A Tool for Analysis of Concurrent Programs

Raghav Karol

Kongens Lyngby 2006
IMM-MSC-2006-88

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-MSC: ISSN 1601-233X

Summary

Writing correct programs is a difficult task and this difficulty increases with the size and the impact of the program on its users. Writing even moderately complex concurrent programs is much more difficult because of the introduction of non-determinism. Most programming languages today support mechanisms that allow a programmer to use concurrency. In fact, all but the smallest software systems utilize concurrency and the requirement to write concurrent programs will increase in the future with the availability of multi-core processors.

Debugging and finding problems in concurrent programs is notoriously difficult because of non-determinism. Here, program analysis techniques can be of great value as they approximate program behavior and allow one to reason about a program without running it. This allows one to find potential problems at compile time, especially attractive for concurrent programs where a problem may manifest itself only for a particular execution of the program.

In this thesis we use program analysis techniques to develop a tool capable of analyzing real world Java programs for race conditions. Evaluation of our tool shows that it provides precise and useful information and can be used to increase the reliability of concurrent programs.

Preface

This thesis was prepared at the Informatics and Mathematical Modelling department, at the Technical University of Denmark in partial fulfillment of the requirements for acquiring an M.Sc., degree in Computer Systems Engineering.

The thesis deals with the use of static analysis based techniques for analyzing concurrent Java programs. In particular, we develop a tool capable of analyzing real world Java programs and reporting potential race conditions in them. Finding race conditions in concurrent programs is a difficult task and the output from our tool can increase the reliability of concurrent programs.

Lyngby, September 2006

Raghav Karol

Acknowledgements

First, I thank my parents for giving me the opportunity and encouraging me to pursue my interest in computer science. I know that not everyone is fortunate enough to have this.

During my first semester at the Technical University of Denmark, I took the course on Concurrent Systems taught by Hans Henrik Løvengreen; after the first few lectures I was convinced that he should be my master thesis supervisor.

I thank Hans Henrik for agreeing to supervise my thesis. Hans Henrik proposed the topic of this thesis and convinced Christian W. Probst to co-supervise my thesis. Working with Hans Henrik is challenging, but at the same time rewarding and great fun. He is a gifted teacher and I have learnt a great deal from him.

I thank Christian W. Probst for agreeing to co-supervise my thesis. Christian gave me good explanations on program analysis topics that I was not familiar with, as well as, many great ideas. Without Christian's advice, I doubt I would have got as far as this. Christian's thorough reading of the draft versions of this thesis found many errors that would have otherwise slipped into the final version.

I am fortunate to have had not one, but two excellent supervisors, both of whom are truly inspiring in their own ways.

I thank the author's of Soot, the framework our tool is built upon, for making Soot available; its functionality and API are awe inspiring. I would also like to thank Fernando Meira, for eye-balling the final version of my thesis and

providing good company during the last days of my thesis.

Last, I thank Tulika, my friend and wife to be, for all her patience and love. I know she does not like all the time I spend on the computer.

Contents

Summary i

Preface iii

Acknowledgements v

1 Introduction 1

1.1 Our Work 2

1.2 Thesis Organization 3

2 Background 5

2.1 Concurrency Theory 5

2.1.1 Applications and Programming Styles 6

2.1.2 Communication and Synchronization 7

2.1.3 A Closer Look at Multi-Threaded Programs 7

2.1.4 Race Conditions and Critical Sections 9

2.1.5 Explicit Atomicity Constructs 10
2.1.6 Deadlocks 13
2.1.7 Summary 13
2.2 Multi-Threading Facilities in Java 13
2.2.1 Creating and Starting Threads 13
2.2.2 Main Thread 14
2.2.3 Thread Priorities and Scheduling 14
2.2.4 Thread Control Methods 14
2.2.5 Synchronization and Monitors 15
2.2.6 Concurrent Programming Utilities in Java 1.5 17
2.3 Program Analysis 18
2.3.1 Lattice Theory 19
2.3.2 Dataflow analysis 21
2.3.3 Inter-procedural Analysis 23
2.3.4 Control Flow Analysis 25
2.3.5 Pointer Analysis 26
2.4 Soot 27
2.4.1 Intermediate Representations 28
2.4.2 Fundamental Soot Objects 31
2.4.3 Program analysis facilities 34
2.4.4 Obtaining more Information on Soot 37
2.5 Related Work 37

2.5.1 Dynamic Race Detection 38

2.5.2 Static Race Detection 39

3 Analysis and Design **43**

3.1 Analysis 43

3.1.1 Scope 43

3.1.2 Problem Decomposition 45

3.1.3 Data Flow for the Tool 46

3.1.4 Architecture 47

3.2 Design and Implementation 49

3.2.1 Choosing an Intermediate Representation 49

3.2.2 Call Graph Building 51

3.2.3 Locks Held Analysis 55

3.2.4 Threads and Memory Accesses 57

3.2.5 Points-to Analysis 59

3.2.6 Finding Race Conditions 66

3.2.7 Imprecision of Context-Insensitive Analysis 67

3.2.8 Implementing Context Sensitivity 68

3.2.9 Output Design 70

3.2.10 Summary 70

4 Evaluation **73**

4.1 Tools Evaluated 74

4.2 Benchmark Programs 75
4.3 Evaluating other Tools on Benchmarks 76
4.4 Motivation for Developing our own Tool 79
4.5 Evaluating our Tool 80
4.5.1 Testing and Validation 80
4.5.2 Comparison with other Tools 80
4.5.3 Analyzing Open Programs 83
4.6 Summary 86

5 Conclusion **87**

5.1 Achievements 87
5.2 Limitations 88
5.3 Applications 89
5.4 Future Work 89

A Benchmark programs **97**

A.1 Confinement 97
A.2 CopyOnWriteList 98
A.3 BoundedBuffer 99
A.4 ImmutableFraction 100
A.5 LazyInitialization 101
A.6 SimpleRace 102
A.6.1 SimpleRace with Race Conditions 102

A.6.2 SimpleRace with False Positives 103

A.7 CopyOnWriteList Annotated for Escjava2 104

A.8 Sample Output from Tool 105

A.8.1 Example Program 105

A.8.2 Results from Tool 107

Chapter 1

Introduction

Concurrent programming originated in the 1960s within the context of operating systems. The motivation was the invention of hardware units called *channels* or *device controllers*. These operate independently of a controlling processor and allow an I/O operation to be carried out concurrently with continued execution of program instructions by the central processor. A channel communicated with the processor by means of an *interrupt*, a signal to tell the processor to stop what it is doing and service the channel. The programming challenge introduced by the invention of the channels was that now parts of a program could be executed asynchronously and in an unpredictable order. Hence, if one part of a program is updating the value of a variable, an interrupt might cause another part of the program to change the value of this same variable.

Shortly after the invention of channels, hardware designers conceived the idea of multi-processor machines, which are now widely used and available. Multi-processor machines and support for multitasking in operating systems presented application programmers with the opportunity to write concurrent programs. Concurrent programs are widely used today and every major programming language has some support for them. In fact, all but the smallest software systems utilize concurrency, and the requirement to write concurrent programs will only increase in the future with the availability of multi-core processors.

Concurrent programs vary along two dimensions — address space and commu-

nication technique used. The address space may be distinct or shared, i.e., we may have multiple operating system *processes* or multiple *threads* within a single process. For communication, multi-threaded programs commonly use shared variables while programs with multiple processes use message passing.

Multi-threaded programs need some way to synchronize accesses to shared memory locations. Failure to synchronize accesses is often the violation of a *program invariant*. However, finding and reproducing such problems is notoriously difficult due to the non-deterministic nature under which they occur. As a result, tools to detect memory access and synchronization problems are of great value for improving the reliability of concurrent programs.

Existing tools fall under two major categories, *dynamic* and *static* tools. Dynamic tools instrument the program code to detect potential memory access problems in the actual execution of a program. These tools have high precision in results reported, but poor coverage as they only deal with one particular execution of the program. Static tools have better coverage as they approximate run time behavior of the program but may produce inaccurate results because of this approximation of program behavior. Despite significant advances in static tools, dynamics tools are still state of the art.

1.1 Our Work

In this project we develop a tool to statically detect potential memory access problems in multi-threaded Java programs. We identify some key properties that a static tool must possess in order to be useful:

1. **Precision.** Does the tool have an acceptable *false positive* rate?
2. **Synchronization Idioms.** Does the tool handle the most common synchronization idioms used in Java?
3. **Open Programs.** Is the tool able to handle programs where the caller or callee information is not available?
4. **Meaningful Output.** Does the output from the tool contain sufficient information to identify and fix a program bug due to incorrect or missing synchronization?

Before constructing our tool, we compared three freely available static analysis tools with capabilities to detect synchronization errors using a suite of "bench-

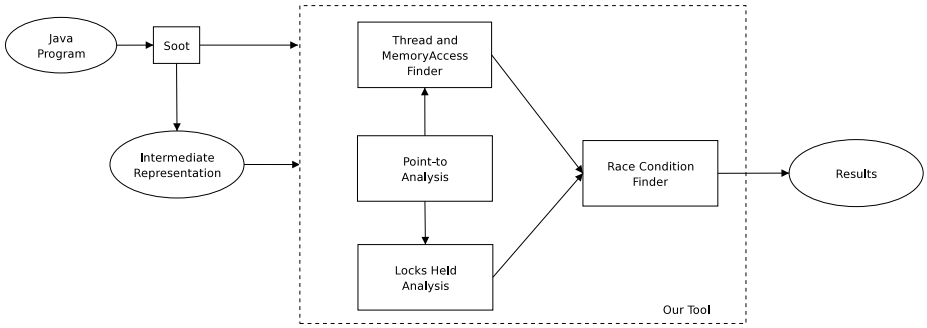

Figure 1.1: Overview of our tool

mark" programs. The purpose of this evaluation was to determine the capabilities of some existing tools and find a framework that is suitable for extension. None of the programs evaluated possess all the key properties identified above and that was our motivation for developing a new tool.

We choose to build our tool on top of Soot [66], a freely[1] available compiler infrastructure capable of parsing Java byte and source codes into a number of different intermediate representations. Soot is written in Java and has an object-oriented API as well a facilities of intra and interprocedural program analyses construction. It is widely used in research and academia for static analysis of Java programs. For an overview to Soot see Section 2.4.

The structure of our tool is given in Figure 1.1. We use an intermediate representation of a Java program obtained from Soot to determine the memory accesses in the program, the locks that may be held for the memory accesses, as well as pointer information to determine if two memory accesses may potentially be involved in a race. The results from the points-to analysis are used by the other components of the tool, and therefore its precision determines the precision of the other components and of the results obtained. We evaluate our tool using the same criteria as the other tools and find it to meet all the key problems identified above sufficiently well.

1.2 Thesis Organization

In the next chapter we discuss some theoretical background. This chapter covers concurrency and program analysis techniques relevant for our tool. We also

[1]Soot is available under the Lesser GNU public license.

introduce the basics of Soot and present a survey on related work.

Chapter 3 covers the analysis and design of our tool. The analysis section introduces the problem of statically detecting interfering memory-accesses and the design section gives a more detailed description on our design and implementation of the different components of the tool.

In Chapter 4 we evaluate some existing tools and compare the results obtained with those from our own tool. Finally, Chapter 5 presents our conclusions.

Chapter 2

Background

In this chapter, we discuss the theoretical background relevant for our tool. We start with a discussion on Concurrency, and then discuss Program Analyses techniques. Next we introduce Soot a compiler infrastructure used for statically analyzing Java programs. Our tool is built on top of Soot. Finally, the section on Related Work, discusses existing work on race detection tools.

2.1 Concurrency Theory

A concurrent program may be defined as a program consisting of more than one *activity* that may run *simultaneously*. This is a broad definition and the interpretation of *activity* and *simultaneously* may be done in different ways. An activity may be a different operating system *process* or a *thread* in a process. Simultaneous execution may be on different processors, loosely or tightly coupled systems or by time-slicing on a single processor.

Regardless of the specific way in which "activity" or "simultaneously" are defined, non-trivial concurrent programs require that the activities communicate with each other. Communication can occur by message passing, commonly used to communicate between different processes, or via the use of shared variables,

commonly used when programming with threads. It is also possible to provide the notion of shared memory for different processes or use message passing with threads; therefore the style of communication used is not exclusive to the kind of activity, but a matter of preference and requirements of the system.

In addition to communication different activities in a concurrent program also need to *synchronize* among each other. By synchronization we mean a constraint in the execution of *actions* that make up an activity. The following are the most basic types of synchronization:

Condition synchronization. An action in one activity waits for an action in another activity, i.e., some actions must always precede others.

Mutual exclusion. In this kind of synchronization, constraints are placed that disallow some actions in different activities from executing simultaneously.

2.1.1 Applications and Programming Styles

In addition to having activities running in parallel, concurrent programming provides a way to organize software that contains several independent parts that communicate and synchronize in order to achieve a result. There are three broad classes of applications of concurrent programming:

Multi-threaded applications. These are applications where multiple threads (activities) execute in the same address space. The threads use shared variables to communicate among each other. Examples of multi-threaded applications are windows systems and real-time control applications.

Distributed applications. These are applications where different processes (activities) execute on machines connected by local or global communication networks. Examples of distributed applications are file and web servers.

Parallel computing. These are applications that require large and computation intensive problems to be solved. The goal is to solve the problem faster by solving some parts of the program in parallel. Examples of such applications are scientific computations like weather forecasting or simulation of physical phenomena.

There is some overlap in these classes of applications. For example, in a distributed computing application consisting of a web server and clients, the clients

server and clients make up a distributed application but may individually be programmed as multi-threaded applications. Applications that use multi-threading for concurrency do so because it is easier to organize the code and data structures as a collection of threads rather than a monolithic program. Distributed applications are typically used to off-load processing, e.g., servers and clients or to manage data that is inherently distributed. Parallel computing is used to achieve high performance using as many processes (threads) as there are processors.

2.1.2 Communication and Synchronization

As mentioned previously, the processes and threads (activities) in concurrent programs must communicate and synchronize with each other; typically, multi-threaded applications use shared variables for this, distributed applications communicate using message passing or remote invocations and parallel programs may use message passing or shared variables, depending on the kind of hardware they run on.

Different concurrent applications also require different synchronization methods. Multi-threaded applications use shared variables to communicate among different threads and therefore may require mutual exclusion between simultaneous access. The different threads may also require conditional synchronization, e.g., one thread produces data consumed by the other, where the consumer thread must wait on the condition that data is available. In distributed applications, where no shared state exists, mutual exclusion is not required but conditional synchronization is used. Parallel computing uses synchronization mechanisms depending on the requirements of the computation.

2.1.3 A Closer Look at Multi-Threaded Programs

Concurrent programs[1] differ from sequential programs as they are *reactive*, i.e., they express an (often infinite) activity rather than computation of some results. Therefore, concurrent programs are characterized by their behavior rather than their input/output relation. The behavior can be expressed with the following properties:

Safety properties. The program does not do anything wrong.

[1] As we consider only multi-threaded programs we, use the term *concurrent programs* and *multi-threaded programs* interchangeably.

Liveness properties. The program makes progress.

It is clear that a concurrent program must possess both safety and liveness properties to make progress and not do something wrong. Safety properties are expressed as *invariants* while liveness properties are expressed using *temporal logic*; liveness properties are dependent on the scheduling.

Interleaving Model for Concurrent Programs

An abstract model for the execution of a process is a sequence of states[2] that the program passes through.

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} \dots$$

Here s_0 is the initial state and an action a transforms the process from one state to another. The states that a program actually passes through are known as the *reachable states*.

In concurrent programs a number of actions may occur overlapping in time. If one is interested solely in properties related to the states a concurrent program may go through it is possible to use the *interleaving model* of concurrency. Here we present the basic idea behind this model, for a more thorough treatment see, e.g., [46].

In the interleaving model, a concurrent program is considered to be composed of several sequential activities. Actions of the different sequential activities must be *mutually atomic*, i.e., do not interfere with each other. Under this assumption, the set of executions of the concurrent program is given by the set of executions of all possible interleavings of actions of the individual sequential activity. A consequence of the actions being atomic is:

> Even if the actions are in fact overlapped in time, the sequence of states of the program will be that of one of its interleavings.

Any property that holds for all of the concurrent program will also be a property of the program under the assumption that actions are atomic.

[2] The set of states is finite but the execution of the process may be an infinite sequence of states.

Note, however, that the number of interleavings, i.e., states that a concurrent program goes through grows exponentially. For three processes with atomic actions n_1, n_2 and n_3 the number of interleavings is:

$$\frac{(n_1 \times n_2 \times n_3)!}{n_1! \times n_2! \times n_3!}$$

Therefore, reasoning about concurrent programs using the interleaving model usually requires an abstraction to reduce the number of actions/states

The interleaving model of concurrency discussed above works under the assumption of atomic actions. On usual machine architectures, reading or writing to a single variable that can be contained in a single word is atomic. Reading or writing of larger types may or may not be atomic. Instructions containing more than one reference to a memory location, e.g., $x := x + 1$ are usually not atomic and broken into a load, increment and store type instructions.

2.1.4 Race Conditions and Critical Sections

Synchronization is used to restrict interleavings, or states, of the program to desirable ones. Consider the program fragment, where actions labeled with a and b may execute simultaneously:

$$\begin{array}{lll} a_1: & \text{if}(x \geq 10) & b_1 : x := 0 \\ a_2: & \quad x := x - 10 & \end{array}$$

Enumerating all the possible interleavings of the program above we get, $(a_1; a_2; b_1)$, $(a_1; b_1; a_2)$, $(b_1; a_1; a_2)$.

Now, if $I : x \geq 0$ was an invariant of the program with an initial value of x as 10, then the interleaving $(a_1; b_1; a_2)$ would violate the invariant (and the related safety property). Clearly, we require that actions $(a_1; a_2)$ and b_1 are *mutually atomic* so that they do not interfere with each other.

Parts of the program that must be mutually atomic with one another so that the program invariants are preserved for all executions of the program are known as *critical sections*. A more appropriate term would be a *critical region*, composed of many critical sections, with the property that at most one thread executes in the critical region at any given time, but we stick to the classic terminology of critical section.

Problems in concurrent programs related to non-determinism and incorrect synchronization while accessing shared memory are termed as *race conditions*. Unfortunately, the term race condition is used implicitly for two different notions. Netzer and Miller [52] classify race conditions[3] into *general races*, for program intended to be deterministic, and *data races* for non-deterministic programs containing critical sections.

A deterministic program should give the same output given a fixed input, even if the executions of the programs are non-deterministic, e.g., a parallelized matrix multiplication program. General races cause non-deterministic results from programs that are intended to be deterministic.

Data races cause non-atomic execution of critical sections and may lead to a violation of program invariants. The example above is an instance of a data race, and for the remainder of this report we use race condition to mean a data race as defined by Netzer and Miller [52].

2.1.5 Explicit Atomicity Constructs

To guarantee that a single thread accesses a critical section, techniques for mutual exclusion are used. It is possible to implement mutual exclusion directly using a machine instruction like test-and-set or via a specialized algorithm to solve the critical section problem [17], e.g., Peterson's algorithm.

The kind of mutual exclusion provided by these techniques are referred to as *spin-locks* or *busy-waiting*, since a thread uses machine cycles while waiting to gain exclusive access to a critical section. Busy-waiting techniques have a number of shortcomings. First, implementation of busy-waiting techniques using an algorithm like Peterson's is quite complex. Second, when using such a technique there is no clear distinction between the program variables and those used for synchronization. Special instructions like test-and-set may not be available on all platforms. A further deficiency is that busy-waiting wastes CPU cycles[4].

In the rest of this section we look at semaphores and monitors, specialized programming constructs that may be used to program explicit atomicity and other forms of synchronization required by concurrent programs.

[3] They use the term data races for race conditions.

[4] Except in multi-processor architectures where the number of threads is equal to the number of processors.

Semaphores

A semaphore is a special kind of shared variable that has a non-negative integer as value and is manipulated only by two atomic operations referred to as `P` and `V`. The `V` operation is used to signal the occurrence of an event while the `P` operation is used to wait for an event to occur. The definitions of these two operations are given by:

$$\begin{aligned} \texttt{P}(s): &\quad < \texttt{await}\ (s > 0)\ s := s - 1 > \\ \texttt{V}(s): &\quad < s := s + 1 > \end{aligned}$$

The `V` operation increments the value of the semaphore atomically. The `P` operation waits till the value of the semaphore is positive and then decrements it atomically. If two threads try to execute a `P` operation simultaneously then only one will succeed at a time. If two threads try to execute a `P` and `V` operation simultaneously both will succeed and the value of the semaphore will be unchanged. A thread waiting for the value of the semaphore to become positive will wait as long as another thread does not perform a `V` operation on the semaphore.

Semaphores come in two different flavors; *general* semaphores which may take on any non-negative value, also known as *counting* semaphores and *binary* semaphores where the value of the semaphore may be 0 or 1.

Semaphores may be used directly to implement mutual exclusion as well as simple forms of conditional synchronization. More elaborate synchronization can be programmed using semaphores. The use of semaphores to solve several concurrent programming problems may be found in [17].

Monitors

Semaphores are a fundamental synchronization mechanism. They can be used to program mutual exclusion and conditional synchronization but have some serious limitations. Semaphores are relatively low level and it is easy to make an error while using them. For example, a programmer must be careful not to omit a `P` or `V` operation. It is the programmer's responsibility to identify and protect all critical sections. Semaphores are global to the whole program, therefore one may need to examine the whole program to see how semaphores are used. Finally, semaphores are used for both conditional synchronization and mutual exclusion. This makes it hard to identify the purpose of a `P` or `V` operation looking at it in isolation.

To overcome these limitations monitors were introduced. Monitors combine synchronization with the two good software engineering approaches of abstraction and encapsulation. Monitors abstract the internal representation of data they contain and provide operations to manipulate this data. They provide implicit mutual exclusion for all operations on the data. Conditional synchronization is provided by use of condition variables.

Mutual exclusion. All operations of a monitor provide implicit mutual exclusion. Since the data contained in a monitor may only be modified by monitor operations, access to the data is implicitly atomic. By placing all shared variables within monitors, the critical section of the program is easily identified, and restricted to the monitor.

Conditional synchronization. Monitors provide a special construct known as conditional variables for conditional synchronization. The conditional variables are used to delay a thread until the internal state of the monitor satisfies some boolean condition and signal the thread when the condition becomes true. The primitive operations available on conditional variables are `wait` and `signal`. The thread calling the `wait` operation blocks until another process calls a `signal` operation on the conditional variable. Several threads may be waiting for the same condition and the order in which they are signalled may or may not be deterministic.

Signalling disciplines. A process that calls a `signal` operation is executing inside a monitor and holds a lock inhibiting other processes from entering the monitor. As the `signal` operation may enable another thread from entering the monitor, it must be decided which of the two processes is allowed to continue in the monitor next. There are two possibilities:

Signal and wait (SW). The thread that executes the signal passes control directly to the signaled thread. Therefore SW is preemptive and the signalling thread goes into the queue of threads waiting to enter the monitor.

Signal and continue (SC). The signaler continues to execute in the monitor. Therefore SC is non-preemptive, the awakened thread goes into a queue of threads waiting to enter the monitor.

2.1.6 Deadlocks

The introduction of mutual exclusion and synchronization causes some threads to wait while another thread executes in a critical section or until the waiting thread is signalled by another thread. In the case of several shared resources, if one thread holds a resource and waits for another thread that requires the resource the first thread is holding, the threads are said to be deadlocked; neither of them can make progress. Deadlocks violate liveness properties of a program.

2.1.7 Summary

This section took a whirlwind tour of concurrency. It introduced the different styles of concurrent programming, namely, multi-threaded, distributed, and parallel programming, and discussed why communication and synchronization are essential for concurrent programs. Next, we took a closer look at multi-threaded programs, presenting the interleaving model as a way to reason about them. We saw the need for atomic actions and the notions of safety and liveness properties. Safety properties are often expressed as invariants, whereas liveness properties are dependent on the implementation of the synchronization constructs, scheduling policies for the different processes or threads as well as the program in the event of deadlocks. Finally, we looked at semaphores and monitors as programming constructs that provide synchronization necessary for concurrent programs. In the next section we introduce the multi-threaded programming facilities available in Java.

2.2 Multi-Threading Facilities in Java

The Java programming language provides facilities for creating and starting threads and synchronizing them using monitors. In addition to language level facilities, Java provides numerous utility classes and libraries with specialized concurrent programming constructs.

2.2.1 Creating and Starting Threads

To create a class with its own thread of control, one may either extend the `Thread` class or implement the `Runnable` interface. Starting a thread depends on how it was created. For subtypes of `Thread`, a new thread starts executing

when the `start()` method is called. Classes that implement `Runnable` must be passed to the constructor of a `Thread` subtype and the start method is called on the latter.

2.2.2 Main Thread

Every Java program has at least one thread known as the main thread. All other threads in the program are started by the main thread, or one of the threads started by the main thread. There is no parent-child relationship between the creator of a thread and the created thread.

2.2.3 Thread Priorities and Scheduling

All Java threads have a priority, which is inherited from the thread that created it. The priority of a thread may be modified at any given time by another thread by calling the `setPriority()` method. At any given time, if multiple threads are ready to execute, the thread with the highest priority is chosen to run. The thread scheduler is preemptive, i.e., a higher priority thread if ready may preempt a running lower priority thread. A thread continues to run until one of the following conditions hold:

- A higher priority thread becomes runnable.
- The thread yields or terminates.
- On multi-tasking systems, the thread's time slice expires.

In general, one should not rely on facilities like thread priorities, preemption or time-slicing for synchronization. The exact scheduling of threads is implementation dependent and the minimum guarantees for thread behaviors can be found in [31]. Following the interleaving model in Section 2.1.3 we assume that any action not explicitly synchronized is executable in parallel with those of other threads.

2.2.4 Thread Control Methods

We have already mentioned the `start()` method that is used by a calling thread to start another thread. Other methods to interact with threads are:

Interrupt. Each thread has an associated interrupt status flag and invoking `interrupt()` on a thread sets it to true, unless the thread is waiting for execution in one of the `wait(), sleep()` or `join()` methods; see below.

isInterrupted. The interrupt status of any thread may be queried by calling this method. It returns true if the thread has been interrupted via the interrupt method but the status has not since been reset.

Join. Invoking the `join()` method on a thread causes the caller to suspend until the called thread completes executing.

The additional control methods `suspend()`, `resume()`, `stop()`, `destroy()` were also supported but have been deprecated because these methods lead to *unsafe* programming practices and unreliable programs; [40] contains a discussion on why these methods are deprecated.

There are also some static thread control methods; these are defined as static as they affect the calling thread, a thread is not allowed to invoke these methods on another thread.

CurrentThread. This method returns a reference to the current thread. This reference may be used to make a call to other non-static methods.

Interrupted. This method clears the interrupted status of a thread, set by the `interrupt()` method and returns the previous value. A consequence of being static is that a thread may only reset its own interrupt status.

Sleep. This method suspends the calling thread for at least the specified amount of time.

Yield. The `yield()` method is a hint to the JVM implementation to run any waiting threads instead of the calling thread. An implementation is free to ignore this call.

2.2.5 Synchronization and Monitors

Java provides a monitor implementation to synchronize access between threads. Mutual exclusion is guaranteed by the use of the `synchronized` keyword.

Every instance of `Object` and its subtypes possesses a lock and may act as a monitor. Scalars of type `int`, `float` etc., are not `Objects` and are protected

using the locks of their enclosing types. Access to individual fields is not marked as synchronized, rather locking may only be applied inside methods.

Synchronized methods and blocks. One may use the `synchronized` keyword in two ways, either by declaring a method as synchronized or by creating a synchronized block. In the case of a synchronized block the object whose lock should be acquired needs to be provided. For methods declared as synchronized, the lock associated with the `this` object is acquired. For example:

```
synchronized void foo() { /* body */ }
```

is equivalent to:

```
void foo() { synchronized(this) { /* body */ } }
```

There is a single lock associated with inherited classes, i.e., the lock of the base class instance is not distinct from that of its derived class, which is as expected. However, the lock of an inner class is different from that of its outer class.

Acquiring and releasing locks. Locking obeys an implicit acquire-release protocol; a result of the `synchronized` keyword being lexically scoped. Therefore, it is not possible to forget to release a lock. Further, a lock is released if an exception is thrown from a synchronized method or block.

A thread that possesses a lock may enter a region protected by the same lock, i.e., locks are re-entrant. This allows a `synchronized` method to make a self call on another `synchronized` method even or call itself recursively. Note that methods that are not synchronized may still be executed by different threads. This is unlike the classic monitor where all methods are implicitly atomic, but the programmer may use any object as a monitor.

The order in which contending threads acquire a lock is not guaranteed, which implies that Java makes not fairness guarantees for contending threads waiting to acquire a lock.

Monitors. Every instance of `Object` has, in addition to a lock, a wait set that is manipulated by the methods `wait()`, `notify()`, `notifyAll()` and `Thread.interrupt()`. The wait sets along with the methods to manipulate them provide an implementation of a condition variable. Each `Object` has a lock and condition variable and may therefore act as a monitor. Note that

unlike classical monitors only a single condition variable is associated with each `Object`. The methods have the following semantics:

Wait. A `wait()` invocation causes the calling thread to suspend execution and block in the target object's wait set. Unless the thread is/was interrupted by the `interrupt()` method it waits until it is signaled by another thread using one of the notify methods. The lock of the target object is released but any other locks held are retained. Upon receiving a signal the lock status of the thread is restored before it continues.

Notify. A `notify()` invocation causes the calling thread to signal *one* of the threads in the targets wait set. The signaling thread continues execution after signalling, i.e., signal and continue (SC) semantics. The signalled thread contends with all other threads to (re)enter the monitor and continues to execute after the place where it called `wait()`.

NotifyAll. A `notifyAll()` works the same way as the `notify()` except that it signals all threads on the wait set of the target.

Interrupt. The behavior of a thread after it receives an `interrupt()` method depends on whether the thread was waiting or running. For a running thread the interrupt status flag is set. A waiting thread is treated as if it were notified, and as soon as it *may* continue execution, an exception is thrown from the `wait()` method.

2.2.6 Concurrent Programming Utilities in Java 1.5

The release of Java version 1.5, or Java 5.0 included a `java.util.concurrent` package for concurrent programming utilities. These utilities are intended to be the basic building block for concurrent programs [5]. The concurrency utilities include:

Task scheduling framework. A framework for standardizing invocation, scheduling, execution, and control of asynchronous tasks according to a set of execution policies. In previous versions of Java Timers, and threads were the available means to create and schedule tasks.

Concurrent collections. Concurrent implementations/adaptors for collection class interfaces `Map`, `List` and `Queue`, as well as specialized concurrent access collections , e.g., `CopyOnWriteList` [40].

Atomic variables. Classes for atomic manipulation of primitive types and references. [5] states that the classes in `java.util.concurrent.atomic` offer higher performance than using synchronization (on most platforms).

Synchronizers. General purpose synchronization classes, including semaphores, mutexes, barriers, latches, and exchangers, which facilitate coordination between threads.

Locks. The `java.util.concurrent.locks` package provides a lock implementation with the same memory semantics as `synchronized`, but also supports specifying a timeout when attempting to acquire a lock, multiple condition variables per lock, non-lexically scoped locks, and support for interrupting threads which are waiting to acquire a lock.

Nanosecond-granularity timing. The `System.nanoTime` method enables access to a nanosecond-granularity time source for making relative time measurements, and methods which accept timeouts can take timeout values in nanoseconds. The actual precision of time measurements is platform-dependent.

The usage of non lexically-scoped locks and semaphores offers greater flexibility to programmers, e.g., allowing to acquire and release locks in different threads, but are more complicated to analyze statically.

2.3 Program Analysis

Program analysis provides compile time techniques to *approximate* the dynamic or run-time behavior of programs. The results are approximate because of the theoretical limits of computability. This is demonstrated with the following example from [59]:

```
x = 17 ; if (TM(j)) x = 18;
```

If we could statically determine that the value of `x` is constant we could also determine that the j'th Turing machine halts on empty input.

Even though the precise behavior of a program can not be predicted the approximate behavior has many useful applications, e.g., in compiler optimization, program understanding, refactoring etc. The approximation calculated is either an *over-approximation* or an *under-approximation* of the actual program behavior. This can be seen in Figure 2.1; in such cases the analysis is said to be *sound*.

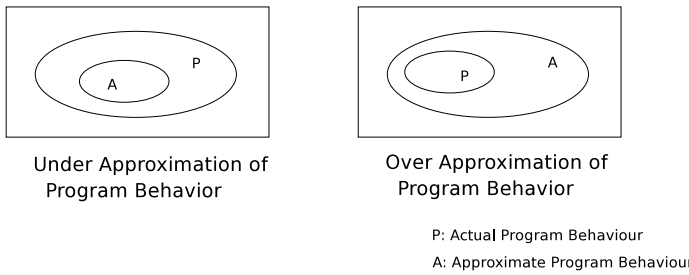

Figure 2.1: Nature of Sound Approximations in Program Analysis

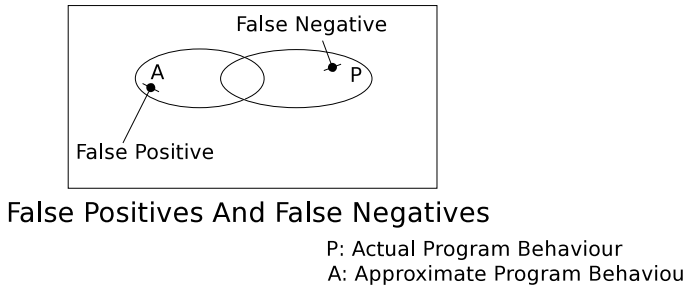

Figure 2.2: False positives and false negatives reported by a program anaysis

Program-analyses that compute over approximations produce results known as *false positives*; a valid behavior for a program but not necessarily the behavior of an execution instance of a program. False positives may be produced by both sound and unsound program analyses. In contrast, unsound program analyses may not report all possible behaviors of a program. These missed behaviors are known as *false negatives* and Figure 2.2 shows examples of both.

2.3.1 Lattice Theory

The techniques for static analysis are based on the theory of lattices. In this section we provide a brief overview; for a more complete treatment see [53].

Lattice

A *partial order* is a mathematical structure $L = (S, \sqsubseteq)$, where S is a set and $\sqsubseteq$ is a binary relation on S that satisfies the following conditions:

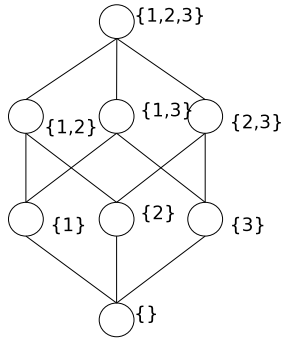

Figure 2.3: Pictorial representation of a lattice $(2^{\{1,2,3\}}, \subseteq)$

- Reflexitivity $\forall x \in S : x \sqsubseteq x$
- Transitivity $\forall x, y, z \in S : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$
- Anti-symmetry $\forall x, y \in S : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$

Let $X \subseteq S$, then $y \in S$ is an *upper bound* for X, written $X \sqsubseteq y$, if $\forall x \in X : x \sqsubseteq y$. Similarly, $y \in S$ is a *lower bound* for X, written $y \sqsubseteq X$, if $\forall x \in X : y \sqsubseteq x$. A *least* upper bound for X, written $\sqcup X$, is defined as:

$$X \sqsubseteq \sqcup X \wedge \forall y \in S : X \sqsubseteq y \Rightarrow \sqcup X \sqsubseteq y$$

Dually, the *greatest* lower bound for X, written $\sqcap X$, is defined as:

$$\sqcap X \sqsubseteq X \wedge \forall y \in S : y \sqsubseteq X \Rightarrow y \sqsubseteq \sqcap X$$

A *lattice* is a partial order in which $\sqcup X$ and $\sqcap X$ exist for all $X \subseteq S$. A lattice must have a unique *largest* element $\top$ defined as $\top = \sqcup S$ and a unique *smallest* element $\bot$ defined as $\bot = \sqcap S$.

In program analysis we often deal with finite sets. Every finite set A defines a lattice $(2^A, \subseteq)$, where $\bot = \emptyset$, $\top = A$, $x \sqcap y = x \cap y$ and $x \sqcup y = x \cup y$. For example, a lattice with three elements is shown in Figure 2.3.

The *height* of a lattice is defined as the longest path from $\top$ to $\bot$. In the above example the height of the lattice is 3.

Monotone Functions and Fixed Points

Monotone functions. A function $f : L \to L$ is *monotone* if $\forall x, y \in S : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$. Note that monotonicity does *not* imply increasing; constant functions are monotone. Viewed a functions, $\sqcup$ and $\sqcap$ are both monotone. The composition of monotone functions is again monotone.

Fixed points. A central result required to calculate analysis results for a program is the *fixed point theorem*. In a lattice L with finite height, every monotone function f has a *unique least fixed point* defined as :

$$fix(f) = \bigsqcup_{i \geq 0} f_i(\bot)$$

The proof of this theorem may be found in [53]. Fixed points are interesting because they allow us to solve systems of equations and calculate the approximate behavior of programs, i.e., the analysis we are interested in, as we shall see in the coming sections.

2.3.2 Dataflow analysis

Dataflow analysis computes an approximation of program behavior for each point of the program that we are interested in. It does so by first representing the program as a *control flow graph* (CFG). A CFG is a graph where the nodes represent the program points of interest, e.g., statements or basic blocks and the arcs represent flow of control through the nodes. For example, the CFG for the program

```
input x;
if ( x > 10 ) y = 0
else          y = 100
output y;
```

is given in Figure 2.4.

Associated with each node v_i of a CFG is a set ranging over the elements in a lattice L. The concrete type of lattice elements depends on the information we are interested in calculating. The value of analysis for node v_i depends on other

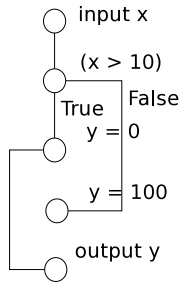

Figure 2.4: Pictorial representation of control-flow graph

nodes in the CFG, typically its neighbors; and is combined using the $\sqcup$ or $\sqcap$ operators. Depending on whether the information at a given node is combined from its immediate successors or predecessor the analysis is known as a *forward* or a *backward* analysis. If the $\sqcup(\sqcap)$ operator is used to combine information at a node the analysis is known as a *may*(*must*) analysis.

The set associated with a node v_i are calculated using the following equations:

$$\begin{aligned} IN(v_i) &= \sqcup\, OUT(v_w), \text{where } v_w \text{ is a neighbor of } v_i \\ OUT(v_i) &= IN(v_i) \backslash kill(v_i) \cup gen(v_i) \end{aligned}$$

where, $\sqcup$ is $\cup$ or $\cap$ depending of if the analysis is a may or must respectively and neighbors are successors or predecessors depending on whether the analysis is a forward of backward respectively.

Notice that the information for each node v_i is expressed as a monotone function over a lattice L; the analysis result, the value of the functions for all nodes v_i may be considered a composition of the individual functions. Using the fixed point theorem we know that a least fixed point for the system exists and may be computed using a *worklist* algorithm.

Classical dataflow analysis include those for *reaching definitions*, *very busy expressions*, *available expressions* and *live variables*. For concrete formulations of the lattice and monotone functions and a worklist algorithm to compute these analyses see [53].

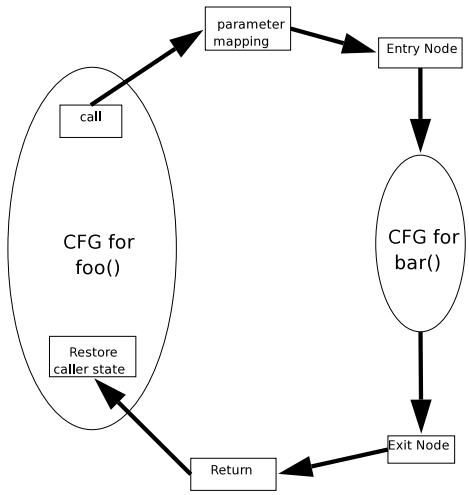

Figure 2.5: Flow graph glueing together cfgs for two functions `foo()` and `bar()`

2.3.3 Inter-procedural Analysis

The analysis techniques discussed in the previous section are *intra procedural* as they analyze the body of a single function. If we analyze whole programs including function calls the analysis is known as *inter procedural*. This section provides an overview into some of the concepts behind inter procedural analysis; a more complete survey is available in [59]; for a thorough treatment see [53].

Flow Graphs for Programs

In the absence of pointers and references, the CFG may be extended for whole programs by constructing the CFG for the individual functions and *glueing* the result together taking parameter passing, and return values into consideration. This can be see in Figure 2.5; the boxes in the flow graph are special nodes that are used to glue the CFGs together.

Context Sensitive vs Context Insensitive

The construction of the flow graph above does not consider multiple call sites, i.e., all calls to a particular function result in control flowing to that function's representation in the flow graph. However, *different* invocations of the same function are associated with a different call site and return to a different location. A naive way to capture this relationship could be to create a *separate* control flow graph for each invocation, but this would generate excessively large flow graphs for non trivial programs and can not handle recursion. Therefore two important concepts are introduced.

Context insensitive. The information about all calling sites is combined and the function body analyzed only once using this combined information. The resulting information about the set of return values is used at all the return points.

Context Sensitive. This approach keeps context information along with each called function. Therefore two different calls are associated with different contexts and the values used to analyze the function called as well as the return value is specific to the call site.

Context sensitivity is clearly more precise than context insensitivity but comes at an additional increase in complexity. The trade-off in complexity versus cost for context sensitive analysis is made by using a limited amount of context while analyzing the program, e.g., restricting the context to at most 'k' callers for a function.

Flow Insensitive vs Flow Sensitive

The classical dataflow analysis seen so far are all *flow- sensitive* analysis as they take the flow of the program into consideration. In general such analysis would yield different results if the program being analyzed changed from $S_1; S_2$ to $S_2; S_1$. Analysis where the order of the statements does not affect the analysis result are known as *flow insensitive*. The applicability of flow insensitivity seems questionable at first but we shall see examples of analysis that use it in the following section.

2.3.4 Control Flow Analysis

The presence of function pointers, higher-order functions or dynamic dispatch introduces non-determinism in the flow of control for a program requiring one to analyze and construct a call graph before inter-procedural analysis is possible. We elide discussing control flow analysis in the presence of function pointers and higher-order functions, for which the reader is referred to [59, 53]. For object-oriented languages the structure provided by the class hierarchy and the type system allows some simpler alternatives that are more scalable than full-blown control flow analysis techniques and equally precise for practical programs [59]. For object-oriented languages control-flow analysis reduces to answering the question, which methods may be invoked at a given method invocation site like:

```
foo.bar(a, b, c);
```

The simplest solution is to match method signatures in the code base with that of the called method, which is not precise because each type (class) is a distinct namespace. Below we consider approaches in increasing order of complexity and precision.

Class Hierarchy Analysis (CHA) [19, 23], considers only those classes that may be a subtype of the method receiver, here `foo`, with matching signatures.

Rapid Type Analysis (RTA). [19], improves CHA by removing those classes from the list of targets that are not instantiated in the program. RTA is an example of a flow insensitive analysis and may be considered as an conservative approximation of which objects reach a particular method invocation.

Variable Type Analysis (VTA). [63] improves the precision of RTA by tracking assignments like,

$$x_1 = \texttt{new Object()}, x_2 = x_1, \ldots, x_{n-1} = x_n, foo = x_n$$

i.e., creation of a type to its use in an method invocation. These assignments may be made directly or indirectly via method calls. VTA like RTA is also flow insensitive but unlike RTA it makes a more fine grained analysis of the actual types of objects that reach a method invocation site.

2.3.5 Pointer Analysis

Pointer analysis determines the set of possible *targets* that a pointer or reference *may* point to during the execution of a program. A target may be an arbitrary memory location on the program heap. As the number of targets during execution may be infinite, we use the abstraction that a unique node, $heap[i]$, is associated with every statement that may create a new object on the heap. We call such a statement a memory allocation site and it is identified by a unique label i. A consequence of this abstraction is that $heap[i]$ represents *all* the object that may be allocated at site i.

Points-to analysis takes place either before or simultaneously with the control-flow analysis or call-graph construction. The result of a points to analysis is a function $pointsTo(p)$ that returns the set of targets that a pointer p may point to. As we need to perform a conservative analysis, these sets may be very large.

Given the points-to analysis, many other facts about a program may be approximated. For example to determine whether two pointers p, q may be aliases we need to check $pointsTo(p) \cap pointsTo(q) \neq \emptyset$.

In the following sections we present two fundamental approaches for the points to analysis, Andersen's algorithm [16] and Steensgaard's algorithm [61].

Andersen's Algorithm

In this approach we associate a set $[\![p]\!]$ ranging over the possible pointer targets with each pointer or reference p variable in the program. Assignments to pointer types introduce the following constraints:

$$\begin{array}{rc} p = new \ldots : & heap[i] \subseteq [\![p]\!] \\ p = \&q : & q \subseteq [\![p]\!] \\ p = q : & [\![q]\!] \subseteq [\![p]\!] \\ p = *q : & r \in [\![q]\!] \Rightarrow [\![r]\!] \subseteq [\![p]\!] \\ *p = q : & r \in [\![p]\!] \Rightarrow [\![p]\!] \subseteq [\![r]\!] \end{array}$$

The constraints are intuitive, for example, for the assignment statement $p = q$, the constraint states that the set of targets for p contains the set of targets for

q which is a conservative approximation of an assignment.

Note that the constraints do not take flow and context sensitivity into account. Indeed, flow and context sensitive points-to analysis like shape analysis [70] exist, and give more precise results but are computationally expensive and feasible only for small sized programs. Static Single Assignment (SSA) intermediate representation may be used to obtain more precise results for flow-insensitive analysis [32].

For a strongly typed language with no pointers but only references[5] only two constraints remain:

$$\begin{aligned} p = new \ldots : &\quad heap[i] \subseteq [\![p]\!] \\ p = q : &\quad [\![q]\!] \subseteq [\![p]\!] \end{aligned}$$

Constraints of the this from can be solved in $O(n^3)$ time using a worklist-based constraint solving algorithm [53, 59]. An implementation of a points-to analysis is generally broken down into two phases. Finding constraints, which is flow insensitive and context insensitive and solving[6] them.

Steensgaard's Algorithm

Steensgaard's algorithm performs a coarser grained analysis but the constraints are solvable in almost linear time. This makes it a popular alternative for Andersen's algorithm trading of precision for scalability.

In Steensgaard's algorithm equivalence constraints are generated instead of subset constraints and therefore the constraints are solvable in almost linear time.

2.4 Soot

Soot[12, 65] is a free compiler infrastructure for Java, written in Java and available under the LGPL license. Soot was originally intended to provide an in-

[5] References are typed and provide a single level of indirection as opposed to pointers where the indirection is limited only by the available memory.

[6] Also known as constraint propagation.

frastructure for implementation and comparison of various program analyses, but has been extended to include facilities like decompilation and visualization. It provides a choice of different intermediate representations, ability to read in Java source or class files and comes with several built in program analyses and transformations.

It is possible to use Soot as a stand alone tool or integrated with the Eclipse development environment[]. It also provides an object-oriented API and may be used as a library. As of this writing, the current Soot version is 2.2.3 and it is available for download at [12].

This section discusses the internal workings of Soot and is organized as follows. We start with the different intermediate representations available in Soot. Next we give the fundamental abstractions behind the Soot API and then introduce the intra and interprocedural analyses facilities.

2.4.1 Intermediate Representations

The current version of Soot is able to read and process Java source and bytecodes. It provides the following intermediate representations:

Baf. Is a stack based compact representation of *B*ytecode.

Jimple. Is *J*ava's s*imple*, typed, 3-address stack-less representation.

Shimple. Is a *S*SA-version of *Jimple*.

Grimple. Is like J*imp*le, but with expressions ag*GR*egated. In Grimple statements like `a = (a + b) * c` are allowed but in Jimple they are split into two statements, one for the addition and the second for the multiplication.

Dava. A structured representation used for *D*ecompiling J*ava*.

For a more complete description on Baf, Jimple and Grimple including the motivations behind each of these refer to [66]; for Shimple refer to [64]. Jimple is the principal Soot intermediate representation and we look at it in more detail in the following section.

Jimple

Jimple is a *typed three address* intermediate representation and does *not* involve stack instructions as is the case for Java bytecodes. As opposed to bytecodes which have more than 200 instructions, Jimple has only 15 statements, where each statement acts explicitly on named variables. The types of each local variable are also known. Listing 2.2 shows the Jimple code for the Java code in Listing 2.1.

```
class Foo {

    protected Integer id;

    public synchronized Object bar() {
        int a=0,b=0,c=0;
        int i;
        Object obj;

        obj = new Object();

        i = (a + b) * c;

        return obj;
    }
}
```

Listing 2.1: Java source for Jimple in Listing 2.2

```
class Foo extends java.lang.Object
{
    protected java.lang.Integer id;

    public synchronized java.lang.Object bar()
    {
        Foo this;
        int a, b, c, i, $i0;
        java.lang.Object obj, $r0;

        this := @this: Foo;
        a = 0;
        b = 0;
        c = 0;
        $r0 = new java.lang.Object;
        specialinvoke $r0.<java.lang.Object: void <init>()>();
        obj = $r0;
        $i0 = a + b;
        i = $i0 * c;
        return obj;
    }

    public void <init>()
    {
        Foo this;

        this := @this: Foo;
        specialinvoke this.<java.lang.Object: void <init>()>();
        return;
    }
}
```

Listing 2.2: Jimple representation for code in Listing 2.1

In Listing 2.2, notice that the variables all have explicit types. The variables prefixed with $ correspond to stack references. Notice also that the expression `(a+b)*c` has been broken down into three instructions. The creation of an `Object` corresponds to two statements, the first creates an `Object` via the `new` operator and the second explicitly invokes the constructor on the newly created object. Finally, notice that the constructor is invoked using the `specialinvoke` instruction and contains the type of the receiver of the method call.

It is worthwhile comparing the Jimple representation with the corresponding Grimple representation in Listing 2.3. Notice that Grimple representation is very close to the Java source code; the expression `(a+b)*c` has not been broken up, neither has the creation of an object and invocation of its constructor as is done in Jimple.

```
class Foo extends java.lang.Object
{

    public synchronized java.lang.Object bar()
    {
        Foo this;
        int a, b, c, i;
        java.lang.Object obj;

        this := @this: Foo;
        a = 0;
        b = 0;
        c = 0;
        obj = new java.lang.Object();
        i = (a + b) * c;
        return obj;
    }

    public void <init>()
    {
        Foo this;

        this := @this: Foo;
        this.<java.lang.Object: void <init>()>();
        return;
    }
}
```

Listing 2.3: Grimple representation for code in Listing 2.1

The different Jimple statements are given below:

Core statements. `NopStmt` and `DefinitionStmt`. A `DefinitionStmt` may be one of `IdentityStmt` or an `AssignmentStmt`. `IdentityStmts`

are used for assignment of parameters passed to a function and assignment of the this reference. These are the first statements[7] in a method.

Ccontrol-flow statements. The `IfStmt`, `GotoStmt`, `TableSwitchStmt` and `LookupSwitchStmt` are used to represent intraprocedural control flow. The `TableSwitchStmt` and `LookupSwitchStmt` are used to represent the corresponding JVM instructions which are both used to implement the `switch` statement.

The `InvokeStmt`, `ReturnStmt` and `ReturnVoidStmt` are used for interprocedural control flow. The `InvokeStmt` has subtypes for virtual, interface, static and constructor method invocations.

ThrowStmt. Used to throw an exception.

MonitorStmt. These are `EnterMonitorStmt` and `ExitMonitorStmt`.

```
public java.lang.Object foo(int, java.lang.String) /* Local variables */
{

  this := @this: Test;      /* IdentityStmt */
  a := @parameter0: int;    /* IdentityStmt */

  if b != null goto label0; /* IfStmt */

  $r0 = new java.lang.RuntimeException;
  specialinvoke $r0.<java.lang.RuntimeException: void <init>()> /* InvokeStmt
      */
  throw $r0;                /* ThrowStmt */

  label0:
  return $r1;               /* ReturnStmt */
}
```

Listing 2.4: Jimple intermediate representation showing different statements

2.4.2 Fundamental Soot Objects

Representing Classes

Soot builds data structures to represent different objects and provides and object-oriented API to access these data structures. In Soot the complete environment is represented by the `Scene` object. A class in represented by a

[7] Local variable declarations are not statements in Soot but instances of `Local`.

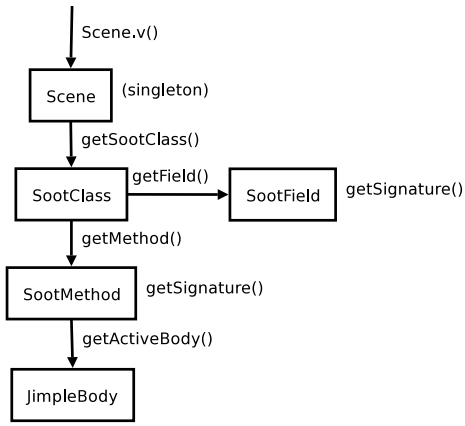

Figure 2.6: Soot Objects for the environment and classes. Figure taken from [34]

`SootClass` object and contains `SootField` and `SootMethod` objects representing the fields and methods of a class. A body of a `SootMethod` is represented by a `Body` object. The exact `Body` object depends on the intermediate representation used, for example, for Jimple the concrete `Body` object is a `JimpleBody`. These relationships are shown in Figure 2.6.

Representing Bodies of Methods

A body contains chains[8] of three distinct types, namely, `Units`, `Locals` and `Traps`. A `Local` represents a local variable declaration in the code. Local variables may be defined by the user explicitly in the code or created implicitly to represent stack memory locations. A `Unit` is an abstract representation for the different statements in the program[9]. These relationships are shown in Figure 2.7.

The body of a `SootMethod` is also used to create a *Control- Flow Graph* object, a subtype of a `UnitGraph`, which captures the intraprocedural control flow between the different units. A control-flow graph may be created ignoring or considering the effect of exceptions and is known as `BriefUnitGraph` and `ExceptionalUnitGraph`, respectively.

[8] A collection with $O(1)$ lookup and insertion time.

[9] For Jimple the concrete statements are subtypes of `Stmt` whereas for Baf they are subtypes of `Inst`.

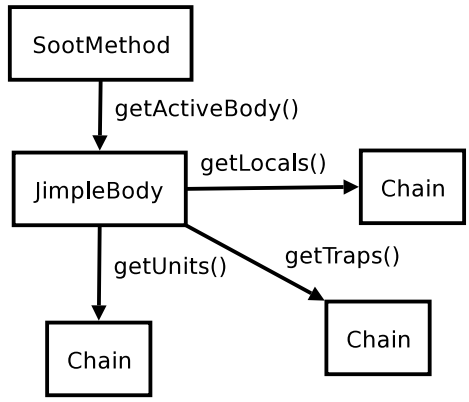

Figure 2.7: Soot Objects for a `SootMethod` Body. Figure taken from [34]

Working with Units

Soot developers recommend that access to data structures should be as abstract as possible. This allows the same analysis and transformation code to be used with the different intermediate representations. Recall that `Units` are Soot's abstract representations for the statements of a program, and different intermediate representations have their own implementations of Statements. To access data used by a statement Soot provides the notion of value and definition *boxes*.

A box may be of thought as a pointer for a value that is either used or defined at a statement. The `getUseBoxes()`, `getDefBoxes()` provide a list of all the values defined or used at a particular statement. For example, for the assignment statement:

```
x = a + b;
```

`getUseBoxes() = [a, b, a+b]` and `getDefBoxes() = [x]`. Boxes are used instead of values directly to make it easier to manipulate the resulting expression trees. To understand this better consider Figure 2.8, that shows two expression trees and try to work out the pseudo code for replacing all occurrences of `y` with `7` for both the trees. It is easier to do a replacement for the tree on the left as one does not have to track the parent node for the value, `y`, being updated.

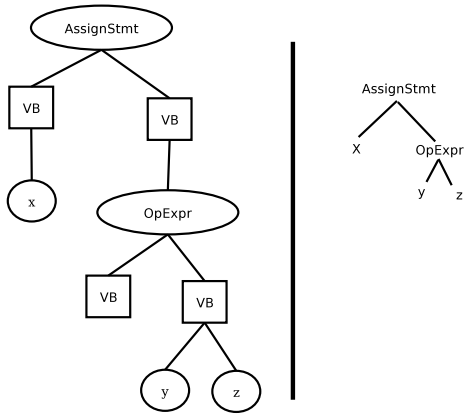

Figure 2.8: Soot Expression trees for `x = y + z` with and without using Boxes. Taken from [34]

2.4.3 Program analysis facilities

Intra Procedural

Soot provides an easy to use framework to implement new intraprocedural dataflow analyses. The dataflow analysis framework in Soot is object-oriented and therefore one implements the new analysis by extending the available classes. The framework provided finds a fixed point for the domain being analyzed using a worklist algorithm. The following information needs to be determined and communicated to the the dataflow framework.

1. Determine if the analysis is forward or a backward analysis.
2. Decide the domain being analyzed, i.e., the operator used to combine information flowing into the nodes. This is usually the $\cup$ or $\cap$ operator, depending on whether the analysis is a may or a must respectively.
3. Write an equation for each kind of intermediate representation statement. This establishes the transfer function relating IN to OUT for each node.
4. Setting the initial values for the sets to $\bot$ or $\top$ depending on whether the analysis is a may or a must respectively.

An excellent description with details on how to implement a backward and a branched forward flow analysis using Soot is available in [34].

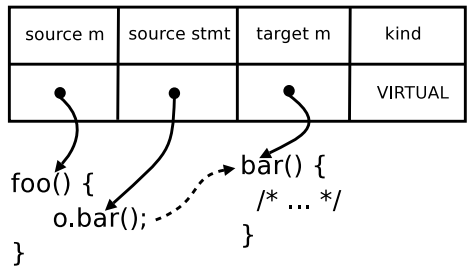

Figure 2.9: Soot A Soot call-graph edge. Figure taken from [34]

Inter Procedural

For inter procedural analysis Soot computes the call-graph for the program and also calculates points-to information. To compute these results it is necessary to put Soot into *whole program mode*.

Call Graphs

Soot calculates the call graph for a given Java program and stores the result as a collection of `Edges` representing *all* method invocations known to Soot; these include:

- Explicit method invocations.
- Implicit invocations of static initializers.
- Implict calls to `Thread.run()`.
- Implict calls of finalizers.
- Implict calls by the `AccessController`.
- ...

The `Edge` object represents an edge in the call graph and contains information on the source method, the source statement(if applicable), the target method and the type of edge. This can be seen in Figure 2.9.

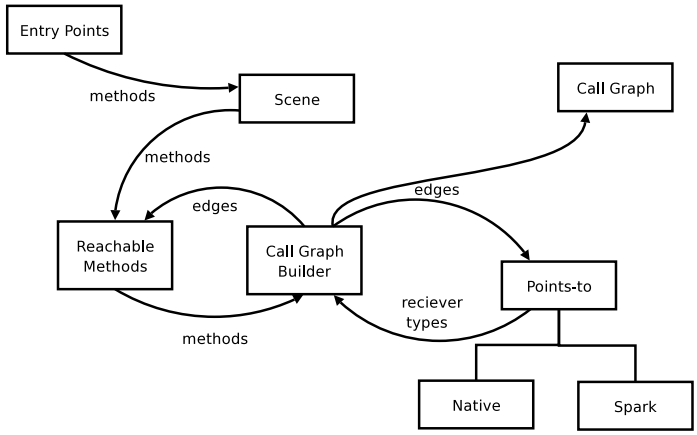

Figure 2.10: Soot Soot call-graph construction — The Big Picture. Figure taken from [34]

To query information in the call graph one of the two methods `edgesOutOf()` or `edgesInTo()` may be used. These methods return iterators over edges flowing out of and into a method respectively. For convenience, adapters are provided to convert an iterator over edges to an iterator over source methods, units or target methods. Classes `ReachableMethods` and `TransitiveTargets` are also provided to determine methods reachable from a source method.

Constructing the call graph requires conservative virtual method calls resolution. Therefore, the call graph works together with the points-to analysis and in the simplest case uses Class Hierarchy Analysis to resolve virtual method calls. Figure 2.10 shows the interaction between different Soot components to compute the call graph.

Points-To Analysis

Soot provides three implementations for points-to analyses. The simplest case assumes that any pointer may point to any object[10]. The other options in Soot are SPARK[41] and PADDLE[42]. Both these points-to analyses come with a large number of settings that one may fine tune. The paddle points-to analysis framework does not come bundled with Soot but has to be separately installed. By default both these frameworks are setup to use the most efficient implementations; a detailed description of the different options used to control the

[10]Respecting subtypes.

points-to is available at the online documentation [12] under the phase options.

Unfortunately, the points-to analyses have relatively long running times for incremental development of a tool such as the one developed in the this master's project. For a trivial program spark runs approximately two minutes[11] while paddle runs three to five minutes. These relatively large running times appear to be caused because of the need to analyze the Java standard library[12] even for trivial applications.

2.4.4 Obtaining more Information on Soot

This section surveys the resources available on Soot that have been helpful in the preparation of this master project. The starting point for all information on Soot is the official website [12]. Soot, its accompanying API documentation and sources and binaries and installation instructions are available at the official website. Also available are tutorials, published papers and technical reports in the Resources section.

Perhaps the best resource to understanding and using Soot is the PLDI 2003 tutorial [34]; indeed most of this section is based on it. [66, 65] are some of the first papers on Soot and discuss the different intermediate representations, namely, Baf, Jimple and Grimple and include the motivations behind each of these intermediate representations. The Shimple intermediate representation is discussed in [64]. Another useful resource on Soot is [25]. Although there is some overlap in the material from [25] and [34], [25] contains instructions to install and setup PADDLE, a points-to framework part of Soot, but distributed separately. Finally, the soot mailing list archives are a good source of information on using the Soot and it's API.

2.5 Related Work

Several tools have been developed to detect race conditions in multi-threaded programs. These tools may be classified into static or compile time, dynamic and hybrid. Each of these approaches differs in along the dimensions of *efficiency*, *precision* and *coverage*.

[11] These running times were verified on an Intel Pentium III 500 Mhz processor as well as a SPARC SUN Fire machine running under light load.

[12] See http://www.sable.mcgill.ca/listarchives/soot-list/msg01304.html.

Dynamic tools have high precision (no false positives) but poor efficiency and coverage (produce false negatives). They require instrumentation of the program code being analyzed and are therefore difficult to apply early in program development. Instrumentation may change the run-time behavior of the program leading to even more false negatives.

Static tools have high coverage as they make conservative approximations of program behavior but may produce a large number of false positives. The accuracy and of the tools and scalability is dependent on those of the underlying static analysis. Hybrid tools aim to combine the advantages of both approaches without the suffering from the disadvantages.

2.5.1 Dynamic Race Detection

Dynamic race detection techniques produce few or no false positives, but are inherently unsound. These tools report only race conditions observed in a single execution and therefore can not prove the absence of race conditions. By definition dynamic tools are difficult to apply early in program development either because the program is open and we lack client code or the program is closed and we lack sufficient input data. Dynamic race detectors may be broadly classified as *happened-before based*, *lockset based* and *hybrid* tools which combine both happened before and lockset approaches.

Happened-Before Based Tools

Happened-before based tools [14, 22, 13, 24, 47, 55, 58] are based on Lamport's happened-before relation [39]. The happened-before relation used to detect race conditions is a partial order on all events of all threads in a concurrent execution.

For a single thread, events are implicitly ordered by the order in which they occur in time. Between threads, events are ordered by constraints imposed by the synchronization objects they access. For example, a synchronization object providing mutual exclusion constrains all accesses in the critical section it guards such that they are related by the happened-before relation. If two threads access a shared variable and the accesses are not ordered by the happened-before relation, then we *may* get a race condition. Dynamic tools based on the happened-before relation monitor every data reference and synchronization operation and check for conflicting access to shared variables that are un-related by the happened-before relation for the execution being monitored.

Tools based on the happened-before relation have two major drawbacks; they are difficult to implement efficiently and their effectiveness is largely dependent on the actual interleavings of the program. These tools produce no false positives but produce a large number of false negatives.

Locket Algorithm Based Tools

Another class of dynamic race detection tools are based on the lockset algorithm, originally proposed by Dinning and Schonberg [24] to improve the happened-before approach, for programs that used common-lock based synchronization. A pair of accesses to the same memory location by two different threads are said to be involved in a race if the threads do not hold a common lock. Eraser [57] is a tool based on the lockset algorithm but introduces a program slowdown of 10-30X. Static and dynamic techniques, e.g., [15, 68, 67], have been used to significantly reduce the associated overhead with [21] having a run-time overhead of 13-42%. The main problem with a lockset based technique is that it produces many false positives when a synchronization idiom other than common-lock synchronization is used. Being a dynamic race detection technique it also suffers the problem of false negatives.

2.5.2 Static Race Detection

Static race detection tools offer much better program coverage than dynamic tools by conservatively approximating the program executions but may suffer from the problem of producing a large number of false positives. Previous approaches are either flow-insensitive type based systems or static versions of the lockset algorithm.

Type checking systems for race conditions have been developed for several languages [27], including Java [28]. In general tools based on type checking perform very well but require input from the programmer in the form of annotations. For example, static race detection in ESC/Java [29] works with programmer supplied annotations and a theorem prover to prove correctness of the program behavior with respect to the annotations. ESC/Java2 [3] is an extension of ESC/Java which analyzes JML-annotated[7] programs for common run-time errors including synchronization errors. Type based analysis tools perform relatively precise analysis, but have only been applied to small programs. However they are not many *real* bugs to find in small programs as the results of these tools, many of which are sound, conclude.

Another approach to statically avoiding race conditions is Guava [18] a race-free dialect of Java. Guava only allows instances of classes belonging to the *class category* called *monitor* to be shared by multiple threads.

Two static versions of the lockset algorithm for C are Warlock [62] and RacerX [26]. Warlock does not trace paths through loops or recursive functions while RacerX is intended for operating system code and uses heuristics to determine locks guarding particular memory location, code that may be executed by multiple threads and which memory accesses are safe. Findbugs [38, 4] is an approach similar to RacerX for Java but has a much larger scope.; it is an extensible framework that uses heuristics, data-flow analysis and pattern matching based approaches to analyze and detect *bug patterns*, e.g., synchronization errors in Java programs. RacerX has been used on large code bases, e.g., Linux (1.8 MLOC) and System X (500 KLOC) and is reported to have found 16 bugs in all, showing that it is capable of handling large programs but imprecisely.

Recent work in static race detection tools are Choi et al., [20], Chord [50] and LOCKSMITH [54]. LOCKSMITH, a race detection tools for C is based on the observation that locks are used to protect critical sections of code. The central approach is a path sensitive analysis to context sensitively infer a *consistent correlation* that relates locks to memory locations they guard. LOCKSMITH uses techniques for inferring thread locality, modeling locks in data structures and heuristics for modeling unsafe features like type casts. It has been used on POSIX applications ($<$7 KLOC) and Linux Drivers ($<$23 KLOC) and has found some race conditions with a few number of false positives.

Both Chord [50] and Choi et al., [20] use a pipeline of stages to filter out pairs of memory accesses potentially involved in a data race to a small and precise set. At a high level both use the same program analysis e.g., points-to analysis and thread-escape analysis but, while, [20] use context-insensitive analysis the authors of Chord have found context sensitivity central to their approach in producing useful results. The ordering of the phases used is different in Chord and [20]; Chord, which has been applied to larger programs more successfully, present the intuitive argument that the cheaper phases should be ahead in the pipeline for scalability. Chord has been applied to both large and open programs and claim to have reported more bugs related resulting from race conditions than previous static analysis tools. The tool from [20] has not been applied beyond small programs and produces a large number false positives for the larger benchmark program it is applied to.

In Chapter 4, we evaluate three tools, JLint [10], Esc/Java2 [3] and FindBugs [38] on a suite of "benchmark" programs, mostly example from Doug Lea's book[40], where we have deliberately introduced race conditions. These are freely available static analysis tools, which claim some race detection capability.

We also compare the results of our tool with those from Chord [50], a recent work in static race detection, that claim to have the best results for static race detection tools till now.

Chapter 3

Analysis and Design

In this chapter we discuss the analysis, desing and implementation for our tool. The analysis identifies the problems that need to be solved to statically detect race conditions keeping in mind the key properties we identify in Section 1.1. In the design and implementation part of the chapter we present the design of the different components of our tool.

3.1 Analysis

The tool to be developed aims to statically detect potential race conditions in concurrent Java programs. A number of different approaches for explicit concurrency and mutual exclusion exist and therefore, we must define the scope of the tool and the synchronization idioms that can be handled before we can analyze the sub problems that need to be solved.

3.1.1 Scope

Consider the trivial Java application given in Listing 3.1. It consists of a single class `SharedCounter` that is shared by two different threads. A race condition

```
class SharedCounter implements Runnable  {
    private final int MAX_COUNT = 500;
    private int counter;

    void inc(int i) {
        counter++;
    }

    void dec() {
        counter--;
    }

    int get() {
        return counter;
    }

    public void run() {
        while (true)
            if (get() > MAX_COUNT)
                dec();
            else
                inc(1);
    }
}

class Runner {

    public static void main(String[] args) {
        SharedCounter aSharedCounter = new SharedCounter();
        Thread threadOne = new Thread(aSharedCounter);
        Thread threadTwo = new Thread(aSharedCounter);

        threadOne.start();
        threadTwo.start();
    }
}
```

Figure 3.1: A trivial java application with race conditions

may occur when the methods `get()` and `inc()` are executed simultaneously by the different threads, i.e., we have a simultaneous read-write on the variable `counter`. For the race condition to occur, the following necessary conditions must hold:

1. The *same* memory location is accessed by *different* threads.
2. At least one of the memory accesses is a write.
3. The threads do *not* hold a common lock.

Java recommends monitor style synchronization for concurrent programs via the `synchronized` keyword. Using `synchronized` makes locking lexically scoped, i.e., there is a syntactic relation between the statements guarded by a `synchronized` block and the lock used. Use of non-lexically scoped locking via specialized lock constructs or semaphores requires one to make a flow-sensitive static analysis to determine the locks held for a given memory access. As the results from tools like [54, 26] indicate, flow-sensitive interprocedural analyses are more complicated, especially when dealing with multiple threads, and have not been successful in finding many race conditions even in large programs. Use of non-lexically scoped locking is much more common for languages like C, C++ than for Java. Therefore, we restrict the scope of this tool to handle locking based on the `synchronized` keyword. We believe that this is a pragmatic approach and will cover a large variety of programs.

3.1.2 Problem Decomposition

The necessary conditions for a race condition give us some insight for decomposing the problem of statically detecting race conditions into sub problems.

The first problem that needs to be solved is detecting and finding the different threads in a Java application (*thread finding*). Once the different threads are known, one must determine as precisely as possible the methods that a particular thread may visit (*thread whereabouts*). Knowledge of a thread's whereabouts requires an accurate call graph for the application (*call graph*); the methods visited by a thread is a slice of the program call graph.

On determining a thread's whereabouts, we can examine the methods visited and collect information about the fields that the thread may read or write (*thread memory-accesses*). We are interested in static and instance fields of classes that a thread may access.

Now that the sub-problems thread finding, thread whereabouts, call graph and thread memory-accesses are defined, we need to determine whether two memory access by different threads may interfere, i.e., be involved in a potential race condition (*race finding*). From the necessary conditions for a race, we need to determine:

1. If the two memory accesses are to the same instance?
2. If the two memory accesses are made by different threads?
3. If the two threads hold any common locks?

To solve the race-finding problem we need a points-to analysis, which can statically determine whether the two references/pointers may be aliases of each other. We also require a mapping between a program statement and the locks held when the statement is being executed by a particular thread (*locks-held analysis*). The locks-held analysis and the points-to analysis will tell us whether a common lock is held when different threads access a shared memory location.

Given solutions to the problems in this section, we could determine whether two memory accesses may be involved in a race by checking if the necessary conditions for a race hold for these memory accesses.

3.1.3 Data Flow for the Tool

Figure 3.2 shows the flow of data in the tool based on the problems identified in the previous section. The boxes in the diagram represent processes and the ellipses data. Notice that most of the processes in the figure are dependent on the call graph; this is to be expected as the analysis that we need to perform are interprocedural.

Notice also that the race finding process uses the data from the locks-held analysis, the points-to analysis and the memory-access finder to check the necessary conditions for a race conditions and report the results accordingly. The locks-held result and memory -access result should be considered as maps keyed on a syntax element, e.g., a statement or a variable used, to the property they represent.

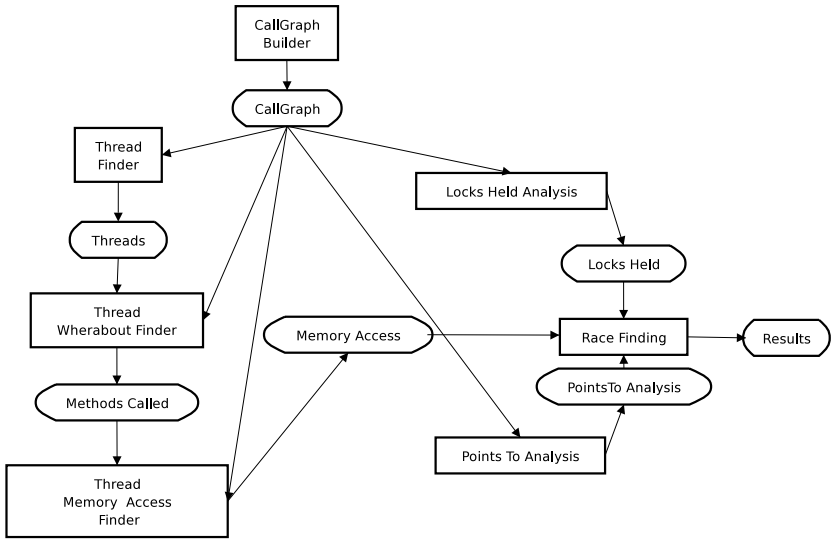

Figure 3.2: Flow of data among various entities in the tool. There is one entity per sub problem defined in Section 3.1.2

3.1.4 Architecture

Considering the data flow in the tool a natural architecture would be a number of stages connected via pipes as shown in Figure 3.3. The processes in the data flow diagram, Figure 3.2, are candidates for stages of the pipeline but may require to be broken down further into stages or merged into a single stage, depending on the design and implementation requirements. For some stages, e.g., call graph building the stage would need to complete, i.e., the call graph be completely available before the next stage(s) dependent on the call graph may be started. For other stages, e.g., memory-access finding, not all memory accesses by different threads need to be available before they can be checked for race conditions.

This architecture has several advantages as it is extensible; stages may be added or removed as per execution time versus precision or memory requirement trade-offs. One may have multiple implementations for a single stage, which may be layered using the Decorator pattern [30].

This architecture also offers scope for parallelization; different stages may be performed in parallel depending on their input output relation to the other stages. For example, stage A may be performed in parallel with stages 2 to $n-1$.

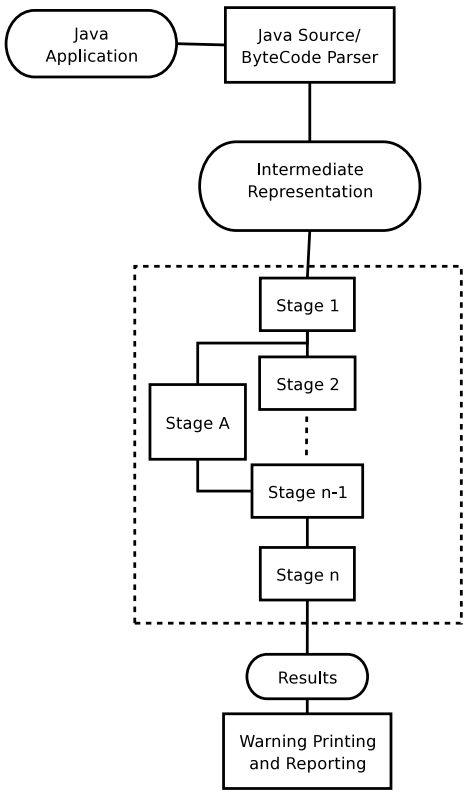

Figure 3.3: Architecture for the tool

```
void inc(int i) {
    counter++;
}
```

Figure 3.4: Method For Baf Body
Method for Baf body in Listing 3.5

Other advantages of this architecture are scalability and response time. While response time is not an important criteria for the tool, previous works shows large memory requirements and polynomial running times affecting scalability. For example, a sufficiently accurate points-to analysis requires $O(n^3)$ running time [16]. A pipelined architecture handles scalability by providing a framework to implement *flow control*, where upstream stages can block a downstream stage when a downstream stage produces data exceeding current memory requirements (provided the output from a downstream stage can be consumed before it is completely available).

3.2 Design and Implementation

3.2.1 Choosing an Intermediate Representation

Our tool builds on top of Soot using it to parse Java source or byte code into an intermediate representation (IR). There are a number of different IRs available, namely Baf, Grimple, Shimple and Jimple, and the first step in the design is choosing the appropriate one.

Baf

The first IR considered is BAF, a stack based compact representation of byte code. BAF is immediately rejected as we do not want to deal with a stack in the IR. A stack based IR is hard to analyze as it requires a flow sensitive analysis and interpretation of the stack. This arguments is made concrete by examining the Baf body in Listing 3.5 for the method `inc()` in Listing 3.4.

Notice that to determine the reference used to access a field, one needs to look behind one instruction and for the general case statically determine the top of the stack at each program point of interest.

```
void inc(int)
{
   word this, i;

    this := @this;
    i := @parameter0;
    load.r this;
    load.r this;
    fieldget SharedCounter.counter;
    push 1;
    add.i;
    fieldput SharedCounter.counter;
    return;
}
```

Figure 3.5: Baf body for method `inc()` in Listing 3.4

We would rather avoid this complexity by using a more suitable intermediate representation.

Grimple

The second IR considered is Grimple, which unlike Baf is not stack based and is very similar to the actual source code of the program. The main feature of Grimple is that expressions are represented in their *aggregated* form, i.e., an expression of the form $a := b + c + d$ is represented as is and not broken down into its constituent expressions and corresponding load and store type instructions. However, we would like any single statement to contain *at most one* field reference. This property makes the intermediate representation simpler to analyze as every statement has a single effect with respect to reading and writing of fields. In fact the only documented use of Grimple found by us is facilitating analysis that remove redundant expression calculations like available expression analysis or very busy expression analysis[1].

Shimple

Shimple is a typed, three address code, static single assignment (SSA) intermediate representation. Being three address code it also has the desired property of containing at most one field reference per statement. SSA form allows a flow-insensitive points-to analysis to give comparable results to that of a flow-sensitive analysis [32]. Unfortunately, we can not use Shimple, as the call-graph

[1]We have not come accross any optimizations based on Grimple in Soot version 2.2.3.

construction and built in points-to analysis frameworks use Jimple[2].

Jimple

With Jimple we run out of choices for IRs, but fortunately, Jimple has all the required properties. It is the *primary* intermediate representation in Soot and call graph construction and the points-to analysis frameworks are built using Jimple. In Soot 2.2.3 Jimple is the only IR for which interprocedural program analysis are available. Jimple is typed three address code and has the property of a single field reference per statement. Soot also contains a feature known as *local splitting* for Jimple that improves the precision of flow-insensitive analysis, although less than SSA, but without the cost of having to calculate *phi-nodes* as in SSA [3].

3.2.2 Call Graph Building

The tool requires us to make interprocedural analysis and therefore we need an accurate call graph for the application being analyzed. As Java allows dynamic dispatch on run-time types, we require a control- flow analysis to construct a call graph. A full blown control-flow analysis for Java is a daunting task and we would like to build on the existing call-graph building infrastructure available in Soot.

Reusing Soots Existing Call Graph Infrastructure

In Soot the call graph is built *on-the-fly* and in two phases. First a fast but imprecise points-to analysis is used to create an initial call graph which is refined using, SPARK or PADDLE, the points-to analyses frameworks part of Soot. The points-to analysis is used to determine the run-time types of the *objects* involved in a method invocation. The call graph construction also adds method calls for class initializers, implicitly called when a class is loaded by the Java Virtual Machine.

Unfortunately, experiments with the existing call-graph building infrastructure were disappointing, requiring run times of 3-5 minutes and a minimum of 512 MBytes of heap space, even for trivial Java applications.

[2]This argument also holds for Grimple and Baf.
[3]See http://www.sable.mcgill.ca/pipermail/soot-list/2006-July/000740.html.

As discovered from the Soot mailing list archives a large amount of time during call graph construction is spent on virtual method call resolution for Java standard library version 1.4 and onwards [4]. Therefore, we do not want to construct a call graph for the Java library, only for our application classes; any library classes that need to be analyzed could temporarily be treated as application classes. Implicit methods like class initializers do not need to be analyzed for race detection as these methods are, by definition, thread safe.

Our Approach

Based on the observations from the previous section we use Rapid Type Analysis (RTA) [19] on a fast but imprecise, application-only[5] call graph to obtain a more precise call graph. First, we build an imprecise interprocedural call graph $ICG_1 = (m_{main}, M, E)$ for the application classes using Soot's call graph builder. ICG_1 is defined as:

- $M = \{m \mid m : \texttt{SootMethod}\}$ is the set of nodes of the call graph
- $E = \{(m_{src}, m_{tgt})\} \subseteq M \times M$ is the edge relation
- m_{main} is the entry point into the call graph

Using ICG_1 we caclulate the methods reachable from m_{main} and the classes that may be created in these methods as:

- $ReachableMethods = \{m \mid m \in TransitiveClosure(m_{main})\}$
- $CreateableClassess = \{c \mid isInstantiatedIn(c, m) \wedge \in ReachableMethods\}$

The $CreateableClassess$ is then used as a filter for the $ReachableMethods$ to caclulate $RTAReachableMethods \subseteq ReachableMethods$ as:

$RTAReachableMethods = \{m \mid m \in ReachableMethods \wedge declaringClass(m) \in CreateableClassess\}$

Finally, $RTAReachableMethods$ may be used to obtain $ICG_2 \subseteq ICG_1$, $ICG_2 = \{m_{main}, M', E'\}$ where:

[4] See http://www.sable.mcgill.ca/listarchives/soot-list/msg01304.html.

[5] Classes in the `java.*` and `sun.*` packages are considered application classes.

- $M' \subseteq M, M' = RTAReachableMethods$
- $E' \subseteq E, E' = \{(m_{src}, m_{tgt}) \mid m_{tgt} \in RTAReachableMethods\}$
- m_{main} is the entry point into the call graph

Evaluation of Call Graph Building

Our approach gives us a reasonably accurate call graph very cheaply. The execution time is of the order of 100 milliseconds, for all evaluated programs, and many spurious edges are removed by RTA; for the `run()` method part of the `Runnable` interface RTA removes approximately 30 classes from the Java standard library that also implement `Runnable` but are not instantiated by the Java application as they are specific to the standard library classes.

At a high level our design for call graph building is the same as in Soot, both approaches use two passes, the first pass produces a cheap but imprecise call graph and the second pass refines the result of the first by approximating the points-to information for references.

The improvement in execution time is because of two reasons. The dominating one for small Java applications is abstraction of the call graph to disregard the standard library. The second, is the use of RTA that uses an $O(n)$, where n is the number of statements in the application, algorithm and to get a coarse grained, yet sufficiently useful approximation of the points-to analysis to resolve virtual method calls.

Call Graph Traversal Strategy

The data flow diagram Figure 3.2 shows that the call graph is used by every component in the tool and therefore a generic call graph traversal component should be introduced to avoid duplication of code in the other components. There are three possibilities for a generic call graph traversal strategy, namely, simple iteration (already available in Soot), breadth first and depth-first traversals.

From a flow-insensitive and context-insensitive perspective all the above mentioned strategies are equivalent but depth-first traversal has a useful property at no extra cost in complexity. As depth-first traversal mimics the execution of a sequential program, one may use it and stack calling contexts during traversal.

This may be useful for a context sensitive analysis or for obtaining stack traces, e.g., paths leading to memory accesses.

The call graph traversal strategy has two main responsibilities. First, to traverse the call graph depth first and keep a track of the calling contexts. Second, to allow a client code to analyze the current method being *visited*. A generic strategy separating traversal from method visiting is achieved by using the Listener pattern [30], where a client of the call graph traversal is *called back* when a method is visited, and may analyze the method as required.

The pseudo code for the call graph traversal strategy is given in Algorithms 1 and 2:

Algorithm 1 visitMethods($mainMethod$)

$seenMethods := seenMethods \cup mainMethod$
Push $mainMethod$ onto $callerStack$
Notify listeners before call
doVisitMethods($mainMethod$)
$seenMethods := seenMethods \setminus mainMethod$
Notify listeners after call
Pop $callerStack$

The pseudo code requires further elaboration. The algorithm is implemented using to functions **visitMethods** a driver function, and **doVisitMethods** the function that implements the actual depth first traversal. Notice that in both methods the method to be visited is pushed onto the $callerStack$, which maintains the current call context.

Algorithm 2 doVisitMethods($currentMethod$)

Notify listeners in method.
for all Methods $callee$ called by $currentMethod$
  if $callee \in seenMethods$
    Notify listners in Method; indicate method recursively called
  $seenMethods := seenMethod \cup callee$
  Push $callee$ onto $callerStack$
  Notify listeners before call
  doVisitMethods($callee$)
  $seenMethods := seenMethods \setminus callee$
  Notify listeners after call
  Pop $callerStack$

The method listeners are called at three distinct points for each methods. These are:

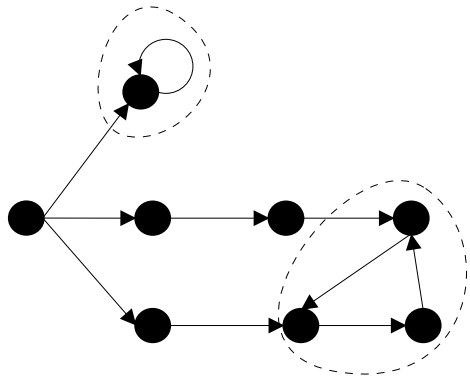

Figure 3.6: Some examples of loops in the call graph due to recursive function calls that are detected by the call graph traversal algorithm

- *Before* the method is called — Notify listeners before call.
- *In* the method call — Notify listeners in method.
- *After* method call — Notify listeners after method call.

Notice that in both the methods, **visitedMethods** and **doVisitedMethods** the method that will be visited is pushed onto the *callerStack before* the "Notify before call" and popped from the stack *after* the "Notify after call." This is a design decision so that the method to be visited, currently being visited, and visited is always on top of the caller stack.

The $seenMethods$ set is used to track and avoid infinite looping during traversal due to cycles in the call graph. The method to be visited is added to the $seenMethods$ set before it is visited and removed from it after it has been visited. Removing the method guarantees that the contents of the $seenMethods$ are specific to the call path taken by a particular method only. Figure 3.6, shows some possible loops that may occur in the call graph; these are all detected by using the $seenMethods$ set.

3.2.3 Locks Held Analysis

The locks-held analysis should answer the question "What locks are held for a particular statement executed by a thread?" Therefore it is a map:

$locksHeldAnalysis_1 : (stmt \times thread \times context) \xrightarrow{m} lock$-**set**

Since we consider only lexically-scoped locking via the `synchronized` keyword the locks held at a particular statement are independent of the thread and the context and we may simplify the map to:

$locksHeldAnalysis_2 : stmt \xrightarrow{m} lock\text{-}\mathbf{set}$

The fact that $locksHeldAnalysis_2$ is keyed on *stmt* seems to suggest that all threads executing *stmt* hold the same locks in all contexts. Every thread executing *stmt* acquires a lock via the same syntactic element, e.g., a variable, field or the `this` reference. To determine exactly which *objects* the locks have been acquired for, one must ask the points-to analysis and therefore, the implementation for *locks* must contain at least the information that is required as input to the points-to analysis.

Lexically scoped locking may be used in several different ways. At a minimum the locks-held analysis should consider the most common of these, namely, synchronized methods, blocks and methods invoked from synchronized methods and blocks. To handle these situations we traverse the methods in the call graph depth first maintaining a stack of *lock*s held at each statement. This requires us to modify the mapping $locksHeldAnalysis_2$ to use a stack instead of a set as:

$locksHeldAnalysis_3 : stmt \xrightarrow{m} lock\text{-}\mathbf{stack}$

Entry (exit) from synchronized methods or blocks causes a *lock* to be pushed (popped) from the stack. With each statement we associate a snapshot of the current *lock* stack.

Memory Usage Optimization

The locks-held analysis outlined above requires $O(n)$ stacks, where n is the number of statements in the application being analyzed. Clearly n will be large for real applications while the size of the stack should typically be less than ten. Therefore, we attempt to reduce n and if required the number of stacks created.

To reduce the memory requirement, we observe that not all statements require us to know the locks held at the particular statement, only those that contain field references. Although tracking only field reference is still $O(n)$ in memory requirements[6], the actual memory required is reduced by a factor of 0.80 for the evaluated programs, as the number of statements with field references in these programs averages at 20%.

[6]Every statement in the program can contain a field reference.

```
void f() {
    for (...) {
        new Thread().start();
    }
}
```

Figure 3.7: Multiple threads started in a loop by a single statement

```
void f() {
    new Thread().start();

}

void g() {
    f();
}

void h() {
    for (...) {
        f();
    }
}
```

Figure 3.8: Multiple threads that may be started by a single statement in a method called in a loop

To reduce the number of stacks created we could use the Flyweight or Prototype pattern[30], but as the size of the stacks is small, it is only a marginal reduction in memory at added cost of implementation complexity and not done in the current version of the tool.

3.2.4 Threads and Memory Accesses

Thread Finding

We require to find the threads that may be started in the Java application and the memory accesses for each of these threads. In general the number of threads created and started is both context and flow sensitive, as can be seen from examples in Listing 3.7 and 3.8.

In Listing 3.7 we see that multiple threads are started at the same statement and one must perform a flow sensitive analysis to detect them.

Listing 3.8 shows that multiple threads *may* be started at the same statement and to detect them one must perform a flow and context-sensitive analysis.

Finding threads using a context sensitive and flow sensitive analysis is complex and only beneficial if the points-to analysis is also both context and flow sensitive as every thread is also an object in the application. Therefore, we use a context and flow insensitive approach to find threads in the program assuming that a statement of the following type identifies a *single* thread:

```
<thread-subtype-reference>.start();
```

The implication of the above assumption is that we are unable to detect the case when multiple threads are started by a single statement, e.g., the statement is in a loop or the invoked in many different contexts. To handle the situation where multiple threads are started at the same statement, we could make an over approximation that *all* statements where a thread is started identify multiple threads. However, the usefulness of this over approximation for race detection depends on whether we can distinguish and identify the objects that may be accessed by the different threads.

Thread Memory Accesses

Once the threads in an application have been located the memory accesses made by each thread may be recorded using two different maps:

$readMap : field \xrightarrow{m} MemoryAccess$
$writeMap : field \xrightarrow{m} MemoryAccess$

The maps are populated by visiting the methods called by a thread depth first and recording the fields that may be read or written by the thread. The depth first traversal is useful to record a stack trace leading to the memory access; the maps may also be populated by simply iterating over the methods called by a thread.

Memory accesses that may be involved in a race are either to a member or static field. As we do not know what objects are accessed when recording memory references, $MemoryAccess$ must contain at least the information required as input to the points-to analysis so that the object involved in the memory access can be determined.

3.2.5 Points-to Analysis

Central to our tool is the points-to analysis that determines what objects a reference *may* point to. Soot provides two points-to analysis frameworks, SPARK and PADDLE, but long running times even for trivial programs, (Section 3.2.2), required us to implement our own points-to analysis from scratch.

Choosing a Points-to Analysis

There is a large body of work on points-to analysis [35]; available literature suggests that Andersen's analysis [16] (inclusion-constraint based) and Steensgaard's analysis [61] (unification based) are two popular options for Java. We choose to base the points-to analysis on Andersen's analysis for several reasons. First, the work of Shapiro and Horwitz [60] and Hind and Pioli [36] found that inclusion constraint based points-to analysis, like Andersen's, produces more precise results than unification-based analysis like Steensgaard's. Second, Liang et al., [45] found that improving the precision of Andersen's analysis would be hard without using the more expensive shape analysis techniques. Third, Lhoták and Hendren [43] state that the speed of Andersens's analysis can be improved by filtering based on programming language types.

Andersen's Analysis for Java

There are a number of papers that describe Andersen's analysis for Java [37, 48, 44, 56, 43, 69]. Our points-to analysis is based on the framework presented in [37] for two main reasons. First, [37] present the data structures used to represent the pointer assignment graph and an algorithm for a constraint propagator, which makes their presentation and framework friendlier than those presented in the other papers. Second, the points-to analysis developed in [37] is done for the Jikes RVM [9], an open source virtual machine written in Java and therefore, their data-structures and algorithms are more amicable for implementation in Java, the language of implementation for our tool.

Overview

Instantiations of Andersen's analysis (inclusion-constraint based analysis) consist of two steps as shown in Figure 3.9. First, a *constraint finder* analyzes the static program code, building a representation that models the constraints

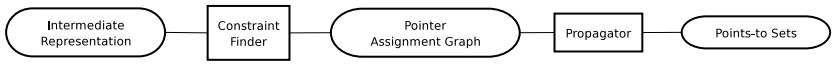

Figure 3.9: The two steps in calculating an inclusion-constraint based analysis

Node Kind	Concrete Entity Represented	Flow sets	Points-to sets
h-node	Set of heap objects, e.g., a heap objects allocated at a particular allocation site	none	none
v-node	Set of program variables, e.g., all occurrences of a local variable	flowTo[v], flowTo[$v.f$]	pointsTo[h]
$h.f$-node	Instance field f for all heap objects represented by h	none	pointsTo[h]
$v.f$-node	Instance field f of all h-nodes represented by v	flowFrom[v], flowTo[v]	none

Table 3.1: PAG representation

arising from the data flow in the program. This representation is known as a *Pointer Assignment Graph* (PAG). Second, a *propagator* processes the PAG to obtain a solution of points-to sets for the references in a program.

PAG Representation

Our PAG has four basic node types. The constraints, or edges, themselves are stored as sets at the nodes. Table 3.1 describes the nodes, introduces the notation used and shows which sets are stored at each node. The [...] show the kind of nodes that are stored in the flow and points-to sets.

The constraint finder models the program code by v-nodes, $v.f$-nodes and their flow sets. Based on these, the propagator computes the point-to sets for v-nodes and $h.f$-nodes. For example, if a client of the points-to analysis would like to know if a local variable o points to an object allocated at site a, it would check if the pointsTo[h] set of the v-node for o contains the object.

Points-to sets represent the set of objects (r-values) that a pointer (l-value) may point to and are associated with v-nodes and $h.f$-nodes. Storing the points-to sets with $h.f$-nodes instead of $v.f$-nodes makes the analysis field-sensitive [43]. Flow-to sets represent a flow of values (assignments, parameter passing, etc.) and are stored with v-nodes and $v.f$-nodes. For example, $v'.f \in \text{flowTo}(v)$

corresponds to the assignment, $v = v'.f$. Flow-from sets are the inverse of Flow-to sets. Arrays are handled by specializing the nodes given in Table 3.1. Each array h-node is associated with a special $h.f_{elems}$ node that stores *all* the h-nodes that have been assigned to the array. Thus, the analysis does not distinguish different elements in the array.

There are many choices for how to implement sets. The SPARK paper [43] evaluates various data structures for representing points-to sets and suggests that *hybrid* sets (using lists for small sets and bit vectors for large sets) yield the best results. The authors of [37] find the *shared bit vector* representation for sets from CLA[33] even more efficient than hybrid sets from SPARK. We observe these results but use the Java standard library `HashSet` implementation for the first version of the tool.

Constraint Finding

The PAG representation used for the point-to analysis is very close to the representation from [37]. However, our approach differs considerably in the way the constraints are discovered and the PAG created.

In [37] constraint finding is implemented as a flow-insensitive pass of the optimizing compiler, part of the Jikes RVM. As they deal with dynamic class loading in their framework, constraint finding also occurs at runtime, e.g., during class loading and reflection. Interprocedural constrains from data flow due to parameter passing and return values are handled by resolving virtual method calls using Class Hierarchy Analysis (CHA).

In our approach, we traverse the reachable methods of the Java application depth first using the call graph created using Rapid Type Analysis (RTA). RTA is more precise than CHA, especially for programs with extensive use of inheritance and/or interfaces. During depth first traversal, each method visited is analyzed for intraprocedural constraints using the rules in Table 3.2.

The column "Actions" in Table 3.2 gives the actions of the constraint finder when it encounters a statement in the column "Statement." The column "Constraint" shows the constrains implicit in the in the actions of the constraint finder. Notice that the constraint is represented by inclusion in the flow set of the corresponding node.

The constraint finding and PAG construction is flow-insensitive but we get results comparable to a flow-sensitive analysis because of a Soot feature known as *local variable splitting*, which splits the uses and definitions of local variables

Statement	Actions	Constraint
$v' = v$	flowTo(v).add(v')	pointsTo(v) $\subseteq$ pointsTo(v')
$v' = v.f$	flowTo($v.f$).add(v')	$\forall h \in$ pointsTo(v) : pointsTo($h.f$) $\subseteq$ pointsTo(v')
$v'.f = v$	flowTo(v).add($v'.f$), flowFrom($v'.f$).add(v)	$\forall h \in$ pointsTo(v') : pointsTo(v) $\subseteq$ pointsTo($h.f$)
$l : v = new$ ...	pointsTo(v).add(h_l)	$\{h_l\} \subseteq$ pointsTo(v)

Table 3.2: Intraprocedural constraints and actions

according to webs [49]. In effect a distinct local variables is used for each dereference context and a variable is only assigned one value that will be used. This is crucial for precision in flow-insensitive analysis as *all* assignments lead to a value flowing into the variable assigned, even if a new assignment overwrites an existing one. The authors of [37] might have had a similar property with their intermediate representation, high-level register based intermediate representation (HIR) and as they state "HIR breaks down access paths by introducing temporaries so that no access path contains more than one pointer dereference."

Interprocedural constraint finding. We now consider interprocedural constraints to handle data flow through parameter passing and return values. For every method, after finding the intraprocedural constraints, the call sites in that method are determined and virtual methods resolved using RTA. For each virtual method that may be invoked, constraints are added corresponding to the assignment of actual to formal parameters. Similarly, for methods that return a value, a constraint is added from the variable containing the return value to the variable receiving this value. In both cases, parameter passing and return values, constraints are only created for assignments to reference types[7]. A constraint is also created corresponding to the `this` reference for instance method calls, i.e., an assignment constraint is created from the receiver of a method call to the `this` reference of all methods that may be called.

While handling return values from methods, we may have to handle *external method calls*; methods for which the intermediate representation is not available, e.g., standard library. We summarize the effect of such methods and make a conservative approximation that each such method returns the same object on every invocation.

[7]Primitive types can only be created on the stack and can not be shared by different threads.

```
class Foo {

    public Object run(Object formalParam) {
        Integer retVal;
        System.out.println("Thanks for " + formalParam);

        if (true) {
            /** 3: */ retVal = new Integer(0);
        }
        else {
            retVal = formalParam;
        }

        return retVal;
    }

    public static void main(String args[]) {
        /** 1: */Foo foo = new Foo();

        /** 2: */Object actualParam = new Object();

        Object value = foo.run(aParam);
    }
}
```

Figure 3.10: A sample java program. the pointer assignment graph for this program is given in Figure 3.11

PAG example. Listing 3.10 contains a small program for which the pointer assignment graph is given in Figure 3.11.

The solid and hollow nodes represent v-nodes and h-nodes respectively. The dashed arrows represent interprocedural constraints, due to method calls and returns, while the solid arrows represent constraints due to intraprocedural assignments. Notice that the h-nodes are identified by the statement at which they are created, i.e., a statement where an object is created represents *all* the objects that may be allocated at that site. Finally, there are two constraints for the v-node of `retVal` corresponding to the two branches of the if block as the constraint finding is flow insensitive.

Propagator

The propagator finds the solution for the points-to sets satisfying the constraints in the PAG. In formal terminology, it finds the *least fixed point* of the system of subset constraints implicit in the flow sets. The propagator is implemented using Algorithm 3, it is a modified version of the algorithm used in [37]. Note that the flow-to (flow-from) sets are used in connection with r-values (l-values); the content of a flow-to (flow-from) set is all the l-values (r-values) a v-node or

Algorithm 3 Constraint propagator

```
1: while workist not empty or isCharged(h.f) for any h.f-node
2:   while workist not empty
3:     remove first node v worklist
4:     for all v' ∈flowTo(v)
5:       pointsTo(v').add(pointsTo(v))
6:       if pointsTo(v') changed
7:         add v' to worklist
8:     for all v'.f ∈flowTo(v)
9:       for all h ∈pointsTo(v')
10:        pointsTo(h.f).add(pointsTo(v))
11:        if pointsTo(h.f) changed
12:          isCharged(h.f) ← true
13:    for all each field f of v
14:      for all v' ∈flowFrom(v.f)
15:        for all h ∈pointsTo(v)
16:          pointsTo(h.f).add(pointsTo(v'))
17:          if pointsTo(h.f) changed
18:            isCharged(h.f) ← true
19:      for all v' ∈flowTo(v.f)
20:        for all h ∈pointsTo(v)
21:          pointsTo(v').add(pointsTo(h.f))
22:          if pointsTo(v') changed
23:            add v' to worklist
24:  for all v.f
25:    for all h ∈pointsTo(v)
26:      if isCharged(h.f)
27:        for all v' ∈flowTo(v.f)
28:          pointsTo(v').add(pointsTo(h.f))
29:          if  pointsTo(v') changed
30:            add v' to worklist
31:    for all h.f
32:      isCharged(h.f) ← false
```

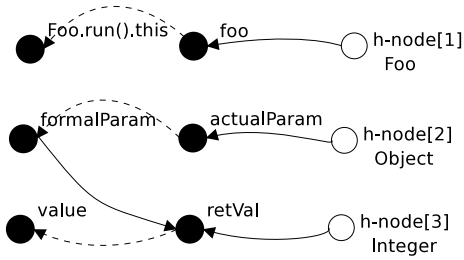

Figure 3.11: Pointer assignment graph for the code in Listing 3.10

$v.f$-node it may be assigned to (from).

The constraint finder populates the worklist, which already contains nodes when the propagator starts. The propagator puts a v-node onto the worklist when its points-to set changes. The propagator does not add any constraints during propagation and hence the constraint finder and propagator are independent of each other, sharing only the worklist. In Algorithm 3 Lines 4-12 propagate the v-node's points-to set to nodes in its flow-sets, i.e., nodes that it has been assigned to. Lines 13-23 update the points-to set for all fields of objects pointed to by the v-node. This is necessary, because for the h-nodes that have been newly added to $v's$ points-to set, the flow to and flow from $v.f$ carries over to the corresponding $h.f$-nodes.

The propagator sets the isCharged-bit for an $h.f$-node to true when its points-to set changes. To discharge an $h.f$-node, the algorithm needs to consider all the flow-to edges from all $v.f$-nodes that represent it (lines 24-30). This is the reason why the algorithm does not keep a worklist of changed $h.f$-nodes; to find their flow-to targets it needs to iterate over $v.f$-nodes anyway.

The constraint finder creates the PAG nodes during constraint finding. An h-node and corresponding $h.f$-nodes, including those that are inherited, are created when a `new` statement is encountered. An alternative to this is to create $h.f$-nodes lazily, the first time an element gets added to their points-to sets as in [37]. A v-node does not contain all the fields of the type it represents, instead the fields part of a v-node are all the those that it has been assigned from. This reduces the amount of iterations in line 13 of the propagator algorithm, as all fields of created v-nodes are not considered, only those involved in data flow.

3.2.6 Finding Race Conditions

Recall that the necessary conditions for a race condition for involving two memory accesses are:

- The two threads access the same memory location.
- The threads hold no common locks.
- At least one of the memory accesses is a write.

With the thread and thread memory-access finding, locks-held analysis and the points-to analysis, we have the infrastructure available to detect race conditions; this is done using Algorithm 4.

Algorithm 4 Race Condition Finding

```
1:  INPUT thread-set, pointsToAnalysis, locksHeldAnalysis
2:  threadPairs = {(t1, t2) | (t1, t2) ∈ thread-set × thread-set ∧ t1 ≠ t2}
3:  for all (t1, t2) ∈ threadPairs
4:      rw = {(r1, w2) | ∀field ∈ (t1.readMap.keys ∩ t2.writeMap.keys) ·
        (r1, w2) ∈ (t1.readMap(field) × t2.writeMap(field))}
5:      for all (r1, w2) ∈ rw
6:          if mayAlias(r1, w2) ∧ noCommonLockAt(r1, w2)
7:              report race condition at (r1, w2)
8:      ww = {(w1, w2) | ∀field ∈ (t1.writeMap.keys ∩ t2.writeMap.keys) ·
        (w1, w2) ∈ (t1.writeMap(field) × t2.writeMap(field))}
9:      for all (w1, w2) ∈ ww
10:         if mayAlias(w1, w2) ∧ noCommonLockAt(w1, w2)
11:             report race condition at (w1, w2)
```

Algorithm 4 is a straight-forward implementation, checking the necessary conditions for race conditions; *threadPairs* is a set of all distinct threads found by our thread finding component. For each pair of threads, we find the simultaneous[8] read-write and write-write to the same memory location, a class field, and check if the references involved may be aliases. If so, and the threads do not hold a common lock, a race condition is reported. The *mayAlias*() and *noCommonLockAt*() functions use the points-to analysis and locks-held analysis; their implementation is not shown here.

[8] All memory accesses by different threads are assumed to be simultaneous.

```
class SharedCounter implements Runnable  {
    private final int MAX_COUNT = 500;
    private int counter;

    void inc(int i) {
        this.counter++;          // this.pointsTo_h = {H[1], H[2]}
    }

    void dec() {
        this.counter--;          // this.pointsTo_h = {H[1], H[2]}
    }

    int get() {
        return this.counter;     // this.pointsTo_h = {H[1], H[2]}
    }

    public void run() {
        while (true)
            if (get() > MAX_COUNT)
                dec();
            else
                inc(1);
    }
}

class Runner {
    public static void main(String[] args) {
 /* 1: */ SharedCounter scOne = new SharedCounter(); //H[1] allocation site
 /* 2: */ SharedCounter scTwo = new SharedCounter(); //H[2] allocation site

 /* 3: */ Thread threadOne = new Thread();
 /* 4: */ Thread threadTwo = new Thread();

 /* 5: */ threadOne.start();
 /* 6: */ threadTwo.start();
    }
}
```

Figure 3.12: Imprecision of context insensitive points-to analysis

3.2.7 Imprecision of Context-Insensitive Analysis

The race condition finding algorithm depends on the points-to analysis for determining if a common object is accessed and if a common lock is held. The precision of the tool, especially the false positives reported, depend on the precision of our points-to analysis. The points-to analysis is context insensitive and basic object oriented features and programming idioms will cause the it produce imprecise results.

Consider the example in Listing 3.12 where we have two instances of the class `SharedCounter` running in distinct threads and therefore no race conditions.

A context-insensitive points-to analysis merges all information that flows into the `this` reference for instance methods of `SharedCounter`. The race condi-

tion finding algorithm will *incorrectly* determine that the heap objects created at location 1 and 2 may be accessed concurrently.

Use of inheritance may also lead to imprecision as the fields of the base class are inherited by all the derived classes. Container classes, e.g., `HashTable` or `Vector`, commonly used in Java will also lead to imprecise results as all objects contained in any container class will be merged due to context insensitivity.

Early experiments with the tool demonstrated the need for context-sensitivity in the points-to analysis. Indeed, this is confirmed by the results of [50] and [20]; [50] claim that context-sensitivity is imperative in their static analysis tool and [20], who use a context-insensitive points-to analysis, have a large false positive rate, 587 potential data races for a program with 990 heap accesses, none of which are actual data races.

3.2.8 Implementing Context Sensitivity

The main design goal while implementing context sensitivity is to reuse the existing context-insensitive points-to analysis implementation as far as possible and therefore we use call sites to distinguish the context for method calls. Call sites allow us to provide context sensitivity with minor modification to the constraint finder and no modifications to the propagator. It is implemented by using two mappings:

$(local \times context) \xrightarrow{m} v\text{-node}$

$(field \times context) \xrightarrow{m} v.f\text{-node}$

The *context* is a list of call-sites leading to a memory access and *local*, $field$ is the local variable or field, involved in the memory access. This mapping is independent of the PAG nodes, propagator and the constraint finder. The *context* is readily available during constraint finding as we use a depth-first traversal over the methods called. A method invocation at a distinct call site causes a fresh set of v-nodes to be created, in effect cloning parts of the PAG. This can be seen in Figures 3.13 and 3.14. Figure 3.13 contains a context-insensitive PAG for the data flow in method `g()` with a single sub graph for all contexts of the method.

Figure 3.14 shows that the PAG representing the data flow in method `g()` is cloned for the different contexts it is called in.

We observe that the call-site approach for context sensitivity is inferior to object

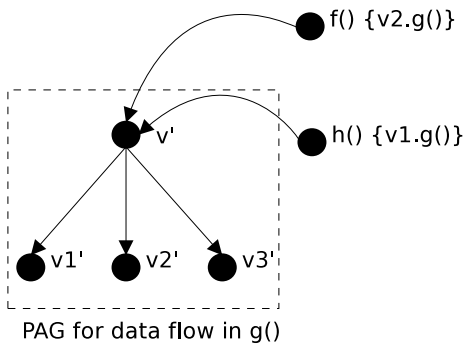

Figure 3.13: PAG for a context-insensitive analysis

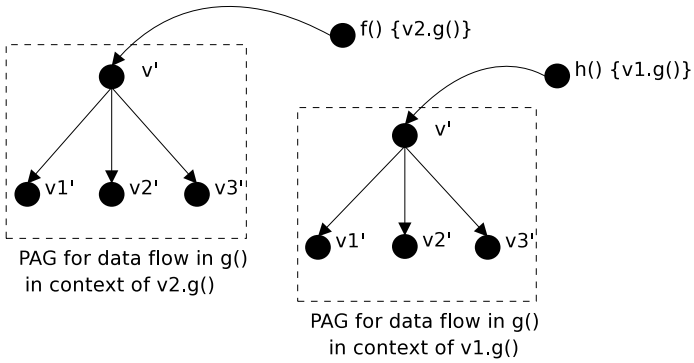

Figure 3.14: PAG for a context-sensitive analysis using call sites

sensitivity [48]. Object sensitivity uses the objects created to provide a context and therefore does not duplicate parts of the PAG for method invocations on the *same* object but at *different* call sites like the call-site approach. This reduces the PAG size and propagation time without compromising precision. We did not implement object sensitivity as it required significant changes to our existing points-to analysis implementation.

3.2.9 Output Design

The output from the tool provides sufficient information for the user to determine the exact cause of a race condition. Further, memory accesses are categorized based on the information available as follows:

1. No common object accessed.
2. Common lock held.
3. Potential race condition.

For each memory access in the categories given above we list the field accessed, the class the field belongs to and the type of the conflict (either read-write or write-write). Also part of the output are the locations where the different threads are started and the call path leading to the memory access. A sample output for a safe memory access is given below in Figure 3.15.

3.2.10 Summary

In this section we considered the design of the various components of our tool based on the analysis of the problem of statically detecting race conditions.

We started by choosing an intermediate representation from Soot and obtaining a call graph for the Java program. The call graph is constructed using the existing call-graph building infrastructure in Soot and Rapid Type Analysis. Next, we designed a generic call-graph traversal component using a depth-first traversal strategy and the Listener pattern.

The necessary conditions for a race serve as the basis of our race detection strategy. Therefore, we use a locks-held analysis component a points-to analysis and a thread and memory-access finding component. The locks-held analysis

```
------ Begin No Common Object Accessed -----
Field           : counter
Declaring Class : SharedCounter
Type of Conflict: Write-Write

Field           : counter
Declaring Class : SharedCounter
Type of Conflict: Write-Write

First  Thread   : SharedCounter.makeAndStartThreads(SharedCounter.java:61)
 Objects Accesed:
  HNode[2]: SharedCounter (at SharedCounter.makeSrcs(SharedCounter.java:45))
 Locks Held     :

Second  Thread  : SharedCounter.makeAndStartThreads(SharedCounter.java:62)
 Objects Accesed:
  HNode[27]: ThreadRun (at SharedCounter.makeThreadRuns(SharedCounter.java:50))
 Locks Held     :

Path to access  : (Write, counter, SharedCounter.inc(SharedCounter.java:7))
        at SharedCounter.inc(SharedCounter.java:7)
        at SharedCounter.run(SharedCounter.java:22)
        at ThreadRun.run(SharedCounter.java:78)
        at SharedCounter.makeAndStartThreads(SharedCounter.java:61)

Path to access  : (Write, counter, SharedCounter.inc(SharedCounter.java:7))
        at SharedCounter.inc(SharedCounter.java:7)
        at SharedCounter.run(SharedCounter.java:22)
        at SharedCounter.makeAndStartThreads(SharedCounter.java:62)

--- End No Common Object Accessed ---
```

Figure 3.15: Sample output from the tool

together with the points-to analysis tells us precisely the locks held at a particular statement. The points-to analysis tells us if two references may aliases each other.

Using the results from the components mentioned above the race-finding component exhaustively checks for memory accesses by different threads that may potentially interfere with each other. In the next chapter, we compare the results of our tool with some other static analysis tools with race detection capabilities.

Chapter 4

Evaluation

One of the first tasks of this project was evaluation of some existing static race detection tools. As the related work, presented in Section 2.5 shows, static race detection has been an area of active research for a number of years and some tools have already been developed. Therefore, before we decided to build a new tool we wanted to evaluate some freely available tools for two main reasons. First, to assess the capabilities of existing tools and second, to decide if one of the tools was suitable for extension. Three freely available tools were selected for evaluation; each one is a static analysis tool based on different approaches and claim race detection capabilities.

To evaluate the tools we used a suite of "benchmark" programs. The benchmark programs are mostly programs from Doug Lea's book [40] where we deliberately added race conditions to some of them. The motivation for choosing small programs (single classes) is to facilitate understanding of the programs so that the race conditions are easy to locate. In fact, choosing single classes with well known race conditions, the benchmark programs serve as functional tests for the tools evaluated. By deliberately introducing race conditions we know the expected output.

Findbugs	Escjava2	JLint
Statically analyzes Java byte-code	Statically analyzes java source	Statically analyzes Java byte-code
No change to source required	Requires annotation as comments known as pragmas	No change to source required
Missing classes are reported	Missing classes are reported; additionally a `.spec` file may be used to supply pragmas about unknown code	Missing classes are not reported
Independent of Java version because of byte-code analysis	Analyzes java version 1.2 source	Independent of Java version because of byte-code analysis
Relatively slow	Relatively slow	Relatively fast
No documented or known unsupported language features	Anonymous classes not supported	No documented or known unsupported language features

Table 4.1: General properties of the tools evaluated

4.1 Tools Evaluated

The tools evaluated are JLint [10], Findbugs [4] and Escjava2 [3]. These tools are static-analysis based tools for Java surveyed in Section 2.5, that are freely available. Table 4.1 gives the general properties of each of these tools.

JLint. JLint is built on top of a hand-written parser for Java byte code and performs data-flow analysis and builds a lock graph to detect bugs, inconsistencies and synchronization problems in Java programs. The original version of JLint was written by Konstantin Knizknik (upto and including version 1.21) and was extended by Cyrille Artho with "better support for synchronization." The current version of JLint is version 3.0, and also the version we evaluate.

FindBugs. FindBugs, is an open source project that uses static analysis to inspect java byte code for occurrences of *bug patterns*. FindBugs uses four main strategies for detecting bugs:

- Analyzing the class structure and inheritance hierarchy without looking at code.
- Using a linear code scan over the byte code and using visited instructions to drive a state machine that looks for bug patterns.
- Analyzing a control-flow graph of a method.
- Using both control flow and data flow to detect bugs.

FindBugs uses the BCEL [1] to parse java byte codes and claims to detect many "multithreaded correctness" bugs. The version of FindBugs we evaluated is 0.9.7 (May 2006).

Escjava2. Escjava2 is the third program we evaluate, and was created out of ESC/Java [29] a tool originally developed at Compaq and eventually released as open source. ESC/Java is a static analysis tools that works on annotated Java programs, using Javafe, a front end supplied with the tool to parse Java source code. ESC/Java uses a theorem prover to reason about the semantics of a program and is capable of finding synchronization errors in multithreaded programs. It performs modular checking, where a module is a function or a constructor, and uses programmer supplied annotations to reason about the other functions called by a module. Escjava2 is an extension to ESC/Java by Kind Software [3] where the annotations are written in JML [7].

4.2 Benchmark Programs

JLint, FindBugs and Escjava2 are evaluated using the benchmark programs introduced in this section. The tools are evaluated with two versions of each benchmark program. In the first version a class that may contain race conditions is shared between two threads. This is the race-condition detection test, intended to determine if the tools do in fact detect the race conditions in the program. In the second version, distinct objects are run in their own threads and do not interfere with each other. The second version for the benchmark program is used to detect if the tool reports false positives in these cases. The two *drivers* for the benchmark are shown in Figures 4.1 and 4.2.

Appendix A.6 contains both versions of the benchmark program. Drivers for the other benchmarks are implemented with the same strategy and are therefore not included.

```
...

ClassWithRaces cwr = new ClassWithRaces; // Implements java.lang.Runnable

new Thread(cwr).start();
new Thread(cwr).start();
```

Figure 4.1: Code snippet for driver to start benchmark program for detecting race conditions

```
...

ClassWithRaces cwrOne = new ClassWithRaces; // Implements java.lang.Runnable
ClassWithRaces cwrTwo = new ClassWithRaces; //             ''

new Thread(cwrOne).start();
new Thread(cwrTwo).start();
```

Figure 4.2: Code snippet for driver to start benchmark program to determine if tool reports false positives

For Escjava2 the benchmark programs had to be annotated; Appendix A.7 contains the annotated version of the benchmark in Appendix A.2. Other programs are annotated similarly.

Table 4.2 contains a list of the benchmark programs used along with a short description of each benchmark program. The benchmarks chosen cover the techniques discussed in the Exclusion chapter in Lea's book [40].

4.3 Evaluating other Tools on Benchmarks

Table 4.3 gives the evaluation of the tools on the benchmark programs; here **Race** and **False** correspond to the cases with race conditions and false positives respectively. The 'N' in Table 4.3 means that the tool was unable to detect a race condition while a '$\surd$' means that it produced meaningful results for the benchmark. A '-' is used for the ImmutableFraction benchmark as by definition this program has no race conditions.

The results show that both JLint and Findbugs do *not* report any synchronization related anomaly in the programs evaluated. In fact, they do not produce *any* output making it unlikely that they detect the case for false positives correctly and therefore have an 'N' in the column for **False**. Evaluating these tools we realize that obtaining no output from a tool that is unsound does not

	Benchmark	Description
A.6	SimpleRace	An example program with an obvious race condition. This is intended to be the first program a tool is evaluated with. It follows the "Fully Synchronized Object" exclusion technique.
A.1	Confinement	An example program using the Confinement exclusion technique. The public entry points to the class are protected by locking, and therefore the other methods called automatically hold a lock.
A.2	CopyOnWriteList	An example program that uses a copy-on-write list to increase concurrency by allowing iteration over the list without locking.
A.3	BoundedBuffer	An example program that splits synchronization into two monitor objects to increase concurrency, making the synchronization dependent on locks for distinct objects, and not the `this` reference.
A.4	ImmutableFraction	An example program using the Immutability pattern. This program uses immutability to avoid use of locks, and therefore concurrent access to shared memory occurs without any locking.
A.5	LazyInitialization	An example program using the lazy initialization pattern. Here not all accesses to the shared memory are guarded by locks, yet the program may be thread safe.

Table 4.2: Benchmark programs and their characteristics

Benchmark		Findbugs		JLint		Escjava2	
		Race	**False**	**Race**	**False**	**Race**	**False**
A.1	Confinement	N	N	N	N	√	N
A.2	CopyOnWriteList	N	N	N	N	√	N
A.3	BoundedBuffer	N	N	N	N	√	N
A.4	ImmutableFraction	-	-	-	-	?	-
A.5	LazyInitialization	N	N	N	N	?	N
A.6	SimpleRace	N	N	N	N	√	N

Table 4.3: Race conditions detection results for the different tools. Here **Race** stands for the version of the benchmark program with a possible race condition and **False** for False Positive. The 'N' implies that the tool failed to produce any results for the benchmark, i.e., was unable to detect the presence of a race condition or the absence. Notice that **JLint** and **FindBugs** do *not* report any problems neither races, nor false positives for the benchmarks evaluated. However, they do produce warnings for some restricted cases. The '-' in the ImmutableFraction is used because this program is by definition thread safe. A '?' is used in conjuction with Escjava2 where the benchmarks are known to use thread-safe idioms; Escajava2 does not report any problems without annotations and when annotations are used it reports false positives.

increase our confidence in the program. JLint and FindBugs report no output for the ImmutableFraction benchmark but we are doubtful if they do so for the correct reason, considering that they can not detect presence or absence of race conditions in our SimpleRace benchmark.

Escjava2 performs better than the other tools; provided with annotated programs it can determine unlocked accesses for almost all our benchmark programs. Unfortunately, it reports false positives even when an object is *not* shared by two different threads as it does not consider objects in its evaluation. For Escjava2, two entries are marked with a '?', as these programs use idioms that are known to be thread-safe. In these cases annotating programs is defensive but a burden on the user, In fact annotating the program correctly is not possible for the LazyInitialization program as it uses a programming idiom that does not assume all accesses to shared memory need to be guarded by a lock. Though the results for Escjava2 are better than those from the other tools, the output is not very useful for finding the cause of the race condition as can be seen from Figure 4.3 which contains sample output for a race condition detected by Escjava2.

As the output above shows, there is not sufficient information to determine the cause of the race condition, which in fact could even be due to a missing

```
------------------------------------------------------------------------
aaaTestFiles/SimpleRaceCondition.java:13: Warning: Possible race condition (Race)
                counter--;
                ^
Associated declaration is "../aaaTestFiles/SimpleRaceCondition.java", line 5, col
      4:
        //@monitored_by this;
           ^
------------------------------------------------------------------------
```

Figure 4.3: Sample output for a race condition detected by Escjava2

annotation. In fact, not adding an annotation is much more serious; for the same program, not declaring any annotation produces no warnings at all.

In effect Escjava2 gives a user an enforceable means to capture design intent using annotations for a program and then type check the program to ensure that it conforms to the annotations.

4.4 Motivation for Developing our own Tool

The poor results for race detection from JLint and FindBugs and limited useability of the results from Escjava2 were sufficient motivation to develop our own Tool. After evaluation of these tools we decided that the new tool developed should read and process Java byte codes as opposed to sources. Parsing byte code we have an extra layer of abstraction against any changes to the languages, e.g., programs with generics (part of Java 1.5), are automatically processable as generics are not visible at the byte code level. Parsing byte code is also generally accepted to be simpler than Java sources.

Our aim for the new tool is to be at least as precise as Escjava2 in detecting race conditions along with the possibility of giving more useful output, i.e., counter examples for the cause of a race condition as well causes for why a memory access is considered safe or a race. We would like at least the precision of Escjava2, but without using annotations to specify the design intent. We believe that only the part of the program that is seen by the compiler gives the true intention of the program.

The authors of ESC/Java [29] suggest that the cost of adding annotations is justified if the bugs caught would require the same amount of time to find. While we concur with this argument, knowing and stating exactly the design intent is difficult for most programs; a perfect example of which is documentation

in the form of comments in the source code. Another argument against the use of annotations as in Escjava2 is that they *must* be present for the type checker to report potential problems. We prefer to take another view for the use of annotations; they should be used to suppress false positives if required and believe this is the safer use for them.

4.5 Evaluating our Tool

4.5.1 Testing and Validation

In this section we describe the general approach to testing our tool. All classes with public interfaces that produce output were unit tested using JUnit [11] tests. For all tests that produce data structures for outputs, e.g., Call Graph Builder (Section 3.2.2), or Points-to Analysis (Section 3.2.5), we serialize the output into a string, manually verify it the first time and then store an MD5 checksum of the output string for automated testing. This approach works as long as the output from each of the components is deterministic; to allow for this we use `LinkedHashSet` and `LinkedHashMap` data structures from the Java standard library. An alternative approach, to avoid using specialized data structures for automatic verification is to sort the output and then use the sorted strings or an checksum over them. We tried both approaches and chose the former; an unmodified dump of the program output is important for determining the cause of test failures.

Unit testing was very helpful in finding bugs, especially over time as the size of the program grew. We also used the unit tests extensively while experimenting with new functionality and refactoring. For example, the Points-to analysis, a complicated part of the tool, was implemented incrementally. When adding extensions like support for arrays, interprocedural support and eventually context sensitivity a successful run of the unit tests gave us confidence that an extension was indeed just *extensions* and did not introduce bugs into existing functionality.

4.5.2 Comparison with other Tools

To start with we list the general properties of our tool as for the other tools in Table 4.1:

Benchmark		Our Tool	
		Race	**False**
A.1	Confinement	√	√
A.2	CopyOnWriteList	N	N
A.3	BoundedBuffer	√	√
A.4	ImmutableFraction	√	√
A.5	LazyInitialization	√	N
A.6	SimpleRace	√	√

Table 4.4: Race conditions detection results for our tool. Here **Race** stands for the version of the benchmark program with a possible race condition and **False** for False Positive

.

1. The tool analyzes both Java source and byte code.
2. The tool does not report missing Java library classes, in fact library classes are not analyzed. Missing application classes lead to an error.
3. Analyzed Java sources(upto and including version 1.4), and independent of JDK version for byte code.
4. The tool is relatively slow compared to JLint.
5. There are no unsupported language features.

Our tool is more precise than the other tools considered as can be seen in Table 4.4. Race conditions are detected correctly in all but two of the benchmark programs. For the LazyInitialization program, the false positives result from the lazy initialization idiom used in concurrent programming. This idiom is against our tool's basic assumption that all memory accesses by different threads need to be protected via a locking. For CopyOnWriteList we believe the failure to detect any race conditions is due to a Soot bug[1]. If we ignore the problem and consider how our tool would behave on the example, we would get a false positive warning as there is a read-write reference that occurs in concurrent without locking. As in the LazyInitialization example, this is against the basic assumption of the tool.

The output produced by our tool is more usable than those produced by the other tools; a sample of a potential race condition is shown if Figure 4.4. There is sufficient information to determine the cause of a race condition, if it exists,

[1]Soot fails to create the call graph correctly when an anonymous class is used to implement an interface as in method `iterator` of this example.

```
------ Begin Race Condition Detected ------
Field           : keySet
Declaring Class : java.util.Hashtable
Type of Conflict: Read-Write

First  Thread   : aaaTestFiles.HashtableTest.main(HashtableTest.java:11)
 Objects Accesed:
  HNode[1]: java.util.Hashtable (at aaaTestFiles.HashtableTest.main(HashtableTest.
       java:10))
 Locks Held     :

Second  Thread  : aaaTestFiles.HashtableTest.main(HashtableTest.java:12)
 Objects Accesed:
  HNode[1]: java.util.Hashtable (at aaaTestFiles.HashtableTest.main(HashtableTest.
       java:10))
 Locks Held     :

Common Objects  :
 HNode[1]: java.util.Hashtable (at aaaTestFiles.HashtableTest.main(HashtableTest.
      java:10))

Path to access  : (Read , keySet, java.util.Hashtable.keySet(Hashtable.java:581))
        at java.util.Hashtable.keySet(Hashtable.java:581)
        at aaaTestFiles.HashTableTestRunner.run(HashtableTest.java:26)
        at aaaTestFiles.HashtableTest.main(HashtableTest.java:11)

Path to access  : (Write, keySet, java.util.Hashtable.keySet(Hashtable.java:582))
        at java.util.Hashtable.keySet(Hashtable.java:582)
        at aaaTestFiles.HashTableTestRunner.run(HashtableTest.java:26)
        at aaaTestFiles.HashtableTest.main(HashtableTest.java:12)
--- End  Race Condition Detected ---
```

Figure 4.4: Sample output for a potential race condition

Tool	Time (sec)
JLint	< 3
Escjava2	3−12
Findbugs	3−10
Our Tool	(10 + 3 + 2)

Table 4.5: Comparison of running times for the tools for the benchmark programs. The running time from our tool is divided into three parts, the 10 second component is the time required to initialize Soot, something that we can not influence. The 3 second component is the time required to build the incomplete call graph of the application. Here too we rely on Soots call graph builder to get us an initial call graph on which we apply RTA. The actual analysis time is about 2 seconds, the break for which is given in Figure 4.5

.

as well as reasons to why a pair of memory accesses are considered safe. The output contains references to the source code for the objects accessed, locks held, threads started and the exact path leading to the memory accesses.

Table 4.5 contains a comparison of the running times for the benchmark programs, these tests were measured on a PIII 1.2 GHz, with 512 MBytes of main memory. Our tool is the slowest from all the other tools with approximately 80% of the time required to initialize Soot, which, unfortunately is not under our control. The breakup of running time for our tool is given in Figure 4.5.

4.5.3 Analyzing Open Programs

Chord, a recent work[2] for static race detection on Java [50] claim the best race detection results and has the capability of handling open programs. The define open programs as those for which either the caller or callee is not known. We evaluate three open programs from their suite of programs and compare the our results from Chord. To analyze open programs, they synthesize a *harness*, a main function for an application that calls the methods of the class in many different scenarios. We use a similar approach to construct a harness program where all public methods of the class are invoked in two different threads, hence any public method may be executed concurrently with any other public method, including itself.

The three programs evaluated are Vector and Hashtable from JDK1.5 ([50]

[2]Their paper became available during this project, but the tool is still not available.

```
DEBUG: Race Condition Statistics
DEBUG:  Number of Access to Different Objects :      1698
DEBUG:  Number of Locked Acccess              :      1646
DEBUG:  Unknown Points-To information for     :        62
DEBUG:  Number of Potential Race Conditions   :         3
DEBUG:  Total Number of Accesses              :      3409
DEBUG:
DEBUG: Time measurements
DEBUG:  Setup Soot                    : 12.6s (75.7%)
DEBUG:  Incomplete Call Graph         : 02.8s (17.0%)
DEBUG:  RTA Reachable Methods         : 00.0s (00.2%)
DEBUG:  Points-To Analysis            : 00.5s (02.7%)
DEBUG:  Thread Finding                : 00.4s (02.3%)
DEBUG:  Race Condtions Finding        : 00.2s (01.4%)
DEBUG:  Total Time                    : 16.5s (100.0%)
```

Figure 4.5: Typical execution time breakup. Notice that time used by our tool is less than 8% for the analysis. The remaining 92% is used to initialize Soot and obtain a call graph for the application.

		Time (seconds)	
	LOC	**Chord**	**Our Tool**
vect 1.5	1,889	8	21
htbl 1.5	1,240	7	33
jdbm	11,504	93	15

Table 4.6: Running time comparison for open programs

	Race Conditions	
	Chord	**Our Tool**
vect 1.5	0	0
htbl 1.5	9	6
jdbm	91	83

Table 4.7: Race conditions found in open programs. The difference in the number of problems reported is most likely due to differences in harness construction.

	Classes	**Methods**	**Statements**		**Access**		**Coverage**
			Tot	**Reach.**	**Rd**	**Wr**	(%)
vect 1.5	5	69	871	659	126	39	77
htbl 1.5	5	48	652	520	101	38	80
jdbm	16	80	756	593	157	33	78

Table 4.8: Size metrics for the open programs analyzed

evaluates versions JDK 1.1 and 1.4) and jdbm [8], a transaction persistence engine for Java. Table 4.6 shows the running times and lines of code analyzed. Table 4.7 gives the number of race conditions found by our tool and Chord. Our Tool reports fewer race condition; this might be because of differences in the harness generations. Chord further classifies races into harmful, benign and false, and reports the 9 races in htbl as benign and the 91 in jdbm as harmful. Our results also confirm this, though we have not checked each of the 83 reported races exhaustively.

Chord uses a thread escape analysis to determine unlocked memory accesses that are safe because the objects written to never escape the thread they are created in. Although we do not have an escape analysis, allowing an object allocation site in a method to create different objects for different contexts, as long as the allocation site dominates the return statement emulates an escape analysis. Objects not assigned to references outside the thread do not escape it.

Finally, Table 4.8 shows some information on the classes analyzed. The "Reachable Statements" and "Total Statements", are the number of statements from application-only classes loaded by Soot and those determined to be reachable from our test program using RTA.

It is interesting to compare the LOC in Table 4.6 with the "Total Statements" in Table 4.8, where a large difference exists for the jbdm benchmark. The low number of statements (756) as compared to LOC (11 K), is because the

harness synthesis does not consider the whole public interface for jdbm, rather a small subset, taken from an example program supplied with the source. This underscores the importance of the harness program used.

4.6 Summary

This section contains the evaluation of three other tools and comparison of the results with our tool. Table 4.3 shows the result of evaluation of the other tools on the benchmark programs given in Table 4.2. Table 4.4 shows the result of evaluation of our tool.

Our tool finds *more* race conditions than the tools evaluated and has a *lower* false positive rate than Escjava2, the tool with the best results of those evaluated. Unlike Escjava2, our tool does not require any modification to the original program. Our tool is calso capable of analyzing open programs, like libraries, and gives comparable results to Chord, a recent work in static race detection.

Chapter 5

Conclusion

In the previous chapters we present the design and evaluation of a static race detection tool for concurrent Java programs. Before setting out to develop our own tool we evaluate some other freely available tools, based on different approaches, that also possess static race detection capabilities. The tools are evaluated using real world Java programs and fail to meet the key properties we identify in Section 1.1 for a static race detection tool to be useful. This was our motivation for developing a new tool.

5.1 Achievements

We have developed a static race detection tool that is capable of analyzing real world Java programs that use the `synchronized` keyword for locking and believe that this covers a large class of concurrent programs. The tool performs a lazy whole-program analysis; we use the term lazy because the Java standard library is analyzed only if required, and found this essential to keep running time acceptable.

The tool uses a context-sensitive but flow-insensitive points-to analysis to determine precisely the *objects* that may be involved in a race and the locks currently held. The points-to analysis, especially context-sensitivity, is crucial for keep

the false positive rate low. In fact, the false positives reported by our tool are for situations where a synchronization idiom other than the one we assume is used.

The tool produces a stack trace for all memory accesses, as well as a pointer to the location for objects accessed and locks held. The output is classified by the type of memory access; this allows a user to determine the cause of a race or the absence of one.

Supplied with a suitable driver program the tool is capable of analyzing open programs such as a library. Such a driver program may be generated automatically with little effort.

We believe that the tool has all the key properties, identified in Section 1.1, that a static race detection tool must have to be useful.

Comparison with Tools Evaluated

An evaluation of our tool with the other tools, JLint, FindBugs, and Escjava2, shows that our tool produces much more accurate results. Our tool finds all the race conditions found by Escjava2, the tool with the best result of those evaluated, but *without* annotation of the program as required for Escjava2. Our tool performs better than Escjava2 with respect to the false positive rate. We believe this makes our tool more useable than all the other tools.

A comparison of the results of our tool with Chord [50] shows that our tool is capable of finding race conditions comparable to Chord. Chord claims to have found the most bugs out of all static race detection tools till date. Their paper [50] became available during this project[1].

5.2 Limitations

Our tool has some limitations. First, we use a context-sensitive, but flow-insensitive may points-to analysis which makes the locks-held analysis result unsound.

The tool is limited by the synchronization idioms detected, we only regard

[1]Chord is not yet available, but the home page for one of the authors states that they intended to release it in the future, though no expected date is given.

locking via the use of the `synchronized` keyword and assume that all accesses must be protected via a lock. While this is a reasonable assumption for Java 1.4, Java 1.5 introduces several concurrent programming utilities like specialized locks and semaphores that require path-sensitive static analyses to reason about their usage.

We elide the checking of race conditions in constructors and class initializers but these may be a source of subtle race conditions because of the Java Memory Model [6]. Finally, we ignore the effect of reflection and dynamic class loading.

5.3 Applications

Even with the limitations of our tool mentioned above, we believe it is useful for finding race conditions in programs, that would have previously gone unnoticed. As a small example the reader is invited to examine the program in Appendix A.8 that contains a race condition and the output from our tool.

Often race conditions are violations of program invariants but are notoriously difficult to find because of non-deterministic execution of threads. Using our tool helps to increase the reliability of concurrent programs.

The tool is also useful as a teaching, learning and debugging aid; concurrent programming has a steep learning curve and even experienced programmers may err while fine tuning concurrency in their programs.

5.4 Future Work

There are some candidate tasks that could be done to extend the tool. First, we could add support for different synchronization idioms. Currently, we only support locking via the `synchronized` keyword. In the current implementation all memory accesses by different threads are assumed to happen in parallel. This could be improved using a may-happen-in-parallel analysis [51]. Scalability is not addressed in this implementation — our approach to context sensitivity in the points-to analysis is not scalable, object-sensitivity is a more scalable approach. Finally, a useful addition would be integration with a integrated development environment like Eclipse [2].

Bibliography

[1] The byte code enginneering library (bcel), available at http://jakarta.apache.org/bcel/.

[2] The eclipse homepage, available at: http://www.eclipse.org/.

[3] Esc/java2. http://secure.ucd.ie/products/opensource/escjava2/.

[4] Findbugs homepage, available at: http://findbugs.sourceforge.net/.

[5] Java 1.5 concurrency utilities, available at: http://java.sun.com/j2se/1.5.0/docs/guide/concurrency/index.html.

[6] The java memory model, available at: http://www.cs.umd.edu/~pugh/java/memorymodel/.

[7] The java modeling language (jml), available at: http://www.cs.iastate.edu/~leavens/jml/download.shtml.

[8] Jdbm a transactional persistence engine for java, available at: http://jdbm.sourceforge.net/.

[9] Jikes rvm, available at: http://jikesrvm.sourceforge.net/.

[10] Jlint 3.0., available at: http://artho.com/jlint.

[11] Junit, available at: http://www.junit.org/index.htm/.

[12] Soot: a java optimization framework (march 18, 2006) 2.2.3., available at: http://www.sable.mcgill.ca/soot.

[13] *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*. USENIX, 2001.

[14] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer. Detecting data races on weak memory systems. *SIGARCH Comput. Archit. News*, 19(3):234–243, 1991.

[15] Rahul Agarwal, Amit Sasturkar, Liqiang Wang, and Scott D. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 233–242, New York, NY, USA, 2005. ACM Press.

[16] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[17] Gregory R. Andrews. *Foundations of Multithreaded, Parallel and Distributed Programming*. Addison-Wesley, 2000.

[18] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a dialect of Java without data races. *ACM SIGPLAN Notices*, 35(10):382–400, 2000.

[19] D.F. Bacon and P.F. Sweeney. Fast static analysis of c++ virtual function calls. *SIGPLAN Notices*, 31(10):324–41, 1996.

[20] J. Choi, A. Loginov, and V. Sarkar. Static datarace analysis for multithreaded object-oriented programs. Technical report, 2001.

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

[22] Jong-Deok Choi, Barton P. Miller, and Robert H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Trans. Program. Lang. Syst.*, 13(4):491–530, 1991.

[23] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. *ECOOP '95 - Object-Oriented Programming. 9th European Conference. Proceedings*, pages 77–101, 1995.

[24] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In *PADD '91: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 85–96, New York, NY, USA, 1991. ACM Press.

[25] Árni Einarsson and Janus Dam Nielsen. A survivor's guide to java program analysis with soot.

[26] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252, New York, NY, USA, 2003. ACM Press.

[27] Cormac Flanagan and Martin Abadi. Types for safe locking. In *European Symposium on Programming*, pages 91–108, 1999.

[28] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. *ACM SIGPLAN Notices*, 35(5):219–232, 2000.

[29] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, volume 37, pages 234–245, June 2002.

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[31] James Golsing, Bill Joy, Guy L. Steel Jr., and Gilad Bracha. *The Java Language Specification. Third Edition*. Sun, 2005.

[32] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 97–105, 1998.

[33] N Heintze. Analysis of large code bases: The compile-like-analyze model. 1999.

[34] Laurie Hendren, Patrick Lam, Jennifer Lhoták, Ondřj Lhoták, and Feng Qian. Soot, a tool for analyzing and transforming java bytecode. 2003.

[35] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, UT, 2001.

[36] Michael Hind and Anthony Pioli. Which pointer analysis should i use? In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 113–123, New York, NY, USA, 2000. ACM Press.

[37] Martin Hirzel, Amer Diwan, and Michael Hind. Pointer analysis in the presence of dynamic class loading. Technical report, 2003.

[38] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[39] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[40] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns. Second Edition.* Addison-Wesley, 1999.

[41] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[42] Ondřej Lhoták. *Program Analysis using Binary Decision Diagrams.* PhD thesis, McGill University, January 2006.

[43] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[44] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. pages 73–79, 2001.

[45] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Evaluating the precision of static reference analysis using profiling. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 22–32, New York, NY, USA, 2002. ACM Press.

[46] Hans Henrik Løvengreen. *Basic Concurrency Theory, First Edition.* IMM, DTU, 2002.

[47] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 24–33, New York, NY, USA, 1991. ACM Press.

[48] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java, 2002.

[49] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

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

[51] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. An efficient algorithm for computing mhp information for concurrent java programs. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 338–354, London, UK, 1999. Springer-Verlag.

[52] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.

[53] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

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

[55] Michiel Ronsse and Koen De Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[56] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for java using annotated constraints. In *Conference on Object-Oriented*, pages 43–55, 2001.

[57] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[58] D. Schonberg. On-the-fly detection of access anomalies. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 285–297, New York, NY, USA, 1989. ACM Press.

[59] Michael I. Schwartzbach. Lecture notes on static analysis.

[60] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. *Lecture Notes in Computer Science*, 1302:16–??, 1997.

[61] B. Steensgaard. Points-to analysis in almost linear time. 1996.

[62] Nicholas Sterling. WARLOCK — A static data race analysis tool. pages 97–106, Winter 1993.

[63] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon, and C. Godin. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, 35(10):264–280, 2000.

[64] Navindra Umanee. Shimple: An investigation of static single assignment form. Master's thesis, McGill University, February 2006.

[65] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.

[66] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[67] C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *PLDI 2003*, pages 115–128, June 2003.

[68] Christoph von Praun and Thomas R. Gross. Object race detection. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 70–82, New York, NY, USA, 2001. ACM Press.

[69] John Whaley and Monica S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of the 9th International Static Analysis Symposium*, September 2002.

[70] Reinhard Wilhelm, Shmuel Sagiv, and Thomas W. Reps. Shape analysis. In *Computational Complexity*, pages 1–17, 2000.

Appendix A

Benchmark programs

A.1 Confinement

```
package raceConditions;

class MessageHandler {
    private int stateA;
    public static MessageHandler theInstance = new MessageHandler();

    private MessageHandler() {
        stateA = 0;
    }

    public synchronized void handleMessage(Object message) { // Message
        stateA++; //update some shared state
        doHandleMessage();
    }

    void doHandleMessage() {
        // ...

        method1();
        method2();
    }

    void method1() {
        int localState = stateA;
    }

    void method2() {
        int localState = stateA;
        stateA++;
```

```
        localState = stateA;
    }

    public static MessageHandler getInstance() {
        return theInstance;
    }
}

class SocketListener implements Runnable {

    public void run() {

        while (true) {
            Object m = recieve(); //blocking call
            MessageHandler.getInstance().handleMessage(m);
        }
    }

    //Block ti message available
    private Object recieve() {
        return new Object();
    }
}

class ListenerStarter {
    public static void main() throws InterruptedException {
        Thread[] clientListeners = new Thread[8]; // arbitrary number

        for (int i = 0; i < 8; i++) {
            clientListeners[i] = new Thread(new SocketListener());
            clientListeners[i].start();
        }

        //Try to trigger race condtion;
        MessageHandler.getInstance().doHandleMessage();

    }
}
```

A.2 CopyOnWriteList

```
package raceConditions;
import java.util.NoSuchElementException;
import java.util.Iterator;

class CopyOnWriteList { // Incomplete
    protected Object[] array =  new Object[0];

    protected /*synchronized*/ Object[] getArray() { return array; }

    public synchronized void add(Object element) {
        int len = array.length;
        Object[] newArray = new Object[len+1];
        System.arraycopy(array, 0, newArray, 0, len);
        newArray[len] = element; array = newArray;
    }

    public Iterator iterator() { return new Iterator() {
```

```
            protected final Object[] snapshot = getArray();
            protected int cursor = 0;

            public boolean hasNext() {
                return cursor < snapshot.length;
            }

            public Object next() {

                try { return snapshot[cursor++]; }
                catch
                    (IndexOutOfBoundsException ex)
                    { throw new NoSuchElementException(); }
            }

            public void remove() {}

        };
    }
}
```

A.3 BoundedBuffer

```
package raceConditions;

final class BoundedBuffer {
    private final Object[] array;   // the elements

    private int putPtr = 0;         // circular indices
    private int takePtr = 0;

    private int emptySlots;         // slot counts
    private int usedSlots = 0;

    private int waitingPuts = 0;    // counts of waiting threads
    private int waitingTakes = 0;

    private final Object putMonitor = new Object();
    private final Object takeMonitor = new Object();

    public BoundedBuffer(int capacity)
        throws IllegalArgumentException {
        if (capacity <= 0)
            throw new IllegalArgumentException();

        array = new Object[capacity];
        emptySlots = capacity;
    }

    public void put(Object x) throws InterruptedException {
        synchronized(putMonitor) {
            while (emptySlots <= 0) {
                ++waitingPuts;
                try { putMonitor.wait(); }
                catch(InterruptedException ie) {
                    putMonitor.notify();
                    throw ie;
                }
                finally { --waitingPuts; }
```

```
            }
            --emptySlots;
            ++usedSlots; // Race condtion; requires lock on takeMonitor
            array[putPtr] = x;
            putPtr = (putPtr + 1) % array.length;
        }

        synchronized(takeMonitor) { // directly notify
            if (waitingTakes > 0)
                takeMonitor.notify();
        }
    }

    public Object take() throws InterruptedException {
        Object old = null;
        synchronized(takeMonitor) {
            while (usedSlots <= 0) {
                ++waitingTakes;
                try { takeMonitor.wait(); }
                catch(InterruptedException ie) {
                    takeMonitor.notify();
                    throw ie;
                }
                finally { --waitingTakes; }
            }

            --usedSlots;
            ++emptySlots; // Race condtion; requires lock on putMonitor

            old = array[takePtr];
            array[takePtr] = null;
            takePtr = (takePtr + 1) % array.length;
        }

        synchronized(putMonitor) {
            if (waitingPuts > 0)
                putMonitor.notify();
        }
        return old;
    }
}
```

A.4 ImmutableFraction

```
package raceConditions;

class Fraction {
    protected final long numerator;
    protected final long denominator;

    public Fraction(long num, long den) {
        // normalize:
        boolean sameSign = (num >= 0) == (den >= 0);
        long n = (num >= 0)? num : -num;
        long d = (den >= 0)? den : -den;
        long g = gcd(n, d);
        numerator = (sameSign)? n / g : -n / g;
        denominator = d / g;
    }
```

```
    //Static guarantees that the method doess not access any instance state
    static long gcd(long a, long b) {
            // ... compute greatest common divisor ...
            return 1;
    }

    public Fraction plus(Fraction f) {
        return new Fraction(numerator * f.denominator +
                            f.numerator * denominator,
                            denominator * f.denominator);
    }

    public boolean equals(Object other) { // override default
        if (! (other instanceof Fraction) ) return false;
        Fraction f = (Fraction)(other);
        return numerator * f.denominator ==
            denominator * f.numerator;
    }

    public int hashCode() {              // override default
        return (int) (numerator ^ denominator);
    }
}
```

A.5 LazyInitialization

```
package raceConditions;

class LazyInitialization {
    private int cachedHashCode = 0;

    private long longCachedHashCode = 0;
    // Correct Double-Checked Locking for 32-bit primitives
    public int hashCode_int1() {
        int h = cachedHashCode;
        if (h == 0)
            synchronized(this) {
                if (cachedHashCode != 0) return cachedHashCode;
                h = computeHashCode();
                cachedHashCode = h;
            }
        return h;
    }

    // Lazy initialization 32-bit primitives
    // Thread-safe if computeHashCode is idempotent
    public int hashCode_int2() {
        int h = cachedHashCode;
        if (h == 0) {
            h = computeHashCode();
            cachedHashCode = h;
        }
        return h;
    }

    /**Not thread-safe since because of long */
    public long hashCode_long1() {
        long h = longCachedHashCode;
```

```
        if (h == 0)
            synchronized(this) {
                if (longCachedHashCode != 0) return longCachedHashCode;
                h = computeLongHashCode();
                longCachedHashCode = h;
            }
        return h;
    }

    /**Not thread-safe since because of long */
    public long hashCode_long2() {
        long h = longCachedHashCode;
        if (h == 0) {
            h = computeLongHashCode();
            longCachedHashCode = h;
        }
        return h;
    }

    private int computeHashCode() {
        return 1;
    }

    private long computeLongHashCode() {
        return 1L;
    }
}
```

A.6 SimpleRace

A.6.1 SimpleRace with Race Conditions

```
package raceConditions;

public class SimpleRace implements Runnable {
    protected int counter;

    public void inc() {
        counter++;
    }

    public void dec() {
        counter--;
    }

    public int get() {
        return counter;
    }

    public void run() {

        while (true) {
            if (get() > 500) {
                dec();
            }
            else {
                inc();
```

```
            }
        }
    }

    public static void main(String args[]) {
        SimpleRace sr = new SimpleRace();

        /** Two threads share a common SimpleRace instance */
        new Thread(sr).start();
        new Thread(sr).start();
    }
}
```

A.6.2 SimpleRace with False Positives

```
package raceConditions;

public class SimpleRace implements Runnable {
    protected int counter;

    public void inc() {
        counter++;
    }

    public void dec() {
        counter--;
    }

    public int get() {
        return counter;
    }

    public void run() {

        while (true) {
            if (get() > 500) {
                dec();
            }
            else {
                inc();
            }
        }
    }

    public static void main(String args[]) {
        SimpleRace srOne = new SimpleRace();
        SimpleRace srTwo = new SimpleRace();

        /**
         * Two threads started using distinct SimpleRace instances.
         */
        new Thread(srOne).start();
        new Thread(srTwo).start();
    }
}
```

A.7 CopyOnWriteList Annotated for Escjava2

```
package raceConditions;
import java.util.NoSuchElementException;
import java.util.Iterator;

class CopyOnWrite { // Incomplete
    //@monitored
    protected Object[] array =  new Object[0];

    protected synchronized Object[] getArray() {
        //...
        return doGetArray();
    }

    //@requires \lockset[this];
    private Object[] doGetArray() {
        return array;
    }

    public synchronized void add(Object element) {
        int len = array.length;
        Object[] newArray = new Object[len+1];
        System.arraycopy(array, 0, newArray, 0, len);
        newArray[len] = element; array = newArray;
    }

    public Iterator iterator() {
        return new ALIterator();
    }

    class ALIterator implements Iterator {

        protected final Object[] snapshot = getArray();
        protected int cursor = 0;

        public boolean hasNext() {
            return cursor < snapshot.length;
        }

        public Object next() {

            try { return snapshot[cursor++]; }
            catch
                (IndexOutOfBoundsException ex)
                { throw new NoSuchElementException(); }
        }

        public void remove() {}

    };
}
```

A.8 Sample Output from Tool

A.8.1 Example Program

This section contains an example program of a bounded buffer shared between a producer and a consumer. It contains a potential race condition. The reader is invited to try and find it before looking at the output from our tool in the next section.

```
final class BoundedBuffer {
        private final Object[] array; // the elements

        private int putPtr = 0; // circular indices

        private int takePtr = 0;

        private int emptySlots; // slot counts

        private int usedSlots = 0;

        private int waitingPuts = 0; // counts of waiting threads

        private int waitingTakes = 0;

        private final Object putMonitor = new Object(); // monitors

        private final Object takeMonitor = new Object();

        public BoundedBuffer(int capacity)
                        throws IllegalArgumentException {
                if (capacity <= 0)
                        throw new IllegalArgumentException();

                array = new Object[capacity];
                emptySlots = capacity;
        }

        public void put(Object x) throws InterruptedException {
                synchronized (putMonitor) {
                        while (emptySlots <= 0) {
                                ++waitingPuts;
                                try {
                                        putMonitor.wait();
                                } catch (InterruptedException ie) {
                                        putMonitor.notify();
                                        throw ie;
                                } finally {
                                        --waitingPuts;
                                }
                        }

                        --emptySlots;

                        array[putPtr] = x;
                        putPtr = (putPtr + 1) % array.length;

                        synchronized (takeMonitor) { // directly notify
                                usedSlots++;
                                if (waitingTakes > 0)
                                        takeMonitor.notify();
```

```
                        }
                }
        }

        public Object take() throws InterruptedException {
                Object old = null;
                synchronized (takeMonitor) {
                        while (usedSlots <= 0) {
                                ++waitingTakes;
                                try {
                                        takeMonitor.wait();
                                } catch (InterruptedException ie) {
                                        takeMonitor.notify();
                                        throw ie;
                                } finally {
                                        --waitingTakes;
                                }
                        }
                        --usedSlots;
                        ++emptySlots;

                        old = array[takePtr];
                        array[takePtr] = null;
                        takePtr = (takePtr + 1) % array.length;
                }

                synchronized (putMonitor) {

                        if (waitingPuts > 0)
                                putMonitor.notify();
                }
                return old;
        }

        public static void main(String args[]) {
                final int BUFFER_SIZE = 32;

                BoundedBuffer buffer = new BoundedBuffer(BUFFER_SIZE);
                new Producer(buffer).start();
                new Consumer(buffer).start();

        }
}

class Consumer extends Thread {
        protected BoundedBuffer buffer;

        Consumer(BoundedBuffer bb) {
                this.buffer = bb;
        }

        public void run() {
                try {
                        while (true) {
                                Object o = buffer.take();
                                process(o);
                                // ...
                        }
                } catch (InterruptedException ignore) {}
        }

        public void process(Object o) { }
}

class Producer extends Thread {
```

```
        protected BoundedBuffer buffer;

        Producer(BoundedBuffer bb) {
                this.buffer = bb;
        }

        public void run() {
                try {
                        while (true) {
                                //...
                                Object msg = produce();
                                buffer.put(msg);
                        }
                } catch (InterruptedException ignore) {}
        }

        public Object produce() {
                return new Object();
        }
}
```

This section contains the output from our tool on the program given in Appendix A.8.1. The output has been modified to move the race conditions found and summary of results to the end.

A.8.2 Results from Tool

```
------ Begin Common Lock Held Memory Access ------
Field           : usedSlots
Declaring Class : aaaTestFiles.BoundedBuffer
Type of Conflict: Read-Write

First  Thread   : aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:92)
 Objects Accesed:
  HNode[1]: aaaTestFiles.BoundedBuffer (at aaaTestFiles.BoundedBuffer.main(
       BoundedBuffer.java:91))
 Locks Held     :
  HNode[37]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
       java:20))
  HNode[36]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
       java:18))

Second  Thread  : aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:93)
 Objects Accesed:
  HNode[1]: aaaTestFiles.BoundedBuffer (at aaaTestFiles.BoundedBuffer.main(
       BoundedBuffer.java:91))
 Locks Held     :
  HNode[37]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
       java:20))

Common Locks    :
 HNode[37]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
      java:20))
Path to access  : (Read , usedSlots, aaaTestFiles.BoundedBuffer.put(BoundedBuffer.
     java:51))
        at aaaTestFiles.BoundedBuffer.put(BoundedBuffer.java:51)
        at aaaTestFiles.Producer.run(BoundedBuffer.java:130)
        at aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:92)
```

```
Path to access  : (Write, usedSlots, aaaTestFiles.BoundedBuffer.take(BoundedBuffer
     .java:72))
        at aaaTestFiles.BoundedBuffer.take(BoundedBuffer.java:72)
        at aaaTestFiles.Consumer.run(BoundedBuffer.java:108)
        at aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:93)
--- End Common Lock Held Memory Access ---

------ Begin Common Lock Held Memory Access ------
Field           : waitingTakes
Declaring Class : aaaTestFiles.BoundedBuffer
Type of Conflict: Read-Write

First  Thread   : aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:92)
 Objects Accesed:
  HNode[1]: aaaTestFiles.BoundedBuffer (at aaaTestFiles.BoundedBuffer.main(
       BoundedBuffer.java:91))
 Locks Held     :
  HNode[37]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
       java:20))
  HNode[36]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
       java:18))

Second  Thread  : aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:93)
 Objects Accesed:
  HNode[1]: aaaTestFiles.BoundedBuffer (at aaaTestFiles.BoundedBuffer.main(
       BoundedBuffer.java:91))
 Locks Held     :
  HNode[37]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
       java:20))

Common Locks    :
 HNode[37]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
      java:20))
Path to access  : (Read , waitingTakes, aaaTestFiles.BoundedBuffer.put(
     BoundedBuffer.java:52))
        at aaaTestFiles.BoundedBuffer.put(BoundedBuffer.java:52)
        at aaaTestFiles.Producer.run(BoundedBuffer.java:130)
        at aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:92)

Path to access  : (Write, waitingTakes, aaaTestFiles.BoundedBuffer.take(
     BoundedBuffer.java:62))
        at aaaTestFiles.BoundedBuffer.take(BoundedBuffer.java:62)
        at aaaTestFiles.Consumer.run(BoundedBuffer.java:108)
        at aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:93)
--- End Common Lock Held Memory Access ---

------ Begin Common Lock Held Memory Access ------
Field           : waitingTakes
Declaring Class : aaaTestFiles.BoundedBuffer
Type of Conflict: Read-Write

First  Thread   : aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:92)
 Objects Accesed:
  HNode[1]: aaaTestFiles.BoundedBuffer (at aaaTestFiles.BoundedBuffer.main(
       BoundedBuffer.java:91))
 Locks Held     :
  HNode[37]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
       java:20))
  HNode[36]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
       java:18))

Second  Thread  : aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:93)
 Objects Accesed:
  HNode[1]: aaaTestFiles.BoundedBuffer (at aaaTestFiles.BoundedBuffer.main(
       BoundedBuffer.java:91))
```

```
 Locks Held     :
  HNode[37]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
       java:20))

Common Locks    :
 HNode[37]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
      java:20))
Path to access  : (Read , waitingTakes, aaaTestFiles.BoundedBuffer.put(
     BoundedBuffer.java:52))
         at aaaTestFiles.BoundedBuffer.put(BoundedBuffer.java:52)
         at aaaTestFiles.Producer.run(BoundedBuffer.java:130)
         at aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:92)

Path to access  : (Write, waitingTakes, aaaTestFiles.BoundedBuffer.take(
     BoundedBuffer.java:69))
         at aaaTestFiles.BoundedBuffer.take(BoundedBuffer.java:69)
         at aaaTestFiles.Consumer.run(BoundedBuffer.java:108)
         at aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:93)
--- End Common Lock Held Memory Access ---

------ Begin Common Lock Held Memory Access ------
Field           : waitingTakes
Declaring Class : aaaTestFiles.BoundedBuffer
Type of Conflict: Read-Write

First  Thread   : aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:92)
 Objects Accesed:
  HNode[1]: aaaTestFiles.BoundedBuffer (at aaaTestFiles.BoundedBuffer.main(
       BoundedBuffer.java:91))
 Locks Held     :
  HNode[37]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
       java:20))
  HNode[36]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
       java:18))

Second  Thread  : aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:93)
 Objects Accesed:
  HNode[1]: aaaTestFiles.BoundedBuffer (at aaaTestFiles.BoundedBuffer.main(
       BoundedBuffer.java:91))
 Locks Held     :
  HNode[37]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
       java:20))

Common Locks    :
 HNode[37]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
      java:20))
Path to access  : (Read , waitingTakes, aaaTestFiles.BoundedBuffer.put(
     BoundedBuffer.java:52))
         at aaaTestFiles.BoundedBuffer.put(BoundedBuffer.java:52)
         at aaaTestFiles.Producer.run(BoundedBuffer.java:130)
         at aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:92)

Path to access  : (Write, waitingTakes, aaaTestFiles.BoundedBuffer.take(
     BoundedBuffer.java:69))
         at aaaTestFiles.BoundedBuffer.take(BoundedBuffer.java:69)
         at aaaTestFiles.Consumer.run(BoundedBuffer.java:108)
         at aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:93)
--- End Common Lock Held Memory Access ---

------ Begin Common Lock Held Memory Access ------
Field           : usedSlots
Declaring Class : aaaTestFiles.BoundedBuffer
Type of Conflict: Write-Write

First  Thread   : aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:92)
```

```
 Objects Accesed:
  HNode[1]: aaaTestFiles.BoundedBuffer (at aaaTestFiles.BoundedBuffer.main(
       BoundedBuffer.java:91))
 Locks Held     :
  HNode[37]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
       java:20))
  HNode[36]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
       java:18))

Second  Thread  : aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:93)
 Objects Accesed:
  HNode[1]: aaaTestFiles.BoundedBuffer (at aaaTestFiles.BoundedBuffer.main(
       BoundedBuffer.java:91))
 Locks Held     :
  HNode[37]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
       java:20))

Common Locks    :
 HNode[37]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
      java:20))
Path to access  : (Write, usedSlots, aaaTestFiles.BoundedBuffer.put(BoundedBuffer.
     java:51))
        at aaaTestFiles.BoundedBuffer.put(BoundedBuffer.java:51)
        at aaaTestFiles.Producer.run(BoundedBuffer.java:130)
        at aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:92)

Path to access  : (Write, usedSlots, aaaTestFiles.BoundedBuffer.take(BoundedBuffer
     .java:72))
        at aaaTestFiles.BoundedBuffer.take(BoundedBuffer.java:72)
        at aaaTestFiles.Consumer.run(BoundedBuffer.java:108)
        at aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:93)
--- End Common Lock Held Memory Access ---

------ Begin Race Condition Detected ------
Field           : emptySlots
Declaring Class : aaaTestFiles.BoundedBuffer
Type of Conflict: Read-Write

First  Thread   : aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:92)
 Objects Accesed:
  HNode[1]: aaaTestFiles.BoundedBuffer (at aaaTestFiles.BoundedBuffer.main(
       BoundedBuffer.java:91))
 Locks Held     :
  HNode[36]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
       java:18))

Second  Thread  : aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:93)
 Objects Accesed:
  HNode[1]: aaaTestFiles.BoundedBuffer (at aaaTestFiles.BoundedBuffer.main(
       BoundedBuffer.java:91))
 Locks Held     :
  HNode[37]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
       java:20))

Common Objects  :
 HNode[1]: aaaTestFiles.BoundedBuffer (at aaaTestFiles.BoundedBuffer.main(
      BoundedBuffer.java:91))

Path to access  : (Read , emptySlots, aaaTestFiles.BoundedBuffer.put(BoundedBuffer
     .java:33))
        at aaaTestFiles.BoundedBuffer.put(BoundedBuffer.java:33)
        at aaaTestFiles.Producer.run(BoundedBuffer.java:130)
        at aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:92)
```

```
Path to access  : (Write, emptySlots, aaaTestFiles.BoundedBuffer.take(
     BoundedBuffer.java:73))
         at aaaTestFiles.BoundedBuffer.take(BoundedBuffer.java:73)
         at aaaTestFiles.Consumer.run(BoundedBuffer.java:108)
         at aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:93)
--- End Race Condition Detected ---

------ Begin Race Condition Detected ------
Field           : emptySlots
Declaring Class : aaaTestFiles.BoundedBuffer
Type of Conflict: Read-Write

First  Thread   : aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:92)
 Objects Accesed:
  HNode[1]: aaaTestFiles.BoundedBuffer (at aaaTestFiles.BoundedBuffer.main(
       BoundedBuffer.java:91))
 Locks Held     :
  HNode[36]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
       java:18))

Second  Thread  : aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:93)
 Objects Accesed:
  HNode[1]: aaaTestFiles.BoundedBuffer (at aaaTestFiles.BoundedBuffer.main(
       BoundedBuffer.java:91))
 Locks Held     :
  HNode[37]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
       java:20))

Common Objects  :
 HNode[1]: aaaTestFiles.BoundedBuffer (at aaaTestFiles.BoundedBuffer.main(
      BoundedBuffer.java:91))

Path to access  : (Read , emptySlots, aaaTestFiles.BoundedBuffer.put(BoundedBuffer
     .java:45))
         at aaaTestFiles.BoundedBuffer.put(BoundedBuffer.java:45)
         at aaaTestFiles.Producer.run(BoundedBuffer.java:130)
         at aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:92)

Path to access  : (Write, emptySlots, aaaTestFiles.BoundedBuffer.take(
     BoundedBuffer.java:73))
         at aaaTestFiles.BoundedBuffer.take(BoundedBuffer.java:73)
         at aaaTestFiles.Consumer.run(BoundedBuffer.java:108)
         at aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:93)
--- End Race Condition Detected ---

------ Begin Race Condition Detected ------
Field           : emptySlots
Declaring Class : aaaTestFiles.BoundedBuffer
Type of Conflict: Write-Write

First  Thread   : aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:92)
 Objects Accesed:
  HNode[1]: aaaTestFiles.BoundedBuffer (at aaaTestFiles.BoundedBuffer.main(
       BoundedBuffer.java:91))
 Locks Held     :
  HNode[36]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
       java:18))

Second  Thread  : aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:93)
 Objects Accesed:
  HNode[1]: aaaTestFiles.BoundedBuffer (at aaaTestFiles.BoundedBuffer.main(
       BoundedBuffer.java:91))
 Locks Held     :
  HNode[37]: java.lang.Object (at aaaTestFiles.BoundedBuffer.<init>(BoundedBuffer.
       java:20))
```

```
Common Objects  :
 HNode[1]: aaaTestFiles.BoundedBuffer (at aaaTestFiles.BoundedBuffer.main(
      BoundedBuffer.java:91))

Path to access  : (Write, emptySlots, aaaTestFiles.BoundedBuffer.put(BoundedBuffer
     .java:45))
        at aaaTestFiles.BoundedBuffer.put(BoundedBuffer.java:45)
        at aaaTestFiles.Producer.run(BoundedBuffer.java:130)
        at aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:92)

Path to access  : (Write, emptySlots, aaaTestFiles.BoundedBuffer.take(
     BoundedBuffer.java:73))
        at aaaTestFiles.BoundedBuffer.take(BoundedBuffer.java:73)
        at aaaTestFiles.Consumer.run(BoundedBuffer.java:108)
        at aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:93)
--- End Race Condition Detected ---

---------------------------------------------------------------------
 Threads Found:
  StartLocation : aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:92)
  RunMethod     : <aaaTestFiles.Producer: void run()>
  StartLocation : aaaTestFiles.BoundedBuffer.main(BoundedBuffer.java:93)
  RunMethod     : <aaaTestFiles.Consumer: void run()>

 Race Condition Statistics
  Number of Access to Different Objects :         0
  Number of Locked Acccess              :         5
  Unknown Points-To information for     :         0
  Number of Potential Race Conditions   :         3
  Total Number of Accesses              :         8

 Time measurements
  Setup Soot                    : 14.4s (86.5%)
  Incomplete Call Graph         : 01.8s (10.5%)
  RTA Reachable Methods         : 00.0s (00.1%)
  Points-To Analysis            : 00.2s (01.4%)
  Thread Finding                : 00.1s (00.6%)
  Race Condtions Finding        : 00.0s (00.1%)
  Total Time                    : 16.6s (100.0%)

  Application Metrics
   Number of Classess        : 4
   Number of Stmts           : 192
   Number of Reachable Stmts : 192
   Coverage Percentage       : 100.00
   Number of Methods         : 10
   Number of Assignments     : 101
   Number of Field Reads     : 36
   Number of Field Writes    : 23
---------------------------------------------------------------------
```