet(loc) do
                            WriteSet(loc) ← WriteSet(loc) \ T_loc
                        //v) proc_k, loc: Delete the pairs in main memory or unknown
                        for each pair T_loc ∈ WriteSet(loc) w/ proc = proc_k ∧ flag ≠ false (k ≠ i ∧ k ≠ j) do
                            WriteSet(loc) ← WriteSet(loc) \ T_loc
                    break;
                case FENCE (proc_i):
                    for each variable loc in WriteSet do
                        bool piHasWrite = false;
                        for each pair T = (taid, proc_j, v, flag) from WriteSet(loc) do
                            if proc_i = proc_j then //proc_i: delete all pairs except latest write
                                if T is the latest write action on loc then
                                    (taid, proc_j, v, flag) → (taid, proc_j, v, true)
                                    piHasWrite = true;
                                else WriteSet(loc) ← WriteSet(loc) \ T
                            else if proc_i ≠ proc_j then //Other processes: delete pairs in main memory
                                if (flag ≠ false) ∧ piHasWrite = true then
                                    WriteSet(loc) ← WriteSet(loc) \ T
                    break;
            }
    }
}
```

Figure B-1. TSO algorithm using JPF

TSO (Total Store Order) memory model is described in §2.1.2. The underlying architecture is shown in Fig. 2-4. In TSO, processes communicate with each other by accessing the main shared memory. Each process is associated with an FIFO queue, called store buffer. Different to PSO, only one store buffer is associated with each process in TSO. Writes to any variables are written to the store buffer before flushing to the main memory. TSO is relaxed. It allows a read to see a not up-to-date value. But TSO has more restrictions on reordering of statements than PSO because of the single store buffer.

Similar to Fig. 6-15, we proposed a JPF listener-styled algorithm for TSO (Fig. B-1). The algorithm mainly presents the operations for the three TSO operations STORE, LOAD, and FENCE. The metadata used in the algorithm is the same as those presented in §6.3 except that the *flag* field of *WriteSet* pair can be unknown in addition to true and false. Fig. B-1 is slightly complicated in LOAD case than Fig. 6-15 because TSO has more restrictions than PSO in terms of the single store buffer. This algorithm can be easily implemented in JPF.

REFERENCES

[1] Adve, Sarita V. and Hill, Mark D. "Weak Ordering - A New Definition and Some Implications." Tech. Rep. TR902, University of Wisconsin - Madison, 1989.

[2] ———. "Weak ordering - a new definition." *Proceedings of the 17th annual international symposium on Computer Architecture*. ISCA '90. New York, NY, USA: ACM, 1990, 2–14.

[3] Adve, S.V. and Hill, M.D. "A unified formalization of four shared-memory models." *Parallel and Distributed Systems, IEEE Transactions on* 4 (1993).6: 613 –624.

[4] Akers, S.B. "Binary Decision Diagrams." *Computers, IEEE Transactions on* C-27 (1978).6: 509 –516.

[5] Alur, Rajeev and Martin, Milo M. K. "Specifying Relaxed Memory Models for State Exploration Tools Sela." $(EC)^2$*: Workshop on Exploting Concurrency Efficiently and Correctly*. 2009.

[6] Andrews, Gregory R. *Concurrent programming: principles and practice*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1991.

[7] Aspinall, David and Ševčík, Jaroslav. "Formalising java's data race free guarantee." *Proceedings of the 20th international conference on Theorem proving in higher order logics*. TPHOLs'07. Berlin, Heidelberg: Springer-Verlag, 2007, 22–37.

[8] ———. "Java Memory Model Examples: Good, Bad and Ugly." *Technical Report EDI-INF-RR-1121*. School of Informatics, University of Edinburgh, 2007.

[9] Bacon, David, Bloch, Joshua, Bogda, Jeff, Click, Cliff, Haahr, Paul, Lea, Doug, May, Tom, Maessen, Jan-Willem, Manson, Jeremy, Mitchell, John D., Nilsen, Kelvin, Pugh, Bill, and Sirer, Emin Gun. "The "Double-Checked Locking is Broken" Declaration." 2008.

URL http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html

[10] Ball, Thomas, Majumdar, Rupak, Millstein, Todd, and Rajamani, Sriram K. "Automatic predicate abstraction of C programs." *PLDI 01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. New York, NY, USA: ACM, 2001, 203–213.

[11] Batty, Mark, Owens, Scott, Sarkar, Susmit, Sewell, Peter, and Weber, Tjark. "Mathematizing C++ concurrency." *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '11. New York, NY, USA: ACM, 2011, 55–66.

[12] Boehm, Hans-J. and Adve, Sarita V. "Foundations of the C++ concurrency memory model." *Proceedings of the 2008 ACM SIGPLAN conference on Programming*

language design and implementation. PLDI '08. New York, NY, USA: ACM, 2008, 68–78.

[13] Botinčan, Matko, Glavan, Paola, and Runje, Davor. "Verification of causality requirements in Java Memory Model is undecidable." *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics: Part II*. PPAM'09. Berlin, Heidelberg: Springer-Verlag, 2010, 62–67.

[14] Burckhardt, Sebastian, Alur, Rajeev, and Martin, Milo M. K. "Bounded Model Checking of Concurrent Data Types on Relaxed Memory Models: A Case Study." *Proceedings of the 18th International Conference on Computer Aided Verification*. 2006.

[15] Burckhardt, Sebastian and Musuvathi, Madanlal. "Effective Program Verification for Relaxed Memory Models." *Proceedings of the 20th International Conference on Computer Aided Verification*. 2008.

[16] Burnim, Jacob, Sen, Koushik, and Stergiou, Christos. "Testing concurrent programs on relaxed memory models." *ISSTA*. 2011, 122–132.

[17] Cenciarelli, Pietro, Knapp, Alexander, and Sibilio, Eleonora. "The java memory model: operationally, denotationally, axiomatically." *Proceedings of the 16th European conference on Programming*. ESOP'07. Berlin, Heidelberg: Springer-Verlag, 2007, 331–346.

[18] Choi, Jong-Deok, Lee, Keunwoo, Loginov, Alexey, O'Callahan, Robert, Sarkar, Vivek, and Sridharan, Manu. "Efficient and precise datarace detection for multithreaded object-oriented programs." *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. PLDI '02. New York, NY, USA: ACM, 2002, 258–269.

[19] Christiaens, Mark and De Bosschere, Koen. "TRaDe, a topological approach to on-the-fly race detection in java programs." *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*. JVM'01. Berkeley, CA, USA: USENIX Association, 2001, 15–15.

[20] Clarke, E. M., Emerson, E. A., and Sistla, A. P. "Automatic verification of finite-state concurrent systems using temporal logic specifications." *ACM Trans. Program. Lang. Syst.* 8 (1986): 244–263.

[21] Clarke, Edmund and Emerson, E. "Design and synthesis of synchronization skeletons using branching time temporal logic." *Logics of Programs*. ed. Dexter Kozen, vol. 131 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1982. 52–71. 10.1007/BFb0025774.

[22] Clarke, Edmund M. "The Birth of Model Checking." *25 Years of Model Checking*. 2008, 1–26.

[23] Clarke, Edmund M., Grumberg, Orna, and Long, David E. “Model checking and abstraction.” *ACM Trans. Program. Lang. Syst.* 16 (1994): 1512–1542.

[24] Cohen, Ernie, Moskal, Michal, Tobies, Stephan, and Schulte, Wolfram. “A Precise Yet Efficient Memory Model For C.” *Electronic Notes in Theoretical Computer Science* 254 (2009): 85 – 103. Proceedings of the 4th International Workshop on Systems Software Verification (SSV 2009).

[25] Cousot, Patrick and Cousot, Radhia. “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.” *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. POPL ’77. New York, NY, USA: ACM, 1977, 238–252.

[26] De, Arnab, Roychoudhury, Abhik, and D’Souza, Deepak. “Java Memory Model aware Software Validation.” *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 2008.

[27] Dijkstra, Edsger W. *Cooperating sequential processes*. New York, NY, USA: Springer-Verlag New York, Inc., 2002, 65–138.

[28] Dubois, Michel, Scheurich, Christoph, and Briggs, Faye. “Memory access buffering in multiprocessors.” *25 years of the international symposia on Computer architecture (selected papers)*. ISCA ’98. New York, NY, USA: ACM, 1998, 320–328.

[29] Ferrara, Pietro. “Static analysis via abstract interpretation of the happens-before memory model.” *Proceedings of the 2nd international conference on Tests and proofs*. TAP’08. Berlin, Heidelberg: Springer-Verlag, 2008, 116–133.

[30] Flanagan, Cormac and Freund, Stephen N. “Type-based race detection for Java.” *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. PLDI ’00. New York, NY, USA: ACM, 2000, 219–232.

[31] ———. “Adversarial memory for detecting destructive races.” *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*. PLDI ’10. New York, NY, USA: ACM, 2010, 244–254.

[32] Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John. *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[33] Gao, G.R. and Sarkar, V. “Location consistency-a new memory model and cache consistency protocol.” *Computers, IEEE Transactions on* 49 (2000).8: 798 –813.

[34] Gharachorloo, Kourosh. “Memory Consistency Models for Shared-Memory Multiprocessors.” Tech. Rep. CSL-TR-95-685, Stanford University, 1995.

[35] Goetz, Brian, Peierls, Tim, Bloch, Joshua, Bowbeer, Joseph, Holmes, David, and Lea, Doug. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[36] Gosling, James, Joy, Bill, Steele, Guy, and Bracha, Gilad. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[37] Hatcliff, John and Dwyer, Matthew B. "Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software." *Proceedings of the 12th International Conference on Concurrency Theory*. CONCUR '01. London, UK: Springer-Verlag, 2001, 39–58.

[38] Henzinger, Thomas A., Jhala, Ranjit, Majumdar, Rupak, and Sutre, Grégoire. "Software verification with BLAST." *Proceedings of the 10th international conference on Model checking software*. SPIN'03. Berlin, Heidelberg: Springer-Verlag, 2003, 235–239.

[39] Hoare, C. A. R. "An axiomatic basis for computer programming." *Commun. ACM* 26 (1983): 53–56.

[40] Ivancic, F., Shlyakhter, I., Gupta, A., Ganai, M.K., Kahlon, V., Wang, Chao, and Yang, Zijiang. "Model checking C programs using F-Soft." *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*. 2005, 297 – 308.

[41] Java Memory Model Causality Test Cases. 2012.

URL http://www.cs.umd.edu/~pugh/java/memoryModel/unifiedProposal/testcases.html

[42] Java Pathfinder. 2012.

URL http://babelfish.arc.nasa.gov/trac/jpf

[43] Jin, Huafeng, Yavuz-Kahveci, Tuba, and Sanders, Beverly A. "Java Path Relaxer: Extending JPF for JMM-Aware Model Checking." *The Java Pathfinder Workshop 2011*. 2011.

[44] Jr., Edmund M. Clarke, Grumberg, Orna, and Peled, Doron A. *Model Checking*. The MIT Press, 1999.

[45] JRF-download. "Java Racefinder." 2012.

URL http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-racefinder

[46] Kahlon, Vineet, Yang, Yu, Sankaranarayanan, Sriram, and Gupta, Aarti. "Fast and accurate static data-race detection for concurrent programs." *Proceedings of the 19th international conference on Computer aided verification*. CAV'07. Berlin, Heidelberg: Springer-Verlag, 2007, 226–239.

[47] Kebrt, Michal and Šerý, Ondřej. "UnitCheck: Unit Testing and Model Checking Combined." *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis*. ATVA '09. Berlin, Heidelberg: Springer-Verlag, 2009, 97–103.

[48] Kim, KyungHee, Yavuz-Kahveci, Tuba, and Sanders, Beverly A. "Precise Data Race Detection in a Relaxed Memory Model Using Heuristic-Based Model Checking." *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, 495–499.

[49] ———. "JRF-E: using model checking to give advice on eliminating memory model-related bugs." *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ASE '10. New York, NY, USA: ACM, 2010, 215–224.

[50] Kuperstein, Michael, Vechev, Martin, and Yahav, Eran. "Partial-coherence abstractions for relaxed memory models." *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. PLDI '11. New York, NY, USA: ACM, 2011, 187–198.

[51] Lamport, L. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs." *IEEE Trans. Comput.* 28 (1979): 690–691.

[52] Lamport, Leslie. "A new solution of Dijkstra's concurrent programming problem." *Commun. ACM* 17 (1974): 453–455.

[53] Leungwattanakit, Watcharin, Artho, Cyrille, Hagiya, Masami, Tanabe, Yoshinori, and Yamamoto, Mitsuharu. "Model checking distributed systems by combining caching and process checkpointing." *ASE*. 2011, 103–112.

[54] Lev-Ami, Tal and Sagiv, Shmuel. "TVLA: A System for Implementing Static Analyses." *SAS 00: Proceedings of the 7th International Symposium on Static Analysis*. London, UK: Springer-Verlag, 2000, 280–301.

[55] Lindholm, Tim and Yellin, Frank. *Java Virtual Machine Specification*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999, 2nd ed.

[56] Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A., and Bensalem, S. "Property preserving abstractions for the verification of concurrent systems." *Form. Methods Syst. Des.* 6 (1995): 11–44.

[57] LTSA. "LTSA - Labelled Transition System Analyser." 2012.

URL http://www.doc.ic.ac.uk/ltsa/

[58] Manson, Jeremy and Pugh, William. "The Java Memory Model Simulator." *Workshop on Formal Techniques for Java-like Programs, in association with ECOOP*. 2002.

[59] Manson, Jeremy, Pugh, William, and Adve, Sarita. "SPECIAL POPL ISSUE The Java Memory Model." 2005.

[60] Manson, Jeremy, Pugh, William, and Adve, Sarita V. "The Java memory model." *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '05. New York, NY, USA: ACM, 2005, 378–391.

[61] McMillan, Kenneth L. *Symbolic Model Checking*. Norwell, MA, USA: Kluwer Academic Publishers, 1993.

[62] Mehlitz, Peter C., Tkachuk, Oksana, and Ujma, Mateusz. "JPF-AWT: Model checking GUI applications." *ASE*. 2011, 584–587.

[63] Merz, Stephan. "Model Checking: A Tutorial Overview." *Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes*. MOVEP '00. London, UK: Springer-Verlag, 2001, 3–38.

[64] Mitra, Tulika, Roychoudhury, Abhik, and Shen, Qinghua. "Impact of Java Memory Model on Out-of-Order Multiprocessors." *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. PACT '04. Washington, DC, USA: IEEE Computer Society, 2004, 99–110.

[65] MRMC. "Markov reward Model Checker." 2012.

URL http://www.mrmc-tool.org/trac/

[66] Naik, Mayur, Aiken, Alex, and Whaley, John. "Effective static race detection for Java." *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '06. New York, NY, USA: ACM, 2006, 308–319.

[67] Narayanasamy, Satish, Wang, Zhenghao, Tigani, Jordan, Edwards, Andrew, and Calder, Brad. "Automatically classifying benign and harmful data races using replay analysis." *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '07. New York, NY, USA: ACM, 2007, 22–31.

[68] Nguyen, Anh Cuong and Khoo, Siau-Cheng. "Towards Automation of LTL Verification for Java Pathfinder." *Proceedings of the 15th National Undergraduate Research Opportunities Programme Congress*. NUROP '10. 2010.

[69] Nguyen, Dinh-Phuc, Luu, Chung-Tuyen, Truong, Anh-Hoang, and Radics, N. "Verifying Implementation of UML Sequence Diagrams Using Java PathFinder." *Knowledge and Systems Engineering (KSE), 2010 Second International Conference on*. 2010, 194 –200.

[70] Nielson, Flemming, Nielson, Hanne R., and Hankin, Chris. *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999.

[71] NuSMV. “NuSMV: a new symbolic model checker.” 2012.

URL http://nusmv.fbk.eu/

[72] O’Callahan, Robert and Choi, Jong-Deok. “Hybrid dynamic data race detection.” *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPoPP ’03. New York, NY, USA: ACM, 2003, 167–178.

[73] On-the-fly, LTL Model Checking with SPIN. 2012.

URL http://spinroot.com/spin/whatispin.html

[74] oracle. “Oracle Thread Analyzer’s User Guide.” 2012.

URL http://download.oracle.com/docs/cd/E18659_01/html/821-2124/gecqt.html

[75] Owens, Scott, Sarkar, Susmit, and Sewell, Peter. “A Better x86 Memory Model: x86-TSO.” *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*. TPHOLs ’09. Berlin, Heidelberg: Springer-Verlag, 2009, 391–407.

[76] Owicki, Susan and Gries, David. “An axiomatic proof technique for parallel programs I.” *Acta Informatica* 6 (1976): 319–340. 10.1007/BF00268134.

[77] Peterson, Gary L. “Myths About the Mutual Exclusion Problem.” *Inf. Process. Lett.* 12 (1981).3: 115–116.

[78] Pnueli, Amir. “The temporal logic of programs.” *Foundations of Computer Science, 1977., 18th Annual Symposium on*. 1977, 46 –57.

[79] Pratikakis, Polyvios, Foster, Jeffrey S., and Hicks, Michael. “LOCKSMITH: context-sensitive correlation analysis for race detection.” *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. PLDI ’06. New York, NY, USA: ACM, 2006, 320–331.

[80] Pugh, William. “Fixing the Java memory model.” *Proceedings of the ACM 1999 conference on Java Grande*. JAVA ’99. New York, NY, USA: ACM, 1999, 89–98.

[81] ———. “The Java memory model is fatally flawed.” *Concurrency and Computation: Practice and Experience* 12 (2000): 445–455.

[82] Roychoudhury, Abhik. “Formal Reasoning about Hardware and Software Memory Models.” *Proceedings of the 4th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*. ICFEM ’02. London, UK, UK: Springer-Verlag, 2002, 423–434.

[83] Savage, Stefan, Burrows, Michael, Nelson, Greg, Sobalvarro, Patrick, and Anderson, Thomas. “Eraser: a dynamic data race detector for multithreaded programs.” *ACM Trans. Comput. Syst.* 15 (1997): 391–411.

[84] SPARC International, CORPORATE, Inc. *The SPARC architecture manual: version 8*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992.

[85] ———. *The SPARC architecture manual (version 9)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994.

[86] Stärk, Robert and Börger, Egon. "An ASM Specification of C# Threads and the .NET Memory Model." *Abstract State Machines 2004. Advances in Theory and Practice*. eds. Wolf Zimmermann and Bernhard Thalheim, vol. 3052 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2004. 38–60.

[87] Starke, P. H. "Reachability analysis of Petri nets using symmetries." *Syst. Anal. Model. Simul.* 8 (1991): 293–303.

[88] Tarski, Alfred. "A Lattice-Theoretical Fixpoint Theorem and its Applications." *Pacific Journal of Mathematics* 5 (1955).2: 285–309.

[89] Torlak, Emina, Vaziri, Mandana, and Dolby, Julian. "MemSAT: checking axiomatic specifications of memory models." *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '10. New York, NY, USA: ACM, 2010, 341–350.

[90] Valmari, Antti. "Stubborn sets for reduced state space generation." *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets 1990*. London, UK: Springer-Verlag, 1991, 491–515.

[91] Visser, Willem, Havelund, Klaus, Brat, Guillaume, Park, SeungJoon, and Lerda, Flavio. "Model Checking Programs." *Automated Software Engineering* 10 (2003): 203–232. 10.1023/A:1022920129859.

[92] Ševčík, Jaroslav and Aspinall, David. "On Validity of Program Transformations in the Java Memory Model." *Proceedings of the 22nd European conference on Object-Oriented Programming*. ECOOP '08. Berlin, Heidelberg: Springer-Verlag, 2008, 27–51.

[93] Yavuz-Kahveci, Tuba and Bultan, Tevfik. "Action Language verifier: an infinite-state model checker for reactive software specifications." *Form. Methods Syst. Des.* 35 (2009): 325–367.

[94] Zhang, Xin and van Breugel, F. "Model Checking Randomized Algorithms with Java PathFinder." *Quantitative Evaluation of Systems (QEST), 2010 Seventh International Conference on the*. 2010, 157 –158.

BIOGRAPHICAL SKETCH

Huafeng Jin received his bachelar's degree in computer engineering at Beijing University of Technology in China in July 2006. He started his graduate studies in Computer and Information Science and Engineering department of University of Florida in August 2006 under the supervision of Dr. Beverly A. Sanders. His research interest is static analysis on concurrent programs by software model checking.