JAVA RACEFINDER:
PRECISE DATA RACE DETECTOR IN A RELAXED MEMORY MODEL

By

KYUNGHEE KIM

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

2010

To my parents for their love and sacrifice

ACKNOWLEDGMENTS

I would like to begin with expressing my deepest gratitude to Professor Beverly A. Sanders, my advisor, for her endless advice, guidance, and support throughout my graduate studies. She has been a role model and a mentor for me. She has always been ready with an answer when I questioned, and encouraged me whenever I was lost in the long path to my final goal. She inspired me with her passion, knowledge, and patience. I would also like to thank Professor Tuba Yavuz-Kahveci for her great ideas, advice, and support. She has responded to me with the best advice whenever I asked a question and shared her experience whenever I was afraid to go forward. Three of us have been a great research team and I am very proud of being part of it. My grateful thanks also go to my committee members, Dr. Stephen M. Thebaut, Dr. Jih-Kwon Peir, and Dr. Douglas Cenzer, for their valuable insights and comments.

I would also like to thank Dr. Eric G. Mercer from Brigham Young University and Dr. Neha Rungta from NASA Ames Research Center for their guidance and advice. Their help made the research on the modular extension possible. I am indebted to NASA Java PathFinder team including Mr. Peter Mehlitz for their support and kind comments. Without Java PathFinder, this research would have been impossible. I would like to express my deepest respect and appreciation to Professor Chul E. Kim, my undergraduate advisor, for his encouragement and advices.

Last but not least, I want to thank my family for their love and support. Words are not enough to express my gratitude for what they have given to me. My parents taught me to be a good person. My children, Chris and Han, made me want to become a better person. My husband, YoungSang, was there for me and tries with me to do our best. I love you Chris, Han, and specially YoungSang.

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 RACEFINDER:
PRECISE DATA RACE DETECTOR IN A RELAXED MEMORY MODEL

By

KyungHee Kim

December 2010

Chair: Beverly A. Sanders
Major: Computer Engineering

Widespread use of multicore computers makes multithreaded programs ubiquitous, and concurrent programming as a main software paradigm. As difficult as sequential programming is, concurrent programming is even harder, and the correctness of it is more difficult to prove. Most approaches to reasoning about multithreaded programs, including model checking, make the implicit assumption that the system being considered is sequentially consistent. However, this is not a valid assumption for most current multiprocessor/core systems and this fact is exposed to the programmer for many concurrent programming languages in the form of a relaxed memory model. For example, the Java Memory Model only promises sequentially consistent behaviors for programs that are free from data races, making the ability to detect and eliminate data races essential for sound reasoning about Java programs.

Toward this end, we introduce a new summary function that captures the information necessary for precise data race detection along with an efficient representation of the function that allows data race detection by model checking. We also introduce novel search heuristics specialized for data race detection that lead to shorter counterexample paths than standard search strategies. The ideas have been implemented in Java RaceFinder (JRF), an extension to the model checker Java PathFinder (JPF). In contrast to many data race detection tools that can only deal with a restricted set of concurrent programming idioms, such as lock-based synchronization, JRF correctly

handles programs that contain memory model-relevant features, including volatile fields, final fields, compareAndSwap, and static initialization in addition to both intrinsic and extrinsic locks. As a result, JRF is powerful enough to be effectively used with wait-free and lock-free data structures. In addition to precise data race detection, the tool provides specific advice for eliminating data races from a program by analyzing the counterexample trace provided by the model checker and the acquiring history recorded by our tool. Case studies of widely used multithreaded programs and concurrent libraries have proved the usefulness of our tool and the effectiveness of our advice to correct found races. Finally, we provided an extended framework to guarantee the race-free use of a library using a library programmer's precondition annotations and the modular verification of their correctness using an assume-guarantee reasoning. This has been applied to various concurrent libraries and has verified the correct preconditions successfully. Once a concurrent program is proven to be free from a data race, standard model checking techniques can be soundly used to check other properties of interest. Our approach successfully addresses the relaxed memory model issue in model checking through precise data race detection.

CHAPTER 1
MOTIVATION

The current trend in multicore computers puts more emphasis on concurrent programming. In most programming environments, however, the correctness of a concurrent program's safety depends on the programmer's ability to correctly order every access to shared data. Unfortunately, this is difficult, and no previous research has correctly implemented verification of this concept.

Virtually all approaches to reasoning about the behavior of concurrent programs, both the informal reasoning practiced by programmers writing a concurrent program and the formal methods and tools, such as model checkers, start with the assumption that program executions are sequentially consistent (SC) [5]. In a SC execution, a concurrent program behaves as if all of its atomic actions occur in some global order that is consistent with the program order on each thread. In particular, all threads "see" values written to main memory in a consistent order. In contrast to this widely accepted assumption in the verification phase, modern programming environments do not guarantee sequential consistency. Common optimizations by the compiler and the hardware that significantly speeds up programs without affecting their sequential semantics are not necessarily benign in a concurrent environment.

As an example, consider the following program fragment where `result` and `done` are visible to multiple threads:

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

The variable `done` is initially false and not accessed by `computation()`, which updates `result` and possibly has other side effects. Since the two statements are independent, the order could be reversed without changing the sequential semantics. However, if this fragment occurs in a concurrent program and `done` is intended to be a flag to other threads that `computation()` is finished, then reversing the order could result in another

thread finding `done==true`, and seeing a state reflecting an incomplete execution of `computation()`. This scenario is not SC.

Architectures provide low level instructions that can be used to prevent reordering by the hardware and are typically inserted in the object code as a result of synchronization instructions in the program source. Compilers can refrain from certain optimizations that may cause sequentially inconsistent behavior. Although it would make concurrent programming much easier to use compiler analysis to determine what needs to be done, this currently is not practical and the programmer is expected to insert sufficient synchronization to ensure sequential consistency.

Exactly how threads interact with memory and how the programmer can control this is defined by a *memory model*. Traditionally, memory models have been defined for architectures, but more recently memory models have become part of a programming language's semantics and memory models have been defined for languages including Java [6, Chapter 17], .net based languages such as C# [7, Partition I, section 12.6], C++ [8], and OpenMP [9]. In Java, C#, and C++, data races are the situations that can lead to non-SC executions.

The term data race has often been used where the definition and consequences are subtly different from those of data races in the context of the JMM. Two memory accesses by different threads on the same location are said to *conflict* when at least one is a write, and a data race has been defined to be a situation where conflicting operations are not ordered by synchronization. In a sequentially consistent system, data races may indicate some sort of concurrency related non-determinism that may or may not affect the overall correctness of the program. For example, in an SC system in the example given earlier, the accesses to `done` would be considered a benign data race,[1] while it is a bug with potentially serious consequences in a program executing

[1] SC rules out reordering and the access itself is atomic.

under the JMM. According to the memory models of Java and C# (but not C++), the race could be eliminated by marking `done` and `result` with the `volatile` keyword. The JMM constrains the behavior of Java programs with data races, so one could, in principle, verify that a data race is benign. This is quite difficult and best left to experts. In the C++ memory model, the behavior of programs with data races is undefined, thus no data races can be viewed as benign in that language. In the context of a relaxed memory model, data races are almost always serious bugs[2] . Similarly, a program may be free of data races in the sense of the JMM, and thus guarantee sequential consistency, while still containing concurrency related errors. For example, suppose a class representing a bank account contains a field for the balance and offers a deposit method that executes `balance = balance + amount`. In the JMM, if `balance` is marked volatile (or if each of the accesses to this field occurs inside its own critical section implemented using a common lock) then the program will not have a data race, but it will still be incorrect due to the fact that the entire deposit method is not atomic. Some authors would call this error a ”race”, adding to the confusion around the term. In this dissertation, we will use the term ”data race” as defined in the JMM and discussed in more detail in Section 3.1 with the goal of detecting situations that lead to non-SC behavior.

The JMM satisfies an important fundamental property for memory models [11]:

Programs whose SC executions have no races must have only SC executions.

As a result, we can assume SC, and thus use model checking to demonstrate data race freedom. Then, for programs without data races, standard model checking techniques can *soundly* be used to find other types of concurrency related errors.

Nevertheless various tools to detect data races have been developed, most of them do not precisely find race conditions as defined by a well-defined memory model. Rather

[2] Lazy initialization[10], is the single well-known, reasonably practical programming idiom exhibiting benign data races in Java.

they identify easy-to-detect situations that may indicate a data race. For example, many approaches (including one of the data race detection extensions in the current JPF distribution) attempt to ensure that for all shared variables, all accesses to a particular variable are protected by a common lock. While this condition is sufficient for data race freedom, it is not necessary. In contrast to our approach, tools that check lock usage cannot effectively analyze programs using important idioms that are not based on locking including wait-free and lock-free algorithms. Provided that these algorithms becomes more important according to the wide-spread use of multicores, the ability to handle them correctly is a significant advantage in a race detector. We describe a tool that uses model checking to precisely detect data races in Java programs in order to help programmers ensure sequential consistency.

Model checking was originally designed to be used at the hardware level, but these days, software model checking becomes popular to check properties in concurrent programs. JPF is widely used model checker for Java bytecode, and has been applied to various applications including several mission critical applications. However, as discussed above, one of the problem of using JPF to check the correctness of a concurrent program is that JPF takes into account SC executions only. JPF is unable to find property violations in a non-SC execution. For example, consider the following simple two-threaded java program using the above code fragment.

```
public class Simple {
   static int x;
   static boolean done = false;
   public static void main(String[] args) {
      (new OtherThread()).start();
      x = 1;
      done = true;
   }
```

```
   static class OtherThread extends Thread {
      public void run() {
         while(!done) { /*spin*/ }
         assert (x == 1);
      }
   }
}
```

The expected model checking result would be to find an assertion violation at `OtherThread` when the property "$done \implies (x == 1)$" is violated in any possible execution path. However, JPF ends with no error for `Simple`, even though there is one according to JMM. The result for the different version of `Simple` with `done` as a `volatile`, in which the property always holds, is the same in current JPF. This shows that JPF cannot be used to soundly prove concurrent program safety. We aimed to find a formal method to distinguish the two programs and identify a program with a sound result.

In addition, the most common problem in model checking is a state-space explosion that obstructs it from being a practical correctness proof method. Our tool includes the features to leverage this problem during model checking to search for a race, although it is impossible to thoroughly solve it. The purpose is to minimize the chance of state-space explosion before a data race detection and we applied heuristics in determining search orders to detect a race earlier if one exists. We also used several optimization methods including lazy representation of array elements and threadlocal exclusion to save temporal and spatial resources.

Furthermore, we also suggest code changes to correct a race to address the problem of analyzing found races. Model checking generates a counterexample along with a property violation, and the counterexample path analysis identifies the source of a race. According to our experience in several case studies, more explanation of a data race other than the race location is inevitable.

Modularization is another widely used approach to leverage the state-space explosion problem. The idea is to decompose the system under test into smaller modules and apply the verification algorithm into them. The individual verification results are combined at the end to conclude the overall result given that certain safety conditions are satisfied. We extended our framework to include a modular race checking capability. In addition to the decomposition of the race detection overhead, we addressed the problem of a foreign code execution in the race-free guarantee. Java libraries are distributed in the form of a bytecode in general and the data race freedom of those foreign codes are determined solely by the appropriate use in an application that makes use of them. Either because of a lack of available information about the internal implementation or because of a lack of attention in applying them, the overall data race freedom of the foreign library is vulnerable. To achieve the two goals of decomposing the race detection into modules and specifying the design decisions of a foreign code regarding its data race freedom, we suggested the use of race-free preconditions. The preconditions attached to the bytecode and annotated by the library developer are verified automatically. Our framework ensures that preconditions are correct and satisfied. The lack of precondition violation guarantees the race freedom of a foreign code and its usage.

The remainder of the dissertation is organized as follows: In Chapter 2, we begin with the related work on data race and model checking. In Chapter 3, we introduce the JMM and give a brief overview of its formal definition, including a new result showing that a weaker condition than complete data race freedom is sufficient to ensure sequential consistency. We also introduce model checking and JPF in this chapter. Chapter 4 describes our approach used in JRF. We introduce a summary function that can be used to capture the necessary information to precisely recognize data races during model checking and new search heuristics based on a careful analysis of the properties of data races. Aforementioned ideas are implemented in a

model checker by extending JPF. We introduce a space efficient representation of the summary function and several optimization techniques, including how to represent an array data structure and how to handle threadlocal memories. The extensions to JRF to make it more powerful are described in Chapter 5. We used counterexample path and acquiring history to analyze a race and suggest appropriate code changes to eliminate it. We introduce one more extension, modular race-free precondition verification, in this chapter. Case studies follow in Chapter 6, and Chapter 7 discusses the overall performance of JRF. We present out conclusions in Chapter 8.

CHAPTER 2
RELATED WORK

2.1 Data Race Detection

Various tools and approaches to avoid and detect race conditions and data races have been described in the literature, but those existing tools to detect or prevent race conditions are limited in fundamental ways. Many tools for data race detection or avoidance do not start from the memory model and its definition of a data race, but instead look for conditions that imply data race freedom and are easier to check. For example, many approaches attempt to ensure that all accesses to a particular shared variable are protected by a common lock.[1] While the semantics of the lock primitive in Java imply that this condition is sufficient for data-race freedom, it is not necessary and requiring it rules out increasingly important programming idioms including wait-free and lock-free algorithms. Previous approaches usually fell into three categories: using a type system with additional annotations, dynamic race detection based on the lockset algorithm or vector clock algorithm to compute the happens-before relation, and hybrid methods with combined static and dynamic information.

Type based approaches extend the type system of a concurrent programming language to mark shared variables as guarded by a particular lock, and to ensure that the lock is held when the variable is accessed [12, 13]. A disadvantage of the type based approach is the burden on the programmer to provide the necessary type annotations. Whether or not this is a significant disadvantage for new programs where the annotations can be written while the code is being developed is arguable, but this clearly limits the usefulness of the approach for existing systems. Some work attempts to improve the situation by inferring many of the necessary annotations [14]. Most

[1] This is what is done by the ”DataRace Checker” that is currently part of the JPF distribution.

approaches using additional type annotations are conservative and produce large numbers of false positives. Thus, the tools implementing this approach pay much attention to effectively eliminating them. [13, 15] are other dialects of java for preventing a data race through a type system. Instead of detecting possible races, those type systems force all shared data accesses to be guarded by a lock; synchronization is not accomplished by programmer's code, but by the type system itself. These approaches guarantee the race freedom for programs satisfying suggested typing rules, but cannot be used for legacy codes.

Race detection tools based on static analysis techniques typically sacrifice completeness, in the sense that they can only deal with a particular set of programming idioms, and thus disallow legal data-race free programs. Some tools deliberately sacrifice soundness as well, failing to identify certain data races. For example, Chord [16], which can handle lexically-scoped lock-based synchronization, fork/join synchronization, and wait/notify, starts by constructing a superset of possible conflicting operations, then filters this set using a sequence of analyses, and reports a possible data race for all remaining pairs. Although Chord is both unsound and incomplete, it appears to be extremely well-engineered and in practice has shown significant utility by identifying 387 concurrency related errors in a set of widely used open source programs. Another example is the rcc checker [12] as recently resurrected and extended for the Mobius project [17]. This tool uses a type theory base approach (which requires annotations by the users) to ensure that locking is done correctly. In its most recent incarnation, it also recognizes that volatile variables do not need to be protected by locks to avoid data races. However, in whatever form, the tool cannot deal with happens-before edges obtained via transitivity and generates false positives as a result.

Tools that perform dynamic race detection look for races in particular executions of the program. The disadvantage is that dynamic tools only detect problems in the test cases that are actually examined. These are typically based on maintaining vector

clocks or the lock-set algorithm with checks to see if every shared variable access is consistently locked. Eraser [18] is an influential example of a lock-set based detector. This system predates programming languages with well-defined memory models and was based on the idea that accesses to shared variables should consistently be locked. In practice that requirement is too strict, and Eraser and subsequent lock-based tools typically incorporate special cases, such as the situation where a single thread accesses and modifies a variable or object for some period of time after which it is no longer modified, but can be read by multiple threads. Since these tools do not check for safe publication, they are unsound in the JMM. Very recent work extending the Eraser algorithm includes [19], which uses aspect-oriented programming techniques to instrument the program at the source code level with new point-cuts lock, unlock, and maybe shared.

The dynamic tool most closely related to ours is Goldilocks [20]. The so-called lockset algorithm described there uses a relation that is very similar to the inverse of summary function h explained in Section 4.1. In other words, the goldilocks algorithm maintains a function for each variable that indicates which threads can access the variable. As with all tools performing dynamic analysis, the required instrumentation of the program may change its behavior and the tool is limited to only analyzing paths that happen to be tested.

The standard JPF distribution includes two race detecting tools, RaceDetector and PreciseRaceDetector. The former implements the lockset algorithm based on the assumption that every shared field is protected by a common lock. This lacks the ability to check happens-before ordering other than by locking. The latter checks for a data race based on a generic definition of a race, rather than starting with the JMM. At each choice generator, it checks if there are more than two thread choices trying to access the same memory location and if at least one of them is an update access. Read and write accesses to a volatile field are also detected as a race. A simple modification to

recognize that volatile variables are not involved in data races would eliminate some, but not all, false alarms. Any situation where transitivity of happens-before edges makes an access safe would be reported as a false alarm.

2.2 Model Checking

Recent work has incorporated memory-model awareness into program analysis tools. The CheckFence [21] system verifies that a program executed on a system with a hardware-level relaxed memory model is observationally equivalent to a sequential execution by translating C implementation code and the test program into a SAT formula that is then given to a SAT solver. Other constraint-based approaches are described in [22, 23]. Several studies [24–26] have incorporated memory model awareness to model checking. The approaches presented in [24, 26] can verify sequential consistency. [24] considers a hardware-level memory model and uses bounded model checking and CHESS, a stateless model checker.[27] Therefore, they use vector-clocks to capture the happen-before relation. [26] considers C#'s memory model and a bytecode-level state-based model checker tailored for C#. The technique presented in [25] guides the model checker in generating a subset (i.e., underapproximates the JMM) of program executions varying due to instruction reordering allowed in the JMM. However, this tool does not detect data races.

Debugging is tedious work, so providing automated debugging support for programmers can save much time and effort. One intuitive idea is to compare failing executions with successful ones and use the differences between the two to explain and localize the fault. This concept has been explored in various studies; some [28–32] use model checking to generate the executions, whereas others [33–36] use testing. The approach in [29–31, 34, 35, 37] is to compare a set of successful traces with a set of erroneous ones to localize the errors or to focus the debugging process on a relatively small part of the program. [29] considers both transition and invariant differences on successful traces as well as the counterexample paths. It provides feedback on how

successful traces can be transformed into counterexample paths. In contrast, our analysis is aimed at providing feedback on how to transform a counterexample path into a possibly successful trace. [30] generates multiple error traces having independent causes and for each error cause reports a single error trace. [37] uses dynamic analysis and machine learning to classify program properties as fault-revealing and non-fault-revealing and reports program invariants that are in the fault-revealing set.

[28, 38] focus on error traces only. [28] slices a counterexample path to find the statements that directly or indirectly affect the failure. [38] computes the transactional happens-before edges on the dynamically generated execution traces to detect blocks that cannot preserve their atomicity and hence cannot be serialized.

Modular model checking is a powerful reduction technique based on assume-guarantee reasoning using ”divide and conquer” to achieve model checking scalability. The Calvin checker [39, 40] is a modular approach using assume-guarantee model checking. It uses user specifications about environment assumptions to constrain thread interactions based on locking. In [40], they inferred environment assumptions automatically to relieve the burden of providing them in a loosely-coupled multithread system. [41, 42] took program specifications in a set of state transition diagrams (STDs) as input and analyzed each component separately, then modeled them as synchronous reactive systems (SRS) synchronized by events. The result for the entire system is deduced from the individual analysis result. Model checking isolated components needs a model of an environment interacting with them and [41] used the most general environment, called *universal environment*, and provided automatic assumption generation to take into account parallel components.[42] Unlike the Configuration in this approach, summary function h introduced in Section 4.1 depends on the interleaving of each module's internal microstep, so global state cannot be modeled uniquely. In our modular extension explained at Section 5.2, we restricted the global state and assumptions to

only include h for memories defined in the universal environment and non-internal fields of each module.

More work related to our modular extension approach is described in [43, 44]. [43] implemented automatic assume-guarantee model checking using a learning algorithm to synthesize assumptions. [44] focused on generating precise component interfaces without additional environment information based on learning during state space traversal. [45] generates environments for components automatically using side-effects and points-to analyses in modular model checking. [46] used an interface grammar to generate component stubs to use in compositional model checking. The Bandera [47] implemented automatic environment generation from the environment assumptions. Environment assumptions are provided as an LTL formula or a regular expression to order the actions in a program. Those generated environments are different from our *universal environments* in that their environments are an abstraction of the entire system to approximate the behavior of the rest of the application rather than most general execution environments that can cover all possible concrete usage scenarios. The target of our environment is different in that the most important requirement is to add no additional happens-before order rather than to provide the specific execution context. We do not restrict the environment of the module according to the specification in this sense. [48] suggested using static dataflow analysis to generate the most general environment of an open reactive system. [49] automated the generation of a reasonable behavior model of the *artificial environment* using static analysis focusing on the level of parallelism. None of the above mentioned approaches deals with data race detection.

The race-free preconditions suggested in Section 5.2 resemble the annotations in type based race detection approaches. The RaceFreeJava type system presented in [14] had a method precondition annotation `requires` to specify necessary lockset information at the method entry. This lockset is used to statically check the `guarded_by` clauses in the method body. The Parameterized RaceFreeJava in [50, 51] extends

RaceFreeJava with ownership parameter and is based on the concept of object ownership. Every method is related with `requires` clause and local type inference reduces the burden of annotations. Extended Parameterized Race-Free Java in [52] allows a more accurate analysis of race conditions and combines it with an atomicity check. JAC [15] distinguishes a precondition annotation `if` and a guard annotation `when`. A guard annotation delays the execution of the invoked method until the condition is satisfied. The synchronization in JAC is determined by the type system and `if` precondition is for specifying the legal environments other than lockset to execute the method. Our race-free preconditions in Section 5.2 include a summary function h as well as an explicit or implicit lockset and are verified during model checking rather than statically checked by a type system.

CHAPTER 3
FOUNDATION

3.1 The Java Memory Model

The new Java Memory Model (JMM) revised as part of Java 5.0 in 2005 [6, 53], defines the legal behavior for a multithreaded Java program and it guarantees sequential consistency when a program is free from a data race. More specifically, JMM defines the legal behavior of correctly synchronized programs and out-of-thin-air guarantees for incorrect programs. The semantics of a memory barrier using a volatile field, immutable object using final declaration, and causality and dependencies are explained both formally and informally to determine legal executions using *commit* iterations. At each iteration, JMM commits a set of memory actions that are consistent with already committed actions and forms a set of legal executions.

The first requirement of JMM is to guarantee sequential consistency for correctly synchronized programs. Correctly synchronized programs are programs without any data race in all sequentially consistent executions of them. For example, the following code fragment contains a data race on `x` and `y` in any sequentially consistent execution, thus the result `r2==2, r1==1`, which is not allowed under sequential consistency, is possible [53].

```
//initially, x = y = 0;
//Thread 1        //Thread 2
  r2 = x;           r1 = y;
  y = 1;            x = 2;
```

Conversely, when both `x` and `y` are declared as volatile variables, since all memory actions on volatiles are synchronization actions, the program is correctly synchronized and the above result is prohibited.

In addition, JMM also guarantees no out-of-thin-air for incorrect programs with a data race. This ensures JMM to be secure and safe even for incorrect programs, which

is not the case in other memory models. An example of out-of-thin-air result is as follows [53]:

```
//initially, x = y = 0;
//Thread 1        //Thread 2
  r1 = x;           r2 = y;
  y = r1;           x = r2;
//r1 == r2 == 42
```

In JMM, even though the above program is not correctly synchronized, it is guaranteed that a value such as 42 will not appear in `r1` or `r2`.

The read and write of a volatile variable are synchronization actions and function as a memory barrier to allow all subsequent memory actions to observe most up-to-date values. An example is the class `Simple` in Chapter 1 with `done` declared as volatile. The read of `done` in `OtherThread` is a memory barrier and the subsequent read of `x` is guaranteed to see most recent write of value 1.

Final fields can be used to create thread-safe immutable objects and guarantee to see the correctly initialized value for final fields themselves and all reachable objects using them.

The causality in JMM restricts causal cycles, such as above out-of-thin-air result code, but allows that resulted from compiler optimizations using global analysis and dependence breaking as follows [53],

```
//initially, x = y = 0;
//Thread 1        //Thread 2
  r1 = x;           r3 = y;
  r2 = r1|1;        x = r3;
  y = r2;
//global analysis allows r1 == r2 == r3 == 1
```

```
//initially, x = y = 0;
//Thread 1        //Thread 2
  r1 = x;           r3 = y;
  r2 = x;           x = r3;
  if (r1 == r2)
    y = 2;
//redundant read elimination allows r1 == r2 == r3 == 2
```

To satisfy this requirement, JMM allows the commitment of any of the uncommitted writes that has the same address and value with already committed writes and any of the uncommitted reads that read a previously committed write in both the justifying execution and the execution being justified. The purpose of this requirement is to prevent a committed action including one that depends on an uncommitted data race.

In the rest of this section, we give an overview of the formal definition of the JMM. For the most part, our treatment follows that of [54],[1] which, in turn, is based on the original specification of the JMM given in [6, 53].[2]

An *action* is a memory-related operation a that belongs to a single thread $Thread(a)$. An action affects variable v or monitor (lock) m and has a *kind*, which is one of the following: volatile read from v, volatile write to v, non-volatile read from v, non-volatile write to v, locking of lock m, unlocking of lock m, starting a thread t, detecting termination of a thread t, and instantiating an object with a set of volatile fields, $volatiles$, and a set of non-volatile fields, $fields$. All of these action types, with the exception of non-volatile read, non-volatile write, and object instantiation, are *synchronization actions*.

[1] Our treatment explicitly handles object instantiation.

[2] The most important differences between [53] and [54] 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), formulates the semantics in terms of finite executions, and ignores external actions.

An *execution* E is given by a tuple $\langle A, P, \xrightarrow{po}, \xrightarrow{so}, W, V \rangle$ where

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

Executions are subject to certain well-formedness constraints, but it is not required that the write-seen function returns the "most recent" write to the variable in question or that the write-seen functions for actions on different threads are consistent, thus allowing various sorts of sequentially inconsistent behavior.

The synchronizes-with relation on actions, denoted $\xrightarrow{sw}$, is given below.

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

In the descriptions above, "subsequent" is determined by the synchronization order.

Well-formedness constraints[3] on executions include such unsurprising requirements as type correctness, correct behavior of locks, consistency with the sequential semantics of the program, and consistency of volatile reads and writes with the synchronization order. In addition, a well-formed execution satisfies *happens-before consistency* where *happens-before* order is a transitive, irreflexive partial order on the actions in an execution obtained by taking the transitive closure of the union of $\stackrel{sw}{\rightarrow}$ and $\stackrel{po}{\rightarrow}$. Happens-before consistency means that a read r of variable v is allowed to see the results of a write $w = W(r)$ provided that

- r is not ordered before w, i.e. $\neg(r \stackrel{hb}{\rightarrow} w)$.
- There is no intervening write $w\prime$ to v, i.e., $\neg\exists w\prime : w \stackrel{hb}{\rightarrow} w\prime \stackrel{hb}{\rightarrow} r$.

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

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

- $w \stackrel{sc}{\rightarrow} r$,
- There is no intervening write $w\prime$ to v, i.e., $\neg\exists w\prime : w \stackrel{sc}{\rightarrow} w\prime \stackrel{sc}{\rightarrow} r$.

A Java program is *correctly synchronized* if all sequentially consistent executions are data race free.

[3] In addition to the well-formedness conditions, *legal* executions according to the JMM are also required to satisfy additional causality conditions that constrain the behavior of programs with data races, thus providing certain safety guarantees for program with races. Our goal is to detect data races so that they can be eliminated rather than reason about the properties of programs with data races, consequently, these conditions are not relevant for the work described in this paper.

Theorem 1. *Any legal execution of a well-formed correctly synchronized program is sequentially consistent.*

This is a property of the JMM [6, 53] and is equivalent to Theorem 1 in [54], where a proof can be found.

Theorem 1 is crucial for justifying our approach. It means that we can use a model checker, which assumes SC, to check whether the program is correctly synchronized, and if so, soundly use the model checker to check other properties as desired.

3.1.1 Model Checker Properties

Since we want to use a model checker to check properties related to the JMM, we need to relate the paths generated by a model checker to the executions defined by the JMM. Paths generated by a model checker are totally ordered and sequentially consistent so that the value read by each read is the value written by the most recent write to that variable. A model checker for Java will usually include byte-code level instructions, some of which are not visible to other threads, and thus do not correspond to actions in the JMM. A path corresponds to an execution if it contains all of the actions in the execution, the total order is consistent with both $\xrightarrow{po}$ and $\xrightarrow{so}$, and the same values are read and written for each action. More formally, a path $Path = \langle A_{path}, \leq_{path}, V_{path}, W_{path} \rangle$ where A_{path} is a set of actions (which may include local actions not visible to other threads), $\leq_{path}$ is a total order on A_{path} corresponds to execution $E = \langle A, P, \xrightarrow{po}, \xrightarrow{so}, W, V \rangle$ if $A_{path}|_A = A$, $\leq_{path}|_{\xrightarrow{po}} = \xrightarrow{po}$, $\leq_{path}|_{\xrightarrow{so}} = \xrightarrow{so}$, $V_{path} = V$, and $W_{path} = W$.

We say that a model checker is *sound with respect to the JMM* if every path generated by a model checker for program P corresponds to a well-formed, sequentially consistent execution of P.

Lemma 1. *If a path generated by a sound model checker contains a data race, then there is a data race in the corresponding execution.*

Two paths are *equivalent w.r.t. an execution* E with actions A if they both correspond to E and the events in A appear in the same order in both paths.

Lemma 2. *Two execution equivalent paths will exhibit the same set of data races.*

A model checker is *complete with respect to the JMM* if for every sequentially consistent execution E, at least one corresponding path is generated.

From Theorem 1 and the above definitions and lemmas, if all paths generated by a model checker that is sound and complete with respect to the JMM for a program P are data race free, so are the sequentially consistent executions of P. If a model checker is sound but not complete, it may still be helpful to find bugs (data races) that may lead to sequentially inconsistent behavior, but not to verify that a program will be sequentially consistent. Bounding and (potentially) partial-order reduction may affect the completeness of a model checker.

3.1.2 A Weakened Condition for Sequential Consistency

Data races in paths can be classified into three kinds: WR, WW, and RW, meaning that the conflicting operations are a write followed by a read, a write followed by a write, and a read followed by a write, respectively. We say that program is *weakly correctly synchronized* if all sequentially consistent executions are data race-free or whose corresponding paths contain only RW data races.

Theorem 2. *Any legal execution E of a well-formed weakly correctly synchronized program is sequentially consistent.*

The proof requires only minor modifications to the proof of Theorem 1 in [54]. The most important part of our proof is a weaker version of Lemma 4. Lemmas 3 and 5 do not rely on data race freedom and the proofs given in [54] require no modification.

Lemma 3. *If $\leq_p$ is a partial order on A and $\leq_t$ is a total order on $S \subseteq A$, then $\leq_q = (\leq_p \cup \leq_t)^+$ is a partial order on A.*

Lemma 4. *For any well-formed execution of a weakly synchronized well-formed program P, if each read sees a write that happens-before it, the execution is sequentially consistent.*

Proof. Let $E = \langle A, P, \overset{po}{\rightarrow}, \overset{so}{\rightarrow}, W, V \rangle$ be a well-formed execution of P.

From Lemma 3, $(\overset{po}{\rightarrow} \cup \overset{so}{\rightarrow})^+$ is a partial order. Let $\leq_t$ be a topological sort of $(\overset{po}{\rightarrow} \cup \overset{so}{\rightarrow})^+$. Since $\leq_t$ is a total order on a well-founded set, it is well-founded. We will prove sequential consistency using well-founded induction on t.

Now, suppose we have a read r in E and all reads $x \leq_t r$ except r see the most recent write. Let w be the most recent write to the variable read by r prior to r so that $W(r) \leq_t w \leq_t r$, and from the hypothesis, $W(r) \overset{hb}{\rightarrow} r$. In a well-formed execution [6, Section 17.4.7],[54, rule 9 in definition 6], W must be consistent with $\overset{hb}{\rightarrow}$, thus $r \overset{hb}{\not\rightarrow} W(r)$ and if $W(r) \overset{hb}{\rightarrow} w \overset{hb}{\rightarrow} r$ then $W(r) = w$. There are three possibilities:

1r sees the most recent write in E w.r.t. $\leq_t$ (i.e. $W(r) = w$)

2there is a WW race between $W(r)$ and w in E w.r.t. $\leq_t$

3there is a WR race between w and r in E w.r.t. $\leq_t$

Now, consider an execution E' that contains all the actions of E prior to r w.r.t. $\leq_t$. Let $A' = \{x | x \leq_t r\}$ and $E' = \langle A', P, \overset{po}{\rightarrow}|_{A' \times A'}, \overset{so}{\rightarrow}|_{A' \times A'}, W[r \rightarrow w], V\rangle$. From Lemma 5, E' is a well-formed execution. From the induction hypothesis and choice of w to be the most recent write to the variable read by r, E' is sequentially consistent.

Because of the way that E' was constructed, if there is a WW race between $W(r)$ and w in E w.r.t. $\leq_t$, then there is a WW race between $W(r)$ and w on a sequentially consistent path in E'. Similarly, if there is a WR race between w and r in E, then there is a WR between w and r on a sequentially consistent path of E'. In either case, we have a sequentially consistent path with a race, which contradicts the assumption that P is weakly correctly synchronized. Thus we conclude that r sees the most recent write in E. □

Lemma 5. *Let P be a well-formed program, $\langle A, P, \overset{po}{\rightarrow}, \overset{so}{\rightarrow}, W, V\rangle$ a well-formed execution, $\overset{hb}{\rightarrow}$ its happens-before order, $\leq_t$ a total order on A, $r \in A$ a read action of variable v, and $w \in A$ a write action to v such that:*

- *$\leq_t$ is consistent with $\overset{so}{\rightarrow}$ and $\overset{hb}{\rightarrow}$,*

- *for every read* $r \in A$ *we have* $W(r) \stackrel{hb}{\rightarrow} r$*,*
- w *is the most recent write to* r *in* $\leq_t$*, i.e.* $w \leq_t r$ *and for all writes* w' *to variable* v *either* $w' \leq_t w$ *or* $r \leq_t w'$*.*

Let A' *be* $\{x | x \leq_t r\}$*. Then the execution* $\langle A', P, \stackrel{po}{\rightarrow}|_{A' \times A'}, \stackrel{so}{\rightarrow}|_{A' \times A'}, W[r \rightarrow w], V\rangle$ *is a well-formed execution.*

Proof. Given in [54]. □

The rest of the proof relies on the definition of a legal execution in the JMM. In order to make the presentation self-contained, we will repeat the definition of a legal execution from [54, Definition 7].

Definition: Legal Execution A well-formed execution $E = \langle A, P, \stackrel{po}{\rightarrow}, \stackrel{so}{\rightarrow}, W, V\rangle$ with happens-before order $\stackrel{hb}{\rightarrow}$ 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. $\leq_{hb_i}|_{C_i} = \stackrel{hb}{\rightarrow}|_{C_i}$
3. $\leq_{so_i}|_{C_i} = \stackrel{so}{\rightarrow}|_{C_i}$
4. $V_i|_{C_i} = V|_{C_i}$
5. $W_i|_{C_{i-1}} = W|_{C_{i-1}}$
6. For all reads $r \in A_i - C_{i-1}$, $W_i(r) \leq_{hb_i} r$
7. For all reads $r \in C_i - C_{i-1}$, $W_i(r) \in C_{i-1}$ and $W(r) \in C_{i-1}$

These conditions are intended to rule out causal loops that could lead to situations where data races cause values read to appear ”out of thin air”.

Now, we return to the proof of the Theorem 2. From Lemma 4, it is sufficient to show that every read in a legal execution E of a weakly correctly synchronized program sees a write that happens-before it.

Since E is legal, there is a committing sequence $\{C_i, E_i\}$ justifying E. We show by induction on i that for all reads in C_i, $W(r) \xrightarrow{hb} r$.

The base case, $C_0 = \phi$ holds trivially.

Now, we assume for all reads in $r \in C_{i-1}$, we have $W(r) \xrightarrow{hb} r$, and show that for any read $r \in C_i$, we have $W(r) \xrightarrow{hb} r$.

From the induction hypothesis and legality rules 2 and 5, we get $W_i(r) \leq_{hb_i} r$ for all reads $r \in C_{i-1}$. From legality rule 6, we get for all reads $r \in A_i - C_{i-1}$, $W_i(r) \leq_{hb_i} r$, and since $C_i \subseteq A_i$ and $C_{i-1} \subseteq C_i$, we have for all reads $r \in C_i$, $W_i(r) \leq_{hb_i} r$. From legality rule 7, for all reads $r \in C_i = C_{i-1}$, we have $W_i(r) \in C_{i-1}$ and $W(r) \in C_{i-1}$, and thus from rule 5, $W_i(r) = W(r)$. From rule 2, we can conclude, for all reads $r \in C_i = C_{i-1}$, $W(r) \xrightarrow{hb} r$ and thus with the induction hypothesis the desired result for all reads $r \in C_i$, $W(r) \xrightarrow{hb} r$. Since all reads in A belong to some C_i, this implies that every read in E sees a write that happens before it.

Since correct synchronization implying sequential consistency is considered a fundamental property of memory models [11], this result is of intrinsic interest. As will be discussed in Chapter 4, ignoring RW data races will allow us to give a more efficient implementation of JRF without sacrificing the ability to verify sequential consistency.

3.2 Model Checking and Java PathFinder

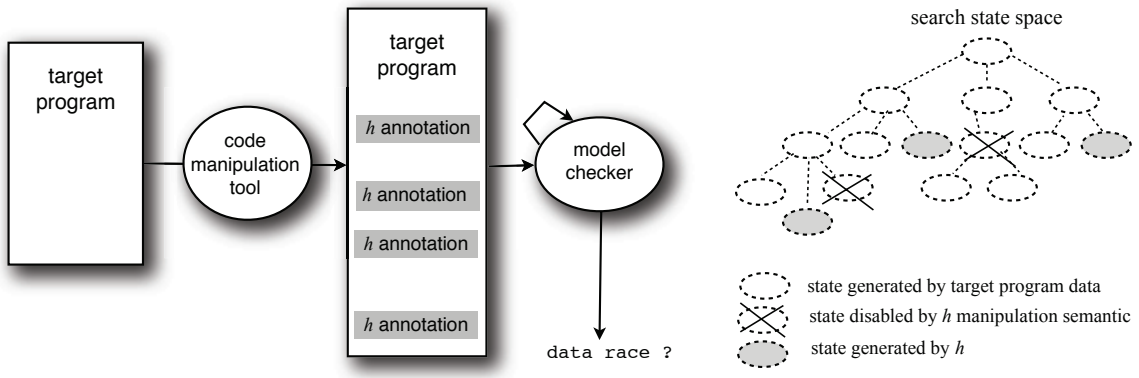
Model checking is a formal verification method for a concurrent system. The system under test will be verified using the correctness specification given in temporal logic by exhaustively searching all possible interleaving of threads. Traditionally, model checking is widely used in hardware circuit design to verify the behavior is consistent with the design specification but recently, software model checkers are accepted as powerful and useful automated tools thanks to their advantages [55]. Unlike logical inference using a theorem prover, model checking does not require a proof and is easy to express concurrency properties in logics. In addition, model checkers provide counterexamples when the property is violated. However, because it is an exhaustive approach, they

inevitably suffer state space explosion. In a modern practical system, there are too many threads and possible data choices. Techniques to address the problem have introduced such as a partial order reduction, symbolic representation of states and transitions, and compositional model checking [56].

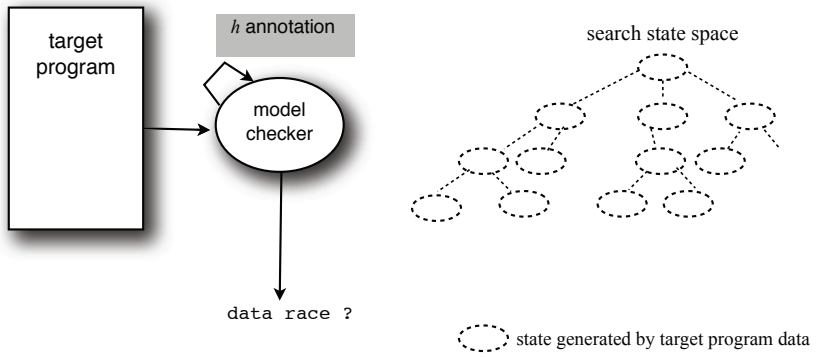
Partial order reduction is a technique using abstraction of states to reduce total number of states. Given a property, partial order methods explore only a reduced part of the global state space that is sufficient for checking the given property [57]. The simplest partial order reduction is to reduce the independent transitions. Symbolic model checking avoids explicit enumeration of states using a binary decision diagram (BDD), a canonical form of representing boolean formula [58]. The compositional approach is an effective technique to achieve scalability in complex system verification. Instead of exploring the entire state space of the system under test, modular model checking decomposes the system into smaller independent modules and applies a verification algorithm to each individual module with an environment model. Once every module is checked with a set of assumptions of the environment model, these assumptions are verified on the environment to discharge them. This assume-guarantee reasoning is widely used with environment generation in automatic verification approaches. Modular model checking based on this technique generally requires additional information provided manually about the module's environment. This manual annotation is an error-prone process and there were attempts to automate the assumption generation [41, 42]. One more requirement using this paradigm is to restrict the generated environment sufficiently to avoid false positives. [43, 44] try to make those interfaces precise through learning search space. Bounded model checking [59] considers only a finite prefix of a path with length k by unrolling the finite state machine for k steps. The closed system in model checking naturally has a bound in the number of threads and the number of repeated iterations. This is another limitation using model checking to check

properties in open systems such as libraries. The bound constraint in JRF modular extension is explained in Section 5.2.

Among different approaches to detect data races discussed in Section 2.1, we choose model checking with indirect code instrumentation to implement summary function h described in Section 4.1. First of all, we ruled out a dynamic approach due to its weakness: it only detects races in a current execution. We also ruled out direct code modification through annotating a target program with codes implementing our algorithm in Section 4.1. It is undesirable that the program semantics would be changed by



A Direct h annotation through an extra annotation tool



B Indirect h annotation through a model checker internal function

Figure 3-1. Operational model of direct vs indirect annotation in model checking

additional code. Figure 3-1 shows two operational models, 3-1A one using direct code annotation to target application and the other 3-1B indirect h annotation accomplished by model checker internal function. The latter does not change the number of states to search and will finish in one pass without preprocessing in code manipulation tool.

The JPF [60] model checks Java byte code by reading Java class files and simulating their execution using its own virtual machine with on-the-fly verification of specified properties. A property violation is reported by JPF along with a counterexample, the execution path that led to the violation. JPF is an explicit state software model checker, and provides a listener interface that we used to extend its functionality for JRF to avoid direct code instrumentation. The interface provides a set of callback functions allowing low level operations, such as object creation, object locking and unlocking, the start of a new thread, and each execution of an instruction to be intercepted and augmented with user-supplied code. The defects JPF is capable of handling are user specific properties provided by programmers, as well as deadlocks and unhandled exceptions. A recent release of JPF includes several extensions in separate runtime modules that are maintained as their own projects. Examples of interesting extensions are *jpf-concurrent* and *jpf-symbc* projects. The *jpf-core* project, which implements virtual machine and core mechanisms, does not support the full functionalities of the standard Java library. Most of all, the *java.util.concurrent* package is an important core module in concurrent data structures, but only is partially implemented in *jpf-core*. The *jpf-concurrent* project is an optimized extension of *java.util.concurrent* and currently implements half of the original constructs. The *jpf-symbc* is the symbolic execution framework of JPF. It uses a configurable instruction set, called *factory*, to change the instruction semantics from immediate value to symbolic execution.

CHAPTER 4
JAVA RACEFINDER

4.1 The Summary Function h

In this section, we introduce a function h that summarizes $\xrightarrow{hb}$ at each point in an SC execution, allowing data races to be detected as they occur during model checking. Let *Addr* be the set of (abstract) memory locations representing non-volatile variables in the program, *SynchAddr* be the set of (abstract) memory locations representing variables with volatile semantics and locks, and *Threads* be the set of threads.

We summarize the happens-before relation as a function

$h : \mathit{SynchAddr} \cup \mathit{Threads} \rightarrow 2^{\mathit{Addr}}$

that maps threads and synchronization variables to sets of non-volatile variables so that $x \in h(t)$ means that thread t can read or write variable x without causing a WW or WR data race.

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

$$h_0 = \lambda z.\text{if } z = \mathit{main} \text{ then } static(P) \text{ else } \phi \tag{4–1}$$

The way that h_{n+1} is obtained from h_n depends on the action a_n. First, we define four auxiliary functions; *release*, *acquire*, *invalidate*, and *new*. The function *release*(t, x) takes a summary function h and yields a new summary function by updating $h(x)$ to include the value of $h(t)$. It is used with actions by thread t that correspond to the source of a $\xrightarrow{sw}$ edge, for example, writing a volatile variable x, releasing lock x, starting thread x, etc.

$$\mathit{release}(t, x)\ h \ \widehat{=}\ h[x \mapsto h(t) \cup h(x)] \tag{4–2}$$

The function *acquire*(t, x) takes a summary function h and yields a new summary function obtained from h by updating $h(t)$ to include the value of $h(x)$. It is used in actions that form the destination of a $\stackrel{sw}{\rightarrow}$ edge, for example, reading a volatile x, locking lock x, and joining or detecting termination of thread x.

$$\textit{acquire}(t, x)\ h \ \widehat{=}\ h[t \mapsto h(t) \cup h(x)] \tag{4–3}$$

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

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

The function *new* is used to incorporate a newly instantiated object into the summary function. It yields a new summary by adding the set *fields* to the value of $h(t)$ and initializing the previously undefined values of h for the new volatile variables.

$$\begin{aligned} &\textit{new}(t, \textit{fields}, \textit{volatiles})\ h \ \widehat{=} \\ &\lambda z.\quad \text{if } (t = z) \text{ then } h(t) \cup \textit{fields} \\ &\qquad\quad \text{else if } (z \in \textit{volatiles}) \text{ then} \{\} \\ &\qquad\quad \text{else } h(z) \end{aligned} \tag{4–4}$$

We define

$$\textit{norace}(x, t) = x \in h(t) \tag{4–5}$$

The definition of h_{n+1}, which depends on h_n and action a_n, is given in Figure 4-1.

To detect data races during model checking, we maintain h and check $norace(x, t)$ before the reading or writing of non-volatile x by thread t. When this condition holds for all non-volatile reads and writes in an execution, we say the execution is *h-legal*.

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

Figure 4-1. Definition of h_{n+1}

We can prove several facts about *h-legal* executions. In the following, we use $last(E)$ to denote the last element of finite sequence E and $E_n|_{w(x)\cup new(x)}$ to denote the subsequence of actions in E_n that write to x plus the instantiation action.

The first, rather obvious lemma confirms our intuition that non-static variables belonging to objects that have not been instantiated yet have not been written in any *h-legal* execution and will serve as the base case for inductive proofs of other results.

Lemma 6. *For an SC execution E, and non-static x, if x has not been instantiated, $E_n|_{w(x)\cup new(x)}$ is empty.*

The following two lemmas, for non-static and static variables, respectively, say that at any point in an h-legal, SC execution, the most recent thread to write a non-volatile variable can access it without causing a WR or WW data race.

Lemma 7. *For an h-legal SC execution E_n and non-volatile, non-static variable x, if x has been instantiated in E_n and $t = thread(last(E_n|_{w(x)\cup new(x)}))$, then $x \in h_n(t)$.*

Proof. The proof is by induction. Since x cannot have been instantiated in E_0, the base case holds trivially. Now, assume the lemma holds for n and show that it holds for $n+1$. There are four cases:

- x has not been instantiated in E_n and action a_n does not instantiate it. Then the lemma continues to hold trivially.

- x has not been instantiated in E_n and action a_n is an action of thread t that instantiates x. Then $last(E_{n+1}|_{w(x) \cup new(x)}) = a_n$ and from the rule for object instantiation in Figure 4-1, $x \in h_{n+1}(t)$.

- x has been instantiated in E_n, $x \in h_n(t)$ where $t = thread(last(E_n|_{w(x) \cup new(x)}))$ and $\mathit{thread}(a_n) = t$. From Figure 4-1, there are no actions performed by thread t that have the effect of removing items from $h_n(t)$ to obtain $h_{n+1}(t)$.

- x has been instantiated in E_n, $x \in h_n(t)$ where $t = thread(last(E_n|_{w(x) \cup new(x)}))$ and $\mathit{thread}(a_n) \neq t$. From Figure 4-1, either $x \in h_{n+1}(t)$ or a_n writes to x with $x \in h_n(\mathit{thread}(a_n))$. This falsifies $t = thread(last(E_n|_{w(x)})$ while establishing $\mathit{thread}(a_n) = thread(last(E_{n+1}|_{w(x)})$. Since the execution is *h-legal*, $x \in h_{n+1}(\mathit{thread}(a_n)$.

□

Lemma 8. *For an h-legal, SC execution E_n and non-volatile, static variable x, if $t =$ (if $E_n|_{w(x)} \neq \theta$ then thread$(last(E_n|_{w(x)})$ else $main$) then $x \in h_n(t)$.*

Proof. Again the proof is by induction. Since x is static, $x \in h_0(main)$ and the base case holds. Now, assume the lemma holds for E_n and show that it holds for E_{n+1}. There are two cases that are similar to the last two cases in lemma 7.

- $x \in h_n(t)$ where $t =$ (if $E_n|_{w(x)} \neq \theta$ then $\mathit{thread}(last(E_n|_{w(x)})$ else $main$) and $\mathit{thread}(a_n) = t$. From Figure 4-1, there are no actions performed by thread t that have the effect of removing items from $h_n(t)$ to obtain $h_{n+1}(t)$.

- $t =$ (if $E_n|_{w(x)} \neq \theta$ then $\mathit{thread}(last(E_n|_{w(x)})$ else $main$) and $\mathit{thread}(a_n) \neq t$. From Figure 4-1, either $x \in h_{n+1}(t)$ or a_n writes to x with $x \in h_n(\mathit{thread}(a_n))$. This falsifies $t =$ (if $E_n|_{w(x)} \neq \theta$ then $\mathit{thread}(last(E_n|_{w(x)})$ else $main$) while establishing $\mathit{thread}(a_n) = thread(last(E_{n+1}|_{w(x)})$. Since the execution is *h-legal*, $x \in h_{n+1}(\mathit{thread}(a_n)$.

□

The next lemma forms the basis of our soundness proof. In this lemma, $a \xrightarrow{hb=} b$ indicates either $a \xrightarrow{hb} b$ or $a = b$.

Lemma 9. *Let E be a well-formed, h-legal SC execution. Then for all non-volatiles x, all threads t, all volatiles v, and all n,*

$$x \in h_n(t) \;\Rightarrow\; E_n|_{w(x)\cup new(x)} \xrightarrow{hb=} last(E_n|_t)$$

$$\wedge$$

$$x \in h_n(v) \;\Rightarrow\; E_n|_{w(x)\cup new(x)} \xrightarrow{hb=} last(E_n|_{w(v)})$$

where for set S, $S \xrightarrow{hb=} s$ means $(\forall s\prime : s\prime \in S : s\prime \xrightarrow{hb=} s)$.

Proof. The proof is by induction. For the base case, we have E_0, which does not contain any actions and where $h(main) = static(P)$ and h is undefined for all other arguments. Since E_0 is empty, the right sides of both implications are trivially true. Now, we assume the property for E_n and show it holds for E_{n+1}. We must consider each kind of action.

- Write a volatile field v by thread t. $h_{n+1}(t) = h_n(t)$, and due to program order and the hypothesis, $E_n|_{w(x)\cup new(x)} \xrightarrow{hb} last(E_n; write(v, t)|_t)$. The second requirement is similar to $x \in h_n(v)$. For the new additions, we already have $x \in h_n(t) \rightarrow E_n|_{w(x)\cup new(x)} \xrightarrow{hb} last(E_n; write(v, t)|_t)$. Since this write will become $last(E_n|_{w(v)})$, the condition is reestablished.
- Read a volatile field v by thread t. This has the effect of adding the elements satisfying $x \in h_n(v)$ to $h_{n+1}(t)$. Since for any such x, all writes of $x \xrightarrow{hb}$ the latest write of v, which $\xrightarrow{hb}$ the read which is the current action on thread t, and thus becomes $last(E_{n+1}|_t)$, thus reestablishing the condition.
- Lock the lock lck. The situation is analogous to reading a volatile
- Unlock the lock lck. The situation is analogous to writing a volatile
- Start thread $t\prime$. This case is somewhat peculiar since it involves two threads, and one of them does not have any actions. To deal with this, we consider that the first action of a thread is a synthetic action where the start action by the starting thread happens-before this action. Then, by transitivity of $\xrightarrow{hb}$, the first condition is reestablished.
- Join thread $t\prime$. This case follows easily from the transitivity of $\xrightarrow{hb}$.
- Detecting termination t' with t'.isAlive(). Similar to join.

- Write a non-volatile field x by thread t. In this case, both conditions are violated–it need no longer be the case, for example, that that all writes of x happen before the last statement of thread t'. The solution is to remove x from $h_{n+1}(t\prime)$, thus reestablishing the condition. Since we require $x \in h(t)$ before performing the write, the required condition for thread t will be maintained.

- Read a non-volatile field x. This does not affect the second condition, and extending E_n with a read operation preserves the first.

- Instantiate an object containing non-volatile fields by thread t non-volatile fields *fields* and volatile fields *volatiles* The instantiate adds the new non-volatiles to $h_{n+1}(t)$ while extending E_n with the instantiate action, thus preserving the first condition. The second condition is not affected by merely defining a new $h(v)$ to be empty.

□

The lemma tells us that if $x \in h_n(t)$, then all the preceding writes to x happen-before the latest action on thread t. Since there are no writes to x between $last(E_n|_t)$ and a_n, by program order, $last(E_n|_t) \xrightarrow{hb} a_n$, and by transitivity all writes to $x \xrightarrow{hb} a_n$. As a result, if a_n is a write by thread t to x, the action can be performed without causing a WW data race.

The next theorem, which follows easily from Lemma 9 justifies our approach.

Theorem 3. *If all sequentially consistent executions of a well-formed program are h-legal, then the program is weakly correctly synchronized and thus all its legal executions are sequentially consistent.*

4.2 Data Race Specific Search Heuristic

Model checking is a way to verify program correctness by exhaustively exploring all possible states of a multithreaded program. The most well-known problem of model checking is a state-space explosion and this limitation makes JPF not able to be scalable.

Our first approach to this problem was program slicing. The primary idea was to divide the whole program into small slices with minimum codes in them and were capable of concluding if a specific data has a race or not. We had used Indus Java slicer[61] in our preliminary experiment, but the result was not as useful as we had

expected. The concurrent programs are complicated and hard to slice into small pieces. Although a non-volatile data access is free from other non-volatile data race analysis in that the h of a non-volatile is independent of that of other non-volatiles, we cannot exclude those accesses if they change the control flow. Another hinderance was the difficulties of incorporating several tools. JPF and Indus have their own requirements about the specific version of Java platform and other third-party tools.

The next approach was to use a method summary to summarize the h-related behavior of modules. Instead of exploring all possible states, we intended to search the states in a method summary. However, the formalization of a method summary was not successful, and the workload to implement this mechanism into JPF was also problematic.

In this section, we explain the heuristic search algorithms that incorporate the rationales about how to detect a race earlier by choosing an execution path with higher probability of a race. Section 4.2.1 includes a motivating small example, and Chapter 6 has more supporting experimental results.

4.2.1 Heuristics

Contemporary model checking research suggest a better way to solve the scalability problem. Instead of reducing the number of states to visit, the focus is to find error earlier if one exists. It is still not guaranteed to terminate, but the successful detection of error can be increased by changing the order of state searching according to heuristics.

At each state, a model checker chooses the next state from all possible candidates and checks the properties that should be satisfied at that point in the program. When there are no candidates for a next state left, it backtracks to check any remaining scheduling sequences.

The order that states are traversed during model checking influences how many states must be visited before errors are found. Since a data race requires the interaction of two threads, a search strategy with more threads interleaving is likely to

find a data race earlier than a depth-first search (DFS) strategy, which has minimum interleaving through sequential scheduling of threads at the beginning. Rather than simply increasing thread interleaving, we also consider the nature of data races and propose the following heuristics, which depend on the current value of h, to choose the next state. In the descriptions, $curr$ refers to the current thread.

- **Writes-first (WF):** Although a data race may occur at either a read or a write, the source of a data race involving memory location m is a write of m that causes an *invalidate* operations to remove m from $h(t)$ for all $t \neq curr$. Any future read or write to m by some other thread results in a data race unless an appropriate synchronization action has occurred. Thus this heuristic prioritizes write operations.

- **Watch-written (WW):** If there has been a write on a memory location, m, it is possible that a future read or write on m by another thread will result in a data race. Thus, this heuristic prioritizes operations on a memory location that has recently been written by a different thread.

- **Avoid *release*/*acquire* (ARA):** A data race free program involves appropriately located matching *release* and *acquire* operations. Although programs with data races may also have *acquire* and *release* operations, the existence of these on a path may indicate a lower probability of the existence of a data race. This heuristic prioritizes operations on threads that do not have a recent acquire operation preceded by a matching release on the execution path.

- **Acquire-first (AF):** When an *acquire* operation is executed after a matching *release*, a happens-before edge is created on the current path. However, if the *acquire* is executed before the matching *release* statement then this does not result in an happens before edge and cannot prevent a data race. This heuristic prioritizes acquire operations that do not have a matching release along the execution path. This situation often corresponds to situations of unsafe publication of an otherwise correctly synchronized object.[1]

The main purpose of these heuristics is to minimize the happens-before orderings that prevents a data race to occur. No heuristic is disjoint and could be combined to

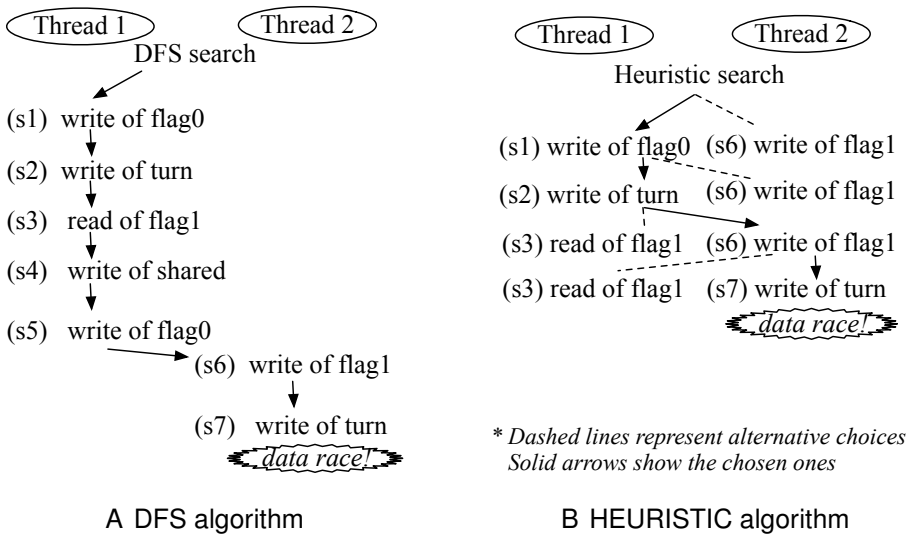
[1] Publication of an object is the act of making its reference visible to other threads. Unsafe publication (section 3.5 of [62]) can lead to a partially constructed object becoming visible to other threads.

```
Initially, flag0 = flag1 = turn = shared = 0;
/* all fields are non-volatile */
        Thread 1                                        Thread 2
=========================================   ========================================
s1: flag0 = 1;                              s6:flag1 = 1;
s2: turn = 1;                               s7:turn = 0;
s3: while (flag1==1 \& turn==1) {/*spin*/}  s8:while (flag0==1 \& turn==0) {/*spin*/}
s4: shared++;                               s9:shared++;
    /*critical section*/                       /*critical section*/
s5: flag0 = 0;                              s10:flag1 = 0;
```

Figure 4-2. One iteration of Peterson's Algorithm

make the heuristic more powerful. The experimental result for different configuration of them is discussed in Chapter 6.

A fragment of the well known Peterson's algorithm in Figure 4-2 shows the advantage of the WF heuristic over a depth-first search. The search space using DFS for this example is shown in Figure 4-3A, and the WF heuristic search in Figure 4-3B. DFS-based model checking takes seven states to find the data race on *turn*, while the



A DFS algorithm B HEURISTIC algorithm

Figure 4-3. Model checking of Peterson's algorithm using different search strategies

heuristic search finds the same race after visiting only four states. The counterexample path is shorter and easier to analyze.

4.2.2 Algorithm

Figure 4-4 shows the heuristic search algorithm used in JRF. It stores the states in a max priority queue, where a priority is the sum of the heuristic value and the depth of a given state times the max heuristic value (MAX=$WRITE_{written\_by\_other}$). The heuristic values only affect the choice of a next state among children of the current state. Once the next state is chosen and advanced, its newly generated children always have higher priority than the states remaining in the queue. This guarantees that the highest priority children and their descendants will be explored first. The search algorithm visits the states in descending priority order until it reaches an error state or no more states are left.

The algorithm for computing the heuristic value for a state is given in Figure 4-5. The heuristic values are determined by considering the data race related heuristics.

```
AlgAlgorithm HeuristicSearch
  Q: max priority queue
  s, s': state
  value: integer
  Q ← empty
  put (initial state, max integer) into Q
  while Q not empty do
      (s, value) ← remove from Q
      if s is an error state then
        print("error")
        break
      for each successor s' of s do
         if s' is not marked then
           mark s'
           put (s', HeuristicValue(s') + depth * MAX) into Q
```

Figure 4-4. Heuristic search algorithm. States are prioritizes based on their heuristic value as computed in Figure 4-5 and the search depth.

```
Algorithm HeuristicValue (s: state)
  v: variable
  if s reached via write to non-volatile v then
    if v most recently written by another thread then
      return 8 /*WRITE_written_by_other (WF ∨ WW)*/
    else return 7 /*WRITE_written_by_self (WW) */
  if s reached via read from non-volatile v then
    if v most recently written by another thread then
      return 6 /*READ_written_by_other (WF) */
    else return 5 /*READ_written_by_self (WF ∧ WW) */
  if s reached via read from a volatile v
  or locking an object not released before then
    return 4 /*ACQUIRE_without_prior_release (ARA) */
  if s reached via read from a volatile v
  or locking an object released before then
    return 2 /*ACQUIRE_with_prior_release (ARA ∨ AF) */
  if s reached via volatile write
  or unlocking an object then
    return 1 /*RELEASE (AF) */
  else return 3 /*OTHER (ALL) */
```

Figure 4-5. Algorithm for deciding heuristic values for states based on their likelihood of leading to a data race. Heuristic values become available according to the heuristics (WF, WW, ARA, AF) presented in Section 4.2.1.

Non-volatile $WRITE$ has higher heuristic value (8 and 7) than non-volatile $READ$ (6 and 5, respectively) whenever *writes-first* is configured. Non-volatile $_{written\_by\_other}$ (8 and 6) is greater than non-volatile $_{written\_by\_self}$ (7 and 5, respectively) with *watch-written*. *acquire-first* and *avoid release-acquire* also determine the 1, 2, and 4 hierarchy. Non-volatile accesses (5 through 8) are always the best candidates for a data race, and $acquire/release$ accesses that generate happens-before edges should be avoided for as long as possible. $ACQUIRE_{without\_prior\_release}$ is chosen earlier than regular operations to avoid a future *release-acquire* possibility.

The main purpose of those heuristics is to follow paths that minimize the happens-before orderings since it is the lack of happens-before orderings that causes data races. The heuristics can be used together and the heuristic search algorithm given in Figure

4-5 can be configured in various ways by turning on and off each of the 4 heuristics: *WF*, *WW*, *AF*, *ARA*. The example seen earlier in Figure 4-3B shows the search path with only *write-first* configured. For example, if the choice of heuristic is *writes-first* and *watch-written*, the only available heuristic values are $WRITE_{written\_by\_other}$, $WRITE_{written\_by\_self}$, $READ_{written\_by\_other}$, $READ_{written\_by\_self}$, and $OTHER$.

In comparison with DFS, the heuristic search algorithm requires more memory. A model checker using DFS only advances and backtracks, while a heuristic search generates and stores all possible child states before advancing, and then later restores them. Although only Δs of h related data are stored during state advance and backtrack, a complete copy of the h is saved when the child states are generated. Heuristic search also tends to take more time because it visits more states in general. As we can see from Figure 4-3, DFS only generates 7 states to find the race, but the WF heuristic search generates 8 states since it takes into consideration all children of the current state to decide next one. The advantage of heuristic search is that, in most cases, it tends to find data races with a shorter counterexample path than DFS. This is important because, in our experience, reasoning about the cause of a data race using the counterexample path is not a straight-forward task, especially when the length of the path is fairly long. Experimental results and further discussion about understanding the counterexample path are given in Chapter 6.

4.3 Implementation

The core of JRF is to maintain a representation of the summary function *h* described in Section 4.1; the listener code intercepts relevant instructions and updates the representation, as described in Figure 4-1. In addition, the norace property is checked prior to all non-volatile reads and writes.

In addition to the indirect code instrumentation to maintain summary function h, JRF implements efficient search heuristics in order to decrease the number of visited states before the race detection. JRF also uses techniques to leverage the overhead of

bookkeeping additional information such as h and acquiring history; lazy representation of array elements and h bitmap entries, storing Δ of h into a stack, and excluding an untracked variable and threadlocal memories from the h entry.

Figure 4-6 shows the components of JRF. All components are implemented as a configurable module to allow the most flexibility in using JRF. In Figure 4-7, the default configuration of JRF from a `jrf.jpf` file can be found. The directory structure of a JRF distribution follows the guidelines of JPF runtime modules. The following sections describe the implementation details of each JRF module.

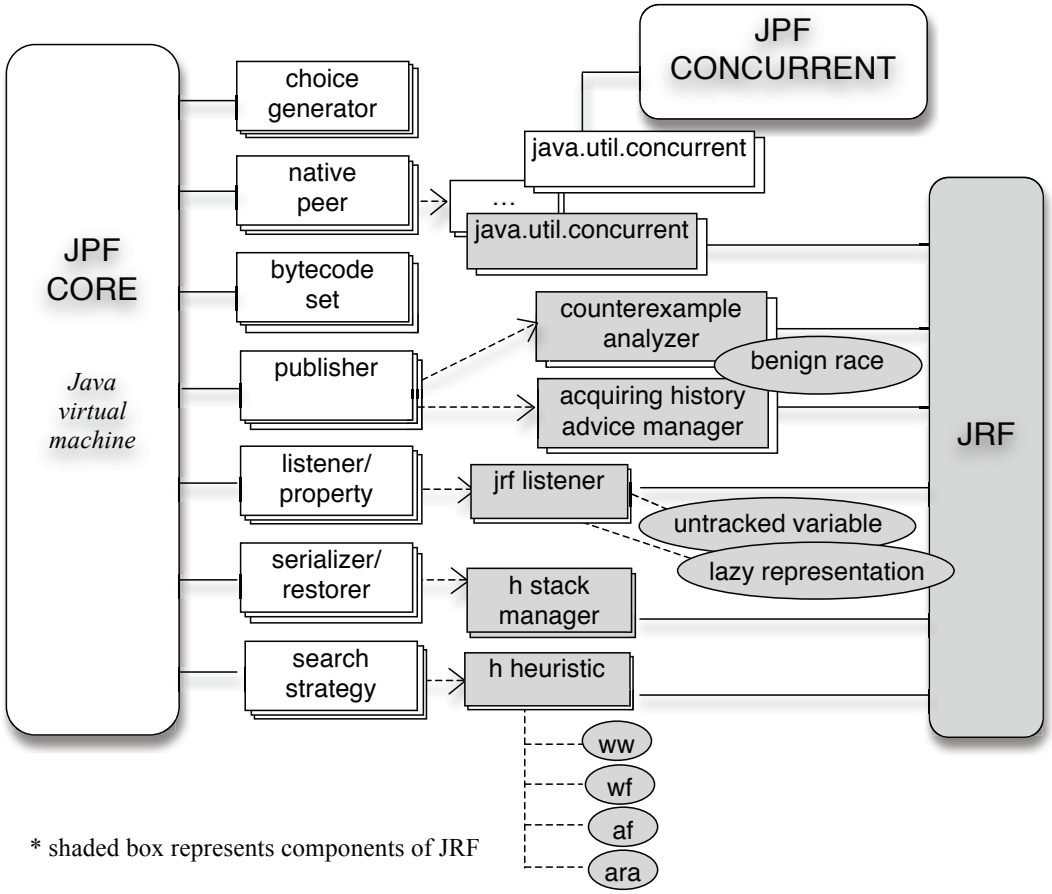


Figure 4-6. JRF components and their JPF counter parts

```
#######################################
## By KyungHee Kim on Dec/06/2009
# jrf.jpf - the property for JRF
# loaded on top of default.properties of JPF
#######################################

# (1) JRF-S prerun
#--------------------------------------
## configure this for threadlocal prerun
#--------------------------------------
## uncomment following three lines and comment out all others
#search.multiple_errors = true
#listener = edu.ufl.cise.jrf.listener.JRFSListener
#vm.fields_factory.class = jrf.factory.JRFSFieldsFactory
#--------------------------------------

# (2) All others
#--------------------------------------
## (2-0) Common configuration
#--------------------------------------

#######################################
## configure this for elementinfo factory
#######################################
vm.fields_factory.class = edu.ufl.cise.jrf.factory.JRFFieldsFactory

#######################################
## to configure benign race & target model classes
#######################################
## specify the position of benign races here
jrf.benign=java/util/concurrent/locks/AbstractOwnableSynchronizer.java:56, java/lang/String.java:652

## specify specific excluding path here
jrf.excluding=true
jrf.excluding.path=java.lang, java.util, java.io, java/io, java/lang, java/util, [JPF_INITIALIZER], sun.misc, gov.nasa.jpf
#--------------------------------------

#--------------------------------------
## (2-1) When to stop search?
#--------------------------------------
## single race or multiple races?
jrf.multiraces=true
jrf.multiraces.threshold=0 ## 0 means find all races

## time limit
#jrf.timeconstraint=0 ## 0 means no time limit

#--------------------------------------
## (2-2) Which search strategy?
#--------------------------------------
search.class = edu.ufl.cise.jrf.search.HBDFSearch
## uncomment following two lines to use jrf heuristic search
#search.class = edu.ufl.cise.jrf.search.HBHeuristicSearch
#search.heuristic.HBHeuristic = acquire_first, watch_written, write_first, avoid_release_acquire
## search.search.HBHeuristic can be configured independently

#--------------------------------------
## (2-3) Using standalone reporter?
#--------------------------------------
jrf.result_path=jrf ## directory where to store counterexample trace
jrf.reporter.standalone=false
#jrf.result_spreadsheet=RESULT_SPREADSHEET # summary of execution written to this file

#--------------------------------------
## (2-4) Using threadlocal optimization feature?
#--------------------------------------
jrfs = false
jrfs.result_path = "jrfs" ## directory where to store shared memory information

#--------------------------------------
## (2-5) Configure Listener
#--------------------------------------
## the order of listener is important : HBSearch > JRF-M > JRF-E > JRF
listener = edu.ufl.cise.jrf.listener.HBSearchListener,edu.ufl.cise.jrf.listener.JRFListener

#--------------------------------------
## (2-6) In case of JRF-M, configure this
#--------------------------------------
## specifies the name of library classes
#jrfm.compose_class = jrfm.DisBarrier,jrfm.UnboundedQueue,jrfm.SimpleTree
#jrfm.verify_class = jrfm.DisBarrier,jrfm.UnboundedQueue,jrfm.SimpleTree,jrfm.PetersonLock

## what to do when found a precondition violation
#jrfm.multi_errors = true
```

Figure 4-7. The default JRF configuration in `jrf.jpf` file

4.3.1 Representation of h

In this section, we describe a representation of h that is suitable for implementation in a model checking tool. This context requires space efficiency, efficient updating, and, since model checking involves backtracking, a way to efficiently save and restore previous incarnations. We take advantage of the fact that, in Java, threads and locks are also objects and handle elements of *SynchAddr*, *Thread*, and *Addr* uniformly as ”memory locations”.

Recall that h maps $SynchAddr \cup Thread$ to 2^{Addr}. This mapping is implemented as an array of bit vectors. Each $SynchAddr \cup Threads$ is given a unique index,[2] conceptually corresponding to a row, and *Addr* an index conceptually corresponding to a column. Then $norace(t, x)$ holds when $h[row(t), column(x)] == 1$.

When an element of $SynchAddr \cup Threads$ is created, an element of the array of bit vectors is reserved. When an element of *Addr* is created, an index is assigned, but space in the corresponding bit vector is not allocated until it is actually used. This implies that the bit vectors are dynamically resized. The *acquire* and *release* operations can be implemented as a single set union step, and the *norace* check, which is used most often, can be done with bit masking. The *invalidate* operation, however, involves a number of rows of set operations. Fortunately, $|SynchAddr \cup Threads|$ is far less than $|Addr|$. In our experiments, the former is on the order of tens, while the latter can easily exceed thousands.

[2] Each memory location is given a unique key constructed from the name of the class, the instance number, and the field name (or array index if the location in question is an array element). The keys are in turn mapped to the corresponding index in the bit vector for elements of *Addr*, and the array holding the bit vectors for elements of $SynchAddr \cup Thread$.

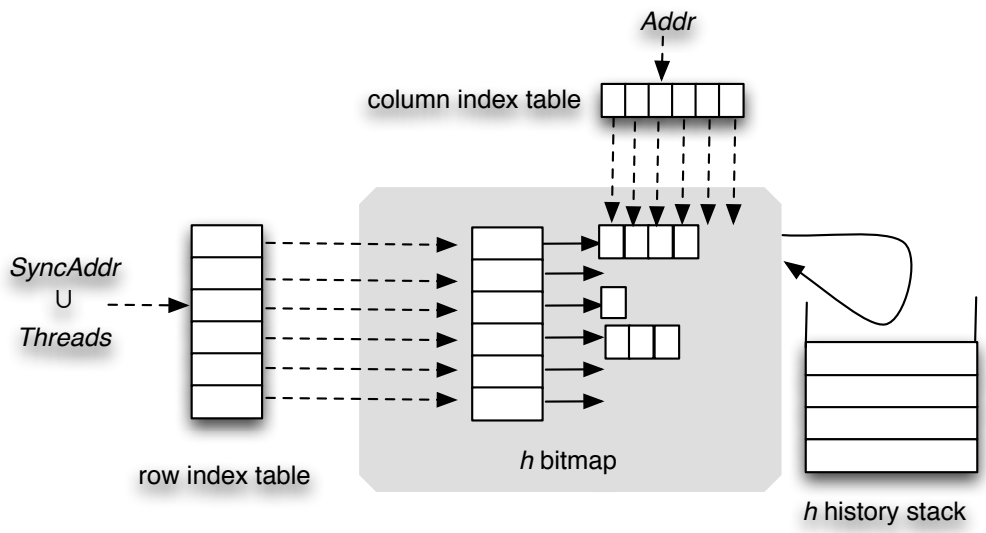


Figure 4-8. Internal representation of h

The changes in h are stored in a separate stack in order to restore the value to an earlier state when the model checker backtracks. Figure 4-8 illustrates the bitmap for h, and a history stack used to store Δs. The Δs of h include the changes following *acquire*, *release*, *invalidate*, and *allocation*, including reuse of garbage collected memory and instantiation in a static initializer. In addition, the changes in *prior_release* and *written_by* information, explained in Section 4.3.2, are also stored in the h history stack. During state backtracking, these are undone to recover h to previous values.

The first representation of h had used *java.util.HashMap* instead of a bit-vector, and the changes had been managed by a model checker rather than storing Δ in a separate stack. From the fact that *java.util.HashMap* stores the *java.util.Entry*s in an array of a linked list, and each entry stores a mapping from a key, *SynchAddr* $\cup$ *Thread*, to value, *Addr*, the bit-vector representation was expected to utilize the space efficiently due to the extra space to maintain heavy weight Java class structures. Experimental results

confirm our expectation and the bit-vector with Δ stack representation outperforms the *java.util.HashMap* representation combined with model checker management.

4.3.2 The Listener Implementation

JPF supports a Listener interface that can be used to extend its functionality. The interface notifies low level events at the JPF java virtual machine level through preregistered callback functions. Types of these events are VM related events, such as *instructionExecuted*, *threadStarted*, and *objectLocked*, search related events, such as *searchAdvanced*, *searchBacktracked* and *propertyViolated*; those events are defined in `VMListener` and `SearchListener` respectively. JRF listener inherits `PropertyListenerAdapter`, which implements both `VMListener` and `SearchListener` interfaces. Callback functions inherited from `SearchListener` manage the stack structure to store Δ of h, and callbacks from `VMListener` manage h, as described in Section 4.3.1. The operations *acquire*, *release*, *invalidate* and asserting *norace* are performed as appropriate when execution of memory model related instructions occur.

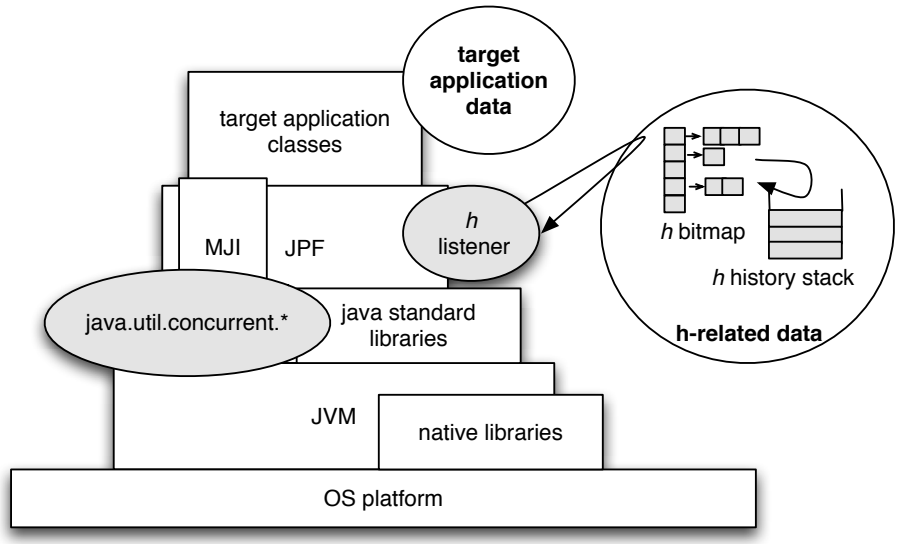
The hierarchical structure of the system is shown in Figure 4-9. The listener lies at the same level as other JPF code, outside the target model classes.

To implement the heuristic search algorithm described in Section 4.2.2, we use the following two auxiliary functions:

- $\mathit{written\_by} : \mathit{Addr} \times \mathit{Thread} \rightarrow \mathit{Boolean}$
- $\mathit{prior\_release} : \mathit{SynchAddr} \rightarrow \mathit{Boolean}$

$\mathit{written\_by}(m, t)$ returns $true$ when the most recent write to the given memory location m was done by thread t, and $\mathit{prior\_release}(m)$ returns $true$ when the given memory location m was released at least once previously.

In an earlier incarnation of our tool, we annotated the byte code with these assertions and then checked for assertion violations using standard JPF. The listener-based approach to extending JPF described above proved to be both more efficient and more flexible.



* shaded area represent our implementation to extend JPF

Figure 4-9. Overall system hierarchy of a JRF extension to JPF and their data position

4.3.3 Pruning the Search Space

JPF, like most model checkers, uses techniques to prune the search space. JPF gives each state a state number determined by the values of all variables in the static and dynamic area along with thread states such as lock and program counter information. If a state is encountered whose state number has already been seen, JPF does not explore that state's subtree again. However, states with the same state number may have different histories that result in different values of h, making this particular optimization unsound for data race detection. In the remainder of this section, we develop a condition, that when satisfied, does allow the search space to be safely pruned when states have the same JPF state number but different h values.

If two states have the same JPF state number, they will have identical subtrees in the search space, except possibly for the values of h. Let $future : State \rightarrow 2^{SynchAddr \cup Threads \cup Addr}$ be a function mapping states to the set of memory locations whose h values are read or written when processing the subtree. If, for

example, a transition between $s\prime$ to a child state $s\prime\prime$ in the subtree of s is a read of x by thread t, then both x and t are in *future*(s). If two states $s1$ and $s2$ have the same state number, then *future*$(s1)$ = *future*$(s2)$. Lemma 10 says that if two states have the same JPF state number, and one of them has an h function value that is a subset of the other for all of the memory locations and threads that will be accessed in their (common) subtree, then that subset property holds in the entire subtree.

Lemma 10. *Suppose $s1$ and $s2$ have the same state number and the value of h at these states, h^{s1} and h^{s2}, respectively satisfy $h^{s1}|_{future(s1)} \subseteq h^{s2}|_{future(s1)}$. Then for all corresponding states $s1\prime$ and $s2\prime$ in the subtrees of $s1$ and $s2$, $h^{s1\prime}|_{future(s1)} \subseteq h^{s2\prime}|_{future(s1)}$.*

Proof. The proof is by induction on the length of the path since all of the actions defined in Figure 4-1 preserve $\subseteq$. □

Theorem 4 justifies pruning of the search space.

Theorem 4. *Suppose that $s1$ is a state in the search tree whose subtree has been found to be free from data races. If $s2$ has the same state number as $s1$ and $h^{s1}|_{future(s1)} \subseteq h^{s2}|_{future(s1)}$, then there are no data races in the subtree of $s2$.*

Proof. follows immediately from lemma 10 and the definition of *norace*. □

4.3.4 Using the Model Java Interface

Java programs rely on a number of platform dependent functions, including thread implementation and low level synchronization primitives that are implemented in native code. The Model Java Interface (MJI) is provided to allow JPF to handle these situations: native code is executed by the host JVM and not model checked by JPF. Unfortunately, this means that the h-related data structures are no longer correct after executing native code. We extended `DefaultFieldFactory` class of JPF to implement

new `JRFFieldFactory` class to include the necessary calls to h manipulation code.[3] To enable complete coverage of Java features by JRF, it was also necessary to extend MJI to include some classes missing from the `java.util.concurrent.atomic` package in the JPF distribution, including all classes supporting atomic arrays. Many lock-free data structures use these classes [2, 3, 63, 64]. Except for finalizers, the current implementation of JRF correctly handles all Java language features related to the JMM.

4.3.5 Problems with State Backtracking

There were several issues with state backtracking that made the task of extending JPF less straightforward than one might expect. Static initializers complicate the management of h since the state backtracking scheme built into JPF does not reload classes or re-initialize their static fields, even when the state is backtracked to a point before the class was loaded. Thus, it is necessary to identify which memory locations are allocated in a static initializer. These locations, if they have not been updated, should be accessible by all threads, including those created later without causing a data race. We maintained another set of *Addr*, $h(static\_initializer)$, where $static\_initializer$ is a synthetic thread representing the class loader thread. This represents the locations corresponding to static variables that have not yet been updated outside the static initializer for the class. In addition to the *release*$(parent\_thread, child\_thread)$, the thread start operation should also perform *release*$(static\_initializer, child\_thread)$.

Another complication in the listener implementation is the unpredictable garbage collection by JPF. When an object is garbage collected during state search, it is no longer in use along that path and might be reused for a new object. This problematic situation occurs when the unique key for the object is no longer unique. The original

[3] Before JPF version 4, it had been necessary to manually modify their Model Java Interfaces to reflect the way the target functions update h. This had been a painstaking task and hard to manipulate. The new `Factory` structure adopted after JPF version 5 allows maximum configurability including extendable bytecode factory and field factory.

object, which is still used in other stored paths, shares its key with a new object, resulting in incorrect sharing of h. State backtracking will use the h of the new object unless it is properly restored to the value before garbage collection. The h history stack stores the necessary garbage collection and reallocation information.

4.3.6 Untracked Variables

JRF offers the option to mark individual non-volatile locations as untracked, so that reads and writes of these variables do not have any effect on h and $norace(x, t)$ is not checked. In other words, we only require $norace(x, t)$ for $x \in \mathit{Addr} \backslash untr$ where $untr$ is the set of variables marked as untracked. The definition of h_0 is changed to

$$h_0 = \lambda z.\text{if } z = main \text{ then } static(P) \backslash untr \text{ else } \phi \tag{4–6}$$

and we modify h_{n+1} in Figure 4-1 as shown in Figure 4-10.

The following lemma justifies the use of untracked variables.

Lemma 11. *Consider an execution E_n and let h_n be the summary function with all $x \in \mathit{Addr}$ tracked, and h_n^{untr} be the summary function for the same execution where the the variables in $untr \subseteq \mathit{Addr}$ are untracked. Then*

$$\forall u \in (\mathit{SynchAddr} \cup \mathit{Threads}) : h_n(u) \backslash untr = h_n^{untr}(u). \tag{4–7}$$

action a_n by thread t	h_{n+1}
write a non-volatile field $x \notin untr$	$\mathit{invalidate}(t, x)\ h_n$
write a non-volatile field $x \in untr$	h_n
instantiate an object containing non-volatile fields $fields$ and volatile fields $volatiles$	$\mathit{new}(t, fields \backslash untr, volatiles)\ h_n$

Figure 4-10. Modified defintion of h_{n+1} with untracked variables. Omitted actions are the same as shown in Figure 4-1.

Proof. The proof is by induction on n. The base case follows immediately from the defintions of h_0 given in Equations 4–1 and 4–6. The case for read non-volatile follows immediately from the induction hypothesis. The rules for write volatile, read volatile, lock, unlock, start, join, and isAlive all have the form $h_{n+1} = h_n[u \mapsto h_n(u\prime) \cup h_n(u\prime\prime)]$ for some u, $u\prime$, and $u\prime\prime$, so we can give a single proof for all of them.

$$
\begin{aligned}
& h_{n+1}(u)\backslash untr \\
&\quad = \quad (h_n(u\prime) \cup h_n(u\prime\prime))\backslash untr \\
&\quad = \quad h_n(u\prime)\backslash untr \cup h_n(u\prime\prime)\backslash untr \\
&\quad = \quad \text{by the induction hypothesis} \\
&\qquad\quad h_n^{untr}(u\prime) \cup h_n^{untr}(u\prime\prime) \\
&\quad = \quad h_{n+1}^{untr}(u)
\end{aligned}
$$

For *invalidate*(t, x) where $x \in untr$,

$$
\begin{aligned}
& h_{n+1}(u)\backslash untr \\
&\quad = \quad (\text{if } (t = u) \text{ then } h_n(u) \text{ else } h_n(u)\backslash\{x\})\backslash untr \\
&\quad = \quad \text{if } (t = u) \text{ then } h_n(u)\backslash untr \text{ else } h_n(u)\backslash\{x\}\backslash untr \\
&\quad = \quad \text{if } (t = u) \text{ then } h_n^{untr}(u) \text{ else } h_n(u)\backslash\{x\}\backslash untr \\
&\quad = \quad \text{if } (t = u) \text{ then } h_n^{untr}(u) \text{ else } h_n(u)\backslash untr \\
&\quad = \quad \text{if } (t = u) \text{ then } h_n^{untr}(u) \text{ else } h_n^{untr}(u) \\
&\quad = \quad h_n^{untr}(u) = h_{n+1}^{untr}(u)
\end{aligned}
$$

For *invalidate*(t, x) where $x \notin untr$,

$$
\begin{aligned}
& h_{n+1}(u) \backslash untr \\
= \;& (\text{if } (t = u) \text{ then } h_n(u) \text{ else } h_n(u) \backslash \{x\}) \backslash untr \\
= \;& \text{if } (t = u) \text{ then } h_n(u) \backslash untr \text{ else } h_n(u) \backslash \{x\} \backslash untr \\
= \;& \text{if } (t = u) \text{ then } h_n^{untr}(u) \text{ else } h_n(u) \backslash \{x\} \backslash untr \\
= \;& \text{if } (t = u) \text{ then } h_n^{untr}(u) \text{ else } h_n^{untr}(u) \backslash \{x\} \\
= \;& h_{n+1}^{untr}(u)
\end{aligned}
$$

The final case is *new*.

$$
\begin{aligned}
& h_{n+1}(u) \backslash untr \\
= \;& (\text{if } (t = u) \text{ then } h_n(t) \cup \textit{fields} \\
& \quad \text{else if } (u \in volatiles) \text{ then} \{\} \\
& \quad \text{else } h_n(u)) \backslash untr \\
= \;& (\text{if } (t = u) \text{ then } (h_n(t) \backslash untr) \cup (\textit{fields} \backslash untr) \\
& \quad \text{else if } (u \in volatiles) \text{ then} \{\} \\
& \quad \text{else } h_n(u) \backslash untr) \\
= \;& (\text{if } (t = u) \text{ then } h_n^{untr}(t) \cup (\textit{fields} \backslash untr) \\
& \quad \text{else if } (u \in volatiles) \text{ then} \{\} \\
& \quad \text{else } h_n^{untr}(u)) \\
= \;& \textit{new}(t, \textit{fields} \backslash untr, \textit{volatiles})\ h_n^{untr} \\
= \;& h_{n+1}^{untr}
\end{aligned}
$$

□

A class can be marked as trusted, which will result in all of its private non-volatiles becoming untracked. A package can be marked trusted, which will result in all package-private non-volatile variables to become untracked.

There are two different motivations for not tracking variables. The first allows JRF to be used with programs that contain so-called benign races. The JMM provides a strong guarantee, namely sequential consistency for properly synchronized programs. It also constrains programs with data races in order to provide some minimal security guarantees. These guarantees include type safety and the guarantee that there are no ”out-of-thin-air” values. As a result, it is possible, in principle, for programmers to write and reason about the correctness of programs that contain data races. If the presence of a race cannot cause a program to violate its specification, even when the non-SC behavior allowed by the JMM is exhibited, it is considered to be benign. Occasionally benign races are allowed in programs for performance reasons. Reasoning about programs with races is quite difficult and should be considered to be a job for experts only. Most programmers should write race-free programs. JRF does not support reasoning about programs with races in the sense that one cannot construct a program with data races, submit it to JRF, and expect the model checker to have explored the states that only occur in sequentially inconsistent executions allowed by the JMM. Marking the location involved in a benign race as untracked allows the rest of the program to be analyzed for races. It is the programmer/JRF user’s responsibility to ensure that when a race is ignored by marking a field untracked, it is indeed benign and that its effects are sufficiently encapsulated that the rest of the model checking will be sound.

The second motivation for not tracking certain variables is to improve the scalability of JRF. Marking classes or packages as trusted is a convenient way to reduce the time and memory requirements by not checking classes that we are willing to assume do not have races, or only benign races. JRF does, however, continue to track happens-before

edges involving volatile fields and locks defined in trusted classes and the non-volatile fields accessible outside of trusted classes (or packages) in order to continue to precisely detect data races in or caused by other classes. For example, a common way to safely publish objects is to make them available to other threads by passing them through a data structure, such as a queue, whose implementation is provided in the `java.util.concurrent` package where the insertion of an object happens-before removal of the object. If the package is marked as trusted, while the happens-before edges related to inserting and removing objects will be preserved, no checking will be done on the internal non-volatile variables in the class unless they are visible outside the trusted code. By marking the classes in the standard Java release as trusted, a significant reduction in the time and space requirements to model check an application class can be achieved. JRF sets `java.lang`, `java.util`, `java.io`, `JPF_INITIALIZER`, `sun.misc`, and `gov.nasa.jpf` as default untracked packages.

4.3.7 Lazy Representation of Array Elements

In the representation of h described in Section 4.3.1, every memory location requires an entry in the h table. This also applies to arrays, which require an entry for each array element.[4] JRF, however, uses a single h entry to abstractly represent the entire array until such time that an individual element is updated and obtains a different value for h. This would occur when the abstract array location is not in the h set of the current thread or is included in some other h set when updated. At that point, an additional entry in the h table for that element is allocated. The abstract location is still used for the remaining elements. The lazy representation of array elements saves *acquire* and *release* time as well as space. For instance, an integer array of

[4] There is no way in Java to make the individual elements of a normal array have volatile semantics. Marking the array declaration with the volatile keyword gives volatile semantics to the reference to the array, but not the individual elements. This is a common misunderstanding of the JMM and a frequent source of errors.

```
Algorithm non-volatile read (x : memory location)
  t: current thread
  if x ∈ untr then
    return;
  else if x is i_th element of an array a then
    if a[i] has h representation then
      norace (a[i], t);
    else /* use abstraction */
      norace (a, t);
  else /* not an array */
    norace (x, t);

Algorithm non-volatile write (x : memory location)
  t: current thread
  if x ∈ untr then
    return;
  else if x is i_th element of an array a then
    norace (a, t); /* check accessibility first */
    if a[i] has h representation then
      invalidate (t, a[i]);
    else /* use abstraction */
      if ∀v ∈ SynchAddr ∪ Threads with v ≠ t, a ∉ h(t) then
        return; /*invalidate (t, a) is redundant */
      else
        new (t, a[i], ∅); /*allocate h for a[i] */
  else /* not an array */
    invalidate (t, x)
```

Figure 4-11. Modified algorithm for non-volatile read and write with lazy representation of array elements and untracked variables

size n requires $n + 1$ different h entries without lazy representation, but JRF initially allocates only one h entry. All h accesses for array elements, such as *norace* and *invalidate*, use this value until additional entries are created. Figure 4-11 summarizes the h manipulation for non-volatile read and write operations.

The representation of strings in Java provides strong motivation for lazy representation of array elements. A string is represented in the String class using a char array.

Although the `java.lang.String` class is immutable[5] , string concatenation with the "+" operator, utilizes the `java.lang.StringBuilder` class, which is not immutable. The char array used to represent the contents could be changed by calling their public methods, such as append and replace. If we had a print message of length 100 concatenated using "+", JRF without lazy array representation, 100 entries would be needed in the h table. A write of a volatile variable v by the current thread would add those 100 memory locations to the h set of v, and later those would be propagated to other threads that read v even though they neither read nor write the message. In practice, without lazy array representation, strings become a significant consumer of time and space in JRF.

4.3.8 Threadlocal Optimization

If we have knowledge of which memory locations are not shared, this can be used to reduce the overhead. In particular, since a data race on a non-volatile variable by definition involves two threads, a non-static variable that is only accessed by the thread that instantiated it can never be involved in a data race. Static variables are accessed by a class loader thread during static initialization. They are then accessed by at most one application thread, they can also never be involved in a data race. This is because the JMM guarantees that static initialization is guaranteed to happen-before the first access of a variable, say x, by any application thread, say t, thus establishing $norace(t, x)$ for that access. Then, $norace(t, x)$ will only be falsified by an invalidate operation caused by another thread writing x, which does not occur since t is the only application thread to access x. As a result, we do not need to maintain h data or check $norace$ for non-volatiles that are only accessed by a single thread. If our knowledge of sharing is imprecise, it may lead to unnecessary overhead (if we do not recognize a thread local

[5] The `String` class is usually considered to be immutable, although technically it is not. The hash code initialization is performed lazily with a benign data race that relies on the JMM final field semantics for correctness.

variable and treat it as shared) or a missed data race (if we treat a shared variable as thread local.)

The situation with volatiles and locks is less obvious, but the outcome is similar. If we know that a particular variable or lock x is only accessed by a single application thread, then $norace(x\prime, t\prime)$ does not depend on $h(x)$.

Lemma 12. *Suppose that variable or lock x is accessed by only one application thread t. Then $h(x) \subseteq h(t)$.*

Proof. The property is true initially since $h(x) = \phi$. It is easy to see that it is maintained by all actions in Figure 4-1. □

Theorem 5. *If a volatile variable or lock x is only accessed during an execution by thread t, then for any thread $t\prime$, $h(t\prime)$ does not depend on $h(x)$.*

Proof. The result is true initially. By lemma 12, it is maintained when $t\prime = t$. For $t\prime \neq t$, we note that the only such dependency would have to be introduced by *acquire*$(t\prime, x)$. However, this is a result of $t\prime$ reading x, which does not happen. □

If knowledge of sharing of locks and volatiles is imprecise, it may lead to unnecessary overhead (if a local variable is considered shared) or a possible false alarm (if a shared variable is considered to be local) but will not result in a data race being missed.

Applying the thread local optimization requires somehow determining which variables are shared. Possibilities include running JPF (a slightly modified version that maintains sharing information), performing some sort of static analysis, or allowing the user to specify thread local variables and verifying the choices.

4.3.9 Benign Race

In most cases, programmers should write programs without data races. However, since the JMM constrains programs with races, it is, in principle, possible to prove that

a data race is benign and to allow data races is sometimes desirable for performance reasons.[6]

JRF provides the mechanism to ignore certain data races by identifying them as benign. The notion of a benign data race is defined by the circumstances. In some approaches a data race is considered benign if the executions are still sequentially consistent, or, in some appropriate sense, equivalent to sequentially consistent executions. This has the advantage that the specification is implicit, but may over-constrain the program. In a model checker, it is natural to consider races to be benign with respect to a specification provided in the form of a checkable property. Thus, a benign race may allow executions that are not equivalent to a sequentially consistent execution, provided they still satisfy the given property.

We do not limit the semantic of a benign race in our tool, instead, we provide the way to mark specific program code as a benign race. Conceptually, a benign race and an untracked variable have same effect on the reported race kind; to ignore the relevant race. The benign race is related to a specific program point, whereas an untracked variable is about a memory location. Another difference is the resource savings. An untracked variable consumes neither the space for h entry nor the time to execute *norace* and *invalidate*. On the other hand, a benign race requires both, and has no resource saving. Users of JRF would choose a benign race for an intentional race on variables that is also accessible from other classes or packages. When a variable is declared class private or immutable, as `hashcode` in `String` class, use of an untracked variable better utilizes the resources and this has no risk of being

[6] Perhaps the most common example of a benign race pattern in Java programs is lazy initialization of fields accessed without synchronization, as in `T getX(){if (x==null){return f(....);} else return x;}` where `x` is not volatile and `f` only depends on final fields and constants. This pattern is found in the initialization of the hashcode value in the java.lang.String class.

involved in a race through third-party codes. The default JRF configuration includes two benign races: `java/util/concurrent/locks/AbstractOwnableSynchronizer.java:56` and `java/lang/String.java:652`. Our experiment in Section 6.5 found one benign race which is a redundant update on shared field `UNIVERSAL_DEBUG` in `montecarlo` benchmark.

CHAPTER 5
EXTENSION

5.1 Eliminating Data Race Using Counterexample

Although the original JPF provides the sequence of statements (the counterexample path) that leads to a property violation, data race in our tool, it requires a tedious manual effort to parse the information hidden in the counterexample to find the interleaving sequence of the threads and the reason why the data race occurred. For instance, the counterexample path for a slightly modified version of `Simple` class in Chapter 1 is shown in Figure 5-1. Both `x` and `done` are involved in races. The omitted output gives similar results for additional (8 more) detected races. To understand what caused the detected race, we need to decode the counterexample path given as `"trace #1"`. Clearly, this is a tedious exercise, even for this simple program where the length of the counterexample path is only six. The path length may be several hundred in realistic examples. In contrast, Figure 5-2 shows the output of the analysis produced by JRF. For each unique race found, the race source statement, the race manifest statement, and suggestions for code modifications that will eliminate that race are given. Note that the tool recognized that marking `done` as volatile is sufficient to eliminate the race on `x` also.

`Simple` class has max depth 6, but our experimental results (Chapter 6) easily grew to hundreds and even thousands in max depth. Decoding all transitions to trace happens-before relations in this counterexample path is a tedious and error-prone procedure. Furthermore, the trace does not include h information from execution paths other than the counterexample. In most cases, those execution paths without errors set a good example of a race free pattern.

We provide a counterexample analysis and an explanation of how to fix the race based on the h information from both a counterexample path and other race free execution paths.

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

Figure 5-1. Partial output from JRF for modified `Simple` class. Eight similar traces have been omitted.

5.1.1 Analysis

Since a data race is defined to be the lack of a happens-before edge, we can leverage the information in h to explain why there is no happens-before edge between the statement that caused the data race, which we call the *source statement*, and the

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

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

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

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

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

Figure 5-2. JRF output which explains the source of the race and suggests how to eliminate it

manifest statement where the data race has occurred, i.e., where `assert norace` has failed. This information can be used to provide suggestions for ways to eliminate the data race by creating a happens-before relationship between those statements.

To improve the quality of the suggestions, we also maintain the acquiring history, $AcquireHis : Addr \Rightarrow 2^{(SynchAddr \cup Threads) \times Threads}$. For a memory location m, $(v, t) \in AcquireHis(m)$ means that, thus far at some point in the computation, thread t performed an operation on v that resulted in m being added to $h(t)$. The actions by thread t that would result in (v, t) being added to $AcquireHis(m)$ for some m could be reading v, locking v, or joining v, where v is a volatile field, lock, or thread, respectively. In contrast to the summary function h, which only applies to a particular path, the $AcquireHis$ is cumulative and contains information from all explored paths.

JRF currently provides four types of suggestions:

- Change the variable that participated in the data race to `volatile` or to use a `java.util.concurrent.atomic` array if the variable is an array element.
- Move the statement that caused the data race before the statement that is the source of an existing happens-before edge.

- Acquire a particular lock (either intrinsic or extrinsic) before the manifesting statement.
- Perform the same type of acquire operation on an agent memory location as has been accomplished earlier by some thread.

5.1.2 Algorithms

This section describes the algorithms used for generating suggestions. When JRF detects a data race, the race eliminator uses the counterexample path (*pathInstr*), the h information for the counterexample path (*pathHB*), the acquire history (*AcquireHis*), the position of the manifest statement (*raceManifestIndex*), and the position of the source statement (*raceSourceIndex*) on the counterexample path. The provided suggestions only eliminate the data race on the current execution path and it is possible that after the user follows the suggestion and makes the suggested changes, JRF will find a different data race. Thus the tool will incrementally guide the user until no more data races can be found.

Change a (non-array element) variable to volatile or implement with an atomic class. Due to the semantics of volatile variables in Java, changing a variable involved in a race to volatile is always sufficient to eliminate a data race involving that variable. Since volatile variables inhibit compiler optimizations and accesses to volatiles incur

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

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

Figure 5-3. Suggest change to volatile or atomic array.

runtime overhead, the trivial way of eliminating races by making everything volatile is undesirable.

Changing a variable to volatile likely to be the most appropriate in situations where this variable is being used for publication (i.e. making the reference to a new object instance visible to other threads). Unsafe publication [62] is a common error in concurrent Java programs written by programmers without a good understanding of the JMM and can lead to a situation where another thread sees a partially initialized object.

Another guaranteed solution is to replace the variable with a final[1] reference to an instance of the atomic class corresponding to the variable's type in the `java.util.concurrent.atomic` package. For example, replace an `int` variable with an instance of the java.util.concurrent.atomic.Atom class. These classes are less convenient than volatiles because they must be accessed with `get` and `set` methods and are the better choice only if the lock-free atomic update methods they provide are needed. These atomic update methods include `compareAndSet`, which is frequently used in lock-free algorithms, and where appropriate for the type, methods such as `getAndAdd`, `addAndGet`, `getAndIncrement`, etc.

Change an array to an atomic array. Atomic arrays for various element types are provided int the `java.util.concurrent.atomic` package. They provide volatile semantics for array elements and thus this change is always sufficient to eliminate data races involving array elements. Arrays are objects in Java and a frequent error is to mark an array reference volatile without realizing that this does not provide volatile semantics for the accesses to the elements.

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

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

as shown in the second path of the state space in Figure 5-7. The algorithm in Figure 5-5 first calls the *findHBEdges* algorithm in Figure 5-4 to compute all the happens-before edges that result from synchronization actions on the counterexample path. A happens-before edge is a pair of instructions where the release instruction is the source vertex and the matching acquire instruction is the destination vertex. Instructions are identified by their positions on the counterexample path. After having

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

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

Figure 5-4. Find the set of happens-before edges through synchronization actions on path *pathInstr*.

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

   Set of integer pairs HBEdges ←findHBEdges(pathInstr)
   foreach pair p =(index1,index2) ∈ HBEdges s.t.
      (raceSourceIndex,raceManifestIndex) intersects p
      AND index1 < raceSourceIndex do
      if same thread executed pathInstr(index1) and pathInstr(raceSourceIndex)
       AND
       same thread executed pathInstr(index2) and pathInstr(raceManifestIndex) then
        print "Move instruction at raceSourceIndex before index1"
```

Figure 5-5. Suggest moving instruction

```
boolean goFlag;
volatile Data publish;

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

Figure 5-6. Via `goFlag` Thread 1 notifies Thread 2 when object `publish` is ready to be used.

all pairs representing the happens-before edges on the counterexample path, the Figure 5-5 algorithm compares the (source statement, manifest statement) pair (*raceSourceIndex*,*raceManifestIndex*) with all other pairs from the set of happens-before edges. If moving the source statement before a statement that is the source of a happens-before edge and has been executed before the source statement creates a happens-before edge between the source statement and the manifest statement, the move is suggested.

As an example, consider the program in Figure 5-6 where goFlag and publish are shared variables. Since `publish` is a reference, the object to which it refers can also be accessed by both threads.[2] Thread 1 creates an object at line `s1` that is currently accessible only to itself. Then it publishes the object by storing the reference in a shared variable `publish` at line `s2`. The state of the object is updated at line `s3` and the shared variable `goFlag` is set to true at line `s4` declaring that the object descriptor has been set and can safely be read by other threads. Thread 2 checks whether `publish` is not null at

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

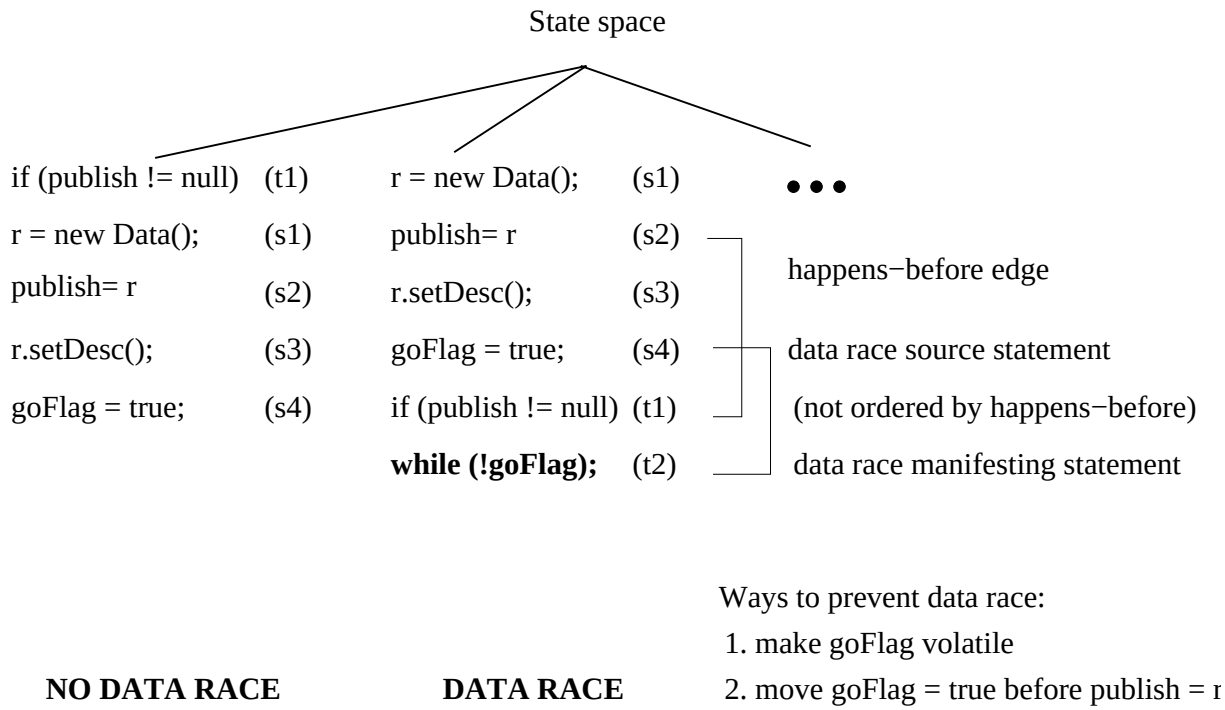


Figure 5-7. Part of the state space showing a data race free path and a path with a data race.

line `t1` and, if so, spins until the global flag becomes true at line `t2` and reads the object descriptor in line `t3`.

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

Figure 5-7 shows part of the state space of the example. As long as JRF does not run out of memory or the user out of patience, it can explore all possible paths in the state space. The first path does not exhibit a data race. When JRF executes `t2` on the 2nd path that is explored, a data race is manifested (e.g., *assert norace* fails),

the currently explored path is reported to the user as a counterexample, and JRF terminates. One way of eliminating this particular data race is making the global flag, `goFlag`, volatile, thus creating a happens-before edge between `s4` and `t2`. Another way is to move `s4` before `s2` and create a happens-before edge between `s4` and `t2`, which follows from the transitive property of the happens-before relationship: `s4` $\xrightarrow{hb}$ `s2`, `s2` $\xrightarrow{hb}$ `t1`, and `s2` $\xrightarrow{hb}$ `t1` implies `s4` $\xrightarrow{hb}$ `t2`. Once this change is made, a new data race between the write at `s3` and the read at `t3` is exhibited. This can be eliminated by moving `s3` before `s2`. At this point the example is both correct (the assertion will never fail) and contains no data races.

Use a synchronized block. Using consistent locking is one way of creating happens-before edges between accesses to shared data. Using synchronized blocks is one way of implementing locking in Java. Figure 5-8 finds all the locks that are released after the source statement and before the manifest statement and suggests protecting the manifest statement with these locks by referring to the specific source lines that perform the locking.

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

   for index from size(pathInstr) to raceSourceIndex+1 do
      if pathInstr(index) is a MONITOREXIT instruction OR
       RETURN instruction of a synchronized method then
       let loc denote the source line for pathInstr(index)
       syncLoc ←syncLoc ∪ {loc}
   foreach source line loc ∈ syncLoc do
      print "Put instruction  pathInstr(raceManifestIndex)
            in synchronized block as in line loc"
```

Figure 5-8. Suggest a synchronized block.

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

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

Figure 5-9. Thread 2 need to synchronize on `lock` to access `data`

Figure 5-9 shows an example in which Thread 1 acquires a lock before accessing the shared data `data` whereas Thread 2 does not acquire any lock before accessing `data`. Figure 5-10 shows the counterexample path that manifests the data race on `data`. At `s3` Thread 1 unlocks `lock` before the manifest statement `t1`. The data race can be eliminated by making Thread 2 acquire `lock` before `t1`.

Change other memory locations to volatile or use atomic arrays.

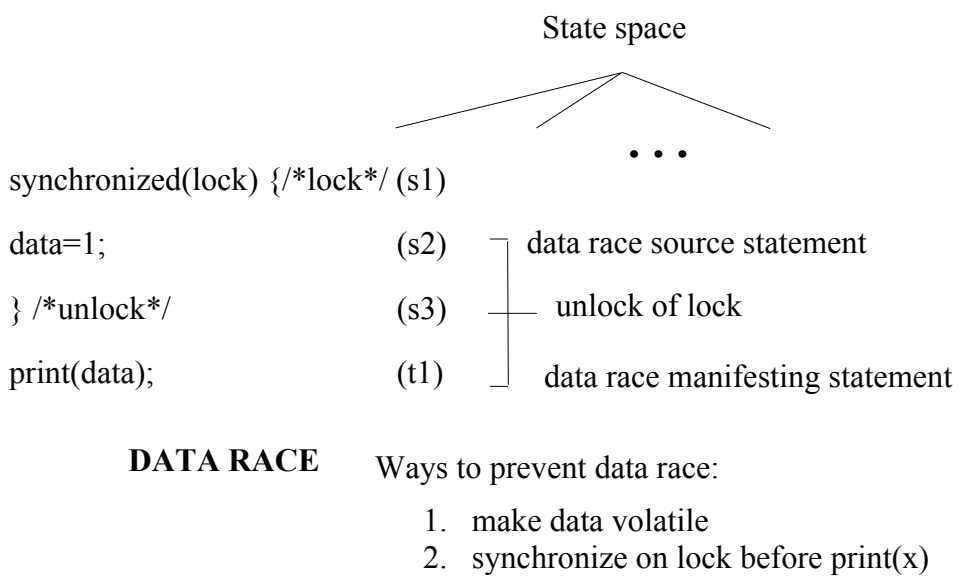


Figure 5-10. Part of the state space with the unlock in between the source statement and the manifest statement

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

If $s1$ and $s2$ are the write and read of a variable v, respectively, then changing v to volatile creates a happen before edge between $s1$ and $s2$. Figure 5-11 shows the algorithm for checking the happens-before relation and the algorithm for this type of suggestion using it.

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

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

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

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

Figure 5-11. Suggest changing a different memory locations to volatile

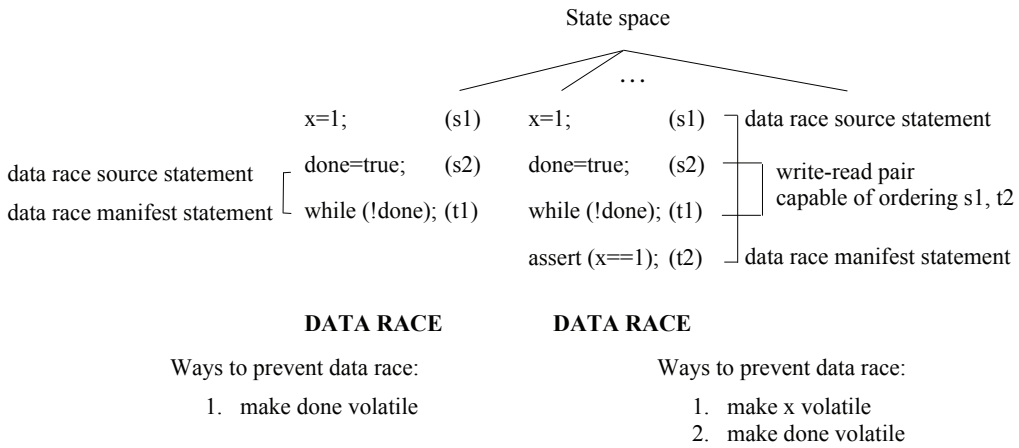


Figure 5-12. Part of the state space that changing `done` to volatile can eliminate a race on `x`

Consider the modified `Simple` class with two threads sharing two variables: `done` and `x`. If JRF is configured with $threshold > 1$, it is possible to find a counterexample that shows a data race manifested in statement `t2` as shown in Figure 5-12. It turns out that between the source statement (`s1`) and the manifest statement (`t2`), there is a write of `done` followed by read of `done`. Since `s1` and `s2` are executed by Thread 1 and `t1` and `t2` are executed by Thread 2, changing `done` to volatile creates a happens-before edge between `s1` and `t2` and eliminates the data race.

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

Perform the same synchronization operation. JRF keeps track of the acquiring history to allow determination of how happens-before edges were created for non-racy accesses to a memory location. Formally, we define the acquiring history as a function:

$AcquireHis : Addr \Rightarrow 2^{(SynchAddr \cup Threads) \times Threads}$. For a memory location m, $(v, t) \in AcquireHis(m)$ means that at some point in the computation so far, thread t performed an operation on v that resulted in m being added to $h(t)$. The actions by thread t that would result in (v, t) being added to $AcquireHis(m)$, for some m could be

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

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

Figure 5-13. Suggest performing an acquire operation that can add the data race memory location to h of the manifesting thread.

reading v, locking v, or joining v, where v is a volatile field, lock, or thread, respectively. In contrast to the summary function h, which only applies to a particular path, the $AcquireHis$ is cumulative and contains information from all the explored paths.

Figure 5-13 finds out how previous accesses to the data race memory location (m) have been ordered by the happens-before relation and suggests performing the same acquire operation. Three possible acquire operation choices are `read`, `lock`, and `join` according to the type of memory location. If the memory location is a field, then it must be volatile and the corresponding acquiring operation is to read it. If the memory location is a lock, the acquire operation is to lock it. When the memory location is a reference to thread, joining it serves as an acquire.

The example in Figure 5-14 motivates the use of the acquiring history. In execution sequence `(r1, r2, s1, s2, t1)` as shown in Figure 5-15, there is a data race between `r1`, a write of `x` by Thread 1, and `t1`, the read of `x` by Thread 3. It should be noted that Thread 2 also performs a read of `x` but it does not result in a data race. The reason is Thread 2 reads volatile `done` before reading `x` and this generates a happens-before edge between the write of `x` by Thread 1 and the read of `x` by Thread 2. The acquiring history

```
int x;
volatile boolean done=false;

     Thread 1              Thread 2            Thread 3
==============   ==============   =========
 r1: x = 1;              s1: if ( done )          t1: print(x);
 r2: done = true;      s2:    assert (x==1);
```

Figure 5-14. Acquiring history of Thread 2 shows Thread 3 can get race free access on `x` by reading `done`.

stores this information and JRF uses it to suggest that Thread 3 reads volatile `done` before reading x to eliminate the data race.

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

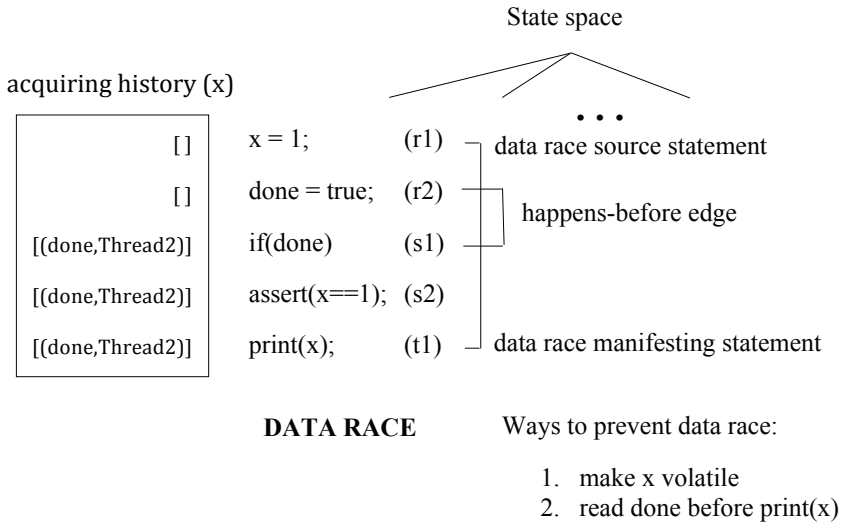


Figure 5-15. Part of the state space with the acquiring history that guides how to eliminate the race

5.1.3 Theoretical Results

In this section, we prove that modifying the program according to the suggestions generated by our tool does not remove any happens-before edges that existed before the modification. Specifically, Theorem 6 and Theorem 7 show that this is the case for all execution paths considering changing a non-volatile to volatile and that adding a new synchronization action via putting in a synchronized block or following an acquiring history-based suggestion, respectively. Theorem 10 shows a similar result for the move suggestion only on the counter-example path under certain conditions.

Theorem 6. *Changing a non-volatile variable to a volatile variable does not remove any of the existing happens-before edges that result from synchronization actions on any of the execution paths, but it may introduce additional happens-before edges.*

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

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

Theorem 7. *Changing a program by adding a synchronization action (joining a thread, acquiring a lock, or reading a volatile variable) that involves an existing memory location does not remove any of the existing happens-before edges that result from synchronization actions or the program order on any of the execution paths.*

Proof. An existing happens-before edge that results from synchronization actions can be removed only by changing the source or the destination statement of the happens-before edge. Adding a synchronization action does not change such a statement. Also, an existing happens-before edge that result from program order does not change as a result of adding a synchronization action because happens-before

is a transitive relation and all such existing happens-before edges would be preserved due to transitivity. □

Theorem 8. *Moving an instruction that accesses a non-volatile variable does not remove any of the existing happens-before edges that result from synchronization actions on any of the execution paths.*

Proof. Same reasoning as in proof of Theorem 6. □

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

Theorem 9. *Moving a data race source instruction,* `write x` *at step h, before an instruction that is the source of a happens-before edge, represented by $[d, i]$ where $d < h < i$, does not introduce a new data race that involves the moved statement on the same counter-example path if and only if the following conditions hold:*

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

  Proof. After the move, the old data source instruction (at step h) would be at step $d - 1$ and would be ordered with the instruction at step g by happens-before. As a concrete example, consider the sample program in Figure 5-16 and the corresponding counter-example path in Figure 5-17A. □

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

  Proof. After the move, the old data source instruction (at step h) would be at step $d - 1$. The instruction at step a and the instruction at step $d - 1$ would be ordered by happens-before. As a concrete example, consider the sample program in Figure 5-16 and the corresponding counter-example path in Figure 5-17B. □

Theorem 10. *Moving a data race source instruction,* `write x` *at step h, before an instruction that is the source of a happens-before edge, represented by $[d, i]$ where*

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

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

Figure 5-16. Via `goFlag` Thread 1 notifies Thread 2 when object `publish` is ready to be used. Thread 3 can also notify Thread 2 by checking a field of the object pointed to by `publish`.

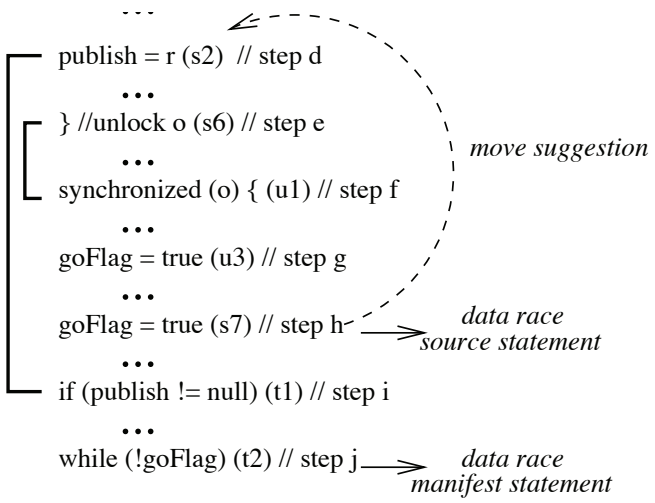
$d < h < i$, does not involve any new data races if and only if conditions[3] 1 and 2 in Lemma 9 hold.

Proof. Follows from Lemma 8 and Lemma 9. □

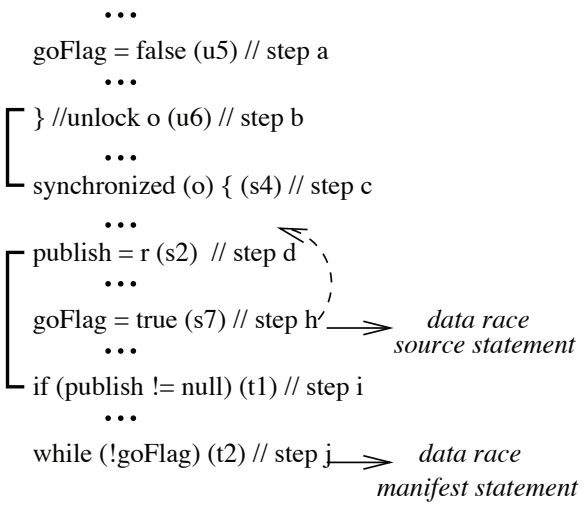
5.1.4 Implementation

Though this extension provides a useful explanation of counterexample, adding more data structures and algorithms to JRF will make the state space to explode earlier. Given that the more counterexample paths, the better suggestions, it is more desirable to use the `multirace` option in the JRF-E extension. Based on our experience 6, one line of a racy code easily produces thousands of counterexample traces. Storing all those traces for analysis is a very memory-consuming operation. We address this problem using standalone eliminator. The idea is to store a race with its counterexample

[3] Checking of the two conditions in Lemma 9 has been implemented in JRF and the suggestion is made only if the conditions hold. For brevity, the algorithm in Figure 5-5 does not include checking for these conditions.



A Instantiation of Condition 1



B instantiation of Condition 2

Figure 5-17. Instantiation of Conditions in Lemma 9 based on sample program in Figure 5-16. Happens-before edges formed by synchronization actions are shown by lines connecting the matching release and acquire instructions.

path in JRF and analyze it later using this stored information. This approach is close to the JPF extension *jpf-trace-server* that provides storing, querying, and analysis of execution trace. Unfortunately, this is an on-going project and not available to JRF at

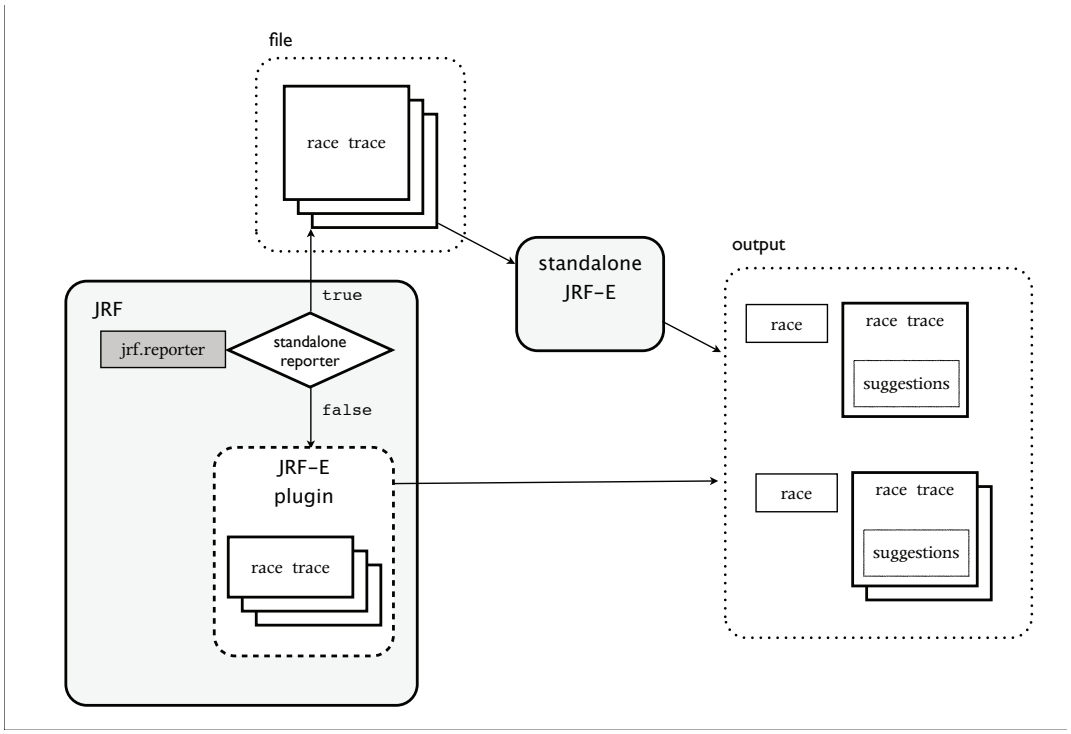


Figure 5-18. The JRF-E execution model: JRF plugin or standalone

this time. Moreover, JRF needs more information in addition to the counterexample trace to analyze a race, such as memory path elements, acquiring history, and h. We store all and only necessary information at a race detection to a file and later standalone JRF-E loads and analyzes it to produce suggestions. In addition, when we only look for one race, it is easier to use to detect and analyze that at one pass. This is configurable in `jrf.jpf` using `jrf.reporter.standalone`.[4] Figure 5-18 shows the structure of `jrf.reporter` and JRF-E extension. Each counterexample trace is directed to file without any analysis when JRF detected it in standalone mode. JRF-E will analyze traces and categorize them into different races, bookkeeping the suggestions accordingly. Additional information, such as happens-before edges in the counterexample path and acquiring history, are available to explain the race better.

[4] The GUI version of `jrf.reporter` will provide a user-friendly environment to JRF-E.

5.2 Modular Verification of the Correctness of Race-Free Preconditions

The foreign code in a concurrent program complicates the verification of the race freedom. The library programmer's specific design decision to avoid a data race is delivered as an inline comment or a separate design document to the application programmer that makes use of it. Naturally, this information is not always available to the user due to the fact that most foreign codes are distributed in an intermediate format such as Java bytecode, rather than an original source format with comments. Under these circumstances, it is dangerous to rely solely on the application programmer's ability to use libraries consistently for a data race-free guarantee. From library programmers' perspective, they are also in need of a mechanism to specify design decisions that is safely deliverable to users and automatically checkable in a concrete application environment. The annotated preconditions for the library routines would play this role. The library programmer annotates the constraints of each library routine as race-free preconditions and the application program analysis will automatically check if these preconditions are satisfied at all invocations of the routine. When all preconditions are satisfied at all invocations of a library routine, there is no need to worry about a data race inside the library code used in an application. In this way, we can verify the race freedom of the entire system using modular race checking of individual libraries by assume-guarantee reasoning. The preconditions can be either a prior locking requirement or a current thread's h inclusion requirement.

For example, assume `UnboundedQueue` [1] and `DisBarrier` [1] in Figures 5-19 and 5-20 are foreign libraries, and an application `FairMessage` [5] in Figure 5-21 uses these.

[5] This code is slightly modified from the junit test driver for `UnboundedQueue` in [1] to include a call to `DisBarrier`

```
public class UnboundedQueue {
   private static final int EMPTY= java.lang.Integer.MIN_VALUE;
   public final ReentrantLock enqLock = new ReentrantLock(), deqLock = new ReentrantLock();
   Node head, tail;

   public UnboundedQueue() {
      head = new Node(EMPTY);
      tail = head;
   }

   public int deq() throws EmptyException {
      int result;
      deqLock.lock();
      try {
         if (head.next == null) throw new EmptyException();
         result = head.next.value;
         head = head.next;
      } finally { deqLock.unlock(); }
      return result;
   }

   public void enq(int x) {
      if (x == EMPTY) throw new NullPointerException();
      enqLock.lock();
      try {
         Node e = new Node(x);
         tail.next = e;
         tail = e;
      } finally { enqLock.unlock(); }
   }

   public int size() { // requires: lock enqLock and deqLock before calling
      int i=(head==tail?0:1);
      for (Node tmp=head.next; tmp != null && tmp != tail ; tmp=tmp.next, ++i);
      return i;
   }

   protected class Node {
      final int value;
      volatile Node next;
      Node(int x) {
         value = x; next = null;
      }
   }
}
```

Figure 5-19. UnboundedQueue library

In `UnboundedQueue`, the non-volatile shared fields `head` and `tail` are protected by explicit lock `deqLock` and `enqLock`, respectively. However, the `size()` method breaks this

```
public class DisBarrier {
   final int size;
   int logSize;
   final Node[][] node;
   final ThreadLocal<Boolean> mySense;
   final ThreadLocal<Integer> myParity;

   public DisBarrier(int capacity) {
    . . .
   }

   public void await() {
    . . .
   }

   private class Node {
      volatile java.util.concurrent.atomic.AtomicBoolean[] flag;
      Node partner;

      Node() {
         AtomicBoolean[] flag = new AtomicBoolean[2];
         flag[0] = new AtomicBoolean(false);
         flag[1] = new AtomicBoolean(false);
         this.flag = flag; // added to ensure safe publication of flag[]
      }
   }
}
```

Figure 5-20. DisBarrier library

constraint and requires prior locking to both locks before its invocation. This constraint is commented, as shown in Figure 5-19, but the application programmer was not aware of it. The `FairMessage` program has a data race on `head` and `tail`. Although it is possible to use JRF to detect these races, it is hard to correct this unless given more information, as suggested in Section 5.1.

Let us assume that the library `UnboundedQueue` has annotated precondition $enqLock \in lockset(current\_thread) \wedge deqLock \in lockset(current\_thread)$ at `size()`. One more assumption is that the library developer has already verified that this precondition would rule out any race in `UnboundedQueue` in any application environment. Given these two assumptions, it is more straightforward to detect a precondition violation in `size()` than data races on `head` and `tail`. It is also possible to exclude `head`, `tail`, and

```
public class FairMessage {
   public static void main(String[] args)
   { (new FairMessage()).run();}

   UnboundedQueue queue = new UnboundedQueue();
   DisBarrier bar = new DisBarrier(NUM_THREAD);
   static final int NUM_THREAD=2;
   static final int PER_THREAD=2;

   private void run() {
      assert (queue.size() == 0);
      for ( int i=0 ; i < NUM_THREAD ; ++i)
      {
         new EnqThread(i).start();
         new DeqThread().start();
      }
      System.out.println("queue size = "+queue.size());
   }

   class EnqThread extends Thread {
      int  id;
      EnqThread(int i) { id = i; }
      public void run() {
         for (int i = 0; i < PER_THREAD; i++) queue.enq(id+i);
      }
   }

   class DeqThread extends Thread {
      public void run() {
         for (int i = 0; i < PER_THREAD; i++)
            try {
               queue.deq();
               bar.await();
            } catch (EmptyException ex) {}
      }
   }
}
```

Figure 5-21. FairMessage application uses UnboundedQueue and DisBarrier

instances of `Node` from h since it is guaranteed to be free from a race when the given precondition is satisfied. When both libraries in `FairMessage` have annotated race-free preconditions, JRF only maintains non-volatiles in `FairMessage` and all volatiles in h. The experiment in Section 7.1.3 shows that this took less time and memory than the original JRF.

5.2.1 Analysis

The modular extension of JRF requires four steps to accomplish the compositional verification. Figure 5-22 outlines the four steps of our modular race detection. First, the library developer should provide appropriate preconditions to convey his/her design decisions regarding the data race-free guarantee. This includes the guarding explicit and implicit lock and h requirement at the entry to each method. This annotation process is followed by universal environment generation. Second, the JRF modular extension generates a universal environment of the target library with minimum happens-before orderings that satisfy given preconditions and the model checking constraints. The third step is to model check the library with the generated context using the h abstraction of JRF. We also check the preconditions at each invocation of the library modules. When a h precondition is violated, we enforce it by expanding h. When a $lock$ or $synch$ precondition is violated, we simply ignore the path.

We use bounded model checking with k as the multiple of the number of concurrent threads and the depth of each method to constrain the search space. This step may or may not discover a race. Adjust the precondition when it can eliminate the found race or modify the library code itself when precondition change cannot eliminate the race. In that case, it is a bug in a library rather than a problem of constraining the usage. Repeat the third step as many times as needed until no race is left. These three steps are performed by the original library developer before releasing the library to others. The last step happens in the application context. The application programmer checks the preconditions at every library method invocation as in the previous step. However, JRF does not manage h for the library internal fields. If any precondition violation is reported, this is an inconsistent use of a library method and it is up to the application programmer to decide how to handle it. Since a precondition violation does not imply the presence of a race, the application programmer might try to use JRF without a modular extension. If there is no precondition violation, we can conclude the application

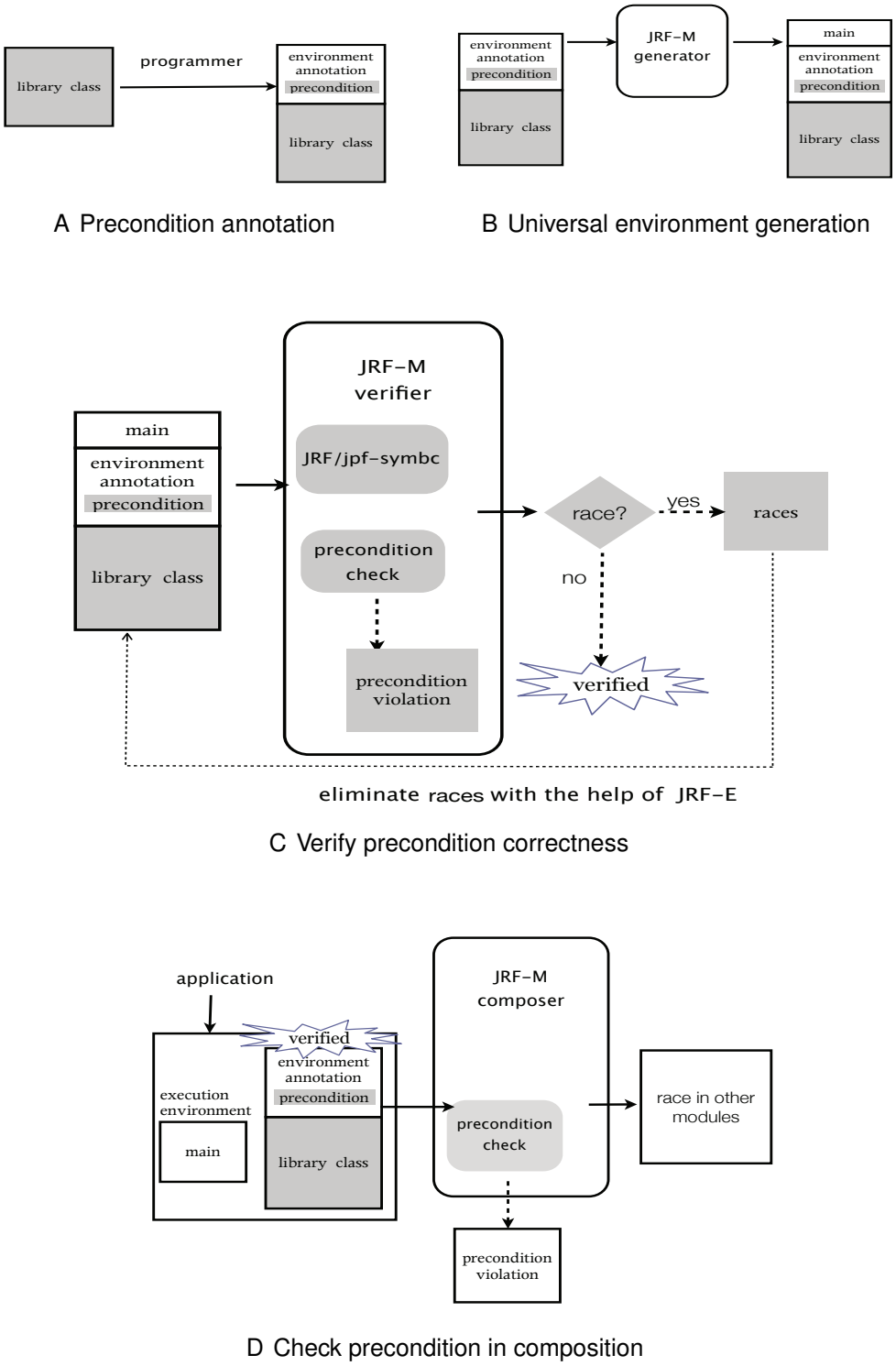


Figure 5-22. The four steps in modular extension of JRF

is free from a data race on library module internal fields. When JRF reports no race on non-volatile fields defined outside the library module, the entire system is guaranteed to be sequentially consistent without checking library internal non-volatiles. We can expect better scalability using JRF when the memory and time overhead required to manage library internal non-volatiles outweigh those to check method preconditions.

5.2.2 Algorithm

This section describes the algorithms used in each step in Figure 5-22.

Step 1. Annotate environment constraints and method preconditions to a library program. When developers implemente a library, they made design decisions on how to order conflicting accesses. Preconditions save this information and constrain the illegal use of the library.

We need to bound the number of concurrent threads invoking the methods of an object to constrain the number of threads in model checking. This bound is inevitable in model checking to limit the search space. In addition, the depth constraint is used to bound the recursive call to a method.

The precondition can be the h requirement, explicit lock, or synchronized block. Methods can have an arbitrary number of preconditions. The deriving rules are as follows:

$$< \mathit{class\_annotation} > := < \mathit{class\_constraint} > < \mathit{object\_constraint} >$$

$$< \mathit{class\_constraint} > := \texttt{threads\_bound}$$

$$< \mathit{object\_constraint} > := \texttt{threads\_bound}$$

$$< \mathit{method\_annotation} > := < \mathit{method\_constraint} > < \mathit{precondition} >^{*}$$

$$< \mathit{method\_constraint} > := \texttt{depth\_bound}$$

$$< \mathit{precondition} > := < \mathit{condition\_type} > \mid < \mathit{precondition} > \ \vee \ < \mathit{precondition} >$$

$$< \mathit{condition\_type} > := \ \mathit{h}(\texttt{field}) \mid \mathit{lock}(\texttt{field}) \mid \mathit{synch}(\texttt{field})$$

`threads_bound` is defined for each object including static object represented by the class itself and bounds the number of threads concurrently running methods of this

instance. In addition to this bound, individual method has a `depth_bound` as allowed number of recursive calls. When a thread tries to invoke the method and there are already `depth_bound` stack frames in working threads stack area, this call violates the given bound and is revoked.

In most cases, the minimum precondition requirement to invoke a library method is whether the object is accessible in the current thread; to have the object itself in h of the calling thread. This is safe publication property and can be specified by h(`this`) precondition. A prior locking requirement can be specified using a similar notation.

```
@ObjectConstraint (threads_bound=3)
public class UnboundedQueue {
   . . .

   @MethodConstraint (depth_bound=1)
   public UnboundedQueue() { . . . }

   @MethodConstraint (depth_bound=2)
   @Precondition ( h={ "CURRENT_THREAD WITH THIS"})
   public int deq() throws EmptyException { . . . }

   @MethodConstraint (depth_bound=2)
   @Precondition ( h={ "CURRENT_THREAD WITH THIS"})
   public void enq(int x) { . . . }

   @MethodConstraint (depth_bound=2)
   @Precondition ( h={ "CURRENT_THREAD WITH THIS"}, lock={"enqLock", "deqLock"} )
   public int size() { . . . }
   . . .
}

@ObjectConstraint (threads_bound=3)
public class DisBarrier {
   . . .

   @MethodConstraint (depth_bound=1)
   public DisBarrier(int capacity) { . . . }

   @MethodConstraint (depth_bound=2)
   @Precondition ( h={ "CURRENT_THREAD WITH THIS"})
   public void await() { . . . }
   . . .
}
```

Figure 5-23. UnboundedQueue library with precondition annotation

For example, when a method needs to lock `priorLock` field, which is an instance of `java.util.concurrent.locks.ReentrantLock`, before invoking it, it has *lock*(`priorLock`) as a precondition. When the requirement is to be invoked inside a synchronized block on the object itself, the precondition would be *synch*(`this`).

Preconditions are compositional as given in the deriving rule as $< \mathit{precondition} >^*$. For instance, the previous prior lock example would have a preconditions *h*(`this`) and *lock*(`priorLock`) since safe publication is the necessary precondition at all times. It is also possible to have choices in a precondition. When a method either requires to lock `priorLock` before calling it or to synchronize on the same field `priorLock`, the precondition can be specified as *lock*(`priorLock`) $\vee$ *synch*(`priorLock`). The precondition is regarded as satisfied when either one is satisfied.

Figure 5-23 shows the constraints and preconditions annotated to the libraries. Both `UnboundedQueue` and `DisBarrier` have annotated with a constraint of 3 concurrent threads and 2 calls to each methods. Note that a constructor have a `depth_bound` one since it is called only once per each instance.

Step 2. generate universal environment with minimum happens-before order.

Though the preconditions are annotated by the original programmer who knows better than anyone else about the library, it is very difficult to be sure that the conditions are strong enough to guarantee race freedom. To prove that those preconditions are correct choices using model checking where we need a closed system, we generate the most general execution environment meeting the environment constraints. The algorithm to generate this *universal environment* is in Figure 5-24.

We spawn one thread for a choice of constructors to instantiate an object and assign it to a non-volatile reference. The non-volatile reference is used to add no additional happens-before order in a universal environment. Note that the safe publication requirement will be fulfilled at the verification step. We also spawn `threads_bound` threads and for `depth_bound` times, make them invoke one of the methods. All

Algorithm *generateUniversalEnvironment*(*object_annotation*, *method_annotations*)

```
declare a non-volatile reference to an object;
choose* a constructor defined in method_annotations
    spawn a thread which
        invokes the constructor with all parameters as symbolic
            and assign new instance to the non-volatile reference;
for i from 1 to object_annotation.threads_bound
    spawn thread which
        for j from 1 to max( method_annotation.depth_bound )
            choose* non-internal method m defined in method_annotations
            foreach lock <condition_type> of m
                lock on the field
            foreach synch <condition_type> of m
                synchronize on the field
            foreach lock <condition_type> in <precondition> ∨ <precondition> of m
                choose* lock on the field or skip
            foreach synch <condition_type> in <precondition> ∨ <precondition> of m
                choose* synchronize on the field or skip
            invokes m with all parameters as symbolic
            foreach lock <condition_type> of m
                unlock the field
            foreach lock <condition_type> in <precondition> ∨ <precondition> of m
                unlock on the field or skip accordingly
```

Figure 5-24. Generate universal environment with all possible combinations of methods. (**choose*** generate different states in model checking)

parameters are symbolic variables so as to prevent specifying an input directly. In this manner, there will be $(1 + \texttt{threads\_bound})$ spawned threads in addition to the *main* in the universal environment. The generated universal environment for `UnboundedQueue` is shown in Figure 5-25.

When more than one locks are specified as preconditions, there is a chance of a deadlock in a concrete environment but the generated universal environment is free from this problem since the algorithm orders the locks according to the specified order. When a concrete environment happen to have a deadlock, it will be detected by JPF during Step 3.

```
public class UnboundedQueueVerify {
   UnboundedQueue obj;

   @Symbolic("true")
   int sym0;
   @Symbolic("true")
   int sym1;

   public static void main(String[] args) {
      new UnboundedQueueVerify().doTest();
   }
   void doTest() {
      for ( int i=0 ; i < 1 ; ++i) new Group1Thread().start();
      for ( int i=0 ; i < 3 ; ++i) new Group2Thread().start();
   }
   class Group1Thread extends Thread {
      public void run() {
         for ( int i=0 ; i < 1 ; ++i) {
            int option = gov.nasa.jpf.jvm.Verify.getInt(1,1);
            if ( option == 1 ) {
               obj = new UnboundedQueue();
            }
         }
      }
   }
   class Group2Thread extends Thread {
      public void run() {
         for ( int i=0 ; i < 2 ; ++i) {
            while ( obj==null);
            int option = gov.nasa.jpf.jvm.Verify.getInt(1,3);
            if ( option == 1 ) {
               obj.deqLock.lock(); obj.enqLock.lock();
               try { obj.size(); }
               finally{ obj.enqLock.unlock(); obj.deqLock.unlock();}
            }
            else if ( option == 2 ) {
               if ( i==0 ) obj.enq(sym0);
               else if ( i==1 ) obj.enq(sym1);
            }
            else if ( option == 3 ) {
               try{
                  obj.deq();
               } catch (jrfm.UnboundedQueue.EmptyException e) {}
            }
         }
      }
   }
}
```

Figure 5-25. Generated universal environment for UnboundedQueue

Step 3. Verify the race freedom for executions with all preconditions satisfied. The annotated library from Step 1 and the universal environment from Step 2 will be fed to JRF modular extension to check both preconditions and races. When a precondition is violated at the entry of any method during model checking, we will ignore the path unless the precondition is *h* type. We handle the *h* precondition violation differently from others since the universal environment did not take it into account as shown in Figure 5-24. We update the h of the current thread at *h* type precondition violation by adding the `field` and all memories last updated by the same thread as `field` when it had happened before the last update of `field` itself.

JRF will check a race as before. When no race is found at the end, we can conclude that the preconditions are correct and the library is verified to guarantee that the internal fields will be free from a data race for any concrete environment with all preconditions satisfied. If a race is found at this step, the preconditions are not strong enough for a race-free guarantee. This step should be repeated either by adding more preconditions or modifying the library code using the advice suggested in Section 5.1 until no race is left as follows:

```
=====================================================
JRFM-VerifyModule results
=====================================================
no race found and preconditions in "UnboundedQueue" are verified
. . .
```

Step 4. Check preconditions of library methods in an application program. The library is ready at this point and an application programmer will use it in a concrete environment. JRF can be used to precisely detect a race at this step, but, as described earlier, the annotated precondition would better guide the application programmer to a consistent use of a library. The modular extension will rule out the internal fields from h. Instead, at all method invocations of the library, the preconditions are checked.

When all preconditions are satisfied at all invocations of library methods, the library modules are consistently used as recommended and guaranteed to be free from a race

```
=====================================================
JRFM-ComposeModule results
===================================================== precondition violation #0
in "jrfm.UnboundedQueue.size()"
    the lock precondition of method (size) "enqLock, deqLock" is violated.
     at "System.out.println("queue size = "+queue.size());" in "jrfm.FairMessage.run(FairMessage.java:23)"
===================================================== precondition violation #1
in "jrfm.UnboundedQueue.size()"
    the lock precondition of method (size) "enqLock, deqLock" is violated.
     at "assert (queue.size() == 0);" in "jrfm.FairMessage.run(FairMessage.java:17)"
. . .
```

Figure 5-26. Precondition violations in `FairMessage` detected by JRF modular extension

inside the library module. When a precondition is violated and more than one thread has executed the methods of an object,[6] it means there is a possibility of a race. The presence of a precondition violation does not automatically mean the presence of a race, though. Rather it means the library is used in an inconsistent way that violates the developer's design. It is the application programmer's choice about how to handle a precondition violation. He may change the code to satisfy the precondition or use JRF without modular extension to precisely detect a data race. When JRF itself detects no race, the application is free from a race in spite of the precondition violation. Figure 5-26 shows the precondition violation in `FairMessage` detected by JRF modular extension.

5.2.3 Theoretical Results

In this section, we justify the soundness of modular race checking with respect to the race-free guarantee. The experimental result is in Section 7.1.

Definition: Equivalent Executions w.r.t. a set of actions Two well-formed executions, $E_1 = \langle A_1, P_1, \xrightarrow{po}_1, \xrightarrow{so}_1, W_1, V_1 \rangle$ and $E_2 = \langle A_2, P_2, \xrightarrow{po}_2, \xrightarrow{so}_2, W_2, V_2 \rangle$, are equivalent w.r.t. a set of actions A which is a subset of A_1 and A_2 if $A_1|_A = A_2|_A$, $\xrightarrow{po}_1 |_A = \xrightarrow{po}_2 |_A$, and $\xrightarrow{so}_1 |_A = \xrightarrow{so}_2 |_A$.

[6] When only one thread is accessing the methods of an object including the constructors, the object is threadlocal and free-from a race regardless of the precondition violation.

Suppose L is a library with internal non-volatiles $fields_L$ that is only accessible in the methods of L. Let us assume a universal environment U with bounding constraint B and any arbitrary concrete execution context C of L satisfying the bounding constraint B with $E^U = \langle A^U, P^U, \overset{po}{\rightarrow}^U, \overset{so}{\rightarrow}^U, W^U, V^U \rangle$ and $E^C = \langle A^C, P^C, \overset{po}{\rightarrow}^C, \overset{so}{\rightarrow}^C, W^C, V^C \rangle$ denote arbitrary sequentially consistent executions of U and C, respectively. When all such E^C satisfy the preconditions P of L in bound B and all E^U have no race on $fields_L$, it is guaranteed that all E^C has no race on $fields_L$ in given bound B. The following two lemmas will justify this.

Lemma 13. *For an arbitrary E^C, there is always an equivalent execution E^U,which is not ignored at step 3, w.r.t. A_L, where A_L is either the set of actions in L or the actions satisfying the lock and synch preconditions of L.*

Proof. The proof is by contradiction. Let us assume that an execution E_1^C has no equivalent execution in the set of E^U w.r.t. A_L, which is not ignored. Given that E_1^C satisfies all preconditions and constraints of L, we can construct an execution E_1^U corresponding to a path of the environment U by choosing the same actions as E_1^C at each transition. Moreover, given that the parameters are represented symbolically, we can choose the same W and V for E_1^U as E_1^C, i.e., $E_1^U = \langle A_1^C, P^U, \overset{po}{\rightarrow}^C_1, \overset{so}{\rightarrow}^C_1, W_1^C, V_1^C \rangle$ Such a transition choice is always available in a universal environment because E^C satisfies the bounding constraint B and there are enough transition choices in U to cover all different interleavings within B. The verification rule at step 3 guarantees that this path is not ignored since this path satisfies all *lock* and *synch* preconditions and $\mathtt{s}$ and $depth\_bound$ constraints. This contradicts the assumption that there is no such E_1^U in U. □

Lemma 14 shows that the h that includes the non-volatile fields in a library is minimal in universal environment. In other words, when a non-volatile field is in the h of a current thread at some execution step in the universal environment, it is guaranteed

to be in the h of a current thread at the equivalent execution step in any equivalent concrete environment.

Lemma 14. *Suppose h^{-1} is the inverse of h where $h^{-1}(x)$ is the set of memory locations $v \in (SynchAddr \cup Threads)$ such that $x \in h(v)$. When h^U and h^C denote h for two equivalent executions, E^U and E^C w.r.t. A_L, respectively, $\forall x \in fields_L$, $(h^U)^{-1}(x) \subseteq (h^C)^{-1}(x)$ holds for all prefixes of $E^U|_{A_L}$.*

Proof. The proof is by induction on the length of the prefix of $E^U|_{A_L}$.

Basis. We have length 0 prefix of $E^U|_{A_L}$. Since no action in A_L happens, $\forall x \in fields_L$, $(h^U)^{-1}(x) = (h^C)^{-1}(x) = \phi$.

Inductive Step. Assume $\forall x \in fields_L$, $(h^U)^{-1}(x) \subseteq (h^C)^{-1}(x)$ holds for $(E^U|_{A_L})_n$.

When the $(n+1)^{th}$ action is an action satisfying the lock and synch preconditions of L, the $(n+1)^{th}$ action is either *release* or *acquire* and $(h^U)^{-1}(x) \subseteq (h^C)^{-1}(x)$ is preserved by the h update rule in Figure 4-1.

When the $(n+1)^{th}$ action is either *release*, *acquire*, *invalidate*, or h irrelevant actions, $(h^U)^{-1}(x) \subseteq (h^C)^{-1}(x)$ is preserved by the h update rule in Figure 4-1.

Otherwise, the $(n+1)^{th}$ action is either an instantiation or a publication of the L object or an invocation of a method in L.

- When the action is an instantiation of the L object, it will add the instantiating thread to both $(h^U)^{-1}(x)$ and $(h^C)^{-1}(x)$ for all x in $fields_L$. This preserves $(h^U)^{-1}(x) \subseteq (h^C)^{-1}(x)$.

- When the action is a publication, given that the reference in U is defined as non-volatile, this publication will not change $(h^U)^{-1}(x)$. If the reference in C is a volatile, this will add the current thread into $(h^C)^{-1}(x)$. Otherwise, $(h^C)^{-1}(x)$ remains the same. This preserves $(h^U)^{-1}(x) \subseteq (h^C)^{-1}(x)$.

- At a method invocation, when h precondition is violated in U, this will add the violated field, suppose the field is f, and all other memory locations Y written prior to that by the same thread t to the h of current thread $currentT$. Since the memories in Y were last updated by t, $Y \subseteq h(t)$. $(h^U)^{-1}(f)|_{n+1} \leftarrow (h^U)^{-1}(f)|_n \cup \{currentT\}$ and $(h^U)^{-1}(y)|_{n+1} \leftarrow (h^U)^{-1}(y)|_n \cup \{currentT\}$ for all $y \in Y$. When h precondition is not violated in U, $(h^U)^{-1}(x)|_{n+1} = (h^U)^{-1}(x)|_n$.

The assumption guarantees that the h precondition is satisfied in C and $\{currentT\} \subseteq (h^U)^{-1}(f)|_{n+1}$. If t is $currentT$, $\{currentT\} \subseteq (h^U)^{-1}(y)|_{n+1}$ for all $y \in Y$. When $t \neq currentT$, f has been added to $currentT$ after the last write of f through *acquire* of v by $currentT$ preceded by *release* of v by t. At the time of *release* of v by t, $\{f\} \cup Y$ had been added to $h(v)$ since $\{f\} \cup Y$ had been in $h(t)$. Following *acquire* of v by $currentT$ would have added $\{f\} \cup Y$ to $h(currentT)$. This concludes that $\{currentT\} \subseteq (h^U)^{-1}(f)|_{n+1}$ and $\{currentT\} \subseteq (h^U)^{-1}(y)|_{n+1}$ for all y in Y. In both cases, $(h^U)^{-1}(x) \subseteq (h^C)^{-1}(x)$ is preserved.

□

Theorem 11 justifies the preconditions that were verified in U can guarantee the data race freedom on internal fields of L in C.

Theorem 11. *When a set of preconditions are verified to be correct in universal environments U with bounding constraint B, any arbitrary concrete environment C within B is guaranteed to be free from a data race on any internal fields of L if all preconditions are satisfied in all sequentially consistent executions E^C of C.*

Proof. The proof follows immediately from lemma 14. Since $\forall x \in fields_L$, $(h^U)^{-1}(x) \subseteq (h^C)^{-1}(x)$ holds for all prefixes of $E^U|_{A_L}$ and $x \in h^U(t)$ at $(E^U|_{A_L})_n$ guarantees $x \in h^C(t)$ at $(E^C|_{A_L})_n$. □

Since C is proved to have no race on $fields_L$, L can be trusted, as discussed in Section 4.3.6, and safely excluded from h without hurting the soundness of JRF in C.

CHAPTER 6
CASE STUDIES

Our extension to JPF[1] make it possible to soundly analyze complex, highly concurrent data structures that do not necessarily use locking, or use a mixture of locking and other synchronization idioms. In order for JPF to be sound when applied to these programs, the program must be free of data races. Lock-free programs typically use volatile variables along with instances of classes in the *java.util.concurrent.atomic* package to create the necessary happens-before edges to prevent data races.

This chapter will cover our experience of applying JRF in various programs from concurrent data structures to a large application framework. Testing was performed on a i386/8 processor with 32GB ram using Linux/2.6.32-24-generic OS, JPF version 5, and Sun Microsystems Inc./1.6.0_16 Java with 2GB JVM heap memory.

6.1 Simple Examples

We started our experiment with simple examples that are proven to be race free. After the first successful verification of a race freedom using JRF, we intentionally seeded races to them and checked if JRF could detect them. JRF was tested for various types of races; moved a shared data access outside a synchronized block, omitted locking, inconsistent use of a lock, and changed a volatile variable to a non-volatile.

The programs used in this phase was from [65], and originally implemented in Zing [66]. The translation from Zing to Java was done manually for total seven programs, and they had code sizes from 43 lines to 514 lines. Brief explanation about each application is as follows,

[1] The standard JPF distribution includes two race detecting tools: RaceDetector and PreciseRaceDetector. Neither correctly deals with programs where data races are avoided by using volatile variable or classes from the `java.util.concurrent.atomic` package.

- **IndependentWorker1**: the main thread creates two threads each of which allocates its own list and traverses it without any sharing of a data.
- **IndependentWorker2**: the main thread creates two threads and two lists and let them traverses one of the lists.
- **FileSystem**: a static array field is shared among threads, but each thread accesses disjoint elements of the array. Even though the container array is shared among different threads, individual data element of it is thread-local.
- **Indexer**: multithreads index each element of the global shared array `table[]` using a hash value. the same as the **FileSystem**, individual data element of the global static array is thread-local.
- **HaltException**: `producer` and `consumer` threads share a lock-protected `buffer`.
- **IOManager**: the main thread create three more threads to model an I/O Request Packet (IRP) ; `NtReadFile` to execute IRP, `IopCloseFile` to cancel it, and `ModelProcessCompletionPortEntries` to post-process either successful or canceled IRP. They synchronize with each other using both lock-protected queues and volatile shared variables.
- **BlinkTree**: this is a B-link tree implementation with configurable number of list tester threads. Each node is protected by a specific lock, and the tree balancing makes this protecting lock to dynamically change.

6.2 Herily-Shavit Examples

To evaluate the usefulness of JRF and its counterexample analysis and to empirically explore the behavior of the search heuristics, we used JRF to analyze an extensive set of concurrent objects described in the a recently published and highly regarded textbook by Herlihy and Shavit [1].

Java implementations were obtained from the book's web site and all the code in this section is derived from these programs. Our test suite includes six barrier implementations from Chapter 17, eight hash sets from Chapter 13, five list implementations from Chapter 9, four mutex codes from Chapter 2, six lock-free queue-based spin lock implementations from Chapter 7, four monitor and blocking synchronization tests from Chapter 8, nine different queue implementations from Chapter 10, nine scheduling and work distribution algorithms from Chapter 16, and four priority queues from Chapter

15. [1] In addition, once JRF found a race, we also corrected it and reran JRF to find any remaining races. We found 19 races from a total 68 test programs and 14 other application assertion violations, such as a deadlock. The purpose of publishing this work is in no way meant to disparage the very impressive and valuable work of Herlihy and Shavit. Rather, analyzing their examples allows us to demonstrate that JRF is a practical and useful tool for finding data races, and that task is worth supporting with a tool because it is difficult.

The Java implementations were provided in the form of a class implementing the algorithm with a test class implementing JUnit test cases. To avoid having the JPF model check the JUnit framework, we modified the test classes to invoke the test methods directly and significantly reduced the number of threads and number of iterations for the tests. In a few cases, such as the `TOLock` class given in Figure 6-5, we found it desirable to modify the source code of the class being tested to eliminate calls to System.nanoTime. In almost all cases, data races were found very quickly, within a few seconds. Once there were no more data races, running an example to completion could take a prohibitive amount of time, with no indication of how much longer it would be–another 30 seconds, several hours, or several days. In most cases, the execution terminated, or was canceled before it ran out of memory. Standard JPF without our extensions (using the default properties) showed the same lack of scalability. Some examples using standard JPF ran for several days without terminating. As a result it was difficult to obtain meaningful data for valid comparisons, but the data we do have indicate that our extensions currently increase the execution time by a factor of two to three. Generally, data races are found quickly because in most programs they tend to occur on multiple paths.

To illustrate how the process of using our tool can work, we describe a selected set of case studies in detail including several hash set implementations from Chapter 13 and two lock-free queue-based spin lock implementations from Chapter 7.

6.2.1 Examples

6.2.1.1 Concurrent Hash Sets

We will look at three different closed-address hash set implementations that use locking in three different ways. `CoarseHashSet` uses a single lock, `StripedHashSet` uses a fixed-size array of locks, and `RefinableHashSet` uses a resizable array of locks. All three are based on the abstract `BaseHashSet<T>` class. An extract of this class is given in Figure 6-1. The implementations of the `contains` and `remove` methods are

```
public abstract class BaseHashSet<T> {
  protected List<T>[] table;
  protected int size;
  public BaseHashSet(int capacity) {
    size = 0;
    table = (List<T>[]) new List[capacity];
    for (int i = 0; i < capacity; i++) {
      table[i] = new ArrayList<T>();
    }
  }

  public boolean contains(T x) {...}
  public boolean add(T x) {
    boolean result = false;
    acquire(x);
    try {
      int myBucket =
        Math.abs(x.hashCode()%table.length);
      result = table[myBucket].add(x);
      size = result ? size + 1 : size;
    }
    finally {
      release(x);
    }
    if (policy())  resize();
    return result;
  }

  public boolean remove(T x) {...}

  public abstract void acquire(T x);
  public abstract void release(T x);
  public abstract void resize();
  public abstract boolean policy();
}
```

Figure 6-1. Abstract class used in three lock-based closed-address hash sets

```
public class CoarseHashSet<T> extends BaseHashSet<T>{
  final Lock lock;
  CoarseHashSet(int capacity) {
    super(capacity);
    lock  = new ReentrantLock();
  }

  /*double the set size */
  public void resize() {
    int oldCapacity = table.length;
    lock.lock();
    ..resize table if oldCapacity==table.length...
    } finally {
      lock.unlock();
    }
  }
  public final void acquire(T x) {lock.lock();}
  public void release(T x) {lock.unlock();}
  public boolean policy() {
    return size / table.length > 4;
  }
}
```

Figure 6-2. Hash Set implementation that uses a single lock

omitted for brevity, but they use a locking scheme and hash calculation similar to the `add` method. The `acquire`, `release`, `resize`, and `policy` methods are abstract and must be implemented in a subclass. The `acquire` and `release` methods implement the locking mechanism, `resize` resizes the HashSet, and `policy` determines if a resize should be done.

The simplest subclass uses a single lock (Figure 6-2). The `acquire` and `release` methods simply delegate to the `lock` and `unlock` methods on a single `java.util.concurrent.Reentra` instance. Analyzing this program with JRF revealed data races on `size` and `table.length` due to the unguarded accesses to these variables in the `policy` method and the first line of the `resize` method that conflict and race with accesses in the `add` and `remove` methods. While there were several options for correcting the problem, we did it by modifying the `resize` method so that it rechecks `policy` instead of `oldCapacity==table.length`, eliminating the need for the unguarded assignment to `oldCapacity`. In addition, the body of `policy` was locked.

```
public class StripedHashSet<T> extends BaseHashSet<T>{
  final Lock[] locks;

  public StripedHashSet(int capacity) {
    super(capacity);
    locks  = new Lock[capacity];
    for (int j = 0; j < locks.length; j++) {
       locks[j] = new ReentrantLock();
    }
  }

  public void resize() {
    for (Lock lock : locks) {
       lock.lock(); //acquire all of the locks
    }
    try {...resize the table ...
    }
    finally {
      for (Lock lock : locks) {
        lock.unlock(); //release all of the locks
      }
    }
  }
   ....

   public final void acquire(T x) {
     int myBucket =
          Math.abs(x.hashCode() % locks.length);
     locks[myBucket].lock();
   }

   public void release(T x) {
     int myBucket =
          Math.abs(x.hashCode() % locks.length);
     locks[myBucket].unlock();
   }

   public boolean policy() {....}
  }
}
```

Figure 6-3. Hash Set implementation that uses a fixed size array of locks

A hash set implementation using an array of locks to allow more concurrency is shown in Figure 6-3. The same problems with races on `size` and `table.length` were

present here. They were eliminated similarly to what was done with the `CoarseHashSet` example. Rerunning JPF revealed a new problem. The increment of `size` in the `add` method defined in `BaseHashSet` is no longer atomic, since threads holding different locks for different buckets will not exclude each other. A good solution to that problem and one that does not require introducing additional locking is to change the type of the `size` variable from `int` to `AtomicInteger`, since the `AtomicInteger` class provides an atomic `getAndIncrement` method that has the same memory semantics as reading and writing a volatile variable. This yields a race free program. We also noticed, and JRF allowed us to verify, that when `size` is an AtomicInteger, locking the body of `policy` is no longer required. The program now uses a mixture of synchronization idioms.

The third lock based hash set, called `RefinableHashSet`, supports a resizable array of locks. The original version suffered from the same problem with the `size` variable as `StripedHashSet`. Once we used the revised version of `BaseHashSet`, where `size` is an `AtomicInteger`, the program was free from data races.

Since these classes use locks for synchronization, some of the described analysis could have been done using a lock-set based race detection tool. However, these would not have been able to deal with the `AtomicInteger` type or allow us to confidently remove the locking from the policy method.

Several other hash set classes, including the concurrent open-address cuckoo hash set implementations [67] `CuckooHashSet`, `StripedCuckooHashSet`, and `RefinableCockooHashSet`, were successfully analyzed without revealing any data races or other errors.[2]

[2] LockFreeHashSet, a lock-free closed-address hash map implementation based on recursive split-ordered list [63] as the underlying data structure, displayed an assertion failure that turned out to be an error in the test driver.

6.2.1.2 Queue-based Spin Locks

In true multiprocessors, it often makes sense to implement locks by having threads spin, i.e., repeatedly check a condition, rather than block and cause a context switch. However, simply having all waiting threads read a single memory location is not scalable in the most common architectures due to the demands it places on the shared system bus. Queue-based locks are designed so that only one thread spins on any memory location, and these locations are not likely to be in the same cache line. In addition, these locks provide fairness and avoid the critical section underutilization inherent to back-off schemes. A detailed discussion of these issues can be found in Chapter 7 of Herlihy and Shavit's book [1].

From the point of view of the memory model, lock algorithms must satisfy two requirements. The algorithms themselves need to be data-race free. In addition, they need to provide a happen-before edge between a release of the lock and a subsequent acquire so that these locks will provide the same memory model related semantics as Java's intrinsic locks.

This section will look at two queue-based locking algorithms. The first algorithm is called a CLH lock after its originators, Craig, Hagersten, and Landin [68, 69] and is shown in Figure 6-4. Variable `tail` always refers to the last node in the queue, or a dummy node if the queue is empty. Each node contains a field `locked`. Waiting nodes form a virtual linked list, and spin on the `locked` field of their predecessor. `Threadlocal` variables cannot be involved in a data race (although a shared object referred to by a Thread local could be), and `tail` is implemented using an `AtomicReference`, so it will not be involved in a data race, and every access, such as the `getAndSet` call in the `lock` method will create happen-before edges. Analyzing this class with JRF, however, reveals that in some executions, there is a data race on the `locked` field. This race can be eliminated by making `locked` volatile.

```
public class CLHLock implements Lock {
  AtomicReference<QNode> tail; // most recent lock holder
  ThreadLocal<QNode> myNode, myPred;  // thread-local variables

  public CLHLock() {
    tail = new AtomicReference<QNode>(new QNode());
    // initialize thread-local variables
    myNode = new ThreadLocal<QNode>() {
    protected QNode initialValue() {
       return new QNode();
    }
  };
  myPred = new ThreadLocal<QNode>() {
      protected QNode initialValue() {
         return null;
      }
    };
  }

  public void lock() {
    QNode qnode = myNode.get(); // use my node
    qnode.locked = true;        // announce start
    // Make me the new tail, and find my predecessor
    QNode pred = tail.getAndSet(qnode);
    myPred.set(pred);           // remember predecessor
    while (pred.locked) {}      // spin
  }

  public void unlock() {
    QNode qnode = myNode.get(); // use my node
    qnode.locked = false;       // announce finish
    myNode.set(myPred.get());   // reuse predecessor
  }
  ...
  static class QNode {  // Queue node inner class
    public  boolean locked = false;
  }
}
```

Figure 6-4. CLH lock

The next algorithm is a variation of the CLH lock that allows waiting nodes to timeout and abort. This version also needs its `QNode` `locked` field to be volatile. Since a waiting thread is part of a virtual list of nodes, it cannot simply abort. Instead, when

```
public class TOLock implements Lock{
  static QNode AVAILABLE = new QNode();
  AtomicReference<QNode> tail;
  ThreadLocal<QNode> myNode;

  public TOLock() {
    tail = new AtomicReference<QNode>(null);
    myNode = new ThreadLocal<QNode>() {
       ..same initialization as CLHLock..
    }

    public boolean tryLock(long time, TimeUnit unit)
         throws InterruptedException {
      long startTime = System.nanoTime();
      long patience =
          TimeUnit.NANOSECONDS.convert(time, unit);
      QNode qnode = new QNode();
      myNode.set(qnode); // remember for unlock
      qnode.pred = null;
      QNode pred = tail.getAndSet(qnode);
      if (pred == null || pred.pred == AVAILABLE) {
        return true;  // lock was free; just return
      }
      while (System.nanoTime()-startTime < patience) {
        QNode predPred = pred.pred;
        if (predPred == AVAILABLE) {
          return true;
        }  else if (predPred != null) {
               pred = predPred; //skip predecessors
      }
    }

    // timed out; reclaim or abandon own node
       if (!tail.compareAndSet(qnode, pred))
         qnode.pred = pred;
         return false;
       }
       public void unlock() {
       QNode qnode = myNode.get();
         if (!tail.compareAndSet(qnode, null))
           qnode.pred = AVAILABLE;
         }

   static class QNode { ..same as CLHLock..}
 }
}
```

Figure 6-5. Variation of CLH lock that allows waiting nodes to timeout

a thread aborts while its `Qnode` is in the middle of the queue, it invalidates its `locked` field in a way that causes its successor to start spinning on the aborting thread's

predecessor. The relevant part of the original code for this class is shown in Figure 6-5. This class is one of the few where we changed the code of the class itself to facilitate the analysis with JPF. In order to avoid calculations involving `System.nanoTime()`, the guard of the loop `while (System.nanoTime() - startTime < patience)` was replaced with `while (Verify.getBoolean())` and the statements assigning to `startTime` and `patience` could be commented out. The `Verify` class is provided by JPF and ensures that both true and false will be explored and thus both timeout and non-timeout paths will be taken.

6.2.2 Experimental Results

6.2.2.1 Race Detection in JRF

Table 6-1 shows empirical results comparing the behavior of an heuristic search with DFS and BFS for a selection of examples, all containing data races identified by JRF. Experimental result shows that 11 of 19 have shorter counterexample paths when using suggested heuristics. Three out of remaining eight cases have same counterexample length with both time and memory overhead. Five examples have longer counterexample and two of those find a different race location.

Table 6-2 summarizes the results of JPF PreciseRaceDetector for the examples in Table 6-1. PreciseRaceDetector found one false race on a volatile field, and missed races on arrays declared as volatile, but their element were not volatiles. JRF heuristic search outperforms PreciseRaceDetector and gives shorter counterexample paths in 11 out of 19 cases.

6.2.2.2 Race Analysis in JRF-E

In this section, the experiment performed with counterexample analysis turned on. Table 6-3 are the experimental results with threshold races as 1, 10, and 100. JRF-E ended with an application assertion error before finding any race in `TCuckooHashSet`. The race given in 6-4 is found when we searched all state space using `search.multiple_errors` and `jrf.reporter.standalone` option.

Table 6-1. Experimental results for [1] examples containing a race found by JRF. Results without thread local optimization for DFS, heuristic search, and BFS are given.

example	search	jpf-states	h-states pruned	length	time (sec)	mem (MB)	volatiles, locks,threads	non-volatiles
DisBarrier	dfs	108	0/28	108	8	81	36	300
	heuristic	74	0/0	38	4	54	36	300
	bfs	1297	0/127	36	117	347	36	306
StaticTreeBarrier	dfs	58	0/86	58	9	58	39	322
	heuristic	88	0/0	46	7	65	39	324
	bfs	10482	8/2282	>27*	12457	1988	39	344
CoarseHashSet	dfs	58	2/46	34	8	65	42	341
	heuristic	139	0/2	34	18	106	60	351
	bfs	495	0/50	>9**	107	300	58	361
LockFreeHashSet	dfs	55	0/0	55	6	70	44	380
	heuristic	181	0/0	46	12	99	43	364
	bfs	1859	35/181	>12**	309	605	41	346
RefinableHashSet	dfs	99	2/76	56	18	100	49	376
	heuristic	198	0/0	56	20	129	49	378
	bfs	5619	142/1516	>15*	10036	1999	81	382
StripedHashSet	dfs	58	2/46	34	9	63	46	351
	heuristic	84	0/0	22	8	75	60	362
	bfs	481	0/52	>9**	133	318	54	371
TCuckooHashSet	dfs	427	724/2372	>240*	1230	549	155	677
	heuristic	1093	0/38	>234*	697	938	146	676
	bfs	3389	4/203	>10*	9836	1999	120	525
LazyList***	dfs	64	0/0	64	9	75	54	353
	heuristic	6226	0/1158	82	2660	1016	72	379
OptimisticList	dfs	55	0/0	55	8	78	54	349
	heuristic	2720	0/285	55	898	487	62	365
	bfs	7155	417/1340	>16*	17221	1992	46	325
Bakery	dfs	33	0/0	33	4	39	38	423
	heuristic	69	0/0	34	5	56	38	410
	bfs	690	0/17	22	94	286	38	378
Filter	dfs	54	0/20	55	5	45	38	300
	heuristic	47	0/0	25	3	35	38	310
	bfs	377	0/18	19	42	178	38	312
Peterson	dfs	24	0/0	24	2	33	35	307
	heuristic	48	0/0	23	2	36	35	300
	bfs	322	0/18	17	23	132	35	302
LockFreeQueue	dfs	25	0/0	25	2	32	29	236
	heuristic	32	0/0	18	2	34	29	233
	bfs	53	0/0	15	3	39	29	233
ALock	dfs	22	0/0	22	2	33	35	245
	heuristic	48	0/0	25	2	35	35	245
	bfs	497	22/28	20	38	191	35	245
CLHLock***	dfs	19	0/0	19	1	34	33	250
	heuristic	40	0/0	21	2	35	33	250
MCSLock	dfs	17	0/0	17	1	29	33	253
	heuristic	43	0/0	20	2	32	33	253
	bfs	204	0/26	15	14	90	33	253
CorrectedMCSLock***	dfs	17	0/0	17	1	31	35	251
	heuristic	21	0/0	11	1	27	34	245
DEQueue	dfs	26	0/0	26	2	33	29	245
	heuristic	28	0/0	18	2	33	29	241
	bfs	34	0/0	10	3	33	29	241
UnboundedQueue	dfs	35	0/0	35	5	45	46	333
	heuristic	139	0/1	29	14	108	46	347
	bfs	1078	11/362	12	333	548	62	336

* JPF out of memory and JRF ended gracefully with result report
** algorithm found application assertion violation before a race is found
*** bfs algorithm ran out of Java heap space and JPF failed to report the result

Table 6-2. Execution result of PreciseRaceDetector for Herlihy-Shavit tests from 6-1

example	state	length	time	mem	result
DisBarrierTest	99	99	1	22	same race
StaticTreeBarrierTest	45	45	1	22	same race
CoarseHashSetTest	58	33	1	22	same race
LockFreeHashSetTest	112	63	1	22	**application assertion error**
RefinableHashSetTest	98	55	1	23	same race
StripedHashSetTest	58	33	1	22	same race
TCuckooHashSetTest	505	318	2	28	**application assertion error**
LazyListTest	224	65	1	27	same race
OptimisticListTest	241	55	1	27	same race
BakeryTest	280	51	1	23	**different race found***
FilterTest	928	0	3	30	**no race**
LockFreeQueueTest	70	25	1	22	same race
PetersonTest	129	22	1	22	same race
ALockTest	772	0	2	31	**no race**
CLHLockTest	96	20	1	22	same race
MCSLockTest	53	19	1	22	same race
CorrectedMCSLockTest	53	19	1	16	false race on volatile field
DEQueueTest	160	20	1	22	same race
UnboundedQueueTest	96	35	1	17	same race

*The results in **boldface** is failure to detect races on volatile array with non-volatile element accesses.

Table 6-4 summarizes suggestions by JRF for each race found. Most races were caused by not using volatile or atomic array. This is because the target codes were selected from concurrent data structures that implement lock-free algorithms. The programmers who implement the concurrent libraries are the experts who understand concurrent programming rules and thread interleaving, and tend to make few concurrency mistakes caused by missing synchronization or inconsistent locking.

6.3 Amino Concurrent Basic Blocks

The Amino open source software project [2] implements concurrent building blocks in highly efficient and scalable codes, and aimes to support a set of lockfree collection classes, parallel patterns, and scheduling algorithms. Amino concurrent building blocks are implemented in two languages, Java and C/C++. The Java version is implemented as `org.amino` package with 6 subpackages, including `utility`, `alg` for parallel graph algorithm, searching and sorting algorithms, `ds` for data structures, such as parallel tree, parallel graph, and lockfree collections, `mcas` for multi-CAS, `pattern` for master-worker

Table 6-3. Experimental results for [1] examples containing a race found by JRF-E. Results without thread local optimization using DFS and threshold as 1, 10, 100 are given.

example	threshold races	jpf-states	h-states pruned	max length	JRF time (sec)	JRF-E time (sec)	mem (MB)	volatiles, locks,threads	non-volatiles
DisBarrier	1	108	0/28	108	8	0	70	36	300
	10	331	0/184	265	27	5	157	36	304
	100	1015	641/3675	268	150	57	263	36	304
StaticTreeBarrier*	1	58	0/86	58	10	0	57	39	322
CoarseHashSet**	1	58	2/46	34	8	0	71	42	341
	10	169	8/214	38	37	2	157	69	374
LockFreeHashSet	1	55	0/0	55	7	0	68	44	380
	10	79	2/28	57	10	7	77	44	386
	100	112	26/146	61	18	26	107	44	406
RefinableHashSet	1	99	2/76	56	19	2	105	49	376
	10	236	15/295	61	66	23	223	63	383
	100	319	41/529	62	95	36	242	76	409
StripedHashSet**	1	58	2/46	34	9	0	63	46	351
	10	169	15/235	38	41	5	159	73	384
LazyList	1	64	0/0	64	10	0	86	54	353
	10	116	2/56	66	20	8	119	54	355
	100	356	133/793	77	108	81	241	68	362
OptimisticList	1	55	0/0	55	8	0	74	54	349
	10	110	8/92	59	22	8	123	54	351
	100	734	911/2693	59	597	24	412	102	407
Bakery	1	33	0/0	33	3	1	40	38	423
	10	36	0/2	34	4	16	39	38	430
	100	71	0/36	51	9	181	72	38	492
Filter	1	54	0/20	55	5	2	47	38	300
	10	72	0/22	70	6	32	55	38	304
	100	197	0/254	81	18	304	102	38	320
LockFreeQueue	1	25	0/0	25	1	0	29	29	236
	10	34	0/8	29	2	0	33	29	238
	100	254	0/454	38	11	5	76	29	238
Peterson	1	24	0/0	24	2	0	34	35	307
	10	29	0/4	26	2	2	33	35	319
	100	141	0/210	39	9	20	73	35	337
ALock	1	22	0/0	22	2	0	31	35	245
	10	25	0/2	23	2	1	34	35	245
	100	161	0/226	36	9	9	72	35	245
CLHLock	1	19	0/0	19	1	0	32	33	250
	10	24	0/4	21	2	0	33	33	255
	100	183	82/528	31	18	3	80	33	255
MCSLock	1	17	0/0	17	1	0	32	33	253
	10	59	12/82	26	4	0	49	33	253
	100	151	161/671	30	18	3	94	33	279
CorrectedMCSLock*	1	17	0/0	17	1	0	29	35	251
DEQueue	1	26	0/0	26	2	0	32	29	245
	10	50	0/26	35	3	1	38	29	245
	100	274	0/464	37	13	11	80	29	246
UnboundedQueue	1	35	0/0	35	5	0	55	46	333
	10	40	0/4	37	6	5	52	46	335
	100	175	21/275	45	31	52	130	46	348

* JRF-E ran out of memory before detecting threshold races
** JRF-E ended by application assertion error before detecting threshold races

pattern, and `scheduler` for scheduling threads. Standard distribution downloaded from [2] includes 20 junit tests and 10 examples using the concurrent building blocks.

Table 6-4. JRF-E suggestions from counterexample and acquiring history analysis for Herlihy-Shavit examples with a race

test suite	race field of class	analysis
DisBarrier	flag[] of Node	**use atomic array for flag[]**
	log[] of DisBarrier	**use atomic array for flag[]** or log[]
StaticTreeBarrier	sense of StaticTreeBarrie	**make sense volatile**
	log[] of StaticTreeBarrie	use atomic array for log[], **make sense volatile**
CoarseHashSet	size of CoarseHashSet	make size volatile, **lock the *lock***
LockFreeHashSet	bucket[] of LockFreeHashSet	**use atomic array for bucket[]**
	head of BucketList	make next volatile, **use atomic array for bucket[]**
	next of Node	make next or head of BucketList volatile, **use atomic array for bucket[]**
RefinableHashSet	size of RefinableHashSet	make size volatile,**lock the locks[]**
StripedHashSet	size of StripedHashSet	make size volatile, **lock the locks[]**
TCuckooHashSet	table[] of TCuckooHashSet	use atomic package for table[], **lock the locks[][]**
LazyListTest	next of Node	**make next volatile**
	marked of Node	**make** marked or **next volatile**
	key of Node	**make** key, marked, or **next volatile**,
	lock of Node	**make** lock or **next volatile**
OptimisticList	next of Entry	make next volatile, **lock the lock**
	key of Entry	make key or next volatile, **lock the lock**
Bakery	label[] of Bakery	**use atomic array for label[] and flag[]**, make counter of Label volatile
	counter of Label	make counter or id of Label volatile, **use atomic array for label[]**
	flag[] of Bakery	**use atomic array for flag[]**
Filter	level[] of Filter	**use atomic array for victim[]** or level[]
	victim[] of Filter	**use atomic array for victim[]**
	counter of Filter	make counter volatile, **use atomic array for victim[]** or level[]
Peterson	victim of Peterson	make victim or counter volatile, **use atomic array for flag[]**
	flag[] of Peterson	**use atomic array for flag[]**
	counter of Peterson	make counter volatile, **use atomic array for flag[]**
LockFreeQueue	tail of LockFreeQueue	**make tail volatile**
	head of LockFreeQueue	**make tail volatile**
	items[] of LockFreeQueue	use atomic array for items[], **make tail volatile**
ALock	flag[] of ALock	**use atomic array for flag[]**, make value of Entry volatile
	counter of ALock	make counter or value of Entry volatile, **use atomic array for flag[]**
CLHLock	locked of QNode	**make locked volatile**
	counter of CLHLock	**make locked of QNode** or counter **volatile**
MCSLock	next of QNode	**make locked**, next, or counter **of QNode volatile**
	counter of MCSLock	**make locked of QNode** or counter **volatile**
	locked of QNode	**make locked of QNode** or next of QNode **volatile**
CorrectedMCSLock	counter of MCSLock	**make locked of QNode** or counter **volatile**
DEQueue	bottom of DEQueue	**make bottom volatile**
	map[] of DeQueue	**use atomic array for map[]**
UnboundedQueue	next of Node	make next volatile, **lock the enqLock**
	value of Node	make value or next volatile, **lock the enqLock**

*__bold__ entry indicates the most appropriate solution.

6.3.1 Examples

The `org.amino.ds.lockfree.LockFreeDeque` implements a doubly ended queue using a Compare-And-Set based lock free algorithm. The lock free algorithm uses

```
public class LockFreeDeque<E> extends AbstractQueue<E> implements Deque<E> {
 ...
   /**
    * Iterator definition of deque. This iterator is NOT thread-safe
    *
    */
   private class DeqIterator implements Iterator<E> {
       private DequeNode<E> cursor = anchor.get().left;

       public boolean hasNext() {
           return cursor != null;
       }

       public E next() {
           if (cursor == null) throw new NoSuchElementException();

           E result = cursor.data;
           cursor = cursor.right.get();
           return result;
       }

       public void remove() {
           throw new UnsupportedOperationException();
       }
   }
 ...
}
```

Figure 6-6. Data Race on cursor in LockFreeDeque.DeqIterator since the implementation is not thread-safe

an `AnchorType` object with left and right pointers, a status field, and a number of elements in deque. One anchor is defined for each deque, and is immutable. It uses `java.util.atimic.AtomicIntegerFieldUpdater` to change the status field. In addition, an anchor for a deque is updated using `java.util.atomic.AtomicReference` wrapping. There is a race in this class implementing iterator when tested for `IteratorTest.testSameIteraterUsedByMT`, which is a test for an iterator shared by multithreads. The code for iterator is given in Figure 6-6. Amino lock free components do not support iterator.remove() to provide better performance for frequently used operations such as addFirst, addLast, pollFirst, and pollLast.

The `org.amino.ds.lockfree.LockFreeQueue` is a lock free FIFO queue. It also uses two pointers, prev and next, instead of a standard singly linked list, stores head and tail of the queue in a volatile field, and is updated using `java.util.atomic.AtomicReferenceFieldUpdater`. `QueueItr` for this class is also not thread safe and has a race on its `nextNode` field when the same iterator is used by multithreads. The code is given in Figure 6-7.

`org.amino.ds.lockfree.LockFreeBlockQueue` implements the blocking version of the lock free FIFO queue. This uses the same data structure to implement the FIFO queue, and blocks when an immediate poll or put is unavailable. `LockFreeBlockQueue` does not support an iterator.

`org.amino.ds.lockfree.LockFreePriorityQueue` is another implementation of concurrent queue implementation. It uses an array list of `java.util.atomic.AtomicMarkableReference` to represent next levels. One iterator cannot be shared among multiple threads since *PQueueIterator.cursor* is not shared properly, as in Figure 6-6

`org.amino.ds.lockfree.EBDeque`, an elimination backoff deque, implements the same algorithm as `org.amino.ds.lockfree.LockFreeDeque` together with considering elimination backoff. This is for high contention cases, and uses `org.amino.utility.EliminationArray` in its implementation. Its main idea is to reduce the number of modification to the main data structure through maintaining two arrays to store two types of operations: add and remove.

In addition to lock free queue, lock free list, lock free ordered list, lock free set, lock free vector, and lock free dictionary are provided in the `org.amino.da.lockfree` package.

When tested with **jpf-core** without **jpf-concurrent**, both directed and undirected `GraphTest` find a race on `java.util.concurrent.ConcurrentHashMap$KeySet` from the code `java.util.concurrent.ConcurrentHashMap`. The reason is **jpf-core** does not

```
public class LockFreeQueue<E> extends AbstractQueue<E> implements Queue<E>, Serializable {
...
  private class QueueItr implements Iterator<E> {
      private Node<E> nextNode;
      private Node<E> lastRet;
      private E nextItem;

      QueueItr() {
          lastRet = null;
          advance();
      }

      private E advance() {
          lastRet = nextNode;
          E x = nextItem; /* value of next node */

          // p point to next valid node
          Node<E> p = (nextNode == null) ? first() : nextNode.getNext();
          while (true) {
              // reach the end
              if (p == null) {
                  nextNode = nextItem = null;
                  return x;
              }
              E item = p.value;
              if (item != null) {
                  /* p is a valid node */ nextNode = p;
                  nextItem = item;
                  return x;
              } else
                  /* skip over nulls */ p = p.getNext();
          }
      }

      public boolean hasNext() {
          return nextNode != null;
      }

      public E next() {
          if (nextNode == null) throw new NoSuchElementException();
          return advance();
      }

      public void remove() {
          throw new UnsupportedOperationException();
      }
  }
...
}
```

Figure 6-7. Data Race on nextNode in LockFreeQueue.QueueItr since the implementation is not thread-safe

implement `java.util.concurrent.ConcurrentHashMap` properly. `OrderedListExample` from the examples package also finds a benign race in `java.lang.String` class.

6.3.2 Experimental Results

6.3.2.1 Race Detection in JRF

Table 6-5 compares JRF results with different search strategies for the examples from [2] with races. The result shows that suggested heuristics performs better than DFS excluding `QueueTest`. BFS returns the optimal counterexample path when there is a race and JRF heuristic search performs close to this optimal case in Table 6-5. It is also noticeable that heuristic search uniquely finds a race in `LockFreeSet` while DFS and BFS ran out of memory after very long computation.

Table 6-6 summarizes the results of JPF PreciseRaceDetector for the examples in Table 6-5. JRF heuristic search gives shorter counterexample paths all eight cases.

Table 6-5. Experimental results for [2] examples containing a race found by JRF. Results without thread local optimization for DFS, heuristic search, and BFS are given.

example	search	jpf-states	h-states pruned	length	time (sec)	mem (MB)	volatiles, locks,threads	non-volatiles
IteratorTest (EBDeque)	dfs	26	0/0	26	32	138	91	564
	heuristic	25	0/0	13	5	45	91	562
	bfs	121	0/0	11	52	159	90	558
IteratorTest (LockFreeDeque)	dfs	26	0/0	26	8	65	58	458
	heuristic	25	0/0	13	4	47	58	456
	bfs	121	0/0	11	34	126	57	452
IteratorTest (LockFreeList)	dfs	38	0/0	38	10	71	52	446
	heuristic	50	0/0	21	5	55	51	440
	bfs	436	0/0	19	119	281	51	440
IteratorTest (LockFreeOrderedList)	dfs	38	0/0	38	10	65	52	458
	heuristic	50	0/0	21	6	51	51	452
	bfs	436	0/0	19	123	279	51	452
IteratorTest (LockFreePriorityQueue)	dfs	33	0/0	33	70	206	92	960
	heuristic	31	0/0	16	18	87	92	958
	bfs	214	0/0	14	384	383	91	954
IteratorTest (LockFreeQueue)	dfs	61	0/0	61	20	99	57	518
	heuristic	33	0/0	17	4	51	55	512
	bfs	830	0/0	13	368	432	54	508
IteratorTest (LockFreeSet)	dfs	53	0/0	53*	36685	2043	1138	2297
	heuristic	49	0/0	21	355	422	1141	2301
	bfs	347	0/0	>17*	19895	1999	1138	2297
QueueTest	dfs	14	0/0	14	3	35	44	314
	heuristic	28	0/0	15	3	41	44	314
	bfs	55	0/3	10	6	48	43	320

* JPF out of memory and JRF ended gracefully with result report

Table 6-6. Execution result of PreciseRaceDetector for examples from Table 6-5

example	state	length	time	mem	result
IteratorTest(EBDeque)	40	21	1	17	same race
IteratorTest(LockFreeDeque)	40	21	1	23	same race
IteratorTest(LockFreeList)	52	33	2	22	same race
IteratorTest(LockFreeOrderedList)	52	33	2	22	same race
IteratorTest(LockFreePriorityQueue)	56	27	1	28	same race
IteratorTest(LockFreeQueue)	280	54	3	28	same race
IteratorTest(LockFreeSet)	818	793	3	47	different race
QueueTest	77	20	1	16	same race

Specifically, JRF significantly outperforms PreciseRaceDetector in example `LockFreeSet` with counterexample length 21 comparing to 793.

6.3.2.2 Race Analysis in JRF-E

In this section, the experiment performed with counterexample analysis turned on. Table 6-7 are the experimental results with threshold races as 1, 10, and 100 excluding `LockFreeSet` since it ran out of memory with DFS strategy. Table 6-8 summarizes suggestions by JRF for each race found.

Table 6-7. Experimental results for [2] examples containing a race found by JRF-E. Results without thread local optimization using DFS and threshold as 1, 10, 100 are given.

example	threshold races	jpf-states	h-states pruned	max length	JRF time (sec)	JRF-E time (sec)	mem (MB)	volatiles, locks,threads	non-volatiles
IteratorTest (EBDeque)	1	26	0/0	26	33	0	139	91	564
	10	36	0/14	27	58	9	172	91	571
	100	47	0/22	28	81	14	228	91	582
IteratorTest (LockFreeDeque)	1	26	0/0	26	8	0	65	58	458
	10	36	0/14	27	14	7	80	58	465
	100	47	0/22	28	19	12	105	58	476
IteratorTest (LockFreeList)	1	38	0/0	38	10	0	66	52	446
	10	48	0/14	39	14	5	92	52	453
	100	59	0/22	40	18	8	95	52	464
IteratorTest (LockFreeOrderedList)	1	38	0/0	38	10	0	65	52	458
	10	48	0/14	39	15	5	99	52	465
	100	59	0/22	40	19	9	98	52	476
IteratorTest (LockFreePriorityQueue)	1	33	0/0	33	89	22	227	96	995
	10	47	0/20	34	137	159	248	93	978
	100	64	0/38	35	187	252	260	92	978
IteratorTest (LockFreeQueue)	1	61	0/0	61	20	0	102	57	518
	10	76	0/6	71	27	5	121	63	533
	100	126	0/146	71	76	58	238	63	533
QueueTest	1	14	0/0	14	2	0	39	44	314
	10	31	0/16	22	7	0	56	45	335
	100	150	54/302	47	43	6	130	47	355

Table 6-8. JRF-E suggestions from counterexample and acquiring history analysis for [2] examples with a race

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

*__bold__ entry indicates the most appropriate solution.

6.4 Google Concurrent Package

The next target of our experiment was two concurrent data structure packages from [3, 64].

6.4.1 Concurrent Adt Experiment Framework

[64] is a framework for an experiment of lock-free and wait-free concurrent data structures. The main goal of this framework is a performance comparison of each concurrent abstract data type; ConcurrentStack, IlConcurrentStack, IlConcurrentStack_v2, and ConcurrentLinkedQueue. The test framework is composed of multiple Producers and a Consumer. `ConcurrentStack` is a simple singly linked list with stack top saved using an `java.util.concurrent.atomic.AtomicReference` wrapper. `ConcurrnetLinkedQueue` is an singly linked list with two `java.util.concurrent.atomic.AtomicReference` wrapped pointers, `head` and `tail`. One important difference between it and `ConcurrentStack` is the use of `java.util.concurrent.atomic.AtomicReference` wrapped pointer for the linked list. Unlike a stack with only one pointer, concurrent accesses to a queue updates two pointers at the same time, and this requires atomic update of both pointers and their internal links. `IlConcurrentStack` and `IlConcurrentStack_v2`

uses the status field. The status of a node is either stable or unstable, and an unstable node needs help from the following stack operations.The status field has a `java.util.concurrent.atomic.AtomicReference` wrapper and forms a happens-before relation among accesses.

None of the four tests found any race, but did find one `NullPointerException` in `ConcurrnetLinkedQueue`.

6.4.2 Google Concurrent Data Structures Workshop Barrier

The second structure is from [3], a barrier implementation from a concurrent programming workshop by Roei Raviv and Jonathan Seroussi. It has implementations of 12 barriers: `simpleBarrier`, `senseBarrier`, `treeBarrier`, `staticTreeBarrier`, `lockBarrier`, `cyclicBarrier`, `senseBarrierWithWait`, `linearSenseBarrier`, `linearSenseBarrierVolatile`, `linearSenseBarrierVolatileWithBackoff`, `binaryStaticTreeBarrier`, and `splittedSenseBarrier`.

```
public class SyncCounterThread extends Thread {
    private CounterWithBarrier _syncedCounter;

    public SyncCounterThread(CounterWithBarrier syncedCounter) {
        _syncedCounter = syncedCounter;
    }

    @Override
    public void run() {
        int cachedVal;
        int countTo = Main.getCountTo();
        for (int iterNum = 0; iterNum < countTo ; iterNum++) {
            cachedVal = _syncedCounter.getValue();
            _syncedCounter.await(); // all threads read the same value.
            cachedVal++;
            _syncedCounter.setValue(cachedVal);
        }
    }
```

Figure 6-8. SyncCounterThread using a barrier with a non-volatile value field has an race on value field since the update by a thread after barrier cannot be overwritten by another thread without causing a WW race

Unfortunately, 8 of 12 barrier implementations had a race on accessing shared value of counter. As shown in Figure 6-8, a barrier cannot be used to order this type of RW sequences since a WW race exists.

The simplest barrier implementation in Figure 6-9 also has a problem. Since the `size` in $s3$ is reset for next use right after the barrier condition satisfied ($s2$), another thread that is spin at $s5$ may miss it. As an example, two threads $T1$ and $T2$ call `await()` to reach a barrier, and an execution sequence $(T1 : s1)(T2 : s1, s2, s3)(T1 : s4, s5)$ will make $T2$ spin at $s5$ forever even though the barrier is already reached.

Another problem JRF found is the use of volatile array. An array declared with a volatile keyword does not guarantee volatile semantics in accessing its elements. Instead, `java.util.concurrent.atomic` package replaces `volatile` array element semantics. `LinearSenseBarrierVolatile` extend `BaseLinearSenseBarrierVolatile`, which has a field `_threadDoneArray` declared to be `private volatile boolean _threadDoneArray[]`. In `LinearSenseBarrier` the same field is declared even without `volatile`, and also has a race.

```
public class SimpleBarrier implements Barrier {
    AtomicInteger count;
    int size;

    public SimpleBarrier(int n) {
        this.count = new AtomicInteger(n);
        this.size = n;
    }

    @Override
    public void await() {
        int position = count.getAndDecrement();          /*s1*/
        if (position == 1) { // If I'm last ...          /*s2*/
            count.set(size);  // reset for next use   /*s3*/
        } else {  // otherwise spin           /*s4*/
            while (count.get() != 0) {}                  /*s5*/
        }
    }
}
```

Figure 6-9. SimpleBarrier using an AtomicInteger is not working properly since the barrier is broken

6.4.3 Experimental Results

6.4.3.1 Race Detection in JRF

The experimental results of [3] ten barrier tests are given in Table 6-9. In this case, heuristic search finds seven shorter counterexamples out of ten examples and all BFS searches are failed to find a race.

Table 6-10 summarizes the results of JPF PreciseRaceDetector for the examples in Table 6-9. In seven out of ten examples, PreciseRaceDetector finds false races on volatile fields and misses two races on volatile array element accesses.

Table 6-9. Experimental results for [3] examples containing a race found by JRF. Results without thread local optimization for DFS, heuristic search, and BFS are given.

example	search	jpf-states	h-states pruned	length	time (sec)	mem (MB)	volatiles, locks,threads	non-volatiles
LinearSenseBarrierVolatile	dfs	39	0/28	39	6	45	43	372
	heuristic	101	0/0	39	6	61	43	368
	bfs	9890	192/461	>17*	2195	1985	39	356
LinearSenseBarrier	dfs	40	0/28	40	6	49	44	400
	heuristic	104	0/0	40	6	58	44	412
	bfs	9682	260/758	>15*	6360	1983	40	376
SimpleBarrier	dfs	36	1/33	29	5	45	38	335
	heuristic	53	0/0	21	3	36	34	337
	bfs	11939	252/655	>16*	6502	1986	36	341
SenseBarrier	dfs	48	0/38	48	7	65	41	362
	heuristic	89	0/0	37	5	55	41	376
	bfs	10606	222/451	>15*	2073	1990	37	346
SenseBarrierWithWait**	dfs	68	0/38	68	8	52	41	360
	heuristic	95	0/0	39	5	57	41	376
TreeBarrier	dfs	71	0/62	71	12	81	44	386
	heuristic	128	0/0	49	8	64	44	396
	bfs	9896	252/615	>15*	4932	1976	39	371
LockBarrier	dfs	34	0/0	34	7	52	58	406
	heuristic	101	0/0	35	10	77	54	407
	bfs	9374	219/447	>15*	7342	1993	49	400
CyclicBarrier	dfs	25	0/0	25	2	33	37	334
	heuristic	68	0/0	24	3	44	33	336
	bfs	13250	253/610	>16*	7285	1978	34	344
BinaryStaticTreeBarrier	dfs	60	0/38	60	10	78	47	391
	heuristic	135	0/0	52	10	70	47	403
	bfs	9328	213/551	>15*	6341	1981	42	378
SplittedSenseBarrier	dfs	71	0/60	71	14	91	49	383
	heuristic	121	0/0	50	10	93	49	396
	bfs	8450	205/514	>14*	6260	1984	43	371

* JPF out of memory and JRF ended gracefully with result report

** bfs algorithm ran out of Java heap space and JPF failed to report the result

Table 6-10. Execution result of PreciseRaceDetector for examples from Table 6-9

example	state	length	time	mem	result
LinearSenseBarrierVolatile	42	42	1	22	**false race on volatile field***
LinearSenseBarrier	44	44	1	22	**false race on volatile field***
SimpleBarrier	46	36	1	22	same race
SenseBarrier	28	28	1	23	false race on volatile field
SenseBarrierWithWait	38	38	1	22	false race on volatile field
TreeBarrier	44	44	1	23	false race on volatile field
LockBarrier	26	26	1	22	same race
CyclicBarrier	20	20	1	22	same race
BinaryStaticTreeBarrier	43	43	1	22	false race on volatile field
SplittedSenseBarrier	45	45	1	23	false race on volatile field

*The results in **boldface** are failure to detect a race on volatile array with non-volatile element accesses

6.4.3.2 Race Analysis in JRF-E

In this section, the experiment performed with counterexample analysis turned on.

Table 6-11 are the experimental results with threshold races as 1, 10, and 100 excluding

Table 6-11. Experimental results for [3] examples containing a race found by JRF-E. Results without thread local optimization using DFS and threshold as 1, 10, 100 are given.

example	threshold races	jpf-states	h-states pruned	max length	JRF time (sec)	JRF-E time (sec)	mem (MB)	volatiles, locks,threads	non-volatiles
LinearSenseBarrierVolatileTest	1	39	0/28	39	6	61	43	372	
	10	53	0/38	54	8	5	66	43	372
	100	277	61/661	67	48	58	151	43	394
LinearSenseBarrier	1	40	0/28	40	6	48	44	400	
	10	56	0/38	57	9	3	67	44	424
	100	303	63/711	72	60	37	176	44	514
SimpleBarrier*	1	36	1/33	29	5	44	38	335	
SenseBarrier	1	48	0/38	48	7	57	41	362	
	10	57	5/69	49	9	3	58	41	362
	100	409	89/1009	50	71	32	206	41	380
SenseBarrierWithWaitTest	1	68	0/38	68	8	54	41	360	
	10	79	11/107	69	10	6	69	41	360
	100	522	195/1519	70	83	63	180	41	378
TreeBarrier	1	71	0/62	71	12	80	44	386	
	10	80	8/102	72	14	2	74	44	386
	100	181	250/1032	72	58	24	215	44	386
LockBarrier	1	34	0/0	34	6	65	58	406	
	10	40	0/6	37	7	6	77	58	406
	100	140	77/365	43	60	67	176	62	407
CyclicBarrier	1	25	0/0	25	2	25	37	334	
	10	41	0/14	31	3	1	58	37	334
	100	430	51/845	36	36	20	162	38	346
BinaryStaticTreeBarrier	1	60	0/38	60	10	66	47	391	
	10	87	0/82	78	16	5	88	47	391
	100	929	283/2601	78	213	48	263	52	483
SplittedSenseBarrier	1	71	0/60	71	14	72	49	383	
	10	80	14/118	72	18	8	82	49	383
	100	175	258/1046	72	73	73	171	49	383

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

Table 6-12. JRF-E suggestions from counterexample and acquiring history analysis for [3] examples with a race

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

*__bold__ entry indicates the most appropriate solution.

`LockFreeSet` since it ran out of memory with DFS strategy. Table 6-12 summarizes suggestions by JRF for each race found.

6.5 Java Grande Forum Test Suite

Java Grande Forum benchmark suite from [4] is widely used in the experiment of several literature research projects. Since the benchmark originally aims to measure the performance of ”grande” applications which require intensive memory or computing power, it cannot be used in a model checking technique as it is. To model check those programs included in Java Grande Forum benchmark multithreaded benchmark version 1.0 [4], we dramatically reduced the number of threads, number of iterations, and size of data in our experiment.

First section of [4] test suites are for testing low level operation such as fork-join of threads and synchronization using a barrier or a synchronized block. *JGFForkJoinBench* tests the performance of thread forking and joining, and each thread has no interactions, so no race found. *JGFBarrierBench* aims to measure the performance of barrier synchronization. It used a lock free *TournamentBarrier* to synchronize the partial computation threads. Volatile array, causes a race on its element access, is used as *IsDone[]* of which each element marks the arrival of the corresponding thread.

```
class CounterClass {
    int shared_cont;
}
class SyncObjectRunner implements Runnable {
    int id,size;
    CounterClass cont;

    public SyncObjectRunner(int id, CounterClass cont, int size) {
        this.id = id;
        this.cont=cont;
        this.size=size;
    }

    public void run() {
        for(int i=0;i<size;i++)
            synchronized(cont) { cont.shared_cont++; }

        // prevent dead code elimination
        if(cont.shared_cont <= 0)
            System.out.println("Benchmark exited with unrealistic counter value "
                                + cont.shared_cont);
    }
}
```

Figure 6-10. Sync benchmark code fragment with a race on *shared_cont*

Third benchmark in this category is *JGFSyncBench* to measure the performance of synchronized method and synchronized block. As shown in 6-10, multithreads share a non-volatile integer counter but access through synchronization, but the code has an unsynchronized part to check if the counter has a reasonable value at the end.

Next section is programs to test computation kernels. *JGFSeriesBench* computes the Fourier coefficients of $f(x) = (x+1)^x$. The data array is divided into independent blocks and distributed among threads, so no sharing of data. JGFLUFactBench benchmark is for LU factorization: transform a matrix into an upper triangle form.

It uses the same *TournamentBarrier* as *JGFBarrierBench*, and has the same race on *IsDone[]* array. *JGFSORBench* performs successive over-relaxation on a N*N grid. It also uses volatile two dimensional array *sync[][]* to communicate with other threads. As *JGFSeriesBench*, *JGFCryptBench*, executes IDEA (International Data Encryption Algorithm) encryption and decryption on an array, has independent computations

```
public class Universal {
    ...
    private static boolean UNIVERSAL_DEBUG;
    ...
    public Universal() {
        super();
        this.DEBUG=true;
        this.UNIVERSAL_DEBUG=true;
        this.prompt="Universal> ";
    }
    ...
}
```

Figure 6-11. MonteCarlo benchmark code fragment with a race on static field *UNIVERSAL_DEBUG*

distributed into threads and no sharing of data involved. The last benchmark is *JGF-SparseMatmult*, computes a matrix multiplication stored in a sparse matrix stored in compressed row format. The computation is distributed uniformly into multithreads, and result vector sharing is avoided using sorted array.

Final section has large scale applications. *JGFMolDynBench* is a dynamic simulation of molecular particles. Each thread in this benchmark generate particles, calculate velocities of moves, move particles and update forces, and then compute full potential energy after barrier to synchronize all computations. Since the original program was written in Fortran, the benchmark is not programmed with an object oriented concept. As *JGFLUFactBench*, volatile array *IsDone[]* in *TournamentBarrier* is involved in a race. *JGFMonteCarloBench* uses a Monte Carlo algorithm in a financial simulation. It has a base class called *Universal* used as a centralized repository for all the functionalities of monte carlo algorithm, but unfortunately, shared a static field *UNIVERSAL_DEBUG* among threads without any protection as given in Figure 6-11.[3]

[3] This is an example of benign race since the sharing of the field is ”redundant update” that does not affect the semantics of the program.

6.5.1 Experimental Results

6.5.1.1 Race Detection in JRF

The experimental results of [4] ten barrier tests are given in Table 6-13. In this case, heuristic search finds four shorter counterexamples out of six examples. As we can deduce from the Table 6-13, the more states are visited before a race detection, the shorter counterexamples are found by heuristic search comparing to DFS.

Table 6-14 summarizes the results of JPF PreciseRaceDetector for the examples in Table 6-13. In four out of six examples, PreciseRaceDetector misses a race on a volatile array element.

6.5.1.2 Race Analysis in JRF-E

In this section, the experiment performed with counterexample analysis turned on. Table 6-15 are the experimental results with threshold races as 1, 10, and 100 excluding `LockFreeSet` since it ran out of memory with DFS strategy. Table 6-16 summarizes suggestions by JRF for each race found.

Table 6-13. Experimental results for [4] examples containing a race found by JRF. Results without thread local optimization for DFS, heuristic search, and BFS are given.

example	search	jpf-states	h-states pruned	length	time (sec)	mem (MB)	volatiles, locks,threads	non-volatiles
BarrierBench	dfs	87	0/18	87	13	79	45	474
	heuristic	139	0/0	72	11	87	45	474
	bfs	1561	0/0	32	328	428	43	446
SyncBench	dfs	111	0/10	103	17	105	48	507
	heuristic	40	0/0	22	2	27	37	251
	bfs	337	0/1	13	17	126	37	252
lufact	dfs	34	0/18	34	4	39	35	274
	heuristic	34	0/0	19	2	34	35	274
	bfs	92	0/0	16	6	56	35	274
sor	dfs	15	0/6	15	8	52	42	690
	heuristic	68	0/0	35	8	74	42	689
	bfs	25	0/0	7	6	42	42	689
moldyn*	dfs	2821	0/1874	2821	598	719	37	521
	heuristic	1896	0/0	950	279	617	37	575
montecarlo	dfs	86	0/0	86	36	174	64	919
	heuristic	178	0/0	90	23	144	64	998
	bfs	151	0/20	17	30	119	52	609

* bfs algorithm ran out of Java heap space and JPF failed to report the result

Table 6-14. Execution result of PreciseRaceDetector for examples from Table 6-13

example	state	length	time	mem	result
BarrierBench	48853	0	28	51	**none***
SyncBench	109	102	1	22	same race
lufact	3583	0	3	34	**none***
sor	2275	0	3	34	**none***
moldyn	14989480	0	7322	1139	**none***
montecarlo	886	87	3	35	same race

*The results in **boldface** are failure to detect a race on volatile array with non-volatile element accesses

Table 6-15. Experimental results for [4] examples containing a race found by JRF-E. Results without thread local optimization using DFS and threshold as 1, 10, 100 are given.

example	threshold races	jpf-states	h-states pruned	max length	JRF time (sec)	JRF-E time (sec)	mem (MB)	volatiles, locks,threads	non-volatiles
BarrierBench	1	87	0/18	87	13	0	103	45	474
	10	128	0/74	102	27	2	137	45	535
	100	1667	0/4650	102	637	22	284	45	556
SyncBench	1	111	0/10	103	18	0	115	48	507
	10	131	0/26	113	25	0	128	48	507
	100	572	94/1500	113	229	5	273	48	507
lufact	1	34	0/18	34	4	0	39	35	274
	10	72	0/30	72	7	1	49	35	276
	100	331	0/152	331	29	64	156	35	300
sor	1	15	0/6	15	8	0	57	42	690
	10	77	0/66	77	43	13	130	42	691
	100	159	0/142	159	78	288	222	42	691
moldyn	1	2821	0/1874	2821	638	32	723	37	521
	10	2861	0/1888	2861	663	295	725	37	521
	100	3136	0/2064	3136	697	3621	754	37	521
montecarlo	1	86	0/0	86	38	2	181	64	919
	10	91	0/4	88	50	19	190	64	984
	100	166	1/77	125	239	202	297	66	1103

Table 6-16. JRF-E suggestions from counterexample and acquiring history analysis for [4] examples with a race

test suite	**race field** of class	**analysis**
BarrierBench	IsDone[] of TournamentBarrier	**use atomic array for IsDone[]**
SynchBench	shared_count of CounterClass	**make shared_count volatile**
lufact	IsDone[] of TournamentBarrier	**use atomic array for IsDone[]**
sor	sync[] of SOR	**use atomic array for sync[]**
	A[][] of RandomMatrix	**use atomic array for** A[][], **sync[]**
moldyn	IsDone[] of TournamentBarrier	**use atomic array for IsDone[]**
montecarlo	UNIVERSAL_DEBUG of Universal	make UNIVERSAL_DEBUG volatile*

bold entry indicates the most appropriate solution.
* This is a benign race in redundant writes.

CHAPTER 7
DISCUSSION

7.1 Performance

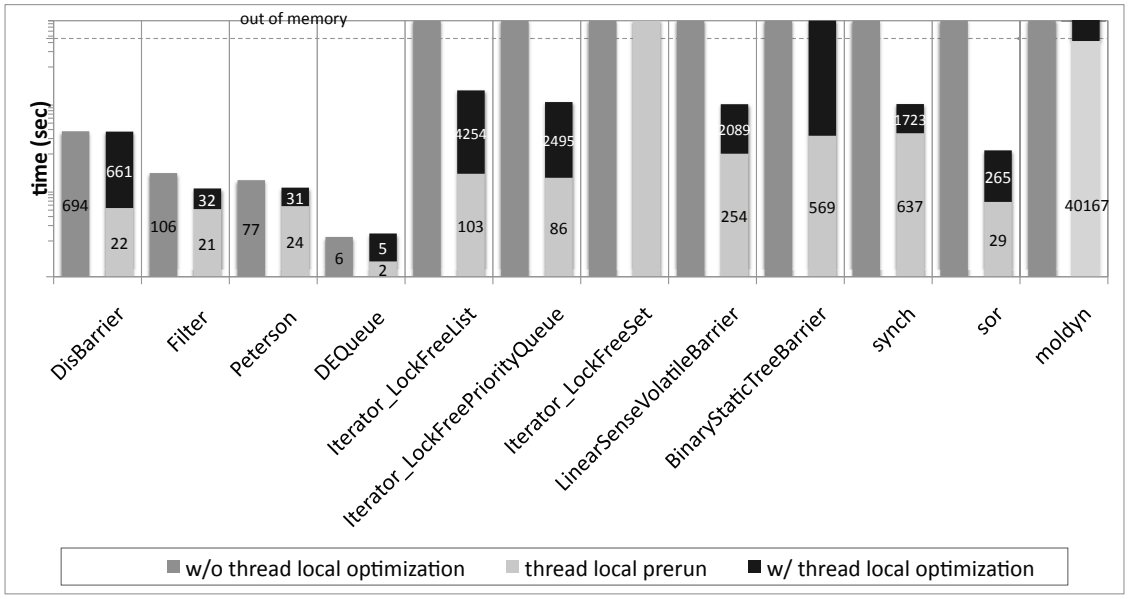
To compare the performance of various configuration of JRF, we used selected set of examples described in Chapter 6. They are `DisBarrier`, `Filter`, `Peterson`, and `DEQueue` from Section 6.2, `LockFreeList`, `LockFreePriorityQueue`, and `LockFreeSet` from Section 6.3, `LinearSenseVolatileBarrier` and `BinaryStaticTreeBarrier` from Section 6.4, and `synch`, `sor`, and `moldyn` from Section 6.5.

7.1.1 Threadlocal Optimization

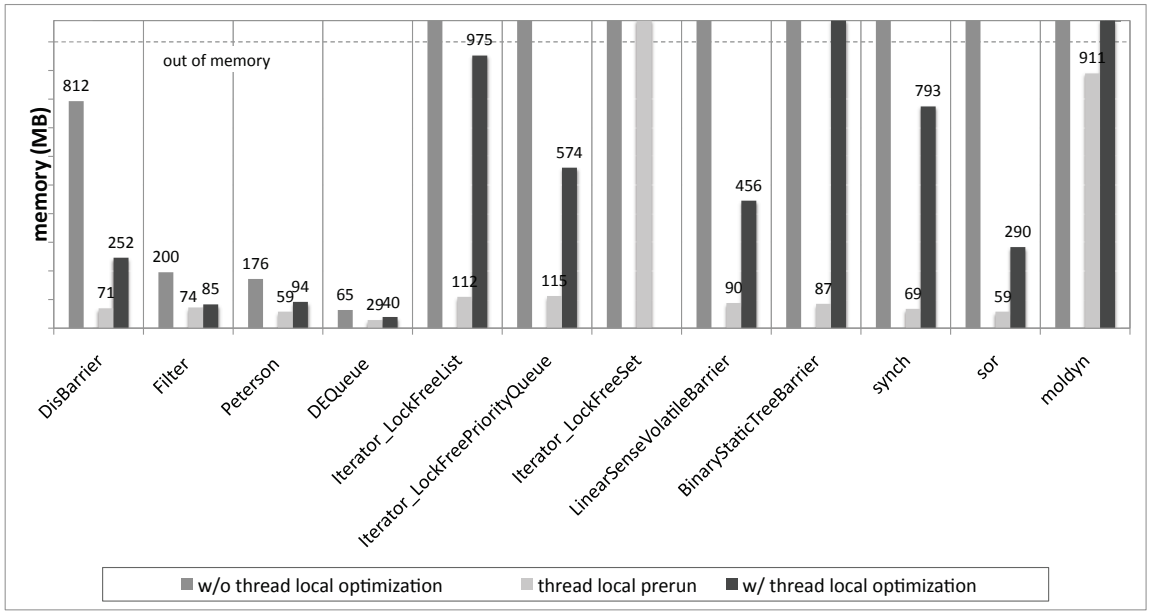
Figures 7-1 compares the performance with and without the optimization of thread local optimization described in Section 4.3.8. We tested for the examples with races using DFS strategy and configured JRF to explore the full search space to find all presenting races rather than to stop at the first race detection. The results show that significant improvements in both execution time and memory requirements can be obtained when information about which variables are thread local is available. For the optimized version, information about thread locality was obtained by a prerun of JPF, with the only modification to the standard distribution being to save the sharing information. In several examples, the prerun ran out of memory before terminating. Since programmers will typically want to use JPF to eliminate normal programming errors before using JRF to detect data races, this is a reasonable approach. Other possibilities will be explored in future work. With threadlocal optimization, JRF successfully found all races in some of examples previously failed to get result due to the out of memory: `Iterator_LockFreeList`, `Iterator_LockFreePriorityQueue`, `LinearSenseVolatileBarrier`, `synch`, and `sor`.

7.1.2 Heuristic Search

In order to evaluate the effectiveness of the ”knowledge” of data races built into the heuristics, we also compared the heuristic strategy with a random choice strategy. The



A execution time



B memory consumption

Figure 7-1. Comparison of the JRF results with/without thread local optimization when DFS strategy is configured to find all races by exploring the full search space

random choice strategy will also fall between DFS and BFS. Figure 7-2 compares the memory, time, and path length of heuristic with a random choice. In the case of random, the line shows the average value obtained in 100 trials with the standard deviation

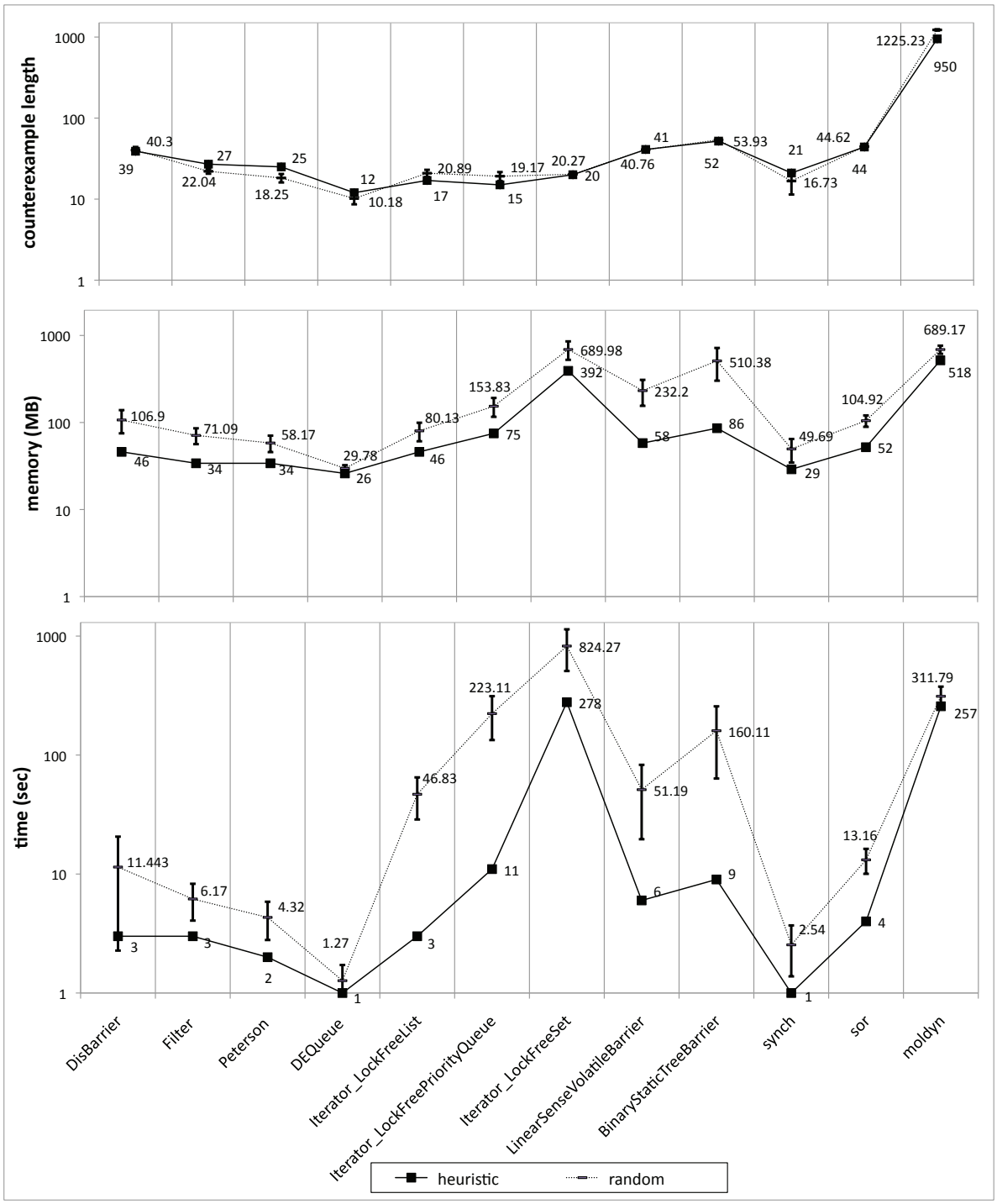
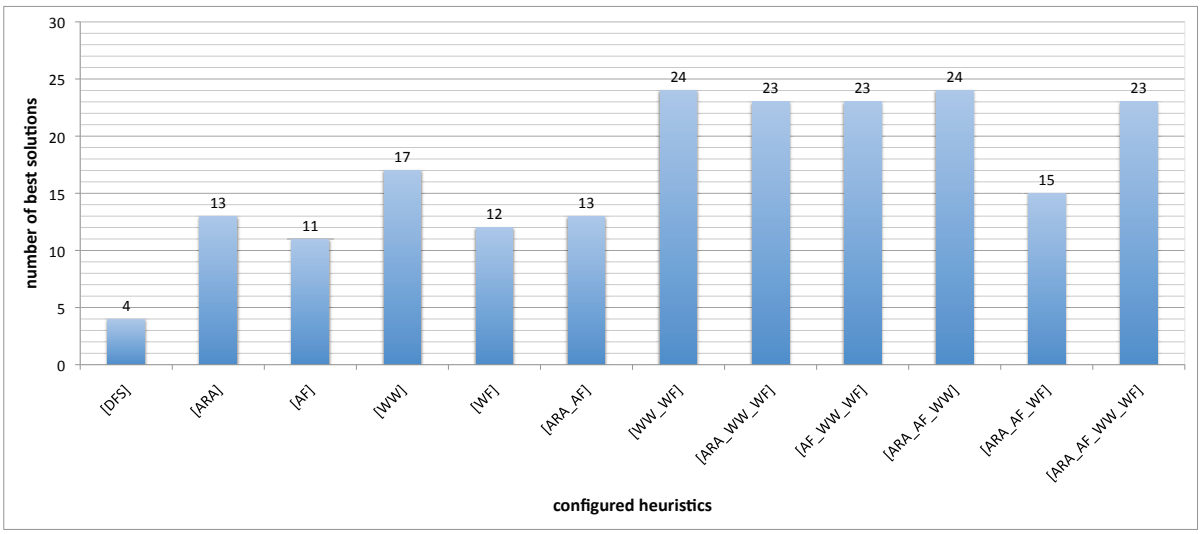
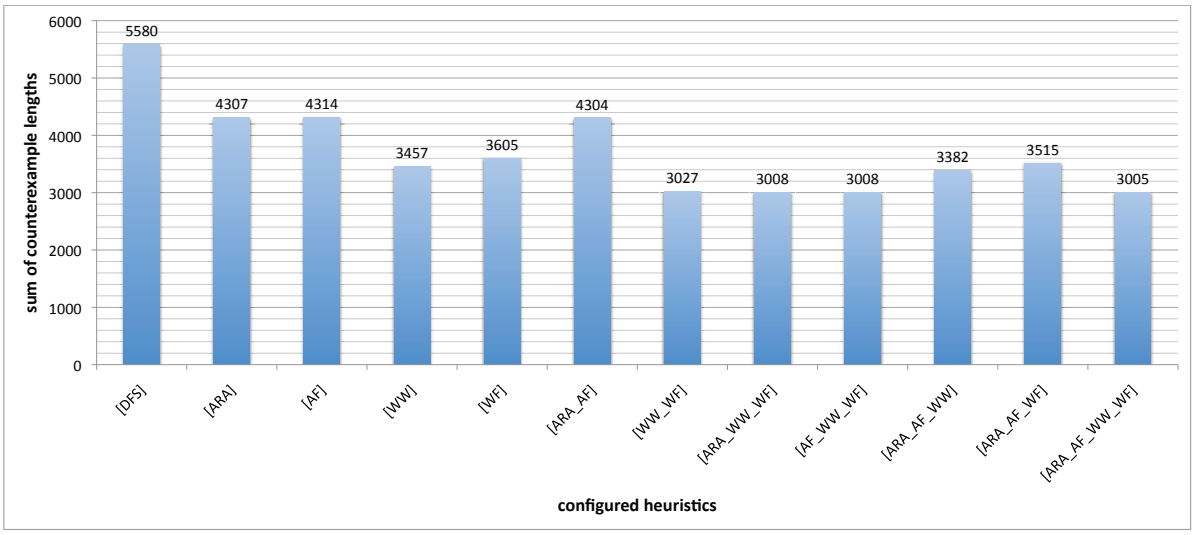


Figure 7-2. Comparison of the heuristic with a random search strategy

indicated. These results do not show a significant benefit for the heuristic, but most of the time the heuristic is somewhat better than random in the quality of the result (shorter counterexample path length) with noticeable saving in memory and times used.



A the number of best solutions (solutions with the shortest counterexample for the test case) per each heuristic congifuration



B the sum of all counterexample lengths per each heuristic configuration

Figure 7-3. Comparison of different heuristic configurations

Recall that several heuristics were presented in Section 4.2.1. We tested various configurations of the heuristic choices and the results show that using all four heuristics or (WW,WF) perform best for most cases in terms of the counterexample length: (WW,WF) configuration gives 24 best solutions for our 41 test cases, which is one more than (ARA,AF,WF,WW), but gives longest path for once case, `moldyn`. The experimental results comparing different heuristic configurations are given in Figure 7-3. Fig 7-3A shows which heuristic configuration found the most best solutions when the shortest counterexample was chosen as the best solution. Fig 7-3B shows the sum of all counterexample lengths for all race examples we had tested. The (ARA,AF,WF,WW) configuration was used for the heuristic results in Figure 7-2 and Table 6-1, 6-5, 6-9, and 6-13.

7.1.3 Modular Extension

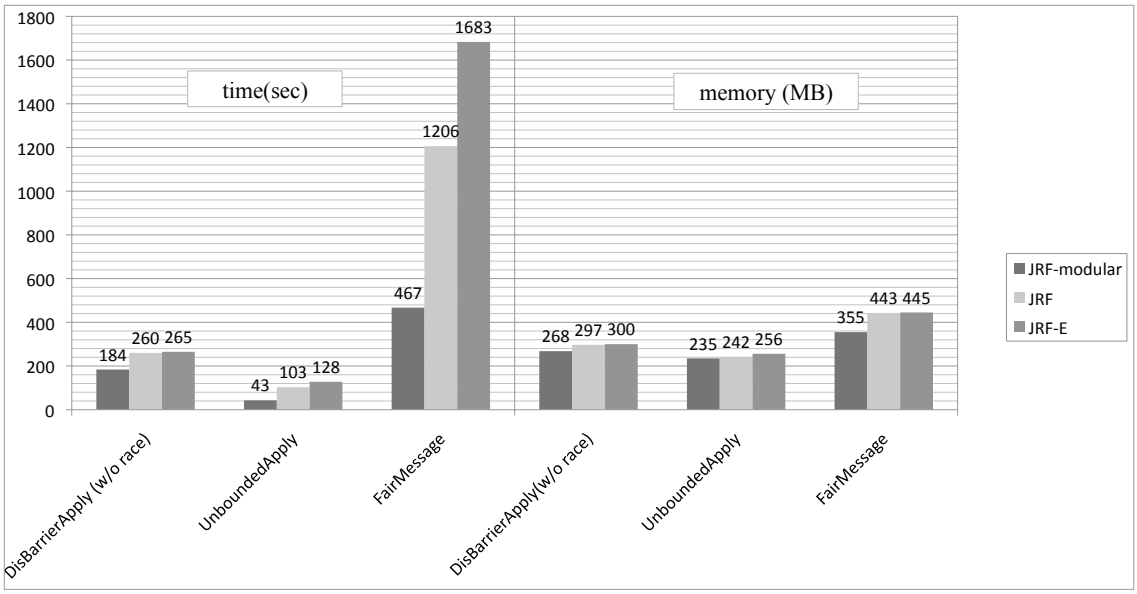


Figure 7-4. The comparison of JRF and JRF modular extension for different configuration of `FairMessage`

In this section, we briefly show the experimental result of JRF modular extension using `FairMessage` in Section 5.2. During the modular extension verification step (step 3) with `DisBarrier`, we identified the unsafe publication of `flag` in `Node` constructor and correct it using `this.flag = flag;`. This had been unidentified using JRF since the concrete environment we had used happen to have a happens-before order there. In Figure 7-4, JRF with modular extension outperforms JRF. `UnboundedQueueApply` and `DisBarrierApply` are the junit driver which we used in Section 6.2. The libraries `UnboundedQueue` and `DisBarrier` in Section 5.2 are corrected according to the suggestion in Table 6-4. The result shows that for all three cases, JRF modular extension outperforms JRF. In addition, `FairMessage` saves more time and memory than `UnboundedQueueApply` and `DisBarrierApply` since it uses both libraries and the savings are compositional. The number of states generated by JPF remains the same as we expected but total number of states visited in JRF modular extension is less than JRF because of the smaller h in all cases. It is clear that the message from JRF modular extension for `FairMessage` in Figure 5-26 in Section 5.2 is easier to understand than the message from JRF or JRF-E.

```
=====================================================
JRF results
===================================================== data race #1
edu.ufl.cise.jrf.util.HBDataRaceException
         at THREAD      (java.lang.Thread@ from null)
         to MEMORY      (jrfm.UnboundedQueue@.tail from "volatile UnboundedQueue queue = new UnboundedQueue();"
                                                  at jrfm/FairMessage.java:10 in (<init>))
         in INSTRUCTION (getfield)
         of SOURCE      ("for (Node tmp=head.next; tmp != null && tmp != tail ; tmp=tmp.next, ++i);"
                                                  at jrfm/UnboundedQueue.java:72)
. . .
=====================================================
JRF-E results
----------------------------------------------------- analyze counter example
data race source statement : "putfield" at jrfm/UnboundedQueue.java:58  : "tail = e;" by thread 1
data race manifest statement : "getfield" at jrfm/UnboundedQueue.java:72:
                       "for (Node tmp=head.next; tmp != null && tmp != tail ; tmp=tmp.next, ++i);" by thread 0

Change the field "jrfm.UnboundedQueue@.tail from "volatile UnboundedQueue queue = new UnboundedQueue();"
         at jrfm/FairMessage.java:10 in (<init>)" to volatile.

    Lock "java.util.concurrent.locks.ReentrantLock@ from "enqLock = new ReentrantLock();"
             at jrfm/UnboundedQueue.java:25 in (<init>)"
             before accessing (jrfm.UnboundedQueue@.tail)
. . .
```

7.2 Overhead

In order to show the overhead of JRF compared with standard JPF, we used the same examples except `Iterator_LockFreeSet` which runs out of memory and ran original JPF with a slight change to make it stop at the same state as JRF. Figure 7-5 represents the time and memory consumed by original JPF, JRF without threadlocal optimization, and JRF with threadlocal optimization respectively. The default of DFS was used for all cases.

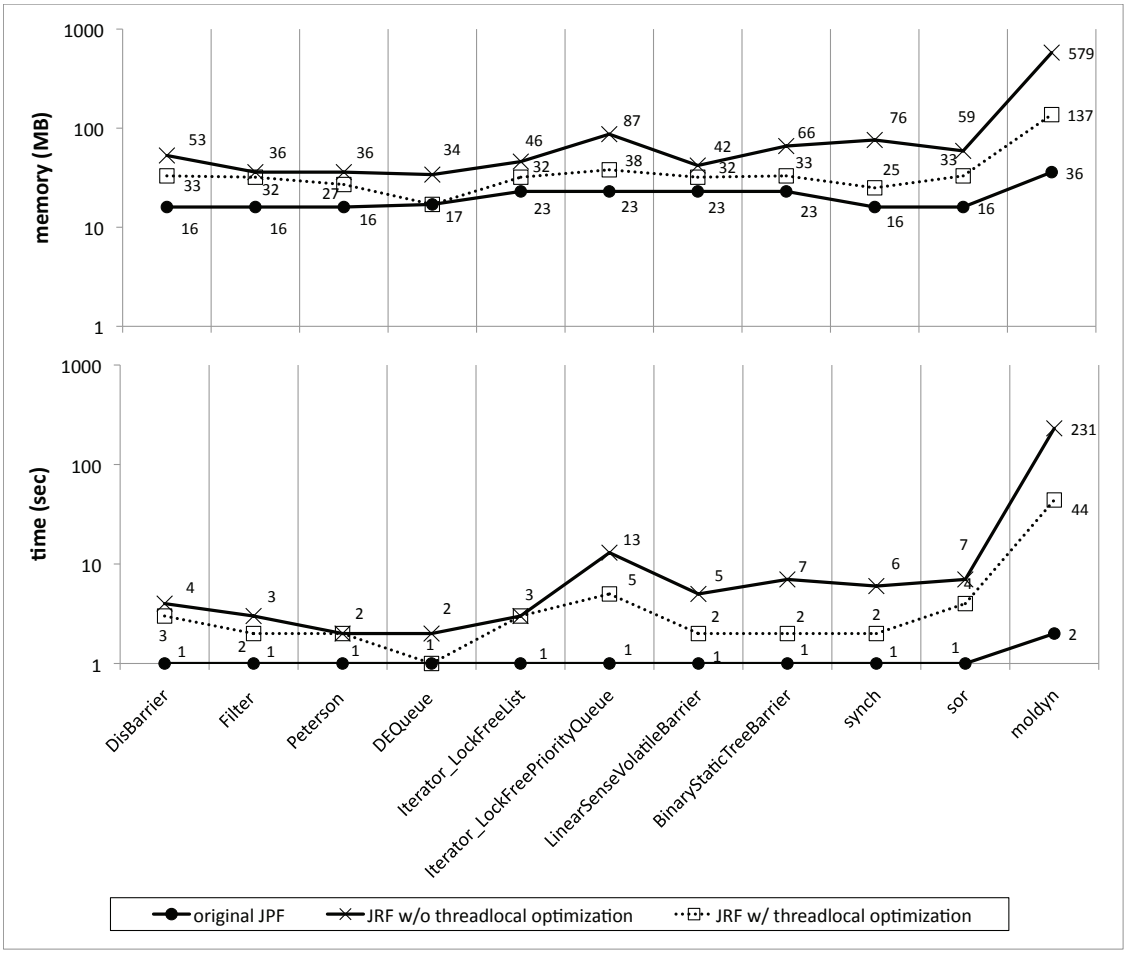


Figure 7-5. Comparison of JRF with/without thread local optimization with original JPF when DFS strategy is configure and forced to stop at the state where JRF find the first race

CHAPTER 8
CONCLUSION

A multithreaded concurrent system is the main programming environment in modern computer architecture and GUI-based applications make it inevitable that we will use multiple threads interacting with each other through shared memories. This puts emphasis on the verification of a concurrent program, but unfortunately there is no existing approach to precisely address this problem. Even the existing analysis tools used to reason about concurrent program behavior are based on a false assumption about sequential consistency, which is ideal but not realistic. It is very difficult to understand a memory model precisely, but the consequence of misunderstanding can be devastating to program correctness.

In this dissertation, we have addressed this problem and provided a method to help programmers find concurrency bugs related to a relaxed memory model. First, we proved that a program without a data race could use existing model checkers to verify its properties. Upon this result, we reduced the problems of a relaxed memory model and a sequential consistency issue into a data race detection problem. We have described an approach based on maintaining a function summarizing the happens-before relation that can be used in a model checker to precisely detect data races. We implemented this into an existing JPF model checker and suggested several techniques to make it more efficient. In addition, we introduced new search heuristics based on a careful analysis of data races that leads to shorter and more easily understood counterexample paths.

Second, we developed techniques to analyze the path that finds a race to make it easier to identify what the problem is, and also to suggest modifications to the code to eliminate races. Without a thorough understanding of the memory model, it is even harder to correct the program with a data race than to find one. Our suggestions are precise and consistent since they based on the analysis of the happens-before relations and information gathered from paths without a race. We have identified important

concurrency bug patterns during the case studies and this improved the analysis of races to provide better suggestions.

In addition, we addressed the problem of foreign code in a relaxed memory model. With restricted information about the internal implementation details, the programmer easily violates the consistent usage patterns. We provided a method to automatically check the consistency of foreign library invocations using preconditions. This approach was implemented using an assume-guarantee technique and decomposed the burden of race detection into smaller modules. The programmer of a foreign library module had a mechanism to check their preconditions' correctness and the user of this foreign code had a mechanism to check their consistency toward them.

The ideas have been implemented in JRF, an extension of JPF that detects data races. This is important, because standard JPF is unsound for programs that contain races. JRF has been shown to be useful in a wide range of important concurrent data structures. In contrast to most other approaches, JRF is precise and can deal with the wealth of synchronization actions in the Java programming language.

Our ideas are implemented in a Java relaxed memory model and its model checker JPF, but we believe that our approach can be applied to other memory models, such as C# and C++, without difficulty and left as a future work.

The main contributions of this work can be summarized as:

- A weaker requirement for a Java program to be SC.
- A summary function that captures the necessary happens-before relation along with a soundness proof.
- An efficient representation of the summary function.
- Data race-specific search heuristics.
- An explanation of a data race counterexample by identifying missing happens-before edges.
- Suggestions for code modifications to eliminate the data race by analyzing the prior acquiring history.

- Preconditions of a library module to ensure data race freedom of internal fields and an automatic tool to verify their correctness in a generated universal environment.
- A tool to check the preconditions at a library module invocation.
- A memory model-aware extension to JPF, thus making JPF sound for JMM and showing feasibility of extending it for new memory models.
- Sound analysis of lock-free and wait-free protocols, such as data race avoidance using any combination of intrinsic and extrinsic locks, volatile variables, join, barriers, compareAndSet operations, and transmitting values through concurrent data structures.
- Case studies of applying JRF to various concurrent programs from concurrent building blocks to the widely used Java development framework.

REFERENCES

[1] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008.

[2] "Amino concurrent building blocks," http://amino-cbbs.sourceforge.net/.

[3] "Google concurrent data structures workshop barriers," http://code.google.com/p/concurrent-data-structures-workshop-barriers/.

[4] "The Java Grande Forum benchmark suite," http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html.

[5] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, September 1979.

[6] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java Language Specification*, Addison Wesley, 3rd edition, 2005.

[7] ECMA International, *Standard ECMA-335 - Common Language Infrastructure (CLI)*, 4 edition, June 2006.

[8] H.-J. Boehm and S. V. Adve, "Foundations of the c++ concurrency memory model," *SIGPLAN Not.*, vol. 43, no. 6, pp. 68–78, 2008.

[9] G. Bronevetsky and B. R. de Supinski, "Complete formal specification of the openmp memory model," *Int. J. Parallel Program.*, vol. 35, no. 4, pp. 335–392, 2007.

[10] J. Manson, "Data-race-ful lazy initialization for performance," December 2008.

[11] V. A. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun, "A theory of memory models," in *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, 2007, pp. 161–172, ACM Press.

[12] C. Flanagan and S. N. Freund, "Type-based race detection for Java," in *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, New York, NY, USA, 2000, pp. 219–232, ACM Press.

[13] D. F. Bacon, R. E. Strom, and A. Tarafdar, "Guava: A dialect of java without data races," in *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA*, 2000, pp. 382–400.

[14] M. Abadi, C. Flanagan, and S. N. Freund, "Types for safe locking: Static race detection for java," *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 2, pp. 207–255, 2006.

[15] K.-P. Lohr and M. Haustein, "Jac-minimizing the differences between concurrent and sequential java code," Tech. Rep. TR B-05-xx, FB Mathematik und Informatik, FU Berlin, 2005.

[16] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for Java," in *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2006, pp. 308–319, ACM Press.

[17] "Mobius consortium. deliverable d3.3: Preliminary report on thread-modular verification," March 2007, http://mobius.inria.fr.

[18] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.

[19] E. Bodden and K. Havelund, "Racer: effective race detection using aspectj," in *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, New York, NY, USA, 2008, pp. 155–166, ACM.

[20] T. Elmas, S. Qadeer, and S. Tasiran, "Goldilocks: a race and transaction-aware Java runtime," in *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation*, New York, NY, USA, 2007, pp. 245–255, ACM.

[21] S. Burckhardt, R. Alur, and M. M. K. Martin, "Checkfence: Checking consistency of concurrent data types on relaxed memory models," in *PLDI*, 2007.

[22] G. Gopalakrishnan, Y. Yang, and H. Sivaraj, "QB or not QB: An efficient execution verification tool for memory orderings," in *CAV LNCS 3114*, 2004.

[23] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind, "Nemos: A framework for axiomatic and executable specifications of memory consistency models," in *IPDPS*, 2004.

[24] S. Burckhardt and M. Musuvathi, "Effective program verification for relaxed memory models," in *CAV '08: Proceedings of the 20th international conference on Computer Aided Verification*, Berlin, Heidelberg, 2008, pp. 107–120, Springer-Verlag.

[25] A. De, A. Roychoudhury, and D. D'Souza, "Java memory model aware software validation," in *PASTE '08: Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, New York, NY, USA, 2008, pp. 8–14, ACM.

[26] T. Q. Huynh and A. Roychoudhury, "Memory model sensitive bytecode verification," *Form. Methods Syst. Des.*, vol. 31, no. 3, pp. 281–305, 2007.

[27] M. Musuvathi, S. Qadeer, and T. Ball, "Chess: A systematic testing tool for concurrent software," Tech. Rep. MSR-TR-2007-149, Microsoft Research, 2007.

[28] S. Basu, D. Saha, and S. A. Smolka, "Localizing program errors for cimple debugging," in *FORTE*, 2004, pp. 79–96.

[29] A. Groce and W. Visser, "What went wrong: Explaining counterexamples," in *SPIN Workshop on Model Checking of Software*. 2003, pp. 121–135, Springer.

[30] T. Ball, M. Naik, and S. K. Rajamani, "From symptom to cause: localizing errors in counterexample traces," in *Principles of Programming Languages*, 2003, pp. 97–105.

[31] A. Groce, "Error explanation with distance metrics," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2004, pp. 108–122.

[32] A. Groce, D. Kroening, and F. Lerda, "Understanding counterexamples with explain," in *Computer-Aided Verification*. 2004, pp. 453–456, Springer. LNCS.

[33] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *ASE*, 2003, pp. 30–39.

[34] H. Cleve and A. Zeller, "Locating causes of program failures," in *ICSE*, 2005, pp. 342–351.

[35] T. Wang and A. Roychoudhury, "Automated path generation for software fault localization," in *ASE*, 2005, pp. 347–351.

[36] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops," in *ASE*, 2005, pp. 263–272.

[37] Y. Brun and M. D. Ernst, "Finding latent code errors via machine learning over program executions," in *ICSE*, 2004, pp. 480–490.

[38] C. Flanagan, S. N. Freund, and J. Yi, "Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs," in *PLDI*, 2008, pp. 293–303.

[39] C. Flanagan, S. Qadeer, and S. A. Seshia, "A modular checker for multithreaded programs," in *In CAV 02: Computer Aided Verification*. 2002, pp. 180–194, Springer.

[40] C. Flanagan and S. Qadeer, "Assume-guarantee model checking," Tech. Rep., 2003.

[41] C. d. l. Riva, J. Tuya, and J. R. d. Diego, "Translating sa/rt models to synchronous reactive systems: An approximation to modular verification using the smv model checker," in *PSI '99: Proceedings of the Third International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, London, UK, 2000, pp. 493–502, Springer-Verlag.

[42] C. de la Riva and J. Tuya, "Modular model checking of software specifications with simultaneous environment generation," in *Automated Technology for Verification*

and Analysis, F. Wang, Ed., vol. 3299 of *Lecture Notes in Computer Science*, pp. 369–383. Springer Berlin / Heidelberg, 2004.

[43] C. Pasareanu, D. Giannakopoulou, M. Bobaru, J. Cobleigh, and H. Barringer, "Learning to divide and conquer: applying the l* algorithm to automate assume-guarantee reasoning," *Formal Methods in System Design*, vol. 32, pp. 175–205, 2008, 10.1007/s10703-008-0049-6.

[44] D. Giannakopoulou and C. S. Păsăreanu, "Interface generation and compositional verification in javapathfinder," in *FASE '09: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, Berlin, Heidelberg, 2009, pp. 94–108, Springer-Verlag.

[45] O. Tkachuk and M. B. Dwyer, "Adapting side effects analysis for modular program model checking," *SIGSOFT Softw. Eng. Notes*, vol. 28, no. 5, pp. 188–197, 2003.

[46] G. Hughes and T. Bultan, "Interface grammars for modular software model checking," in *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, New York, NY, USA, 2007, pp. 39–49, ACM.

[47] O. Tkachuk and M. B. D. O. Cis, "Automated environment generation for software model checking," in *In Proceedings of the 18th International Conference on Automated Software Engineering*, 2003, pp. 116–129.

[48] C. C. Loyola, C. Colby, P. Godefroid, and L. J. Jagadeesan, "Automatically closing open reactive programs," in *In Proceedings of 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1998, pp. 345–357, ACM Press.

[49] P. Parizek, J. Adamek, and T. Kalibera, "Automated construction of reasonable environment for java components," *Electron. Notes Theor. Comput. Sci.*, vol. 253, no. 1, pp. 145–160, 2009.

[50] C. Boyapati, R. Lee, and M. Rinard, "Ownership types for safe programming: Preventing data races and deadlocks," 2002.

[51] R. Agarwal and S. D. Stoller, "Type inference for parameterized race-free java," in *In Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation*. 2004, pp. 149–160, Springer-Verlag.

[52] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller, "Automated type-based analysis of data races and atomicity," in *Proc. ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming (PPoPP)*. June 2005, ACM Press.

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

[54] D. Aspinall and J. Sevcik, "Formalising Java's data-race-free guarantee," in *TPHOLs 2007 (LNCS)*. 2007, vol. 4732, pp. 22–37, Springer.

[55] E. M. C. Jr., O. Grumberg, and D. A. Peled, *Model Checking*, The MIT Press, 1999.

[56] E. A. Emerson, "The beginning of model checking: A personal perspective," pp. 27–45, 2008.

[57] P. Godefroid, "Partial-order methods for the verification of concurrent systems - an approach to the state-explosion problem," 1995.

[58] K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[59] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," 2003.

[60] "Java Pathfinder," http://javapathfinder.sourceforge.net/.

[61] "Indus," http://indus.projects.cis.ksu.edu/.

[62] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java Concurrency in Practice*, Addison Wesley Professional, 2006.

[63] O. Shalev and N. Shavit, "Split-ordered lists: lock-free extensible hash tables," *Journal of the ACM*, vol. 53, no. 3, pp. 379–405, 2006.

[64] "Concurrent adt experiment framework," http://code.google.com/p/concurrentadt/.

[65] T. Elmas, S. Qadeer, and S. Tasiran, "Precise race detection and efficient model checking using locksets," Tech. Rep. Technical report MSR-TR-2005-118, Microsoft Research, 2006.

[66] "Zing," http://research.microsoft.com/en-us/projects/Zing/.

[67] M. Herlihy, N. Shavit, and M. Tzafrir, "Concurrent cuckoo hashing," Tech. Rep., Brown University, 2007.

[68] T. Craig, "Building FIFO and priority-queueing spin locks from atomic swap," Tech. Rep. TR 93-02-02, Department of Computer Science, University of Washington, 1993.

[69] P. Magnussen, A. Landin, and E. Hagersten, "Queue locks on cache coherent multiprocessors," in *Proceedings of the Eighth International Symposium on Parallel Processing (IPPS)*, 1994, pp. 165–171.

BIOGRAPHICAL SKETCH

KyungHee Kim received her bachelor's degree in computer science and engineering at Pohang University of Science and Technology in Korea in 1995. She worked for the Samsung Electronics, Inc., Korea from 1995 to 1999. She was involved in the development of a next generation Realtime operating system, transporting pJava on top of pSOS realtime operating system, and the design of an execution unit for digital television. She also worked for Alticast Inc. in Korea in implementing the independent data server for the digital broadcasting system of Korean Broadcasting System (KBS). Since 2005, she has been conducting research with Dr. Beverly A. Sanders in Department of Computer and Information Science and Engineering at the University of Florida. Her research interests are concurrent programming, formal verification, static analysis, and software model checking.