NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

JORGE LOPES FERREIRA

BSc in Software Engineering

STATIC ANALYSIS FOR DATA-RACE DETECTION IN JAVA

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon

April, 2024

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

STATIC ANALYSIS FOR DATA-RACE DETECTION IN JAVA

JORGE LOPES FERREIRA

BSc in Software Engineering

Advisers: Hervé Miguel Cordeiro Paulino
Associate Professor, FCT-NOVA
António Maria Lobo César Alarcão Ravara
Associate Professor, FCT-NOVA

Examination Committee

Chair: João Carlos Gomes Moura Pires
Associate Professor, FCT-NOVA

Rapporteur: Tiago Soares Cogumbreiro Garcia
Assistant Professor, UMass Boston

Adviser: Hervé Miguel Cordeiro Paulino
Associate Professor, FCT-NOVA

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
April, 2024

Static Analysis for Data-race detection in Java

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.18) [17].

Abstract

Over a decade ago the architectural paradigm shifted from single-core to multi-core/multi-threaded CPUs. Since then programming in parallel became crucial for developing efficient and faster programs. With concurrent programming came the challenge of writing programs free of data races. There are mechanisms to control the concurrency by means of synchronizing the threads, but using them does not always guarantee a correct program.

To overcome this issue, in this thesis we investigate and present how we can use static program analysis to detect data races in Java classes without any extra input from the programmer. We used static analysis in Java to perform points-to analysis to find the accesses that cause data races.

We benchmarked our tool with some open source code examples and compared the results of our solution with an existing tool for static analysis. We found that, while our tool still needs more refinement and optimization, it can already detect most, if not all of the data races in Java classes.

Keywords: program analysis, points-to analysis, static analysis, data race

Resumo

Há mais de uma década, o paradigma arquitetural mudou de CPUs single-core para CPUs multi-core/multi-threaded. Desde então programar em paralelo tornou-se crucial para desenvolver programas mais rápidos e eficientes. Com a programação concorrente veio o desafio de escrever programas livres de data races. Existem mecanismos que efetuam controlo de concorrência ao sincronizarem as threads, mas usá-los por si só não garante um programa correto.

Para resolver este problema, neste trabalho investigamos e apresentamos como podemos usar análise estática de programas para detetar data races em classes Java sem qualquer informação extra dada pelo do utilizador.

Fizemos testes à nossa ferramenta usando exemplos de código de fonte aberta e comparámos os resultados da nossa solução com os de uma ferramenta de análise estática já existente. Descobrimos que, apesar de a nossa ferramenta ainda precisar de ser aperfeiçoada e otimizada, esta já consegue detetar a maioria, senão todas as data races em classes Java.

Palavras-chave: análise de programas, points-to analysis, análise estática, data race

Contents

List of Figures

List of Algorithms

List of Listings

Acronyms

API Application Programming Interface *(p. 2)*
AST Abstract Syntax Tree *(p. 60)*

CLR Common Language Runtime *(p. 10)*
CPU Central Processing Unit *(pp. 1–3)*

ILP Instruction-Level Parallelism *(p. 1)*

OS Operating System *(pp. 2, 3, 16)*

PTA Points-to Analysis *(pp. viii, 9, 16–19, 23–29, 33, 52, 59, 60, 71)*

RAM Random Access Memory *(pp. 1, 2)*
ROB Release-on-Block *(p. 16)*

1 Introduction

This chapter aims to present the context of computer systems and a common problem with it, data races in concurrent programs. We describe what they are, how they exist and why. Towards the end of the chapter, we propose a solution to this problem.

1.1 Motivation

In the twentieth century, almost every Central Processing Unit (CPU) had a single core, which led to programming languages that focused on a linearization of the code since that made the job simpler for programmers. As the years passed, the performance of computers increased thanks to architectural changes and optimizations that allowed for faster switching speeds from the transistors that made faster clock speeds possible and also implicit parallelism that allowed the hardware to execute more than an instruction at a time when it was possible and required no changes in the code of the already written programs.

However, in 2005 this architectural paradigm had reached its limits and could not offer much more performance without running into problems. Mainly three problems converged at this time and made future improvements to single core CPUs no longer viable. These became known as the "three walls", which consisted of the power wall, the Instruction-Level Parallelism (ILP) wall and the memory wall [30].

The power wall was caused by the growth in power usage with the clock rate. Power usage does not increase linearly with clock speeds, which meant that even faster clock speeds would lead to power-inefficient systems, which could not be air cooled properly and therefore required more expensive cooling solutions [30].

The ILP wall appeared, because these parallelism mechanisms, that had been already in use for four decades, had finally reached their limits. By this time, the point of diminishing returns in performance from ILP mechanisms had passed and further gains from them were of little use, since they only offered performance bursts, or could not be sustained by real processors with limited resources [30].

The memory wall manifested due to the speed difference between CPUs and Random

Access Memory (RAM). Off-chip memory speeds did not increase as much as CPU clock speeds, because of the power usage and the number of pins that can be incorporated on RAM chips. In consequence, the performance of many programs is limited by the latency of the RAM [30].

The most viable solution to keep improving computer performance was and still is the use of CPUs with multiple threads and/or multiple cores, an architecture that allows for even more parallelism with less power usage and less heat produced. Almost all computers need to run multiple programs in parallel. With more parallelism, programs will have more CPU resources available and less time spent competing for them.

The added parallelism from multiple cores and threads also benefits distributed systems and micro-services due to their parallel nature. Distributed systems are composed by hardware and software interlinked by a communication infrastructure, where the computers need to communicate with each other through messages to coordinate their execution of the service. Micro-services is a program architecture where the program is a group of independent small services that communicate with each other using Application Programming Interfaces (APIs) [33, 32].

Since multi-core systems generate less heat, that means air cooling will still be enough for future systems and devices like smartphones can remain cool without bulky cooling solutions. Power efficiency is also specially desirable on portable computers, since these need to run off of their batteries which are a limited source of energy.

A process is a running program and contains its own context and memory address space. A thread is a lighter process, which despite having its own context, shares the memory address space, data and the used libraries with the main process and the other threads [2]. Single threaded programs are usually seen by the Operating System (OS) as running processes, but concurrent programs have other processes or threads associated with the main process.

For programs to benefit from the power of multi-threaded/multi-core processors and for programmers to have more control over the multiple threads of the program, when necessary and possible, programs need to be concurrent. This way various instructions can be executed at the same time, provided the hardware the program runs on has more than one core or hardware thread available.

Harnessing the power of multi-threaded/multi-core systems is the way to keep improving the performance of programs, but it is not an easy one. Because of the way these systems function, making good use of them with a single core paradigm background and/or languages designed during that paradigm. Section 1.2 explains why that is the case.

1.2 The Problem

Due to the nature of multi-thread/multi-core hardware and due to the non-determinism behavior of OS CPU schedulers, there are many ways a concurrent program may execute.

```
1 function add(e) {
2     atomic {
3         arr[count] = e;
4         count++;
5     }
6 }
```

Listing 1.1: Example function

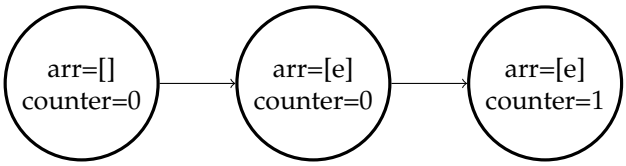


Figure 1.1: State chart of the single-threaded add function

Each way differs in the order of instructions executed. Each one of the possible instruction sequences is called an *interleaving*.

Sequential programs are easy to reason with since they only have one interleaving, making it is easy to analyze each and every single state of the program. Concurrent programs have two or more interleavings and as the amount of interleavings grows linearly, the amount of states grows exponentially, making it impossible for a programmer to make sure the program is correct when many threads are involved. This is the state explosion problem.

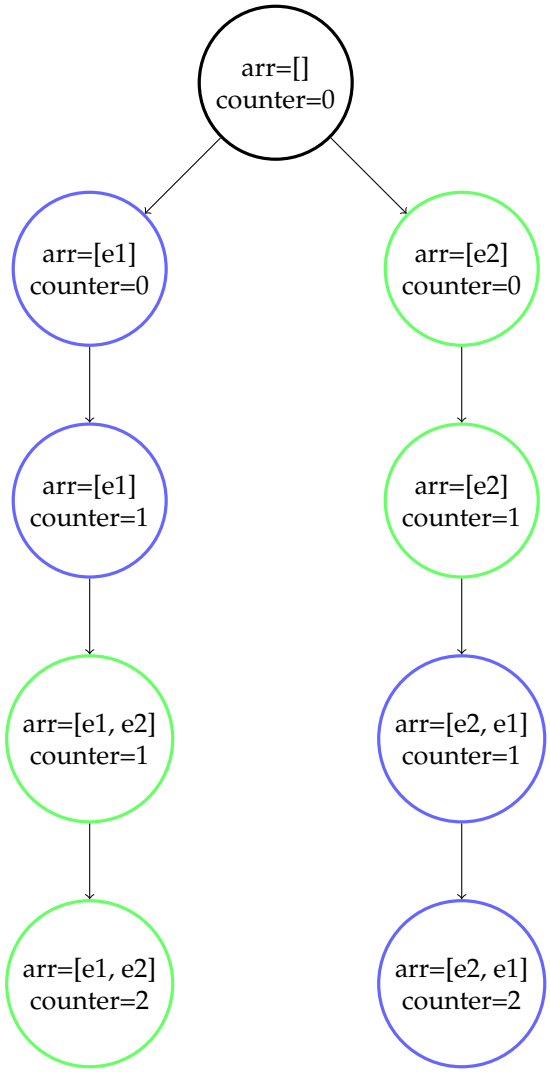
Lets pay attention to listing 1.1. Assume that *e* is an object, *arr* is an array of objects and *count* is a number. Also assume *arr* and *count* are variables defined outside of the function itself and the code inside the atomic block is executed atomically.

If we assume the program is sequential, we have a finite number of states, which means a programmer can mentally observe every single state possible. In this case, without counting the initial state, we have two states as shown in figure 1.1.

However, if we assume multiple threads are running this function, since only one thread at a time can execute the whole atomic block and we do not know which thread will execute the code next, we have many possible states. In that case, we cannot observe every possible state within a useful amount of time. In figure 1.2 is the example for 2 threads. In this case, we now have 2 possible interleavings and 8 different states.

As it is often the case, some of the possible interleavings are wrong and cause issues. These manifest themselves in the form of data races or race conditions and both can happen in the same program. While data race and race condition are often used interchangeably, they are actually two different things.

A race condition happens when the timing and ordering of code execution affects the code correctness, which makes these harmful for the proper execution of a concurrent program [28, 16]. The same group of code instructions is executed by multiple threads and due to the non-deterministic behavior of the OS CPU scheduler, threads are able to



The blue circles represent the states when thread 1 is executing the code.
The green circles represent the states when thread 2 is executing the code.

Figure 1.2: State chart of the add function with 2 threads

perform sequences of code instructions that are incorrect and break invariants of the program. Listing 1.2, taken from the function *transfer2* found in [28] exemplifies a situation in which a race condition can occur.

Assume that *amount* is a positive number, *account_from* and *account_to* are bank accounts and that anything inside an atomic block is executed atomically. This is a function that transfers money from *account_from* to *account_to*. While this is a rather silly example, it is trivial enough for one to see the problem with this code. When executed concurrently by two threads for the same bank accounts, there is a chance money can be created or lost, which results in an erroneous execution and an anomalous result.

A data race is a more specific case of a race condition and occurs when two or more threads concurrently access the same memory location and at least one of these threads writes data to it [18]. This can happen on concurrent programs that read and write data

```
function transfer(amount, account_from, account_to) {
    atomic { bal = account_from.balance; }
    if (bal < amount) return false;
    atomic { account_to.balance += amount; }
    atomic { account_from.balance -= amount; }
    return true;
}
```

Listing 1.2: Example with a race condition

```
function inc(counter) {
    counter++;
}
```

Listing 1.3: Example with data race

shared with multiple threads without proper synchronization. Data races can cause unexpected behavior, depending on the logic of the program, since there may at least be one thread that will be able to see data in an inconsistent state. Listing 1.3 provides a simple example of code with a data race.

Assume that *counter* is a number. This is a function that increments the value of counter by one. When used by multiple threads with the same variable, it is possible that some of the increments get lost since each write operation is not atomic.

To fix problems like these, the programmer should use immutable and non shared variables where possible and must control the execution flow of the program to avoid all thread interactions with shared memory that are problematic. To control the execution, the programmer needs to synchronize the access to the shared memory, this can be done with monitors, locks or semaphores.

Sadly, these solutions are no silver bullet, they can be misused and cause data races and/or race conditions in the program, so they do not guarantee to fix the problems. If you do not synchronize the threads enough, data races and race conditions will still occur, but with too much synchronization, you can still have race conditions and possibly deadlocks [30]. A deadlock occurs when two or more threads wait circularly for each other to release a lock. Deadlocks can cause the program to completely stop progressing and never terminate.

Reasoning about concurrent programs is fundamental, but very difficult at times. Many things can possibly go wrong and debugging data races can be very time consuming. For this reason, there have been attempts at designing and implementing tools to perform program analysis. These tools can analyze the program and detect issues in it such as data races. However, most of these tools have various problems that limit their usability and practicality, such as the requirement for custom annotations. This can lead to the previously mentioned problem of misusing locks, monitors or semaphores.

It would be convenient and more reliable for programmers to have an automated tool that could check for data races in the program without the need for extra information. In section 1.3 we propose a solution for this problem. Later in chapter 2 we will clarify what

is program analysis and show previous work on that field.

1.3 Proposed Solution

Our objective is to be able to detect data races in a program without the need for the programmers to provide more information. To that end, we wanted to devise and implement a tool that accepts a Java program without any concurrency control as input and reports data races that can occur in the program. We want to employ static analysis for the data race detection. The tool must not require any information other than the source code of the input program. It also should detect all possible data races in the program and should report no false positives.

The end goal is a solution that can analyze any Java program, compile it and inject the necessary concurrency control code. We plan to to use an existent tool for concurrency control injection called *AtomiS* [23, 25, 24, 21], which requires annotations to generate the code. Since this tool is not completely automated, our solution is to integrate it with our solution for data race detection. This thesis is a step towards that goal, focusing on the program analysis needed.

1.4 Contributions

With this work, we expect to make the following contributions:

1. Provide an algorithm for static data race detection.
2. Provide an implementation in Java.
3. Provide an evaluation of the solution with a comparison against an existent solution.

1.5 Document Outline

In chapter 2, we clarify what is program analysis and the differences between static and dynamic analysis. We also mention the advantages and disadvantages of static and dynamic analysis and justify why we chose static analysis over dynamic analysis.

Next, the details of how we solved the problem in section 1.2 are shown in chapter 3. We give an overview of our solution and describe how we got to it, by explaining the algorithm behind it and highlight the relevant information. Finally, at the end of the chapter, we talk about the implementation of the algorithm.

In chapter 4, we detail the means to evaluate our solution and analyse the limitations of the algorithm. Then, we analyse the results obtained with our implementation and compare them with those given by an existent solution. To finalise, we make our conclusions and refer the future work in chapter 5.

2

Background and Related Work

In this chapter we will explain what is program analysis, how it works and what is its general objective. We will also compare static and dynamic analysis and justify our choice between them. Finally, we will mainly analyze work previously done in this field and discuss how we can possibly use past knowledge to solve the current problem.

2.1 Program Analysis

Program analysis involves getting information about dynamic properties of a program. These usually include the set of values, locks used, aliases, traces and execution paths. This information can be useful to optimize the program or to correct errors, which can range from trivial logical issues to data races [5, 22]. Depending on the type of analysis, it may require input from the programmer in the form of annotations or specifications to provide more information or it can be completely automatic. There are different techniques and methods to extract the program information, but these can be divided between two main types of program analysis, static and dynamic analysis.

Static Analysis [22] is the category of techniques performed to analyze either the source code of a program or a compiled program, which means no execution is done during the analysis. This type of analysis normally attempts to predict safe and computable approximations of values and/or behaviors of the program. Static analysis can be used by compilers to generate code for avoiding unnecessary computations in the program, but can also be used to check a program for (potential) errors.

Dynamic Analysis [34] is the category of techniques performed to analyze the execution of a program. Usually this analysis checks the values of variables and state transitions of the program at every execution step or only for critical steps. This type of analysis is often used by profilers, but can also be used to catch errors in the programs much like static analysis. Dynamic analysis tools can analyze a program execution either by doing so while the program is running or by analyzing execution logs. These two techniques are called online checking and postmortem checking respectively. Among the

dynamic analysis tools, two main analysis methods are also used, lockset analysis and happens-before analysis.Other methods and techniques are used as variations of lockset or happens-before analysis or used to enhance the algorithm.

Lockset analysis involves verifying if the program obeys a locking discipline that ensures the absence of data races. For this the program looks at the sets of locks used during the execution and checks if shared memory accesses/code blocks are protected by locks and that the locksets used on both threads correspond. Happens-Before analysis instead uses the Lamport's happens-before relation and synchronization events to try defining an order of the program statements. When two conflicting memory accesses have no definable temporal ordering, this indicates a data race. Some dynamic analysis algorithm are designed to include forms of both lockset and happens-before analyzes to create a hybrid analysis.

2.2 Dynamic vs Static Analysis for Data-Race Detection

As said in the previous section, both static and dynamic analyzes can be used to perform data race detection on a program. However, they do similar tasks in different ways and there is no fully flawless analysis, both will have different advantages and disadvantages. In this section we compare both analyzes and look at a brief review of work previously done in both.

Static analysis relies on analyzing the code of the program instead of its execution, meaning the analysis can be done before the program is compiled. This allows for the exploration of all possible paths in the program, which reduces the possibility of data races being missed. It also makes the tool more practical for analyzing programs that must run without extra overhead or cannot be run at the site the analysis is done. Unfortunately, this analysis does not come without problems to solve.

Since this analysis works with approximations of the program behavior, it can report many false positives, making the inspection of the output rather time consuming. A simple solution to reduce false alarms involves the use of escape mechanisms to tell the analysis to not report certain issues in parts of the code like in [11]. Another solution is filtering the possible data races according to the documented thread safety of the code [27].

Both of these solutions require apriori knowledge to be manually inserted into the code, usually in the form of annotations. This can be problematic when the annotation overhead is too high and without annotation inference [10] the analysis becomes impractical for large programs. These techniques also compromise the soundness of the analysis since it no longer can guarantee to find all data races.

Most static analysis algorithms require the whole program for analysis [11, 12, 8, 20, 14, 27, 26], which in some situations implies analyzing hundreds of thousands of lines of code. This makes the analysis less scalable for large programs and less efficient with the resources used. Some work has been done to solve this problem. In [14] whole-program

Points-to Analysis (PTA) is used to build a concurrent control flow graph. This graph is then used by the algorithm to decide if it is worth exploring each function call in the program. This reduces the set of contexts the analysis needs to take into account and reduces execution times.

Dynamic analysis executes the program in order to analyze it. Because it works with real values instead of approximations, races reported by it have less chances of being false positives. It also does not need the whole program to run in order to analyze it, which means it can scale better in large programs. This comes at a cost of soundness since it can miss data races hidden in program code that was not executed during the analysis. Running the program more than once is a possible solution, but this adds extra time needed for the analysis and it is still difficult to cover the entire code of the program.

Online dynamic tools analyze the code as it executes, while postmortem dynamic tools analyze a log of the program execution. Both types of tools need to add their own instrumentation to the program to make it perform the necessary calls for the analysis or the logging, which means they introduce additional overhead to the execution to be able to perform the analysis [34, 19]. Postmortem dynamic tools are also impractical for the analysis of programs that have a long runtime, since these require the program to terminate [34]. Work has been done on dynamic analysis tools to try reducing the performance overhead of the program execution. We will show some work previously done on dynamic analysis in section 2.3.

Static analysis can catch all data races when correctly done. In dynamic analysis this is very difficult to do since it can only report data races that have happened. The best that could be done in this case is to execute the tool every time the program executes, which is far from ideal due to the execution overhead. For this reason, we chose to use static analysis for our solution.

2.3 Dynamic Analysis for Data Race Detection

In this section we will present some of the work already done on dynamic analysis and what techniques were employed to reduce the execution overhead.

TRaDe [4] In 2001, Mark Christiaens and Koen De Bosschere presented their topological method for data race detection for the Java language implemented into the *TRaDe* tool. Since an object can only be manipulated through a reference in Java, the method simply maintains an interconnection graph of used objects during the program execution and looks for reference manipulations that alter this graph. The complete data race detection using happens-before analysis is only employed for objects reachable by more than one thread, which reduces the runtime and memory usage.

The analysis itself uses vector clocks to check if two segments are ordered. A segment is a set of operations of a thread between 2 consecutive synchronization operations and a vector clock is an array of integers of each thread that is updated at each synchronization

operation. The analysis flags a data race when two parallel segments access a common variable and one of the segments performs a write. These vector clocks grow as large as the total amount of spawned threads, which means they can use a lot of memory unnecessarily. To prevent this, the authors invented the accordion clocks technique, which removes dead threads that did not touch any data structures that still exist, reducing the memory usage.

RaceTrack [34] In 2005, Yuan Yu, Tom Rodeheffer and Wei Chen presented their research on a hybrid analysis algorithm they implemented into *RaceTrack* inside of the Common Language Runtime (CLR) from Microsoft. This tool performs online checking on the input program. The tool tracks the lockset and a vector clock for each thread and only tracks the lockset for each object. Whenever the lockset of an object is updated to the empty set, the thread set is tracked and if it contains more than one thread, a possible data race occurred. This object granularity and the adaptive thread set tracking reduce the runtime and memory usage.

Since object granularity is less precise, once a possible data race is detected in an object, *RaceTrack* performs field granularity monitoring for that object. On a field of a suspicious object, if the tracked thread set has more than 2 threads, the data race is confirmed and flagged. To avoid cases where memory usage could still be excessive, for arrays the tool only tracks a specific amount of indexes at the beginning, middle and end, which can make it miss some data races. The size of vector clocks is also bounded, and when one reaches the imposed size limit, the tool shrinks it in half by removing the older elements. With these enhancements, according to the tests of the authors, *RaceTrack* execution is up to 3 times slower and uses up to 52% more memory compared to a normal execution.

LiteRace [19] In 2009, Daniel Marino, Madanlal Musuvathi and Satish Narayanasamy showed an approach to data race detection based on sampling, which they implemented on the *LiteRace* tool. This algorithm is based on the cold-region hypothesis that:

> data races are likely to occur when a thread is executing a "cold" (infrequently accessed) region in the program.

At the start of the execution the tool will sample all code regions equally, but as a region is sampled, its sampling rate is reduced until reaching a lower bound. This is to avoid slowing down the execution of code that runs more frequently, at the cost of possibly missing data races. *LiteRace* creates 2 copies of each code region for this purpose.

The sampling involves logging synchronization events with a logical timestamp and logging read/write operations in the program order, logically happening at the timestamp of the preceding synchronization event. For frequently executed code, only the synchronization operations are logged. The logical timestamp is updated atomically. Once the execution ends, these logs are analyzed with happens-before analysis. With the cold-region based sampling technique, *LiteRace* could keep the execution slowdown below two

and a half times compared with the normal execution according to the benchmarks done by the authors.

BigFoot [29] In 2017, Dustin Rhodes, Cormac Flanagan and Stephen Freund showed their work in dynamic analysis with the *BigFoot* tool. This tool employs intraprocedural static analysis to place the checks to be done in the program. The checks are deferred as long as possible and only inserted when deferring them further could false positives or false negatives. This reduces the amount of checks performed on the input Java program. After the static analysis, one last step coalesces the sets of checks placed in the program. The deferred check placement and check coalescing reduce the amount of checks that the tool needs to perform during the execution of the program, which reduces the execution slowdown.

The technique of check coalescing allows the use of two other techniques, shadow compression of object fields and shadow compression of arrays. Using static shadow proxy analysis, *BigFoot* compresses the shadow locations of object fields before execution. Compression of the shadow state of arrays is done dynamically during the execution. At first it compresses the state of the entire array, then refines the representation as the program executes. In the tests done by the authors, the tool had a mean execution slowdown of two and a half times. However, in specific cases, the execution could be over 10 times slower compared with the normal execution.

2.4 Static Analyses for Data Race Detection

Various static analysis tools have been devised in the past to perform data race detection and while most require the whole program for analysis, more recent ones can instead analyze parts of it at a time. As these tools evolved, with different methods and techniques, there was a general focus on making the analysis process more automatic and on reducing false positives. In this section, we will show work previously done, including an overview of the devised methods and of their implementations.

RCCJava [11] In 2000, Cormac Flanagan and Stephen Freund investigated the implementation of type system analysis, starting with the type system previously devised by Cormac in 1999. A concurrent subset of Java was defined and named "ConcurrentJava", which includes the fork operation to spawn threads and the synchronized expression to represent the synchronized statement in Java. They also defined the grammar to show how programs, classes, variables, etc are defined in the subset.

The authors defined an extension to the subset and named it "RaceFreeJava". This extension includes annotations to track the locks used on a program, ghost variables to allow programs using locks outside of classes to be correctly verified and an extra *thread_local* modifier, with which one can define a class of objects used only in a single thread without using the *guarded_by* annotation.

```
1  class Account {
2      final Object lock = new Object();
3
4      //# guarded_by lock
5      int balance = 0;
6
7      //# requires lock
8      void update(int n) { balance = n; }
9
10     void deposit(int x) {
11         synchronized(lock) {
12             update(balance + x);
13         }
14     }
15 }
```

Listing 2.1: RCCJava bank account example

```
1 f.a = 3; //# no_warn race
2
3 //# holds f
4 f.a = 3;
```

Listing 2.2: RCCJava escape annotations

The type system was redefined on top of this extended Java subset and then adapted to the entirety of the Java language with the implementation of the *RCCJava* tool. The objective of this tool is the detection of race conditions on Java programs with locks. According to Cormac and Flanagan, race conditions occur when "two threads manipulate a shared data structure simultaneously, without synchronization". The listing 2.1 adapted from [10] shows an example of a class in Java using the *guarded_by* and *requires* annotations.

The type system is sound as proved later with the *Race-Freedom* theorem supported by various lemmas. The theorem states that, if a program P yields type t, then the program does not have a race condition [1]. Rules were also provided to extend the lemmas and the theorem to thread-local classes.

Unfortunately, as mentioned by the authors, the system proved too restrictive to be effectively used in non-exemplified situations. Cormac and Flanagan included escape mechanisms in *RCCJava*. These consisted of an annotation and a flag "no_warn" to disable some warnings, an annotation "holds" to assert a lock is held at a certain line of code and a flag "-constructor_holds_lock" to assert that constructors hold the lock *this*.

Listing 2.2 shows examples of what the escape annotations look like. Firstly we have an example of the *no_warn* annotation being used to supress race condition warnings. Then we have an example for the use of the *holds* annotation to assert that a particular lock is being held.

The analysis escape mechanisms, albeit unsound, are necessary to greatly reduce the amount of false positives the tool would otherwise report. For a program with data races, the output of the tool includes errors related with lock usage mistakes when accessing fields and the relevant code location in the program.

The implemented tool takes as input a Java program with annotations. Since it requires proper annotated code to correctly find data races, this means lack of experience using the tool may lead to improper use of the annotations, which in turn can lead to false negatives. Additionally, the authors noticed the tool needs an average of twenty annotations per thousand lines of code, which can be done in roughly an hour according to them. For really large code bases, this translates to hundreds of hours required to annotate everything. Such annotation overhead renders the tool less useful for inexperienced users and makes it scale poorly for larger programs. The tool was also limited in the range of synchronization methods it could support and would report false positives.

Houdini/rcc [10] To solve the annotation burden of RCCJava, in 2001, Cormac and Flanagan worked on a tool based on the Houdini framework, the *Houdini/rcc*. This tool is meant to provide annotation inference to automate the task for the programmer. The tool first infers all the annotations for a program, then proceeds to run *RCCJava*, as many times as needed, on the program to check which annotations are correct and to delete the annotations that are not correct.

To further reduce the amount of reported false positives, the authors created extensions in the form of flags for both *Houdini/rcc* and *RCCJava*. The extensions added were the "no_override" to disable warnings on thread-local classes that override methods from thread-shared superclasses, the "cons_lock" to make the tools assume the lock *this* is held inside constructors, the "read_only" to ignore constant shared fields not marked as final and "main_lock" to make the tools assume the main thread holds an implicit lock when accessing a static field. These extensions are not significant sources of false negatives since the warnings they mitigate normally do not cause data races, according to the experience of Cormac and Flanagan.

When running tests on *Houdini/rcc*, the authors found out the tool would incorrectly infer certain classes *C* as thread-local and in consequence infer all classes transitively reachable from *C* as thread-local, causing the *RCCJava* tool to report false warnings on the accesses of the fields from these classes. For an easier analysis of these warnings, *Houdini/rcc* clusters the warnings reported as a consequence of a class being deemed thread-local, so that the programmer can ignore the warnings in the cluster when the class is indeed thread-local. While this helps the programmer, it only partially addresses the issue with these false positives.

Thread-Modular Analysis [12] In 2002, Cormac, Flanagan and Shaz Qadeer tried a different approach to static analysis, thread-modular verification with assume-guarantee reasoning. For the basis of their work, they defined a language for parallel programs, the "Plato" language. Listing 2.3 shows an example of a program written in this language.

In this language all operations are atomic and include assignment and lock operations. The constraint of each operation needs to be satisfied by a store of values in order to be executed, otherwise the thread is blocked. If the resultant store of values does not

```
1    || ((assume tid = 1; t1())□(assume tid = 2; t2()))
```

Listing 2.3: Simple example of a Plato program

correspond to the chosen store, the operation goes wrong. These atomic operations are enough to express primitives and control constructs such as *if* and *while* statements. The execution of a "Plato" program is the sequence of operations of the sequential threads.

The idea of the thread-modular verification is to analyze the individual threads of the program. Each thread needs the specification of an environment assumption and a guarantee, both need to be reflexive and the guarantee must be stronger than the assumption of every other thread. These pairs of assumptions and guarantees give us an assume-guarantee decomposition of the program. The *Thread-Modular Verification* theorem states if none of the statements of the program goes wrong, the program does not go wrong, which allows the desired decomposition of the program. This decomposition is used together with a translation of a program to the "Plato" language to get the abstraction of each thread.

The authors also expanded the verification to data invariant checking to show that a program also keeps its invariants. According to the *Invariant Verification* theorem, given an initial store of values and a program with an assume-guarantee decomposition, if an invariant holds at the beginning, the guarantee of every statement preserves the invariant and none of the statements goes wrong, then the invariant is preserved by the program. The programmer would have to supply the invariants for the input program, which could then be used to strengthen the statement assumptions and derive their guarantees.

The implementation of this verification takes as input a Java program with locks and annotations of the environment assumptions, the invariants and assertions of properties. It translates the program to a language like Plato, then derives the thread abstractions from it using the information in the annotations and converts these into verification conditions. Since the implementation is based on the *ESC/Java* tool, the verification conditions are validated using the theorem prover Simplify [6]. For the invalid conditions, the prover generates a counterexample to be processed into an error message that shows the atomic step violating either an invariant, or an environment assumption, or shows an invariant that does not hold. If all conditions are valid, the program is considered correct .

The authors claimed the annotation overhead of this tool is moderate and that it supports more synchronization methods than the previous *RCCJava* tool [11]. But this tool lacks scalability for larger programs, because the conversion of the abstractions to conditions inlines procedure calls. For this reason, larger programs require a combination of thread-modular and procedure-modular verification.

Calvin [13] Cormac, Flanagan *et al* investigated how to combine both thread-modular and procedure-modular verification methods. They extended the Plato language to give semantics to method calls with *Push* and *Pop* statements and semantics for program paths.

As for the verification, what changes is that procedure-modular analysis is performed on procedure calls using procedure specifications.

The authors formalized the *Modular Verification* theorem that states a program P is simulated by a program Q in which every thread executes a specification of the procedure l and if the procedure specification is simulated by a statement that does not go wrong, meaning its execution is correct according to the specification, then the execution of the operations of the specification by a thread satisfy the environment assumption of every procedure transitively called from l. From this, the authors conclude the program Q will not go wrong if it starts executing with a store of values that satisfies the invariant of l.

The modular verification was implemented on the *Calvin* tool, which takes as input a Java program with locks and the necessary annotations. *Calvin* parses the program into Abstract Syntax Trees, which are then type checked and translated to a language with "Plato" syntax. Then the verification conditions are derived through the same process, except that procedure calls are checked with procedure-modular analysis when the specification is available. If the theorem prover fails to terminate verifying a condition in 5 minutes, *Calvin* reports a time-out on the output. The rest of the output is essentially like the previous implementation.

Two optimizations were made on the environment assumptions. If the simplification of the iterated environment assumption is deemed reflexive and transitive by the theorem prover, it is used instead of the full assumption. Environment assumptions can often be split into "a conjunction of actions mentioning disjoint sets of variables, and any two such actions commute". Annotations of procedure specifications are optional, but without them the procedure-modular analysis is not carried out and procedure calls are inlined instead much like in the tool implemented in 2002 [12], which means *Calvin* can only scale to larger programs at the cost of extra annotated information.

RacerX [8] In 2003, Dawson Engler and Ken Ashcraft worked on a static analysis tool with flow-sensitive interprocedural analysis to flag race conditions and deadlocks. The *RacerX* tool takes as input a program written in the C language with locks and optional annotations indicating routines that are single-threaded, multi-threaded, or interrupt handlers and a table with the functions for acquiring and releasing locks and for enabling and disabling interrupts. *RacerX* starts the analysis by writing the control flow graph from the program files.

With the graph files, the tool proceeds to the lockset analysis. It iterates over the roots of all graphs and does a depth first search, analyzing the effects of each path along with the lockset at each call site and storing both in caches. The caching allows the tool to not need to check the same effects and locksets twice. The result of the analysis are the exit locksets.

As the lockset analysis is running, it calls the race condition checking algorithm at every statement. The algorithm relies on heuristics to attribute a score for each error and

errors are ranked based on their total score. The scores given depend on whether the lockset is valid, whether the code is concurrent and whether a variable needs to be protected. The race checking has three modes of operation with different precision levels. To avoid false negatives, the algorithm also implements statistical inferece of locking functions and looks for various lockset analysis errors.

The analysis output of the tool consisted of error messages reporting either a race condition or a deadlock and containing a trace and a score. The errors are ranked according to their score.

The researchers figured semaphores caused false positives since binary semaphores are used as locks and signal-wait semaphores are used for scheduling dependencies. To distinguish both cases, they used belief analysis as a means to classify the semaphores through statistics of lock acquisitions and releases. Other problems with Release-on-Block (ROB) locks and lockset mistakes causing false deadlock reports were also found and solved.

Dawson and Ken claimed that *RacerX* requires less than a hundred annotations for millions of lines of code, making the annotation overhead at most less than one percent of that from *RCCJava* [11]. They did point out some limitations like lack of PTA, inability to resolve functions used as arguments and a speed problem when analyzing Operating System (OS) code due to functions which call large portions of the code that may be called from many different spots. Instead of PTA, pointer variables are represented by their types, which leads to these errors having a lower ranking. The problematic OS functions are skipped when their set of entry locksets is deemed too large, which leads to some missed errors.

Chord [20] In 2006, Mayur Naik, Alex Aiken and John Whaley presented an analysis method consisting of different static analyses used in various stages. This race detection method is meant to provide k-object sensitivity to treat abstract contexts and objects uniformly by defining the contexts as objects with the *this* argument bound at runtime.

The race detection algorithm starts with the Harness Synthesis. This step is only executed for "open programs" with no main method where the execution starts, such as libraries. This step essentially consists of creating a class with an execution starting point that performs calls to the different methods of the program, creating various different scenarios. This extra class makes it more straight forward for the algorithm to find the races. When the input program is a "complete program", the *Soot* framework is used to compute the *OriginalPairs*, an approximation of the unordered pairs of memory accesses that may cause a race.

The bulk of the algorithm is divided into four stages, reachable-pairs computation, aliasing-pairs computation, escaping-pairs computation and unlocked-pairs computation. The stages use four static analyses, call-graph construction, PTA, thread-escape analysis and lock analysis.

The first stage, reachable-pairs computation, filters the *OriginalPairs* using k-object sensitivity to remove pairs of accesses that aren't reachable from the main method, thus computing the *ReachablePairs* subset. The computation is done with call-graph construction and PTA running concurrently to get better precision from both analyzes. Optional annotations can be provided to remove call pairs from this subset and annotations can be provided at the declaration of the interface to specify which methods should not be called in parallel.

The second stage, aliasing-pairs computation, filters the *ReachablePairs* set to remove pairs of accesses that do not access the same memory location to compute the *AliasingPairs* subset. For this computation a form of alias analysis that is k-object sensitive is used.

The third stage, escaping-pairs computation, filters the *AliasingPairs* set to remove pairs of accesses that do not access thread-shared data, resulting in the *EscapingPairs* subset. To get this pair subset thread-escape analysis is used, which depends on call-graph construction and alias analysis. An object is considered thread-shared when it can be accessed through an argument of a call site that spawns threads or it is reachable from a static field. The resultant *EscapingPairs* subset may contain pairs that are not involved in a race since a thread-shared object may not be accessible by more than a thread at the same time, so the programmer may optionally provide annotations per field or class to exclude access pairs that reference that field or fields.

The final stage, unlocked-pairs computation, filters the *EscapingPairs* set to remove pairs of accesses that are not executed by a pair of threads that do not hold a common lock. The *UnlockedPairs* subset is computed with lock analysis, which depends on call-graph construction and alias analysis. Each pair of accesses in the subset is treated as a possible race.

The researchers implemented the algorithm in the *Chord* tool. This takes as input a Java program with locks and optional annotations and outputs any possible races found as error messages with the pairs of accesses represented on a graph with the call paths. *Chord* also provides two views of the errors, field based and object based. From their experiments the annotation overhead is very low with often one annotation needed per tens of thousands of lines of code.

The analysis is unsound and reports a few false races. The analysis of libraries can miss races due to the fact the harness synthesis does not account for all scenarios nor for all the methods. Out of the four main stages of the analysis, the three first are flow insensitive and the fourth stage lacks must-alias analysis when checking common locks of pairs of accesses. The effects of reflection and dynamic class loading are also ignored.

CoBE [14] In 2009, Vineet Kahlon et al showed their work for static analysis on C concurrent programs with asynchronous calls and locks. The data race detection method they devised builds a control flow graph and relies on PTA and may-happen-in-parallel analysis.

The method starts with the construction of the concurrent control flow graph of the program to be analyzed, using flow and context-sensitive PTA. This analysis uses bootstrapping to find the statements that affect the aliases of lock or function pointers and uses summarization to compute these aliases. The PTA is done together with the graph construction to resolve function pointers as they are encountered. The information of the graph is then used for data flow analysis to detect the accesses to shared variables. The lockset analysis uses the graph and the shared variable access locations to compute the locksets. Both data flow and lockset analyses are performed while the graph is being constructed to avoid the need for a separate computation phase.

With the shared variables identified, the data race detection algorithm creates a list of conservative data race warnings related with pairs of accesses from two different threads to a shared variable. To purge the lots of false alarms this list will contain, may-happen-in-parallel analysis is used. This analysis creates a set of call pairs using the may-happen-in-parallel rules. From these rules, two accesses may happen in parallel if their locksets are disjoint or if these accesses come from a parent thread and a forked child thread between the respective fork and join points. The set of call pairs is then used to filter the warnings that are related with each call pair.

The PTA computes aliases following the *Aliasing* theorem, stating that two pointers p and q are aliased at a control location l if there is a sequence of successive control locations λ starting at an entry point l_0 of the program and ending at l such that there is a third pointer a from which there are update sequences of all pointers from a to p and from a to q, both from l_0 to l along λ. The use of update sequences allows aliases to be summarized.

When the input program has recursive calls, the number of contexts becomes infinite, which is a problem for constructing the concurrent control flow graph. If a function call is found again, the algorithm only explores it if it can lead to new aliases for a lock, function or thread or if the exploration can lead to new locksets. To make this decision, the algorithm uses the knowledge of the *Finitization* theorem. This theorem works with must-locksets which are the intersection of locks held along all paths leading to a control location.

The theorem mentions that for a set of pointers and two contexts Con_1 and Con_2 of the same function call, if the aliases of every pointer from the Steensgaard closure *Cl(R)* and the must-lockset are the same at the location of the function call in both contexts and for any sequence of function calls going to a function h, if the contexts resulting from applying the chosen call sequence to Con_1 and Con_2 result in Con'_1 and Con'_2, both valid contexts, then the aliases of each pointer in *Cl(R)* and the must-lockset are the same at the location of h in Con'_1 and Con'_2. In this case, exploring the same function call again will not lead to any new alias.

The graph construction algorithm does not miss any data races. According to the *Soundness* theorem, given a must-lockset L at a location l in a valid context c, then there is a valid context c' in the constructed graph that the must-lockset at l in c' is L.

The method was implement on the CoBE Framework for analyzing concurrent C programs. Data races detected by the method are reported as warnings with the information on the affected shared variable. The false alarm rate is low. The bootstrapping employed by the PTA helps this method scale to large programs. The combination of the execution of multiple analyzes at the same time and the reduction on the set of contexts in the constructed graph decrease a lot the time taken by the analysis.

IteRace [27] In 2013, Cosmin Rădoi and Danny Dig. Cosmin presented three techniques to detect data races while supporting the new lambda-style parallel loops and the thread safety of the collections to be released in Java 8. The techniques are 2-Threads, Bubble-up and Filtering. They provide context-sensitivity for the construction of a call graph of the program.

The 2-Threads analysis allows the analysis method to be aware of the way the program manages the threads and of the data-flow structure of loop-parallel operations. It disambiguates identical threads by modeling them as different abstract threads. With the threads differentiated, the amount of shared objects is reduced, leading to less false positives. The technique matches threads operating on the same collection using may-alias. It also adds object sensitivity by tracking relevant collections in the program and modeling their elements as abstract fields so as to identify to which threads the elements belong. This modeling is also used for sequential loops. Potential data races are pairs of accesses from different threads to the same field of an object with at least one of those accesses being a write.

The Filtering technique filters race warnings taking in consideration the thread safety of the libraries being used. It assumes libraries are implemented correctly and uses their documented behavior to judge the correctness of the program. To that end, information about the thread safety of the method call is used. A method is considered thread safe when it cannot be involved in a data race. A thread safe method m may also be thread safe on closure if all the methods reachable from m are also not involved in data races. A method m is considered to only instantiate thread safe objects if any object it instantiates is thread safe and cannot be instantiated by methods called from m. A method circulates unsafe objects if it returns any possibly non thread safe object or receives it as a parameter.

Thread safe on closure methods get a *ThreadSafeOnClosure* flag. Methods may also receive the flag *Interesting* or *Uninteresting* according to the rules of the 2-Threads and Bubble-up techniques. These flags are also propagated to the methods called by the methods that have them. The warnings are filtered out when the called method is not thread safe, when the object does not instantiate thread safe objects or the context of the method call is not thread safe on closure.

The Bubble-up technique is used to avoid reporting races on library code and to instead report them on the code of the program using them. From a pair of conflicting accesses in library code, the call graph is traversed backwards until reaching the respective calls outside of the library code. The warnings on objects in the program code are

also grouped. This adds a layer of object sensitivity between the program and the libraries used for more precision.

Additionally, the method has a Synchronized phase where locksets are computed conservatively in a way similar to the *Chord* tool, but with the locks being represented as abstract objects. Locksets are used to determine if the program is properly synchronized. A pair of accesses is considered safe if the intersection of their locksets is not empty. Safe accesses are filtered twice, first on an initial set and then a second time after bubble-up.

The described technique was implemented on the *IteRace* tool. It takes as input a Java program with locks and reports the data races it finds. Warnings are reported as pairs of read/write accesses on fields of abstract objects. The programmer needs to give a specification of which classes are thread safe, thread safe on closure, which methods only instantiate thread safe objects and which circulate unsafe objects. The programmer should also give a specification of which classes belong to libraries used by the program for the Bubble-up technique. The authors claim the tool is "fast and precise enough to be practical" and that it scales to hundreds of thousands of lines of code, according to their evaluation.

When the classes and methods are correctly characterized for the filtering and the library classes are specified, out of the reported warnings, only a few are false alarms. However, the lack of information on library classes can greatly increase the amount of reported warnings and a wrong class characterization can the tool to miss data races. The filtering specification requirement can have a significant overhead on larger programs, rendering the tool less practical. Specifying the library classes should be easier for a programmer analyzing their own programs, but more problematic when it is a third party program.

The tool does not handle reflection and native method calls very well due to limitations on the WALA library [31]. The Synchronized phase of the analysis also uses may-alias information in an unsafe manner for must-alias lock relations, but the authors claim it can be adapted to do must-alias analysis if there is a scalable solution. This phase is the least effective on reducing warnings and the programmer can choose to deactivate it for safer results.

Another limitation is the fact *IteRace* only supports threads spawned by lambda-style parallel loops and will give "potentially unsafe thread spawn" warnings for threads spawned by other means. The authors do claim the tool can be extended to support other parallel constructs and the Bubble-up and Filtering techniques could be reused. The 2-Threads technique cannot be reused.

JCBMC [26] In 2015, Quoc-Sang Phan, Pasquale Malacaria and Corina S. Păsăreanu worked on a method based on symbolic execution. Their approach uses symbolic execution to translate a program with assertions into a disjunctive formula that encodes path conditions for each complete path explored. To make the explanation simpler, they defined the "guarded command" language with a grammar that supports statements and

```
1    {assume e | assert e | v = e | if e then goto s else goto s}
```

Listing 2.4: Example of statements written in the Guarded Command language

if-then-else constructs.

This language also includes the commands *assume* and *assert*, which encode properties of the program at the line of code where they are "invoked". A program is a set of states and an execution of the program, or trace, is a sequence of states. A trace can be seen as the set with the initial state in a conjunction with the conjunction of transition relations. Listing 2.4 shows an example of statements written in this language, where e is an expression, v is a variable and s is a statement.

This method starts with a form of bounded symbolic execution to translate the program, in which path conditions are not checked as soon as they are updated during the symbolic execution, but instead are checked when the symbolic execution reaches a specified bound. This way, more possible paths in a program are enumerated depending on the given bound. For the paths that are enumerated, a controller collects the path conditions that may lead to the violation of assertions in a formula in the disjunctive form. Once a certain amount of disjuncts is collected, they are sent to a worker thread for checking. If no path conditions violate assertions, no checking will be done. Here is an example of the formula.

$$\bigvee_{i=0}^{M} (pc_i \wedge \neg P|_{\sigma_i})$$

Each worker thread checks the satisfiability of its path conditions using constraint solving. If a thread finds a model that satisfies the path conditions, that means it found a path that leads to an error state in the program. In this case the execution stops and the verification of the program is deemed as failed. If no worker finds a path to an error state, the symbolic execution and the worker threads eventually terminate and the verification is deemed as successful. In the case no path condition checking is done, the verification is also deemed successful. The concurrency depends on the amount of works available and on the amount of disjunct path conditions sent to each.

The researchers implemented the developed method on the *JCBMC* tool. This tool takes as input a Java program with locks and assertions, along with a property to verify and three parameters, the search bound, the number of workers and the number of disjuncts per worker. The tool leverages off of the Symbolic PathFinder framework for the symbolic execution of the code and off of the Z3 solver for satisfiability checking. According to the tests, the tool performs better at finding a counterexample rather than ensuring their absence.

In programs where the enumeration of paths becomes expensive, the performance of the tool drops since the cost of generating path conditions is higher than the cost of solving them. For some of the tests, the tool ran out of memory. Both of these problems can be solved by replacing Symbolic PathFinder with a lighter weight tool with better memory

management. The authors do mention the tool does not run more than a thread at a time, which is also another aspect that should be improved. The overhead of the assertion and assumption annotations is unknown due to lack of information on the paper. Since this tool merely attempts to find only one counterexample to a given condition on a program, it also does not list every single problem with the program, but this could be changed with tweaks on the logic of the method.

RacerD [3] From late 2016 to 2018, Sam Blackshear et al investigated an approach to automatic static analysis of data races on Java programs. A method that reports true races quickly with low annotation overhead and can scale up to very large programs. Its implementation allowed Facebook engineers to convert the news feed from a sequential architecture to a concurrent one and flagged thousands of issues before they reached production during the background migration of the Litho library.

This method devised by the authors has the objective of making an over-approximation of all the heap accesses in the program. To do this, the analyzer computes syntactic access paths, locks, concurrency and ownership to create a set of access snapshots. For the locks, the analyzer simply counts the locks held on each access to be able to tell when an access is not protected by any locks and to avoid analysis errors on nested locks.

For the concurrency, the analyzer looks for evidence of the access being able to be done in a concurrent context, such as usage of locks, usage of java.util.concurrent annotations and the use of Android utility functions like *assertMainThread()* and *assertOnBackgroundThread()*. With this evidence the analyzer classifies the access as *NoThread* if it does not run while another thread is running, *AnyThreadButMain* if it only runs on the main thread, or *AnyThread* if it runs while another thread is running.

For the ownership, the analyzer uses ownership analysis to assign abstract ownership values to an access path. This analysis leverages the fact an access path assigned to a new memory allocation is owned and propagates the ownership through the assignments. The ownership of a value can be *Unowned* if it has no ownership, *OwnedIf({1,2})* if the value is only owned when the first and second parameters are owned at the call site or *OwnedIf(∅)* if the value is owned, this last value can be shortened to *Owned*. Initially local and global variables have the *Unowned* value and the *i*th formal parameter is bound to *OwnedIf({i})*. The argument *this* is assumed to be Owned in constructors.

With the computed access snapshots, the analyzer then computes a summary for each method, recording potential read or write accesses to a variable with information saying if these are protected by locks or performed by a single thread. During analysis, summaries of calls from the non private methods of each class are grouped into a set of access snapshots. When the summary of a requested method is not cached, it is requested on the fly.

Given a pair of access snapshots, the analyzer finds out if the accesses touch the same address and if both can happen concurrently. If the two access paths are syntactically equal, it is reasonable to assume they refer to the same address. If both accesses are not

```
@ThreadSafe class ImmutableData {
    private final int mData;
    public ImmutableData(int data) { this.mData = data; }
    int getData() { return mData; }
}
```

Listing 2.5: Example use of the @ThreadSafe annotation

```
private Object mlock = new Object();

@GuardedBy("mLock")
private int a;
```

Listing 2.6: Example use of the @GuardedBy annotation

owned by the same thread, at least one is not protected by a lock and at least one runs in a thread different from the main thread, then both accesses can happen concurrently. And if at least one of the accesses is a write, the analysis considers this pair of accesses can cause a data race.

The described method is implemented on the *RacerD* tool using the Infer analysis framework. The objective of this race detector tool was the incremental analysis of Java programs, however support was later extended to C, C++, C# and Objective-C. It takes as input a program in any of these languages and outputs warnings for detected data races with information on the type of violation and the traces of their conflicting accesses. While the tool does not require any annotations to start analyzing a program, in the case of Java programs, it does look for specific annotations such as those in the java.util.concurrent package. These annotations can be applied to variables, methods, classes or interfaces.

Listing 2.5 shows an example of a class annotated with the *@ThreadSafe* annotation, which tells *RacerD* that objects of this class are safe to be used in threads.

Listing 2.6 shows an example of a variable annotated with *@GuardedBy*, which was taken from the GitHub repository [9] of the implementation. This annotation tells *RacerD* that the variable is guarded by the declared lock of the same name.

For *RacerD* to be deployed at Facebook as part of the continuous integration system it needed to do compositional analysis, reason sequentially about memory accesses, locks and threads using the employed synchronization and memory accesses and report races between syntactically identical paths, while being fast and efficient. Compositional analysis allows the tool to give the same results by analyzing either the entire program or separated parts of it. Whole-program analysis, exploration of interleavings and PTA were discouraged, since the tool needed to be fast and to avoid reporting false positives as much as possible.

RacerD is not very fast when analyzing a large code base at once for the first time, but the compositional analysis allows it to quickly analyze portions of the code and changes to that code. For the Java language, the amount of missed races can be higher when the code is not annotated, however the annotation overhead is low. In this case, *RacerD* also

relies on existing annotations of the language, which are more likely to be known by the developers. The authors only know 3 cases of the tool missing a race during the changes on the Litho library, which happened due to implementation issues that were resolved afterwards. No false negatives due to design issues had been reported in production when this paper was published.

2.4.1 Discussion

Table 2.1 presents a summary of information about the static analysis tools presented in section 2.4. Column one displays the name and references of the tools. Column two presents the language(s) these tools support. The third column presents the type of analysis. The fourth column shows a list of the methods and techniques used by the tools. Column five says if the tools support separate compilation. Column six displays the guarantees of these tools. The seventh column shows the details of the output of the tools. The last column says if there is an implementation available. Some of the references unfortunately do not have details about the guarantees given by some of the tools, while other references do not disclose many details on the tools output.

In general, static analysis tools use variations of the lockset algorithm or graph based algorithms. To resolve aliases, the tools use PTA, which is essential for a precise data race detection. Lack of PTA can lead to missed data races as seen in with *RacerX* [8]. The analysis on most tools is not compositional, requiring the entire program for analysis.

Out of all the analyzed tools, only two have data race detection algorithms with proof of soundness, *RCCJava* [1] and *CoBE*. However, the type system used in *RCCJava* is too restrictive when applied to the Java language and requires analysis escape techniques that can introduce false positives. The algorithm based on the construction of a control flow graph from *CoBE* looked more promising and we took some inspiration from it for our work.

Since lack of PTA leads to false negatives, we incorporated it in our solution. Whole-program analysis could also help to ensure no data races are missed, but it is expensive. On the other hand, if our analysis was compositional it could support separate compilation and scale better, so we also want our analysis to be compositional similar to what was implemented in *RacerD* [3].

Table 2.1: Analysis Tools Summary.

Tools	Language Analyzed	Methods and Techniques	Separate Compilation	Guarantees	Output	Implementation Available
RCCJava Houdini/rcc [11, 10, 1]	Java + annotations RaceFreeJava	Type Based Analysis Analysis Escape Mechanisms Annotation Inference	No	Race-Freedom	Errors contain relevant code location Errors show lock usage mistakes Has a few false positives Has few false negatives	No
Calvin [12, 13]	Java + annotations Plato	Thread-modular reasoning Assume-guarantee decomposition Procedure-modular reasoning Thread abstraction Automatic theorem proving	No	Thread-Modular Verification Invariant Verification Modular Verification	Theorem proving timeouts are reported Output messages are errors or warnings Messages show invariant violation Messages show environment assumption violation Messages show abstraction step violation Messages show assertion that does not hold	No
RacerX [8]	C + annotations	Lockset Algorithm Flow-sensitive Interprocedural Analysis Unlockset Analysis Belief Analysis Simple Function Pointer Resolution Statistical Inference of Locking Functions	No		Errors report either a race condition or a deadlock Errors contain the smallest backtrace Errors ranked from most to least severe Has false positives Has false negatives	No
Chord [20]	Java + annotations	K-object sensitivity K-object sensitive PTA Thread-escape analysis Lock analysis	No		Each error has a pair of conflicting calls Each pair is shown on a graph Each error has a field-based view and an object based view Has a few false positives	No
CoBE [14]	C	Flow & context-sensitive PTA with Bootstrapping Context-Sensitive Lockset Analysis Thread order analysis	No	Aliasing Theorem Finitization Soundness	Warnings report data races Warnings contain the affected shared variable	No
IteRace [27]	Java + filtering specifications + bubble-up specification	2-Threads Technique Filtering Technique Bubble-up Technique Pointer Analysis	No		Sets of warnings on fields of abstract objects reported Each warning is a read or write access to the field A few false positives False negatives depend on specifications given for the filtering	Yes
JCBMC [26]	Java + annotations Guarded Command	Symbolic Model Execution	No		Errors show assertion violations	No
RacerD [3]	C C++ Objective-C C# Java	Compositional analysis Full call stack report Grouping races	Yes	Compositional	Warnings show violations in methods Violations can be lock misuse Violations can be potential data races Violations can be use of non threadsafe interfaces from threadsafe code Races are reported between conflicting accesses Has a few false positives Has false negatives	Yes

3

Solution

In this chapter we will explain our solution for the problem mentioned at the end of Section 1.2. In Section 3.1 we give an overview of the methodologies we used to design and implement our solution. Then in Section 3.2 we describe the algorithm used and its limitations. Finally in section 3.3, we show how we implemented our solution.

3.1 Solution Methodology

The main goal of this work is a tool that takes as input a Java program with no concurrency control and detects which fields have access conflicts that cause data races. The intention is to provide the information needed by the programmer to know where concurrency control is needed.

For this tool, we developed an algorithm that we describe in detail in Section 3.2. The algorithm can gather all the field accesses in a program using Points-to Analysis (PTA) and use the information to detect data races. The algorithm is context-sensitive and provides a compositional analysis. We also show an algorithm for the detection of data races in 3.2.3. In section 3.3 we explain how we implemented this solution.

3.2 The Algorithm

The algorithm has to perform 3 main tasks: resolve the fields and the field accesses, resolve the method call results and decide which accesses constitute a possible data race. The algorithm is mainly composed of a PTA algorithm that will perform an intra-method analysis, followed by an inter-method analysis and a data race algorithm to detect the data races. The intra-method analysis is explained in Section 3.2.1 and the inter-method analysis is covered in Section 3.2.2. The data race detection is explained in Section 3.2.3.

3.2.1 Points-to Analysis

To explain how the algorithm functions, we must first explain how the different elements of a program are represented. We assume that the input program is written in an imperative object-oriented language and that we will receive an abstract representation of the code to analyze. When the PTA runs through the input program, it will resolve the resources of that program.

There are different types of resources: constants, objects, arrays, fields, parameters and method call results. An object represents the result of an object creation statement. Some resources might be associated with more than a single value. Resources that can have multiple different values are represented by a resource list.

Not all resources can be resolved by an intra-method analysis and will require the inter-method analysis of Section 3.2.2. These resources are called unresolved resources. Method call results are sub-types of unresolved resources, because they require inter-method analysis and because they contain information that differs from that of standard unresolved resources.

Some resources cannot be resolved because the source code of their types is missing. They differ from unresolved resources for this reason and are classified as undetermined resources.

No matter their resource type, resources always contain information, such as the type of what they represent, their visibility in the program and a list of methods that have accessed the resource. The methods list is empty before the PTA algorithm starts analyzing the methods of the class.

Each resource type also has specific information. The constant resources contain the value they represent. Object resources have a map of fields indexed by their identifiers. Array resources have a map of values indexed by their index position on the array. Resource lists have list of resources. Field resources have a resource list with the possible values assigned, their identifier and their parent resource. Parameter resources store their index in a method declaration and the resource of their argument which is not initialized at the beginning of the execution of the algorithm.

Unresolved resources have the information necessary for a posterior resolution of a statement or an expression that will result in another resource. This information includes a list of dependencies that need to be resolved beforehand, the condition that needs to be true for this resource to be resolved, a lambda expression and a list of dependents. The lambda expression contains a function to be executed and the arguments. In order to resolve an unresolved resource, prior to the execution of the lambda, the resources must be updated after the dependencies of the unresolved are resolved.

The method call result resources contain the information needed for a posterior resolution of a method call expression that results in another resource and/or may produce side effects. Like the typical unresolved resources, method call results contain a list of

dependencies and dependents, and a condition. However, instead of a lambda expression, this type of resources stores the called method, the resource of the target and the resources of the arguments.

The list of dependencies of an unresolved resource can contain either unresolved resources or parameter resources and is never empty, unless the unresolved resource is a method call result that does not depend on any other unresolved resources. The condition may or may not be initialized depending on the location of the resource and the list of dependents may or may not be empty.

Undetermined resources store a list of resources that, at one point, were involved in operations or assignments with the given resource. The purpose of these resources is to at least provide enough information on possible conflicts even when the original object cannot be resolved.

Before the PTA algorithm is executed, the input program is parsed into a more abstract representation structured according to the grammar shown in table 3.1. Let $Types$, ranged over by t, be the set of all types known to the program. Consider $CSTypes$, ranged over by c, as the set of conditional statement types such as `if` and `else` for `if` and `else` statements respectively and $WSTypes$, ranged over by w, as the set of while statement types such as `while` and `do-while` for `while` and `do-while` statements respectively.

Let $Vars$, ranged over by x, be the set of variable identifiers, let $MethodNames$, ranged over by m, be the set of method identifiers and let Vis, ranged over by z, be the set of identifier visibility such as `private`, `public` and `package-private` in Java. Let u be a unary operator, such as $-, !, ++, ...$, let b be a binary operator, such as $+, -, *, /, \&, |, ==, !=, \&\&, ||, ...$ and let v be a primitive value. Lastly $\hat{\alpha}$ denotes a sequence of elements from the category α.

In the grammar, method call expressions are considered to be the calls at the right of an assignment statement. Method call statements are the calls that are not part of an assignment statement, albeit these are handled by the algorithm in a way similar to method call expressions as will be shown in this section. In a `for` statement, the first $\hat{S}$ contains the statements at the head of the `for` statement, the 2 expressions represent the condition and unary expression respectively and the last $\hat{S}$ contains the statements inside the block of the `for` statement.

For a better understanding of the PTA algorithm, a specific struct of the *dataStruct* type is used throughout most of the functions. A *dataStruct* contains, a map of classes indexed by their type, a map of local variables indexed by their identifier, a map of local variable maps indexed by methods, a map of parameters indexed by their identifiers, a list of classes analyzed, a list of method call results, the stack of classes being visited, the stack of methods being visited, the stack of the current object resources and the stack of the current condition resources. The object resource at the top of the object stack is considered the current "this" resource.

To keep the algorithm simpler, many auxiliary functions are used. The tables 3.2, 3.3 and 3.4 show the functions that we did not write pseudo-code for. It also shows the return

Table 3.1: Abstract Program Representation Rules

$P := \hat{C}$	Program
$C := (\hat{F}, \hat{M})$	Class
$F := (z, D)$	Field
$D := (t, x, E) \mid (t, x)$	Variable declaration
$M := (z, t, m, \hat{D}, \hat{S})$	Method declaration (including constructors)
$S := D$	Local variable declaration
$\mid (x, E) \mid (E, x, E)$	Assignment on var x or field $E.x$
$\mid \hat{S}$	Block statement
$\mid (c, E, \hat{S})$	Conditional (If or Else) statement
$\mid (w, E, \hat{S})$	While statement
$\mid (\hat{S}, E, E, \hat{S})$	For statement
$\mid$ **ret** $E \mid$ **ret**	Return statement
$\mid$ **break**	Break Statement
$\mid$ **continue**	Continue Statement
$\mid (E, m, \hat{E})$	Method call $E.m(\hat{E})$ (including calls to constructors)
$E := (u, E)$	Unary operation
$\mid (b, E, E)$	Binary operation
$\mid (E, E, E)$	Conditional operation
$\mid (E, m, \hat{E})$	Method call $E.m(\hat{E})$ (including calls to constructors)
$\mid (E, x)$	Field selection $E.x$
$\mid (t, E)$	Cast
$\mid v$	Primitive value
$\mid t$	Type
$\mid$ **this** $\mid$ **super**	Special objects

Where $t \in Types; c \in CSTypes; w \in WSTypes; x \in Vars; m \in MethodNames; z \in Vis$.

type, arguments and their types and a description of the role for each function.

The algorithm is outlined in Algorithm 1. The *p2a* function requires the Java classes, along with the relevant class type to start the analysis on. The *typeOf* function used can take either a class, statement or expression and returns their corresponding type. The algorithm starts by indexing each class in the *CM* map using their types at line 5 and discovering the *startClass* to start the analysis on at line 6.

The *startClass* is then visited at line 11 and yields the object resource that represents it, a list of method call result resources, the classes that were analyzed, the map with all the field resources indexed and a map of local variable maps indexed by methods. The results of this analysis are stored in the *p2aData* struct, which may contain unresolved resources that are solved at line 12 using the *resolve* function described in section 3.2.2. Once the PTA algorithm resolved everything it can, it returns the *p2aData* struct.

The function *visitClass* contains the process of visiting a class. The function *getThisResource* takes a class and returns its object resource. The *leftExpression* function takes a statement and returns the expression at the left side. The *identifierOf* function takes an

Table 3.2: Auxiliary Algorithm Functions (part 1)

Function → Return Type	Argument : Type	Description
getClass → class	p2aData : dataStruct type : type	Retrieves the class of the given *type* from the *p2aData* struct.
methodsOf → methodsList	cls : class	Retrieves the list of methods from the class *cls*.
signatureOf → string	m : method	Retrieves the signature of the method *m*.
getClassResource → resource	p2aData : dataStruct type : type	Retrieves the resource of the class of the given *type* from the *p2aData* struct.
getCurrentObject → objectResource	p2aData : dataStruct	Retrieves the object resource on the top of the object stack stored in the *p2aData* struct.
getFieldsMap → fieldMap	p2aData : dataStruct	Retrieves the map of fields of the "this" object resource stored in the *p2aData* struct.
getVarsMap → localVarMap	p2aData : dataStruct	Retrieves the map of local variables stored in the *p2aData* struct.
updateVarsMap	p2aData : dataStruct L : localVarMap	Updates the local variables map in the *p2aData* struct using the data in the *L* map.
getMethodVarMaps → methodVarMap	p2aData : dataStruct	Retrieves the map of local variable maps indexed by methods stored in the *p2aData* struct.
updateMethodVarMaps	p2aData : dataStruct ML : methodVarMap	Updates the map of local variable maps in the *p2aData* struct using the data stored in the *ML* map.
get → any	M : anyMap key : any	Retrieves the value indexed by *key* in the *M* map.
getField → fieldResource	p2aData : dataStruct name : string	Retrieves the field "this.name" from the *p2aData* struct.
getVar → resourceList	p2aData : dataStruct name : string	Retrieves the resource list corresponding with the local variable of the given *name* from the *p2aData* struct.
getParam → paramResource	p2aData : dataStruct name : string	Retrieves the parameter resource with the given *name* from the *p2aData* struct.
isAnalyzed → boolean	p2aData : dataStruct cls : class	Checks if the class *cls* in the *p2aData* struct was already analyzed.
getMCalls → mcallsList	p2aData : dataStruct	Retrieves the list of method call results stored in the *p2aData* struct.
updateMCalls	p2aData : dataStruct mcalls : mcallsList	Stores the method call results in *mcalls* in the *p2aData* struct.
enterClass	cls : class p2aData : dataStruct	Initializes the relevant data in the *p2aData* struct according to the *cls* class that will be visited.
leaveClass	p2aData : dataStruct	Initializes the relevant data in the *p2aData* struct according to the class being previously visited.
getCurrentMethod → method	p2aData : dataStruct	Returns the method stored in the *p2aData* struct that is being currently visited.
enterMethod	p2aData : dataStruct m : method	Initializes the relevant data in the *p2aData* struct according to the method *m* that will be visited.
leaveMethod	p2aData : dataStruct	Resets the relevant data in the *p2aData* struct related with the method that was visited.
getCurrentCondition → boolean	p2aData : dataStruct	Return the condition of the block being currently visited stored in the *p2aData* struct.
pushCondition	p2aData : dataStruct blockCondition : boolean	Pushes a new *blockCondition* condition into the condition stack of the *p2aData* struct.
popCondition	p2aData : dataStruct	Pops the condition at the top of the condition stack of the *p2aData* struct.
conditionExpression → boolean	b : block	Retrieves the expression of the *b* block.
blockBody → block	b : block	Retrieves the body of the *b* block.
copyOf → any	o : any	Retrieves a copy of the *o* object.
isEquivalent → boolean	data1 : dataStruct data2 : dataStruct	Checks if the data in the structs *data1* and *data2* is equivalent.

Table 3.3: Auxiliary Algorithm Functions (part 2)

Function → Return Type	Argument : Type	Description
leftExpression → expression	s : statement	Retrieves the expression from *s* on the left side of the "=" sign.
rightExpression → expression	s : statement	Retrieves the expression from *s* on the right side of the "=" sign.
typeOf → type	e : expression	Retrieves the type of the field/variable from the expression *e*.
targetOf → resource	e : expression	Retrieves the target of the field/variable from the expression *e*.
identifierOf → string	e : expression	Retrieves the identifier of the field/variable from the expression *e*.
visibilityOf → visibility	e : expression	Retrieves the visibility of the field/variable from the expression *e*.
eval → resource	e : expression	Evaluates the expression *e* and returns the corresponding resource.
getInitExpressions → expList	e : expression	Returns the expressions inside of the array initialization expression *e*.
argsOf → resourceList	mc : methodCall	Retrieves the list of argument resources from the method call *mc*.
methodSignatureOf → string	mc : methodCall	Retrieves the method signature from the method call *mc*.
operatorOf → operator	e : expression	Retrieves the operator from the infix expression *e*.
conditionOf → expression	c : conditional	Retrieves the condition expression from the conditional expression *c*.
thenExpression → expression	c : conditional	Retrieves the left expression from the conditional expression *c*.
elseExpression → expression	c : conditional	Retrieves the right expression from the conditional expression *c*.
getCondition → resource	cs : conditionalStruct	Retrieves the resource of the condition from the conditional struct *cs*.
getThenExpression → resource	cs : conditionalStruct	Retrieves the left expression from the conditional struct *cs*.
getElseExpression → resource	cs : conditionalStruct	Retrieves the right expression from the conditional struct *cs*.
getExpression → resource	cs : conditionalStruct	Retrieves the entire conditional expression from the conditional struct *cs*.
getValue → resource	cr : constantResource	Retrieves the value of the constant resource *c*.
getObjectFields → fieldList	o : objectResource	Retrieves the field resources of the object resource *o*.
getObjectField → fieldResource	o : objectResource name : string	Retrieves the field resource of the object resource *o* with the identifier *name*.
getFieldIdentifier → string	f : fieldResource	Retrieves the identifier of the field resource *f*.
getFieldValues → resourceList	f : fieldResource	Retrieves the resource list with the values of the field resource *f*.
getFieldParent → objectResource	f : fieldResource	Retrieves the object resource where the field resource *f* is stored.
addData	arr : arrayResource i : int r : resource	Adds the resource *r* to the array resource arr at index *i*.

Table 3.4: Auxiliary Algorithm Functions (part 3)

Function → Return Type	Argument : Type	Description
getResources → resList	rl : resourceList	Retrieves the list of resources stored in the resource list *rl*.
addResources	rl : resourceList r : resource	Adds the resource *r* or the resources stored in *r* to the resource list *rl*.
addFieldValue	f : fieldResource r : resource	Adds the resource *r* as a value of the field resource *f*.
addDependents	u : unresolved r : resource	Adds the resource *r* or the resources stored in *r* as dependents to the unresolved resource *u*.
getParamArg → resource	p : paramResource	Retrieves the resource of the argument of the parameter resource *p*.
getType → type	expData : expStruct	Retrieves the type stored in the *expData* expression struct.
getTarget → resource	expData : expStruct	Retrieves the target stored in the *expData* expression struct.
getName → string	expData : expStruct	Retrieves the identifier stored in the *expData* expression struct.
getVisibility → visibility	expData : expStruct	Retrieves the visibility stored in the *expData* expression struct.
setResource	r1 : resource r2 : resource	Replaces the values in resource *r1* with the resource *r2*.
getMethodAccesses → accessesList	m : method	Retrieves the list of accesses done by the method *m*.
getMethodVisibility → visibility	m : method	Retrieves the visibility of the method *m*.
updateMethodAccesses	m : method al : accessesList	Updates the list of accesses of the method *m* with the accesses in the access list *al*.
addMethodAccessedResources	m : method rl : resList	Adds the resource in the list *rl* to the list of resources accessed by method *m*.
addMethodThatAccess	r : resource m : method	Adds the method *m* to the list of methods that access the resource *r*.
updateAccess	al : accessesList a : access	Update the access list *al* with the access *a*.
addAccessType	a : access at : accessType	Adds an access type *at*, which can be a read or write access type, to the access *a*.
leftOperandOf → expression	e : infixExpression	Retrieves the operand expression at the left of the infix expression *e*.
rightOperandOf → expression	e : infixExpression	Retrieves the operand expression at the right of the infix expression *e*.
binaryOperation → resource	op : operator left : resource right : resource	Perfoms a binary operation between resources *left* and *right* according with the operator *op*.
operandOf → expression	e : expression	Retrieves the operand expression of the expression *e*.
unaryOperation	op : operator r : resource p2aData : dataStruct operand : expression	Performs a unary operation with the resource *r* according with the operator *op* and, if necessary, changes the resource of the *operand* expression stored in the *p2aData* struct.
getResourceVisiblity → visibility	r : resource	Retrieves the visibility of the resource *r*.
getMethodReturnType → type	m : method	Retrieves the return type of the method *m*.
getCurrentStatementType → type	p2aData : dataStruct	Retrieves the type of the statement being currently analyzed from the *p2aData* struct.
getMethodSignature → string	m : method	Retrieves the method signature from the method *m*.
getPreviousIfCondition → resource	p2aData : dataStruct	Retrieves the condition resource of the previous if condition stored in the *p2aData* struct.
invert → resource	r : resource	Inverts the value of the condition resource *r*.

Algorithm 1 PTA algorithm

```
1:  function P2A(classes : classList, startType : type)
2:      CM ← {}                                          ▷ maps classes to their types
3:      for cls ∈ classes do
4:          type ← TYPEOF(cls)
5:          CM ← CM ∪ {(type, cls)}
6:          if type = startType then
7:              startClass ← cls
8:          end if
9:      end for

10:     p2aData ← (CM)
11:     VISITCLASS(p2aData, startClass)
12:     RESOLVE(p2aData)

13:     return p2aData
14: end function

15: function VISITCLASS(p2aData : dataStruct, currentClass : class)
16:     ENTERCLASS(currentClass, p2aData)

17:     for f ∈ FIELDDECLARATIONS(currentClass) do
18:         r ← VISITFIELD(p2aData, f)
19:         left ← LEFTEXPRESSION(f)
20:     end for

21:     ML ← GETMETHODVARMAPS(p2aData)          ▷ maps methods to local variable maps
22:     for m ∈ METHODSOF(class) do
23:         L ← VISITMETHOD(p2aData, m)          ▷ maps local variable to resources
24:         ML ← ML ∪ {(m, L)}
25:         UPDATEMETHODVARMAPS(p2aData, ML)
26:     end for

27:     LEAVECLASS(p2aData)
28: end function
```

expression as argument and returns the identifier within it. The function *fieldDeclarations* takes a class and returns its field declarations and the *methodsOf* function takes a class and returns its methods.

When visiting a class, the *enterClass* function is called at line 16 to initialize the relevant values in the *p2aData* struct. The field declarations in the *currentClass* are accessed via the *fieldDeclarations* function and each field and method declaration of the class is then visited at line 18 to collect the field resources and the local variable resources into the fields map of the current object resource of *p2aData*.

To prepare for visiting the class methods, the method variable maps of *p2aData* is accessed and the methods of the current class are accessed with the *methodsOf* function. Then each method is visited at line 21. The local variables are indexed by their name in the L map returned by the *visitMethod* function and the variable maps are indexed by the methods in the ML map accessed with the *getMethodVarMaps* function and updated by the *updateMethodVarMaps* function.

Algorithm 2 pictures the process of visiting field declarations. The *visitField* function starts by first retrieving the expression at the left side of f using the *leftExpression* function

Algorithm 2 Visiting Fields

```
1: function visitField(p2aData : dataStruct, f : fieldStatement)
2:     left ← leftExpression(f)
3:     name ← identifierOf(left)
4:     if f is assignment then
5:         right ← rightExpression(f)
6:         fieldResource ← visitE(p2aData, right)
7:         updateField(data, name, fieldResource)
8:     end if

9:     F ← getFieldsMap(p2aData)                ▷ maps field names to field resources
10:    r ← get(F, name)
11:    return r
12: end function
```

Algorithm 3 Visiting Methods

```
1: function visitMethod(p2aData : dataStruct, m : method)
2:     enterMethod(p2aData, m)
3:     for s ∈ statementsOf(m) do
4:         if s is block then
5:             L ← L ∪ visitBlock(p2aData, s)
6:         else
7:             visitS(p2aData, s)
8:         end if
9:     end for

10:    leaveMethod(p2aData)
11:    caller ← getCurrentMethod(p2aData)
12:    v ← getMethodVisibility(m)
13:    if m is constructorMethod or v = private then
14:        accesses ← getMethodAccesses(m)
15:        accessedResources ← getMethodAccessedResources(m)
16:        updateMethodAccesses(caller, accesses)
17:        addMethodAccessedResources(caller, accessedResources)
18:        callerVisibility ← getMethodVisibility(caller)
19:        if caller not constructorMethod or callerVisibility not= private then
20:            for f ∈ accessedResources do
21:                addMethodThatAccess(caller, f)
22:            end for
23:        end if
24:    end if

25:    return L
26: end function
```

Algorithm 4 Visiting Code Blocks

```
 1: function visitBlock(p2aData : dataStruct, b : block)
 2:     condition ← getCurrentCondition(p2aData)
 3:     if not mayBeTrue(condition) then
 4:         return {}
 5:     end if

 6:     L ← getVarsMap(p2aData)
 7:     if b is ifBlock then
 8:         L ← L ∪ visitIfBlock(p2aData, b)
 9:     else if b is elseBlock then
10:         L ← L ∪ visitElseBlock(p2aData, b)
11:     else if b is whileLoop then
12:         L ← L ∪ visitWhileLoop(p2aData, b)
13:     else if b is doWhileLoop then
14:         L ← L ∪ visitDoWhileLoop(p2aData, b)
15:     else if b is forLoop then
16:         L ← L ∪ visitForLoop(p2aData, b)
17:     else
18:         enterBlock(p2aData, b)
19:         for s ∈ statementsOf(b) do
20:             if s is block then
21:                 L ← L ∪ visitBlock(p2aData, s)
22:                 updateVarsMap(p2aData, L)
23:             else
24:                 visitS(p2aData, s)
25:             end if
26:         end for
27:         leaveBlock(p2aData)
28:     end if

29:     updateVarsMap(p2aData, L)
30:     return L
31: end function
```

at line 2 and then uses the *identifierOf* function with *left* as an argument to get the identifier in the *left* expression.

Afterwards, the *visitField* function checks whether f is an assignment or not. In the case of f being an assignment, the expression on the right is visited at line 6 to retrieve its corresponding resource to be stored in the fields map of the current object in *p2aData*.

The *visitMethod* function in Algorithm 3 uses the function *enterMethod* to initialize the necessary values in the *p2aData* struct in preparation for the visit to the method m. At line 3 it accesses all the statements in the body of m through the *statementsOf* function and visits them at line 7.

After visiting every statement of the method m, the function leaves the method at line 10. Then the visibility of m is accessed using the *getMethodVisibility* function. If m is a constructor method or is a private method, its accesses and accessed resources are stored in the *caller* at lines 16 and 17. If *caller* is also not a constructor nor a private method, it is added to the list of acessing methods of every field resource accessed by m at line 21.

Algorithms 4 to 7 show how all the statements are analyzed and Algorithm 8 shows

Algorithm 5 Visiting If-Else Blocks

```
1: function visitIfBlock(p2aData : dataStruct, b : ifBlock)
2:     ce ← conditionExpression(b)
3:     blockCondition ← visitE(p2aData, ce)
4:     L ← getVarsMap(p2aData)
5:     pushCondition(p2aData, blockCondition)
6:     body ← blockBody(b)
7:     L ← L ∪ visitBlock(p2aData, body)
8:     popCondition(p2aData)
9:     return L
10: end function

11: function visitElseBlock(p2aData : dataStruct, b : elseBlock)
12:     blockCondition ← getPreviousIfCondition(p2aData)
13:     blockCondition ← invert(blockCondition)
14:     L ← getVarsMap(p2aData)
15:     pushCondition(p2aData, blockCondition)
16:     body ← blockBody(b)
17:     L ← L ∪ visitBlock(p2aData, body)
18:     popCondition(p2aData)
19:     return L
20: end function
```

how expressions are analyzed. Visiting each block statement is a process shown in Algorithm 4. Before proceeding, the function *visitBlock* checks if the current condition related to the block can possibly be true and in the case it is surely false, an empty map is returned. This confers context-sensitivity to the whole algorithm by making it not visit statements when the condition of the block in which they reside is known to be false. On the other hand, if the condition can be true, the function updates the local vars map at line 22 before visiting the block.

Before visiting the block, the *visitBlock* function checks whether the block *b* is a if block, an else block, a while loop block, a do-while loop block or a for loop block and updates the local variables map with the result of calling the functions in the Algorithms 5, 6. The *visitDoWhileLoop* and *visitForLoop* functions are shown in annex I. If *b* is just a simple block, the *visitBlock* function enters the block at line 18 and accesses its fields using the *statementsOf* function at line 19. Each statement *s* is then visited at lines 21 or 24 and the function leaves the block at line 27. At the end of the *visitBlock* function, there is one final update of the local variables in *p2aData* at line 29.

Before moving to the next few algorithms, we will explain some of the functions that are used. The *blockBody* function takes a block as argument and returns its body. The function *copyOf* takes an argument of any type and returns a copy of it. The *isEquivalent* function takes 2 dataStruct arguments, and returns true if both structs have the same variables and fields initialized and if the values of fields and variables are still the same.

For if/else blocks, the *visitIfBlock* and *visitElseBlock* functions in Algorithm 6 are used. The *visitIfBlock* function starts by retrieving the expression of the condition at the head of the block *b* using the *conditionExpression* function at line 2 and then visits that expression. The result of visiting the expression is stored on *blockCondition*.

Algorithm 6 Visiting While Loop Blocks

```
1: function visitWhileLoop(p2aData : dataStruct, b : whileLoop)
2:     ce ← conditionExpression(b)
3:     blockCondition ← visitE(p2aData, ce)
4:     L ← getVarsMap(p2aData)
5:     body ← blockBody(b)
6:     pushCondition(p2aData, blockCondition)
7:     oldData ← copyOf(p2aData)
8:     doLoop ← mayBeTrue(blockCondition)
9:     i ← 0
10:    while doLoop and i < 10 do
11:        L ← L ∪ visitBlock(p2aData, body)
12:        if isEquivalent(p2aData, oldData) then
13:            i ← i + 1
14:        else
15:            oldData ← copyOf(p2aData)
16:            blockCondition ← visitE(p2aData, ce)
17:            popCondition(p2aData)
18:            pushCondition(p2aData, blockCondition)
19:            doLoop ← mayBeTrue(blockCondition)
20:            i ← 0
21:        end if
22:    end while
23:    popCondition(p2aData)
24:    return L
25: end function
```

Next, the map of local variables is retrieved from *p2aData* to be stored in L and the *blockCondition* is pushed onto the top of the conditions stack of *p2aData* in preparation for visiting the if-else block. The body of b is then accessed and visited at line 7. The new local variables from the block are stored in L and the condition of the block is popped out of the conditions stack of *p2aData*. Finally, the var map L is returned.

The *visitElseBlock* function works similarly to the previous function, but since an else block has no condition expression, it uses the *getPreviousIfCondition* at line 12 to get the resource of the condition from the previous if block stored in the *p2aData* struct. Then the *invert* function is used on the *blockCondition* at line 13 to invert the value of the condition. After getting the condition of the else block b, the function performs the same steps is the *visitIfBlock* function.

While loop blocks are visited by the function *visitWhileLoop* of Algorithm 6. Like the previous function, it starts by getting the expression of the condition at the head of the block b, so it can then visit the expression and get the condition resource. The process only differs between the first *pushCondition* function call and the last *popCondition* function call.

As part of the preparation to visit the body of the while loop block multiple times, this function creates a copy of *p2aData* which it calls *oldData*, it also initializes the *doLoop* with the result of calling *mayBeTrue* with the *blockCondition* and it sets i to 0. During the while loop, the body of the block is visited at line 11 and then the function checks if the data in *p2aData* is equivalent to the data in *oldData*.

If the data in both structs is equivalent, that means the data did not actually change at all and the i counter is therefore increased by 1 at line 13 for each loop, until it reaches

Algorithm 7 Visiting Statements

```
1: function visitS(p2aData : dataStruct, s : statement)
2:     condition ← currentCondition(p2aData)
3:     if mayBeTrue(condition) then
4:         right ← rightExpression(s)
5:         r ← visitE(p2aData, right)
6:         if s is returnStatement then
7:             currentMethod ← getCurrentMethod(p2aData)
8:             setReturnResource(currentMethod, r)
9:         else if s is mcallStatement then
10:            return
11:        else
12:            left ← leftExpression(s)
13:            type ← typeOf(left)
14:            target ← targetOf(left)
15:            name ← identifierOf(left)
16:            visibility ← visibilityOf(left)
17:            expData ← (type, target, name, visibility)
18:            changeResource(p2aData, expData, r)
19:        end if
20:    end if
21: end function
```

10 or until the resource values of *p2aData* change. When the values in *p2aData* change, the *oldData* struct is updated with a new copy of *p2aData* at line 15, the block condition is updated and used to update the top of the *p2aData* condition stack and the *doLoop* boolean and i is set back to 0.

Non block statements in methods are evaluated using the *visitS* function in Algorithm 7. After checking if the *condition* may be true, the function evaluates the expression at the right of the statement s at line 5. Then it checks whether the statement s is a return statement, a method call statement or another type of statement.

If it is a return statement, the function accesses the current method and sets its return resource as the evaluated resource r using the *setReturnResource* function at line 8. If s is a method call statement, the method finishes executing since the method call expression was already visited at line 5. On the contrary, the function retrieves the expression at the left of s using the *leftExpression* function and accesses its identifier. Finally, the resource r is stored as a value in the resource corresponding to the *left* expression by calling the function *changeResource* at line 18.

The expressions are visited in the *visitE* function in Algorithm 8. Like with the statements, the process starts with a check of the *condition* and, depending on the type of expression e, it is then visited at lines 8, 11, 13 and 15 by different functions in the Algorithms 10, 11, 12 and 13. The simpler expressions are evaluated by the *visit* function at line 17. In the case of method call expressions, the returned resource is stored in the *mcalls* list. The mcalls list in the *p2aData* struct is then updated using the *updateMCalls* function at line 19.

The *visit* function in Algorithm 9 starts with obtaining the target, type and visibility of expression e. The target is accessed using the *targetOf* function, the type is accessed

Algorithm 8 Visiting Expressions

```
 1: function visitE(p2aData : dataStruct, e : expression)
 2:     condition ← currentCondition(p2aData)
 3:     if not mayBeTrue(condition) then
 4:         return _
 5:     end if

 6:     mcalls ← getMCalls(p2aData)
 7:     if e is methodCall then
 8:         r ← visitMCall(p2aData, e)
 9:         mcalls ← mcalls ∪ {r}
10:     else if e is infixExpression then
11:         r ← visitBinOp(p2aData, e)
12:     else if e is prefixExpression or e is postfixExpression then
13:         r ← visitUnOp(p2aData, e)
14:     else if e is conditional then
15:         r ← visitConditional(p2aData, e)
16:     else
17:         r ← visit(p2aData, e)
18:     end if

19:     updateMCalls(p2aData, mcalls)
20:     return r
21: end function
```

by the *typeOf* function and the visibility is accessed with the *visibilityOf* function. The function *visit* proceeds to check if the target is initialized.

If the expression does have a target, its type, identifier and visibility are accessed and then stored in the *texpData* struct at line 9. An attempt is made to get the target from the resource data already present in *p2aData* and store it in *t* at line 10. If the target is not found, the target expression is evaluated at line 12 and the value is stored at line 13. Then the resource corresponding to the given identifier *name* in the expression is retrieved at line 17. If the expression has no target, the function will only retrieve the resource using the same process it uses for the target at lines 23, 24 and 26.

Algorithm 10 is used to visit method call expressions. The signature of the method in the expression *e* is retrieved using the function *methodSignatureOf* function at line 2. The expression target is accessed at line 3 using the *targetOf* function and visited at line 4. Each of the arguments is also visited at line 9.

The execution of a method call requires the target and the arguments of the call to be read at lines 6 and 11 respectively, so those accesses are updated using the *read* function. The visibility of the target is accessed at line 13 using the function *getResourceVisiblity* to be used when creating the resource that the *visitMCall* function returns.

Given the type of *cls*, if it is initialized, the method is retrieved from it at line 19 the unresolved resources are grouped in a list and the return type of the *calledMethod* is accessed at line 26 to create the *mcall* method call result at line 28. Finding the *calledMethod* involves comparing signatures of the methods in *cls*. Method signatures are accessed by the use of the *getMethodSignature* function.

If the class is not initialized, this means the method cannot be retrieved and the class

Algorithm 9 Visiting Generic Expressions

```
 1: function VISIT(p2aData : dataStruct, e : expression)
 2:     target ← TARGETOF(e)
 3:     type ← TYPEOF(e)
 4:     visibility ← VISIBILITYOF(e)
 5:     if ISINITIALIZED(target) then
 6:         ttype ← TYPEOF(target)
 7:         tname ← IDENTIFIEROF(target)
 8:         tvisibility ← VISIBILITYOF(target)
 9:         texpData ← (ttype, _, tname, tvisibility)
10:         t ← GETRESOURCE(p2aData, texpData)
11:         if not ISINITIALIZED(t) then
12:             t ← EVALEXPRESSION(p2aData, target)
13:             CHANGERESOURCE(p2aData, texpData, t)
14:         end if

15:         name ← IDENTIFIEROF(e)
16:         expData ← (type, t, name, visibility)
17:         r ← GETRESOURCE(p2aData, expData)
18:         if not ISINITIALIZED(r) then
19:             r ← undetermined{type, visibility, {}, {}}
20:         end if
21:     else
22:         name ← IDENTIFIEROF(e)
23:         expData ← (type, _, name, visibility)
24:         r ← GETRESOURCE(p2aData, expData)
25:         if not ISINITIALIZED(r) then
26:             r ← EVALEXPRESSION(p2aData, e)
27:         end if
28:     end if

29:     return r
30: end function
```

cannot be visited, so the algorithm assumes conservatively that, whatever the method being called may be, it will both read and write to all the arguments. The type of the current statement is accessed at line 37 to then create an undetermined method at line 38 for the sake of registering the read and write accesses to all the argument resources. An *undetermined* resource is then created to contain the resources of the target and of the arguments at line 43.

Function *visitBinOp* of Algorithm 11 is used to visit binary operations. It begins by accessing the operand expressions at the left and right of the infix expression *e*, using the functions *leftOperandOf* and *rightOperandOf* at lines 2 and 3 respectively.

With both operand expressions, the function then starts visiting the expressions on either side at lines 4 and 30 and then checking if the operation itself is valid. To validate the operation, the operator is also accessed using the *operatorOf* function 6. Since the details related to the validation of operations depend directly on the rules of the Java language, they are not relevant to this paper.

If the operation is validated, the resources of the operands that need to be resolved are grouped in a list at lines 12 to 18 and if the list is not empty, an *unresolved* resource is then created at line 24. If there are no empty resources, the *binaryOperation* function performs

Algorithm 10 Visiting Method Call Expressions

```
1: function visitMCall(p2aData : dataStruct, e : methodCall)
2:     signature ← methodSignatureOf(e)
3:     texp ← targetOf(e)
4:     target ← visitE(p2aData, texp)
5:     currentMethod ← getCurrentMethod(p2aData)
6:     read(currentMethod, target)

7:     argResources ← {}
8:     for arg ∈ argsOf(e) do
9:         argResource ← visitE(p2aData, arg)
10:        argResources ← argResources ∪ {argResource}
11:        read(currentMethod, argResource)
12:    end for

13:    visibility ← getResourceVisiblity(target)
14:    condition ← getCurrentCondition(p2aData)

15:    cls ← classOf(target)
16:    if isInitialized(cls) then
17:        for method ∈ methodsOf(cls) do
18:            if getMethodSignature(method) = signature then
19:                calledMethod ← method
20:            end if
21:        end for

22:        ul ← getUnresolved(target, argResources)
23:        if isUnresolved(condition) then
24:            ul ← ul ∪ {condition}
25:        end if

26:        type ← getMethodReturnType(calledMethod)
27:        mc ← mcall{type, visibility, {}, calledMethod, target, argResources,
28: condition, ul, {}}
29:        if calledMethod is constructorMethod then
30:            object ← getThisResource(cls)
31:            r ← copyOf(object)
32:            updateMCalls(p2aData, {mc})
33:        else
34:            r ← mc
35:        end if
36:    else
37:        type ← getCurrentStatementType(p2aData)
38:        calledMethod ← undeterminedMethod{type, signature}
39:        for argResource ∈ argResources do
40:            read(calledMethod, argResource)
41:            write(calledMethod, argResource)
42:        end for

43:        r ← undetermined{type, visibility, {}, argResources ∪ {target}}
44:    end if

45:    return r
46: end function
```

Algorithm 11 Visiting Binary Operation Expressions

```
 1: function visitBinOp(p2aData : dataStruct, e : infixExpression)
 2:     leftExp ← leftOperandOf(e)
 3:     rightExp ← rightOperandOf(e)
 4:     left ← visitE(p2aData, leftExp)
 5:     right ← visitE(p2aData, rightExp)

 6:     operator ← operatorOf(e)
 7:     leftType ← typeOf(leftExp)
 8:     rightType ← typeOf(rightExp)
 9:     if not validOperation(operator, leftType, rightType) then
10:         return _
11:     end if

12:     unresolvedList ← {}
13:     if isUnresolved(left) then
14:         unresolvedList ← unresolvedList ∪ {left}
15:     end if
16:     if isUnresolved(right) then
17:         unresolvedList ← unresolvedList ∪ {right}
18:     end if

19:     if |unresolvedList| > 0 then
20:         le ← lambda{visitBinOp, {p2aData, e}}
21:         type ← typeOf(e)
22:         visibility ← visibilityOf(e)
23:         condition ← getCurrentCondition(p2aData)
24:         r ← unresolved{type, visibility, {}, condition, le, unresolvedList, {}}
25:     else
26:         r ← binaryOperation(operator, left, right)
27:     end if

28:     currentMethod ← getCurrentMethod(p2aData)
29:     read(currentMethod, left)
30:     read(currentMethod, right)
31:     return r
32: end function
```

the operation on the values of the resources according to the rules of the Java language at line 26. To finalize, the current method is accessed using the *getCurrentMethod* function at line 28 and the read access is recorded for both the *left* and *right* resources at lines 29 and 30 respectively.

The *visitUnOp* function of Algorithm 12 works with a principle similar to that of the previous function. The only difference is that the the value of the operand is also changed according to the rules of the Java language. The operation itself, excluding its side effects, is performed by the *unaryOperation* function at line 17. Since prefix and postfix expressions only have one operand, this is accessed using the *operandOf* function at line 2. At the end, the function accesses the current method in the *p2aData* data struct at line 29 and records the read access on the *operandResource* at line 30.

For visiting conditional expressions, the algorithm uses the function *visitConditional* in Algorithm 13. This function starts by acessing the condition expression of the conditonal *e* at line 2 before visiting it at line 3 and then the function accesses the *then* and *else*

Algorithm 12 Visiting Unary Operation Expressions

```
1:  function visitUnOp(p2aData : dataStruct, e : expression)
2:      operand ← operandOf(e)
3:      operandResource ← visitE(p2aData, operand)
4:      operator ← operatorOf(e)

5:      if not validOperation(operator, operandResource) then
6:          return _
7:      end if

8:      type ← typeOf(e)
9:      visibility ← visibilityOf(e)
10:     condition ← getCurrentCondition(p2aData)
11:     if e is postExpression then
12:         r ← operandResource
13:         if isUnresolved(r) then
14:             le ← lambda{visitUnOp, {p2aData, e}}
15:             operationRes ← unresolved{type, visibility, {}, condition, le, {r}, {}}
16:         else
17:             operationRes ← unaryOperation(operator, r, p2aData, operand)
18:         end if

19:     else
20:         r ← operandResource
21:         if isUnresolved(r) then
22:             le ← lambda{visitUnOp, {p2aData, e}}
23:             operationRes ← unresolved{type, visibility, {}, condition, le, {r}, {}}
24:         else
25:             operationRes ← unaryOperation(operator, r, p2aData, operand)
26:         end if

27:         r ← operationRes
28:     end if

29:     currentMethod ← getCurrentMethod(p2aData)
30:     read(currentMethod, operandResource)
31:     return r
32: end function
```

expressions using the functions *thenExpression* and *elseExpression* at lines 4 and 5 respectively.

If the condition resource *c* is a *resourceList*, the function creates the *conditionalData* struct with the relevant information at line 11, evaluates the result of the conditional operation at line 12 by calling the *evalConditional* function for each value in the list and, at line 13, stores the resultant resource in the resource list created at line 9. If the condition is not a resource list, the function creates the *conditionalData* struct at line 16 and makes a single call to the *evalConditional* function at line 17 and returns its result.

The *evalConditional* function retrieves the data from the *conditionalData* struct using the functions *getCondition*, *getThenExpression*, *getElseExpression* and *getExpression* at lines 22 to 24. It then starts by checking the type of the resource *c*. If *c* is a constant, given its value, it visits the *then* expression at line 30 or the *else* expression at line 32 and returns the resource *r*. If *c* needs to be resolved, a new unresolved resource is created at line 36 and returned. If none of the previous cases occurs, it is assumed the *c* is undetermined

Algorithm 13 Visiting Conditional Expressions

```
 1: function visitConditional(p2aData : dataStruct, e : conditional)
 2:     condition ← conditionOf(e)
 3:     c ← visitE(p2aData, condition)
 4:     then ← thenExpression(e)
 5:     else ← elseExpression(e)

 6:     if c is resourceList then
 7:         type ← typeOf(e)
 8:         visibility ← visibilityOf(e)
 9:         r ← resourceList{type, visibility, {}, {}}
10:         for value ∈ getResources(c) do
11:             conditionalData ← (value, then, else, e)
12:             values ← evalConditional(p2aData, conditionalData)
13:             addResources(r, values)
14:         end for
15:     else
16:         conditionalData ← (c, then, else, e)
17:         r ← evalConditional(p2aData, conditionalData)
18:     end if

19:     return r
20: end function

21: function evalConditional(p2aData : dataStruct, conditionalData : conditionalStruct)
22:     c ← getCondition(conditionalData)
23:     then ← getThenExpression(conditionalData)
24:     else ← getElseExpression(conditionalData)
25:     e ← getExpression(conditionalData)
26:     type ← typeOf(e)
27:     visibility ← visibilityOf(e)
28:     if c is constResource then
29:         if getValue(c) then
30:             r ← visitE(p2aData, then)
31:         else
32:             r ← visitE(p2aData, else)
33:         end if
34:     else if isUnresolved(c) then
35:         le ← lambda{visitConditional, {p2aData, conditionalData}}
36:         r ← unresolved{type, visibility, {}, c, le, {c}, {}}
37:         addDependents(r, c)
38:     else
39:         r ← undetermined{type, visibility, {}, {c, left, right}}
40:     end if

41:     return r
42: end function
```

and a new undetermined resource will be created at line 39 and returned.

The expressions that were not yet evaluated, go through the function *evalExpression* in Algorithm 14. This function starts with checking the type of the expression to evaluate. If the type is an array type, an array resource r is created at line 5. Then the initialization expressions in e are accessed using the function *getInitExpressions* at line 7 and each of the these is visited at line 8 to get their corresponding resource, which is stored in r by the *addData* function at line 9.

If the expression type is primitive, it is then evaluated by the *eval* function at line 13

Algorithm 14 Evaluating Expressions

```
 1: function evalExpression(p2aData : dataStruct, e : expression)
 2:     type ← typeOf(e)
 3:     visibility ← visibilityOf(e)
 4:     if type is arrayType then
 5:         r ← arrayResource{type, visibility, {}, {}}
 6:         i ← 0
 7:         for exp ∈ getInitExpressions(e) do
 8:             re ← visit(p2aData, exp)
 9:             addData(r, i, re)
10:             i ← i + 1
11:         end for
12:     else if type is primitiveType then
13:         value ← eval(e)
14:         if isInitialized(value) then
15:             r ← constResource{type, visibility, {}, value}
16:         else
17:             r ← undetermined{type, visibility, {}, {}}
18:         end if
19:     else
20:         cls ← getClass(p2aData, type)
21:         if isInitialized(cls) then
22:             if not isAnalyzed(p2aData, cls) then
23:                 visitClass(p2aData, cls)
24:             end if

25:             r ← getClassResource(p2aData, type)
26:         else
27:             r ← undetermined{type, visibility, {}, {}}
28:         end if
29:     end if

30:     return r
31: end function
```

which may return a constant value or nothing. The values returned by this function are required by the function *visitBlock* in algorithm 4 so that the whole algorithm is context-sensitive. If *eval* returns a value, a constant resource is created at line 15.

If *eval* does not return a value, an undetermined resource is created instead at line 17. If the expression type is not an array type nor a primitive type, the function assumes that *e* refers the identifier of an object. In this case it retrieves the class *cls* of the expression type at line 20 and checks if it is initialized.

If *cls* is initialized, the function proceeds to check if it has has not yet been analyzed. If the class was not analyzed, it is visited using the *visitClass* function at line 23 which stores its resource in the *p2aData* struct. Once *cls* is analyzed, the function *getClassResource* is called at line 25 with the data struct and the class to get the object resource of *cls*, which is then returned. On the contrary, if the class is not initialized, it is assumed the expression *e* cannot be evaluated and an undetermined resource is created at line 27 and returned.

Accessing resources from resource maps is taken care of by the function *getResource* in Algorithm 15. Initially all the relevant data is accessed from the *expData* struct. The functions *getType*, *getTarget*, *getName* and *getVisibility* are used to return the type, target

Algorithm 15 Auxiliary Function to Retrieve Resource Values

```
1: function getResource(p2aData : dataStruct, expData : expStruct)
2:     type ← getType(expData)
3:     target ← getTarget(expData)
4:     name ← getName(expData)
5:     visibility ← getVisibility(expData)

6:     if isInitialized(target) then
7:         if target is resourceList then
8:             r ← resourceList{type, visibility, {}, {}}
9:             for object ∈ getResources(target) do
10:                for field ∈ getObjectFields(object) do
11:                    if getFieldIdentifier(field) = name then
12:                        values ← getFieldValues(field)
13:                        addResources(r, values)
14:                    end if
15:                end for
16:            end for

17:            resources ← getResources(r)
18:            if |resources| > 0 then
19:                return r
20:            end if
21:        else
22:            for field ∈ getObjectFields(target) do
23:                if getFieldIdentifier(field) = name then
24:                    return getFieldValues(field)
25:                end if
26:            end for
27:        end if
28:    end if

29:    thisResource ← getCurrentObject(p2aData)
30:    l ← getVar(p2aData, name)
31:    p ← getParam(p2aData, name)
32:    f ← getField(p2aData, name)

33:    if name = "this" then
34:        return thisResource
35:    else if isInitialized(l) then
36:        return l
37:    else if isInitialized(p) then
38:        arg ← getParamArg(p)
39:        if isInitialized(arg) then
40:            return arg
41:        else
42:            return p
43:        end if
44:    else if isInitialized(f) then
45:        return f
46:    end if

47:    return _
48: end function
```

resource, identifier and the visibility stored in the *expData* struct at lines 2 to 5.

If the target resource is initialized and is a resource list, a new resource list is created to hold all the possible values of the field of the *target* and is then returned. If the target resource is not a resource list, the function just returns the values of the field. The function *getFieldValues* is used at line 24 to retrieve the values from a given field resource. In this case the values are retrieved from the field resource with the *name* identifier that belongs to the *target* resource or one of its values.

If the target resource is not initialized or none of its fields has the identifier *name*, other possibilities are then checked. In this case, the relevant resources stored in the maps of the *p2aData* struct are accessed using the functions *getVar getParam* and *getField* at lines 30 to 32. The "this" object resource is also accessed using the *getCurrentObject* function at line 29.

If the identifier *name* is "this", the *thisResource* is returned. Otherwise, the function checks if the resources l, p or f are initialized, by that order. If l or f are initialized, they are then returned. In the case l is not initialized and p is initialized, the function returns the argument of p if it is initialized or returns p on the contrary. The argument of p is accessed at line 38 using the function *getParamArgs*.

Whenever a statement with an assignment is evaluated, the *changeResource* function in Algorithm 16 is called to add or set resource values on fields or local variables. Like with the *getResource* function, the data in the *expData* struct is accessed at lines 2 to 4 to check if a resource with the given *name* identifier is one of the fields of the given *target* resource or if it is stored in the local variables map, parameters map or the fields map in the *p2aData* struct.

If the *target* resource is initialized, the field of that object will be updated. On the contrary, the relevant resources are accessed in the resource maps of the *p2aData* struct at lines 9 to 12. If the resource is initialized in the L map, the new value will replace the previous value of the resource at line 18. If the parameter resource p is initialized, the new value r will be added to a new resource list created at line 24 and added to the L map. If the field resource f is initialized, it will be updated at line 28. If the resource is not yet indexed anywhere, a new resource will be indexed in the L map and it will receive the new value.

To update fields, the function in algorithm 16 uses the auxiliary function *updateObjectField* in algorithm 17. Function *updateObjectField* retrieves the field with the identifier *name* from the object *target* and updates its value using the function *updateFieldValue*.

The function *updateFieldValue* checks if resource r undetermined. In that case, all the values of f will be added to r at line 21, which will then become the only value of f at line 22. In the opposite scenario, r is simply added to the values of f at line 24. Finally, *updateFieldValue* updates the accesses to f and r using the function *updateAccesses* at line 26. This last function simply records the read access to r and the write access to f in the case the method m is initialized.

Algorithm 16 Auxiliary Function to Change Resource Values

```
 1: function changeResource(p2aData : dataStruct, expData : expStruct, r : resource)
 2:     type ← getType(expData)
 3:     target ← getTarget(expData)
 4:     name ← getName(expData)
 5:     currentMethod ← getCurrentMethod(p2aData)

 6:     if isInitialized(target) then
 7:         updateObjectField(expData, currentMethod, r)
 8:     else
 9:         L ← getVarsMap(p2aData)
10:         l ← get(L, name)
11:         p ← getParam(p2aData, name)
12:         f ← getField(p2aData, name)

13:         if isInitialized(l) then
14:             if r is undetermined then
15:                 values ← getResources(l)
16:                 addResources(r, values)
17:             end if

18:             setResource(l, r)
19:         else if isInitialized(p) then
20:             if r is undetermined then
21:                 values ← getResources(p)
22:                 addResources(r, values)
23:             end if

24:             rl ← resourceList{type, getVisibility(r), {}, {r}}
25:             L ← L ∪ {name, rl}
26:             updateVarsMap(p2aData, L)
27:         else if isInitialized(f) then
28:             updateFieldValue(currentMethod, f, r)
29:         else
30:             if r is resourceList then
31:                 L ← L ∪ {(name, r)}
32:             else
33:                 rl ← resourceList{type, getVisibility(r), {}, {r}}
34:                 L ← L ∪ {(name, rl)}
35:             end if

36:             updateVarsMap(p2aData, L)
37:         end if
38:     end if
39: end function
```

Algorithm 17 Auxiliary Sub-Functions to Change Resource Values

```
function updateObjectField(expData : expStruct, currentMethod : method, r : resource)
    target ← getTarget(expData)
    name ← getName(expData)

    if target is resourceList then
        for object ∈ getResources(target) do
            f ← getObjectField(object, name)
            if isInitialized(f) then
                updateFieldValue(currentMethod, f, r)
            end if
        end for
    else
        f ← getObjectField(target, name)
        if isInitialized(f) then
            updateFieldValue(currentMethod, f, r)
        end if
    end if
end function

function updateFieldValue(m : method, f : fieldResource, r : resource)
    if r is undetermined then
        values ← getResources(f)
        addResources(r, values)
        setResource(f, r)
    else
        addFieldValue(f, r)
    end if
    updateAccesses(m, f, r)
end function

function updateAccesses(m : method, f : fieldResource, r : resource)
    if isInitialized(m) then
        read(m, r)
        write(m, f)
    end if
end function
```

The functions *read* and *write* in Algorithm 18 are used to record read and write accesses on resources and to add the accessed resources to methods. These use the function *getMethodAccesses* to obtain the accesses of the method m, from which the given access to a resource can be accessed using the *get* method.

If the resource *res* is not a field resource no accesses are recorded. On the contrary, the read or write access, depending on the function, is registered on the access map of the method using the *addAccessType*, *updateAccess* and *updateMethodAccesses* functions. Then, the read access of the parent of the *res* resource is registered and the field accessed is added to m by the function *addAccessedResources*. Finally, if the method m is not a constructor nor a private method, it is added to the *res* resource by the function *addMethodThatAccess*.

Algorithm 19, shows some of the most used auxiliary functions. The *isInitialized* function takes an object of any type and returns true if it is not a null object. The *isUnresolved* function takes a resource object and returns true if its type is unresolved, methodCall or paramResource. The reason for parameter resources being classified as unresolved here

Algorithm 18 Auxiliary Functions to Read/Write Resources

```
 1: function READ(m : method, res : resource)
 2:     if res is fieldResource then
 3:         accesses ← GETMETHODACCESSES(m)
 4:         access ← GET(accesses, res)
 5:         if not ISINITIALIZED(access) then
 6:             access ← {"r"}
 7:             accesses ← accesses ∪ {(res, access)}
 8:         else
 9:             ADDACCESSTYPE(access, "r")
10:             UPDATEACCESS(accesses, access)
11:         end if

12:         UPDATEMETHODACCESSES(m, accesses)
13:         parent ← GETFIELDPARENT(res)
14:         READ(m, parent)
15:         ADDMETHODACCESSEDRESOURCES(m, {res})
16:         v ← GETMETHODVISIBILITY(m)
17:         if m not constructorMethod or v not= private then
18:             ADDMETHODTHATACCESS(res, m)
19:         end if
20:     end if
21: end function

22: function WRITE(m : method, res : resource)
23:     if res is fieldResource then
24:         accesses ← GETMETHODACCESSES(m)
25:         access ← GET(accesses, res)
26:         if not ISINITIALIZED(access) then
27:             access ← {"w"}
28:             accesses ← accesses ∪ {(res, access)}
29:         else
30:             ADDACCESSTYPE(access, "w")
31:             UPDATEACCESS(accesses, access)
32:         end if

33:         UPDATEMETHODACCESSES(m, accesses)
34:         parent ← GETPARENT(res)
35:         READ(m, parent)
36:         ADDMETHODACCESSEDRESOURCES(m, {res})
37:         v ← GETMETHODVISIBILITY(m)
38:         if m not constructorMethod or v not= private then
39:             ADDMETHODTHATACCESS(res, m)
40:         end if
41:     end if
42: end function
```

Algorithm 19 Other Auxiliary Functions

```
 1: function getUnresolved(target : resource, args : resList)
 2:     list ← {}
 3:     if isUnresolved(target) then
 4:         list ← list ∪ {target}
 5:     end if

 6:     for arg ∈ args do
 7:         if isUnresolved(arg) then
 8:             list ← list ∪ {arg}
 9:         end if
10:     end for

11:     return list
12: end function

13: function isUnresolved(r : resource)
14:     return r is unresolved or r is methodCall or r is paramResource
15: end function

16: function mayBeTrue(condition : resource)
17:     if isInitialized(condition) then
18:         if condition is constResource then
19:             return getValue(condition)
20:         else if condition is resourceList then
21:             for value ∈ getResources(condition) do
22:                 if mayBeTrue(value) then
23:                     return true
24:                 end if
25:             end for

26:             return false
27:         end if

28:         return true
29:     end if

30:     return true
31: end function

32: function isInitialized(o : any)
33:     return o not= _
34: end function
```

is the fact that not only their argument is not known during this intra-method analysis, but it changes from call to call.

The *getUnresolved* function takes a *target* resource and a list of resources *args* and returns a list with all the resources that are deemed unresolved by the function *isUnresolved*. The function *mayBeTrue* takes a *condition* resource and checks if it can be true. If the resource is not initialized, the function returns true. If it is a resource list and any of its values can be true, the function returns true. If the resource is a constant, the function returns its value. If none of the previous cases applies, the function returns false.

Algorithm 20 Inter-Method Resolution

```
1: function RESOLVE(resultStruct : dataStruct)
2:     mcalls ← GETMCALLS(resultStruct)
3:     keepResolving ← true
4:     resolveLater ← {}
5:     while keepResolving do
6:         for mc ∈ mcalls do
7:             dependencies ← GETDEPENDENCIES(mc)
8:             if |dependencies| > 0 then
9:                 resolveLater ← resolveLater ∪ {mc}
10:            else if mc ∉ GETRESOLVEDCALLS(resultStruct) then
11:                RESOLVEMCALL(resultStruct, mc, _)
12:            end if
13:        end for

14:        keepResolving ← |resolveLater| > 0
15:        resolveLater ← {}
16:    end while
17: end function
```

With these functions, the PTA algorithm discovers all the possible values of field resources, the methods that access them and part of the read/write accesses. This information is used on the data race algorithm described in Section 3.2.3 to detect access conflicts.

3.2.2 Method Call Resolution

With the intra-method analysis done, the next step is to resolve all the method call results, all the parameters and all the other unresolved resources that depend on the first two. This requires an inter-method analysis that re-analyzes relevant statements while replacing target objects and parameter arguments with their real values. The following algorithms use auxiliary functions from the previous tables in section 3.2.1 along with others from the table 3.5.

For this analysis, our algorithm starts with the method call result objects generated during the intra-method analysis. Their dependencies are resolved, followed by the call itself and then the resources that depend on it. As Algorithm 20 shows, all of the method call results that were collected during the intra-method analysis are iterated and resolved if they were not resolved already, until no method call results are left unresolved. The method call results are retrieved from the *resultStruct* data struct using the *getMCalls* function and the resolved calls are accessed with the *getResolvedCalls* function.

Method call results are resolved according to the Algorithm 21. First, the function checks if the *mc* method call has already been resolved by using the *getResolvedOf* function and testing if the result *r* of that is initialized. If *r* is initialized, it is returned.

If *mc* is not yet resolved, the method call dependencies are retrieved with the function *getDependencies* and then each dependency is iterated and resolved. The resolution result is then used by the *updateDependency* function to update the *mc* data, such as the target, an argument or the condition. After all dependencies are resolved, the condition is accessed with the *getMCallCondition* function.

Table 3.5: Auxiliary Algorithm Functions (part 5)

Function → Return Type	Argument : Type	Description
getDependencies → resList	r : resource	Retrieves the resource list with the dependencies of the resource *r* if it is unresolved or a method call result.
getDependents → resList	r : resource	Retrieves the resource list with the dependents of the resource *r* if it is unresolved or a method call result.
getArgs → resList	mc : mcall	Retrieves the list of argument resources stored in the *mc* method call result and ordered by their index.
getMCallCondition → resource	mc : mcall	Retrieves the condition resource from the method call result *mc*.
getMcallTarget → objectResource	mc : mcall	Retrieves the target of the method call result *mc*.
getMCallMethod → method	mc : mcall method call result *mc*.	Retrieves the method from the
getResolvedCalls → mcallsList	resultStruct : dataStruct	Retrieves the resolved method call results stored in the *resultStruct* struct.
enterObject	resultStruct : dataStruct o : objectResource	Initializes the relevant data in the *resultStruct* struct according to the object resource *o*.
leaveObject	resultStruct : dataStruct	Resets the relevant data in the *resultStruct* struct to leave the object resource being visited.
getIndexParamName	P : paramsMap i : int	Retrieves from the map *P* the parameter with index *i*.
getParamType → type	p : parameterResource	Retrieves the type of the parameter resource *p*.
getParamVisibility → visibility	p : parameterResource	Retrieves the visibility of the parameter resource *p*.
getResolvedOf → resource	resultStruct : dataStruct r : resource	Retrieves the resource resultant from resolving the *r* resource stored in the data struct *resultStruct*.
setResolvedResource	resultStruct : dataStruct u : resource r : resource	Sets resource *r* as the result of resolving resource *u* in the data struct *resultStruct*.
getUnresolvedCondition → resource	u : unresolved	Retrieves the condition resource from the unresolved resource *u*.
getLambdaExpression → lambda	u : unresolved	Retrieves the lambda expression from the unresolved resource *u*.
execute → resource	l : lambda	Executes the lambda expression *l* and returns the resultant resource.
getUnresolved → resource	update : updateStruct	Retrieves the unresolved resource, which can also be a method call result, from the update struct *update.*
getResolved → resource	update : updateStruct	Retrieves the resolved resource from the update struct *update*.
getRequester → resource	update : updateStruct	Retrieves the resource that requested the resource update in the update struct *update*.
updateDependency	u : resource d : resource r : resource	If *u* is a mcall or an unresolved, updates the data in *u* by replacing the dependency resource *d* with the resource *r*.
addResolvedCall	resultStruct : dataStruct mc : mcall	Sets the method call result *mc* as resolved for cases when the resolution of one of the dependencies is _.
setArgument	param : parameterResource arg : resource	Sets the resource *arg* as the argument of the parameter resource *param*.
getMethodReturnResource → resource	m : method	Retrieves the resource returned by the method *m*.
img → anyList	M : anyMap	Retrieves the list of keys used to index data in the *M* map.

Algorithm 21 Resolving Method Call Results

```
1: function resolveMCall(resultStruct : dataStruct, mc : mcall, req : resource)
2:     r ← getResolvedOf(resultStruct, mc)
3:     if isInitialized(r) then
4:         return r
5:     end if
6:     dependencies ← getDependencies(mc)
7:     for dependency ∈ dependencies do
8:         if dependency is mcall then
9:             r ← resolveMCall(resultStruct, dependency, mc)
10:        else if dependency is unresolved then
11:            r ← resolveUnresolved(resultStruct, dependency, mc)
12:        else
13:            r ← getParamArg(dependency)
14:        end if

15:        updateDependency(mc, dependency, r)
16:    end for

17:    condition ← getMCallCondition(mc)
18:    if mayBeTrue(condition) then
19:        m ← getMCallMethod(mc)
20:        args ← getArgs(mc)
21:        P ← getMethodParams(m)
22:        ML ← getMethodVarMaps(resultStruct)
23:        target ← getMcallTarget(mc)
24:        enterObject(resultStruct, target)
25:        i ← 0
26:        for arg ∈ args do
27:            paramName ← getIndexParamName(P, i)
28:            param ← getParam(resultStruct, paramName)
29:            type ← getParamType(param)
30:            visibility ← getParamVisibility(param)
31:            expData ← (type, _, paramName, visibility)
32:            changeResource(resultStruct, expData, arg)
33:            i ← i + 1
34:        end for

35:        L ← visitMethod(resultStruct, m)
36:        r ← getReturnResource(resultStruct, L, mc)
37:        ML ← ML ∪ {m, L}
38:        updateMethodVarMaps(resultStruct, ML)
39:        leaveObject(resultStruct)
40:    else
41:        r ← _
42:    end if

43:    setResolvedResource(resultStruct, mc, r)
44:    update ← (mc, r, req)
45:    updateDependents(resultStruct, update)
46:    return r
47: end function
```

If the condition can be true, the method, the arguments and the target are retrieved from the method call and the *enterObject* function is called to initialize the relevant information in the *resultStruct* struct. For each argument, its identifier is accessed from the *P* map, given the index *i* using function *getIndexParamName* and then the parameter value is changed to that of the argument. Once all parameters are changed, the method is visited, which updates the fields in the *resultStruct* and gives the map of local variables, which are then used to resolve the return resource of the call that is stored on *r*. If the condition cannot be true, *r* is set to a non initialized value.

Afterwards, the *setResolvedResource* function is used to replace the *mc* resource with *r* and record the method call result resolution in the *resultStruct* struct. Finally an *update* struct is created to update the dependents of *mc*.

The functions *getMethod* and *getArgs* take a method call as argument and return its method and arguments respectively. The *getMethodParams* functions takes a method as argument and returns its map of parameter resources. The *getParamArg* function is used to get the current argument of a parameter resource. The function *updateData* updates the target, argument or condition of *mc* that are equal to that of the *dependency* and replaces them with the new resource *r*.

The unresolved resources are resolved in the *resolveUnresolved* function of Algorithm 22. Like with the previous function, it checks if the unresolved resource *u* was previously resolved, returning the resolved value in that case. If not, the dependencies are resolved and the results of their resolution are used to update the data of *u*, such as its condition, one of the arguments or the lambda expression.

If the condition can be true, the lambda is accessed with the *getLambdaExpression* function, then executed by the *execute* function and the execution result is stored on *r*. On the contrary, *r* gets a null value. Then, the unresolved resource is indexed by *u* in the map of resolved resources of *resultStruct*. Finally, the function uses *r* to update the fields map of the data struct and to update the resources depending on *u*.

The function *updateDependents* of Algorithm 23 is used to update the method call results, unresolved resources or parameters that depend on a resource that is either a method call or an unresolved resource. From the *update* struct the resource that was resolved, the resolution result and the resource requesting the resolution are accessed using the functions *getUnresolved*, *getResolved* and *getRequester* respectively.

The dependents of *u* are retrieved using the function *getDependents* and iterated. For each *dependent*, if the resource *r* is not initialized, it is assumed that *u* cannot be resolved because its condition is false. In this particular case, if the *dependent* is a method call result, it is marked as a resolved call in the *resultStruct* struct.

When *u* cannot be resolved, the dependents of *dependent* cannot be resolved either and if *u* is a method call, it is added to the list of resolved calls by the function *addResolvedCall*. If *u* was resolved, and the dependent is a method call or an unresolved resource, the dependency of the dependent is updated and removed from the dependencies list. If *u* was

Algorithm 22 Resolving Unresolved Resources

```
1: function resolveUnresolved(resultStruct : dataStruct, u : unresolved, req : resource)
2:     r ← getResolvedOf(resultStruct, u)
3:     if isInitialized(r) then
4:         return r
5:     end if

6:     dependencies ← getDependencies(u)
7:     for dependency ∈ dependencies do
8:         if dependency is mcall then
9:             r ← resolveMCall(resultStruct, dependency, u)
10:        else if dependency is unresolved then
11:            r ← resolveUnresolved(resultStruct, dependency, u)
12:        else
13:            r ← getParamArg(dependency)
14:        end if

15:        updateDependency(u, dependency, r)
16:    end for

17:    condition ← getUnresolvedCondition(u)
18:    if mayBeTrue(condition) then
19:        l ← getLambdaExpression(u)
20:        r ← execute(l)
21:    else
22:        r ← _
23:    end if

24:    setResolvedResource(resultStruct, u, r)
25:    update ← (u, r, req)
26:    updateDependents(resultStruct, update)

27:    return r
28: end function
```

resolved and the dependent is a parameter resource, its argument is updated by the *setArguments* function. When a method call or unresolved dependent has no dependencies left, it will be resolved. If the dependent is the same resource that triggered the resolution of u, no updates are done on the dependent.

Algorithm 24 shows the function *getReturnResource*, which resolves the return value of a method call, given the data struct and a map of local variables. The return resource of the method m is obtained by the *getMethodReturnResource* function, which takes a method as argument.

This function attempts to find a resource equals to the *returnResource* by comparing it to the local variables, parameters, the method call target or one of the fields in the target. Otherwise, if the *returnResource* is a method call or an unresolved resource, the function will return its resolution. And if the *returnResource* is a parameter resource, its current argument is returned. If none of the previous cases occurs, the *returnResource* itself is returned.

The algorithms in this subsection ensure that all method call results and all resources depending on them are resolved, which gives information on the field accesses that had

Algorithm 23 Updating Dependents

```
1: function updateDependents(resultStruct : dataStruct, update : updateStruct)
2:     u ← getUnresolved(update)
3:     r ← getResolved(update)
4:     req ← getRequester(update)
5:     dependents ← getDependents(u)
6:     for dependent ∈ dependents do
7:         if dependent ≠ req then
8:             if not isInitialized(r) then
9:                 if dependent is mcall then
10:                    addResolvedCall(resultStruct, dependent)
11:                end if
12:            else if dependent is mcall or dependent is unresolved then
13:                updateDependency(dependent, u, r)
14:                dl ← getDependencies(dependent)
15:                if |dl| = 0 then
16:                    if dependent is mcall then
17:                        resolveMCall(resultStruct, dependent, _)
18:                    else if dependent is unresolved then
19:                        resolveUnresolved(resultStruct, dependent, _)
20:                    end if
21:                end if
22:            else
23:                setArgument(dependent, r)
24:            end if
25:        end if
26:    end for
27: end function
```

not been found in the functions of Section 3.2.1. With this information obtained, the data race analysis in Section 3.2.3 can take place.

Algorithm 24 Resolving the Method Call Return Resource

```
1: function getReturnResource(resultStruct : dataStruct, L : varMap, mc : mcall)
2:     target ← getTarget(mc)
3:     m ← getMethod(mc)
4:     P ← getMethodParams(m)
5:     returnResource ← getMethodReturnResource(m)

6:     for l ∈ img(L) do
7:         if returnResource = l then
8:             return l
9:         else if l is objectResource then
10:            for f ∈ getObjectFields(l) do
11:                if returnResource = f then
12:                    return f
13:                end if
14:            end for
15:        end if
16:    end for

17:    for p ∈ img(P) do
18:        r ← getParamArg(p)
19:        if returnResource = r then
20:            return r
21:        else if r is objectResource then
22:            for f ∈ getObjectFields(r) do
23:                if returnResource = f then
24:                    return f
25:                end if
26:            end for
27:        end if
28:    end for

29:    if returnResource = target then
30:        return target
31:    else
32:        for f ∈ getObjectFields(target) do
33:            if returnResource = f then
34:                return f
35:            end if
36:        end for
37:    end if

38:    if returnResource is mcall then
39:        return resolveMCall(resultStruct, returnResource, returnResource)
40:    else if returnResource is unresolved then
41:        return resolveUnresolved(resultStruct, returnResource, returnResource)
42:    else if returnResource is paramResource then
43:        return getParamArg(returnResource)
44:    else
45:        return returnResource
46:    end if
47: end function
```

3.2.3 Data Race Detection

Our PTA algorithm is supposed to take the abstract representation of a program without concurrency control, as mentioned in section 3.1, we have to assume that any and all methods can be executed by more than one thread. In addition, we also assume that read and write accesses are not atomic operations. This means that, any write access performed by a method will be a source of conflicts.

Once the PTA is done, we can now start to detect the fields which are targets of data races. Since most of the heavy lifting is already done, finding access conflicts is as simple as checking whether a field resource gets written by at least a method and, in that case, adding marking that field resource as one with conflicting accesses.

The *detect* function in Algorithm 25 illustrates how the data race detection is done. It starts by reading the fields map F from the *resultStruct* and initializes a collection *conflictResources* of resources that will include the fields that have conflicting accesses. For each field resource f in F, the list of methods that access f is read by the function *getMethodsThatAccess* before the algorithm iterates over the methods.

For each iterated method, the function *getMethodAccesses* is used at line 7 to retrieve all the accesses done by the method and then the function *getFieldAccess* takes the method accesses and the field resource f and retrieves the access done to f at line 8. Then the algorithm checks at line 9 if the access includes a write and in that case, it adds f to the collection of resources with access conflicts.

With the list of field resources with conflicts, it is finally possible to report all of the data races. At line 15, the *detect* function calls the *reportDataRaces* auxiliary function with the *conflictResources* resource list which will gather all the accesses of the conflicting resources and report all the possible data races.

Algorithm 25 Detecting Data Races

```
1:  function DETECT(resultStruct : dataStruct)
2:      F ← GETFIELDSMAP(resultStruct)
3:      conflictResources ← {}

4:      for f ∈ IMG(F) do
5:          methods ← GETMETHODSTHATACCESS(f)
6:          for m ∈ methods do
7:              accesses ← GETMETHODACCESSES(m)
8:              access ← GETFIELDACCESS(accesses, f)
9:              if "w" ∈ access then
10:                 conflictResources ← conflictResources ∪ {f}
11:                 break
12:             end if
13:         end for
14:     end for

15:     REPORTDATARACES(conflictResources)
16: end function
```

3.2.4 Algorithm Limitations

The algorithm presents a limitation regarding the visibility of methods. While private methods are not considered towards access conflicts, protected and package-protected methods are taken into account the same way as public methods. For methods that cannot be analyzed, our algorithm assumes that they both read and write to their arguments, which may lead to false positives.

3.3 Implementation

The algorithm described in the previous section was partially implemented for the Java language using the Eclipse JDT [7]. The JDT handles the parsing of Java source code, provides class information in the format of Abstract Syntax Tree (AST) nodes and also provides a visitor that can be used to visit all the statements in a node.

3.3.1 Points-to Analysis Implementation

The PTA was implemented using the abstract AST visitor of the JDT and the java analysis interface of AtomiS [23]. The JDT provides an AST node tree of the input program, where every node represents a different part of the source code. The visitor provides the means to access every node of the tree. The implementation of the analysis interface exposes the methods for the PTA analysis.

In the implementation, the abstract visitor is extended by the *TypeCollectorVisitor* and by the *PointsToAnalysisVisitor*. To explain how the visitants work in conjunction with the analysis interface, we will describe how the visitors work and then we describe the sequence of calls in the *PointsToAnalysisVisitor* for a code example.

The type collector visitor visits each statement in the source code to collect all the types in the program. It visits declaration statements of types, fields, methods and variables, along with initialization expressions. After the type collector visitor finishes, the PTA visitor visits the program statements to perform the intra-method PTA analysis. Below follows the calls performed by the *PointsToAnalysisVisitor* for the example in the listing 3.1.

```
1 public class ExampleClass {
2   int i;
3
4   public void m1() {
5     m2();
6   }
7
8   public int m2() { ... }
9 }
```

Listing 3.1: Simple Java Code Example

```
1  public class ExampleClass2 {
2    int n = 0;
3    int x = 5;
4
5    public void m() {
6      if (x > 0) {
7        n = n * x;
8      }
9    }
10 }
```

Listing 3.2: Java Code Example with if-else

1. The visitor visits the type declaration at line 1, sets the current class to the evaluated type and calls the *enterType* method of the analysis interface. This initializes the class resource information. Since this class is not an interface and has no explicit constructor the interface method also resolves the implicit constructor and initializes its information.

2. The visitor visits the field declaration at line 2.

3. The visitor visits the method declaration at line 4, resolves the method *m1* and calls the *enterMethod* method of the analysis interface, which initializes the information of the method.

4. The visitor visits the method invocation at line 5 and calls the *call* method.

    a) The *call* method calls the visitor to visit the target of the invoked method, which in turn calls the *variable* method of the analysis interface to resolve the target.

    b) The *call* method resolves the invoked method and initializes the method call resource.

5. The visitor proceeds to visit the next method declaration at line 8, which will finish once all statements within the body of the method *m2* are visited.

Next we show the calls performed for the example in the listing 3.2. This example contains field declarations with initializations an if-else structure binary operations and field updates.

1. The visitor visits the type declaration at line 1, sets the current class to the evaluated type and calls the *enterType* method of the analysis. This initializes the class resource information. Just like the previous example, the interface method resolves the implicit constructor and initializes its information.

2. The visitor visits the field declaration at line 2 and calls the *update* method of the analysis.

    a) The *update* method resolves the value 0 on the right side of the declaration by calling the method *literal* of the analysis.

   b) The *update* method resolves the target of the field *n* by calling the *variable* method of the analysis to get the "this" instance of the class being visited.
   c) The *update* method adds the value 0 to the field *n* of the "this" instance.

3. The visitor repeats step 2 and its sub-steps for the declaration of the field *x* at line 3.
4. The visitor visits the method declaration at line 5, resolves the method *m* and calls the *enterMethod* method of the analysis, which initializes the information of the method.
5. The visitor visits the if statement at line 6 and calls the *enterIf* method of the analysis, which resolves the condition expression before entering the if block.
   a) The binary operation in the condition expression is resolved by calling the *binaryOperator* method.
   b) The *binaryOperator* method of the analysis resolves the expression at the left of the operation by calling the *select* method.
   c) The *select* method of the analysis resolves the target expression by calling the method *variable* to retrieve the "this" instance of the class being visited and returns the field *x*.
   d) The method *binaryOperator* resolves the expression at the right of the operation by calling the *literal* method.
   e) The method *binaryOperator* evaluates the binary operation and returns its value.
6. The visitor visits the field update at line 7 and calls the *update* method.
   a) The *update* method resolves the binary operation in the expression at the right of the assignment by calling the *binaryOperator* method.
   b) The *binaryOperator* method of the analysis resolves the expression at the left of the operation by calling the *select* method.
   c) The *select* method of the analysis resolves the target expression by calling the method *variable* to retrieve the "this" instance of the class being visited and returns the field *n*.
   d) The previous 2 sub-steps are repeated for the expression at the right of the operation, which returns the field *x*.
   e) The method *binaryOperator* evaluates the binary operation and returns its value.
   f) The *update* method checks if the condition of the block being visited is true. Since that is the case here, the value of the operation is added to the field *n* of the "this" instance.
7. The visitor calls the *exitIf* method of the analysis to restore the block context of the method *m*.

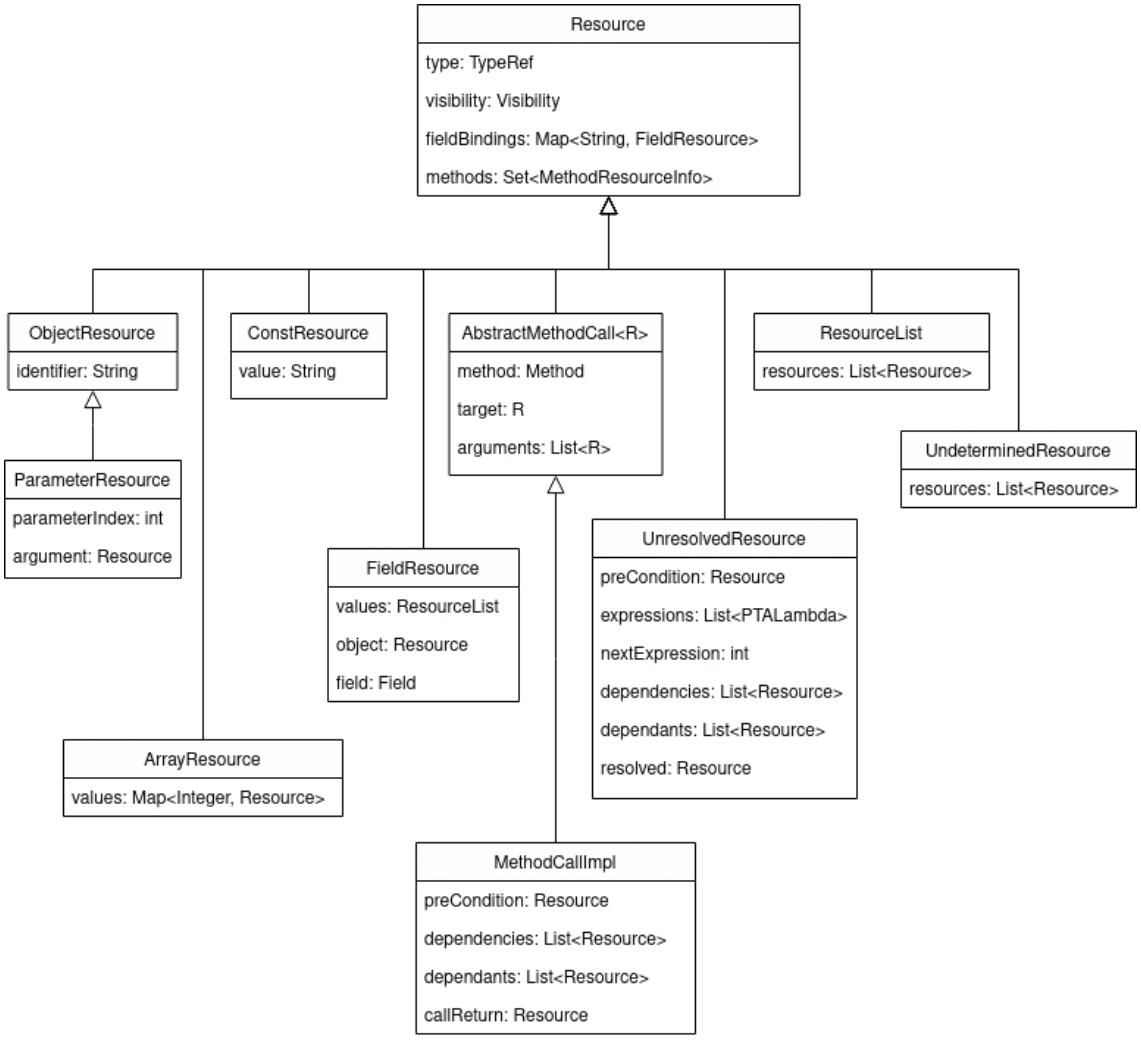


Figure 3.1: Resource Data Structures

As the nodes are visited, the analysis creates representations of the program resources. Figure 3.1 shows a diagram of the data structures of the program resources. They are implemented in classes that extend the Resource class. Each local variable in a method will have a ResourceList associated with it. When method calls or parameter resources are assigned to a variable, an unresolved resource is created to be assigned to the variable instead.

3.3.2 Implementation Limitations

Our implementation currently lacks concurrency, which means it currently will not scale well on large programs. It also has an incomplete support for unary operations and array accesses that limits the context-sensitivity when unary operations are involved and limits the accuracy of data race reports for fields that are arrays, since the implementation does not consider the index used for the array access.

4

Evaluation

In this chapter we describe our methods of evaluation in section 4.1 and proceed to evaluate the results obtained in section 4.2.

4.1 Evaluation Methodology

The goal of our solution is the static analysis of a Java program. The tool should analyze the program and output the necessary information relative to field accesses. Ideally the tool should execute in a reasonable amount of time. The implementation evaluation should answer the following questions:

- Are the data races being properly detected? How many false positives? How many false negatives?
- What is the average runtime of our solution? Is it acceptable for a typical software development process?

Since we assume the code to analyze has no concurrency control and may be executed concurrently, any write accesses to fields directly done by non-private methods or indirectly done through calls to private methods can conflict with read accesses, with other write accesses or even with themselves in the case the same method is executed by 2 or more threads. Each possible conflict is counted as a data race. However, in the cases we have multiple read or write accesses for the same field in the same method, we do not count all the individual accesses.

We define false positives as reported data races that do not actually exist and false negatives as data races that exist but have not been reported. In the context of our solution, the false positives and false negatives are also restricted only to the code of the program that can be visited by our tool.

To evaluate the implementation of the analysis algorithm we devised, we used the *RacerD* [3] tool to compare the detection of data races and the runtime on real open source code examples. As mentioned in section 2.4, *RacerD* is a part of the infer framework available on github. To use the *RacerD* annotations in the code to analyze, we also needed to

```
infer --racerd-only -- javac -cp .:infer-annotation-0.18.0.jar File.java
```

Listing 4.1: RacerD Analysis Command Example

Table 4.1: Points-to Analysis Test Results
* The points-to analysis only supports method calls within the same class or static method calls.

Features	Value	Object	Resource List	Field	Method Call	Parameter	Unresolved	Undetermined
assign	✓	✓	-	✓	✓	✓	✓	-
update	✓	✓	-	✓	-	✓	✓	-
method call	✓*	✓*	-	-	✓*	✓*	✓*	-
new	-	-	-	-	-	-	-	-
binary op	✓	-	-	-	✓	✓	✓	-

use the infer-annotations package found in the MVN repository. For our testing, we used the version 1.1.0 of the infer framework and version 0.18.0 of the annotations jar.

To run the code analysis we ran commands similar to the example shown in the listing 4.1. When running the benchmarks, *RacerD* outputs the issues found on the terminal along with a total counter of the issues found. However, the terminal output only lists a maximum of 5 issues, so the data for each example, shown in the next section, was taken from the report file generated at `infer-out/report.txt`.

4.2 Results Evaluation

To test the analysis of our implementation, some JUnit tests were made to test different features. Table 4.1 shows what was tested given various resource types and the test results. While the analysis of conditionals, casts and parenthesized expressions was implemented, it was not yet tested, so they do not show up in the table.

For the benchmarks of our solution, we used source code from a public repository, namely 6 classes of the benchmarks developed for the deuce framework [15], which can be found here. The classes used are the *HashTableSet*, the *IntJavaConcurrentHashSet*, the *IntSetHash*, the *IntSetLinkedList*, the *IntSetSkipList* and the *PrimeFinder*.

All class examples, except the *PrimeFinder* class, provide different implementations for the IntSet interface. *HashTableSet* uses a hash table for the implementation, the *IntJavaConcurrentHashSet* uses a concurrent hash map, the *IntSetHash* uses an array of integers, the *IntSetLinkedList* uses nodes linked to the next node and the *IntSetSkipList* uses nodes with arrays of the nodes in the next positions. The classes *IntSetHash*, *IntSetLinkedList* and *IntSetSkipList* also make use of an Atomic annotation which can be found here. The *PrimeFinder* class contains an array of prime numbers and a static function to retrieve a prime number equal to or greater than the desired integer passed as argument.

In the table 4.2 we can see information of the code examples. It contains the number of code lines written, discounting any leading or trailing white lines. It also shows the amount of classes declared inside the example class, the number of methods, the amount

Table 4.2: Benchmark Statistics

Example	# lines of code	# inner classes	# methods	# reads expected	# writes expected	# races expected
HashTableSet	24	0	3	0	0	0
IntJavaConcurrentHashSet	23	0	3	0	0	0
IntSetHash	152	0	7	15	10	56
IntSetLinkedList	99	1	6	1	1	2
IntSetSkipList	200	1	10	8	3	16
PrimeFinder	126	0	1	0	0	0

```
24 public boolean add(int value) {
25   int index = findIndex(value);
26   if (index < 0) {
27     return false;       // already present in set, nothing to add
28   }
29
30   byte previousState = states[index];
31   set[index] = value;
32   states[index] = FULL;
33   postInsertHook(previousState == FREE);
34
35   return true;
36 }
```

Listing 4.2: IntSetHash add method

of conflicting read and write accesses that cause conflicts and the amount of data races that are expected to be detected if the tool used works as we intend. While every example has accesses to mutable fields, only the *IntSetHash*, *IntSetLinkedList* and *IntSetSkipList* have conflicting accesses in the code.

The *IntSetHash* class has a total of 56 possible data races. One of those races can be found in the add method shown in listing 4.2. At line 30 there is a read access for the field `states` which can conflict with the write access at line 32.

There are also three data races found in the postInsertHook method of the *IntSetHash* class in the listing 4.3. At line 120 there is a read access to the field `free` which conflicts with the write access at line 124. At line 124 there is also a read access to the field `maxSize` which conflicts with the write access at line 131. And at line 124 there is a write access to the field `size` which conflicts with the read access at line 129.

In listing 4.4 we can find another 2 possible data races of the *IntSetHash* example in the method rehash. At line 137 there is a read to the field `set` which can conflict with the write access at line 140. Another read to the field `states` at line 138 conflicts with a write at line 141.

The *IntSetLinkedList* has a total of 2 data races, one of them can occur when the write access to the field `m_next` at line 29 in the setNext method of the internal Node class conflicts with the read access at line 33 in the method getNext of the internal class, as depicted in listing 4.5.

The *IntSetSkipList* has a total of 16 data races. One of these races is found at lines 31 and 35 in the methods setForward and getForward of the internal Node class. As seen in

```
118 protected final void postInsertHook(boolean usedFreeSlot) {
119   if (usedFreeSlot) {
120     free--;
121   }
122
123   // rehash whenever we exhaust the available space in the table
124   if (++size > maxSize || free == 0) {
125     // choose a new capacity suited to the new state of the table
126     // if we've grown beyond our maximum size, double capacity;
127     // if we've exhausted the free spots, rehash to the same capacity,
128     // which will free up any stale removed slots for reuse.
129     int newCapacity = size > maxSize ? PrimeFinder.nextPrime(states.length << 1) :
      states.length;
130     rehash(newCapacity);
131     maxSize = states.length/2;
132   }
133 }
```

Listing 4.3: IntSetHash postInsertHook method

```
135 protected void rehash(int newCapacity) {
136   int oldCapacity = set.length;
137   int oldSet[] = set;
138   byte oldStates[] = states;
139
140   set = new int[newCapacity];
141   states = new byte[newCapacity];
142
143   for (int i = oldCapacity; i-- > 0;) {
144     if(oldStates[i] == FULL) {
145       int o = oldSet[i];
146       int index = findIndex(o);
147       set[index] = o;
148       states[index] = FULL;
149     }
150   }
151 }
```

Listing 4.4: IntSetHash rehash method

```
28 public void setNext(Node next) {
29   m_next = next;
30 }
31
32 public Node getNext() {
33   return m_next;
34 }
```

Listing 4.5: IntSetLinkedList Node.setNext method

```
30 public void setForward(int level, Node next) {
31   m_forward[level] = next;
32 }
33
34 public Node getForward(int level) {
35   return m_forward[level];
36 }
```

Listing 4.6: IntSetSkipList Node.setForward and Node.getForward methods

```
89 public boolean add(int value) {
90   boolean result;
91
92   Node[] update = new Node[m_maxLevel + 1];
93   Node node = m_head;
94
95   for (int i = m_level; i >= 0; i--) {
96     Node next = node.getForward(i);
97     while (next.getValue() < value) {
98       node = next;
99       next = node.getForward(i);
100    }
101    update[i] = node;
102  }
103  node = node.getForward(0);
104
105  if (node.getValue() == value) {
106    result = false;
107  } else {
108    int level = randomLevel();
109    if (level > m_level) {
110      for (int i = m_level + 1; i <= level; i++)
111        update[i] = m_head;
112      m_level = level;
113    }
114    node = new Node(level, value);
115    for (int i = 0; i <= level; i++) {
116      node.setForward(i, update[i].getForward(i));
117      update[i].setForward(i, node);
118    }
119    result = true;
120  }
121
122  return result;
123 }
```

Listing 4.7: IntSetSkipList add method

the listing 4.6 the write and read accesses to the field `m_forward` conflict with each other. Another race is located in the add method shown in the listing 4.7 where the write to the field `m_level` at line 112 can conflict with the read at line 110. The method remove in the listing 4.8 also contains a possible data race, since the read at line 149 can conflict with the write access at line 150.

There are 2 more classes from the deuce framework that we wished to use on our benchmarks but we could not get our tool to function in them without errors, the RBTree class and the IntJavaHashSetClass. In the RBTree class there is a declaration of an enum, which is not supported yet in our solution and leads to one of the arguments of a method call being null. In the IntJavaHashSet, the visitor that collects the types fails when visiting

```
126 public boolean remove(int value) {
127   boolean result;
128
129   Node[] update = new Node[m_maxLevel + 1];
130   Node node = m_head;
131
132   for (int i = m_level; i >= 0; i--) {
133     Node next = node.getForward(i);
134     while (next.getValue() < value) {
135       node = next;
136       next = node.getForward(i);
137     }
138     update[i] = node;
139   }
140   node = node.getForward(0);
141
142   if (node.getValue() != value) {
143     result = false;
144   } else {
145     for (int i = 0; i <= m_level; i++) {
146       if (update[i].getForward(i) == node)
147         update[i].setForward(i, node.getForward(i));
148     }
149     while (m_level > 0 && m_head.getForward(m_level).getForward(0) == null)
150       m_level--;
151     result = true;
152   }
153
154   return result;
155 }
```

Listing 4.8: IntSetSkipList remove method

Table 4.3: Points-To Analysis Benchmark Results

Example	Our Solution				RacerD			
	# reads detected	# writes detected	# races detected	average time (ms)	# reads detected	# writes detected	# races detected	average time (ms)
HashTableSet	0 (100%)	0 (100%)	0 (100%)	159	0 (100%)	0 (100%)	0 (100%)	39
IntJavaConcurrentHashSet	0 (100%)	0 (100%)	0 (100%)	165	0 (100%)	0 (100%)	0 (100%)	37
IntSetHash	15 (100%)	10 (100%)	56 (100%)	199	8 (53%)	9 (90%)	40 (71%)	54
IntSetLinkedList	1 (100%)	1 (100%)	2 (100%)	167	1 (100%)	1 (100%)	2 (100%)	39
IntSetSkipList	8 (100%)	3 (100%)	16 (100%)	226	6 (75%)	2 (67%)	8 (50%)	54
PrimeFinder	0 (100%)	0 (100%)	0 (100%)	164	0 (100%)	0 (100%)	0 (100%)	38

a call to the *eq* static method in the inner SimpleEntry class.

Table 4.3 shows the results of the analysis of our implementation on the benchmarks and the results of the analysis of RacerD on the same examples. These results were obtained in an Omen by HP laptop with an Intel Core i7-7700HQ with up to 3.8GHz and 16 GB of RAM.

In the *IntSetHash* class, *RacerD* reported indirect conflicting accesses resulting from calls to a non-private method where the conflicting accesses occur. We deemed these to not be relevant since we only care about direct accesses in non-private methods or indirect accesses that result from calls to private methods as previously mentioned in section 4.1. In the same class, *RacerD* also reported 2 reads to the field `states` in the *findIndex* method, which we will only count as a single access since our tool does not count multiple accesses of the same type to the same field and in same method.

As we can observe our tool detects all the conflicting accesses we expected in every

example. Our tool has a runtime of 180 milliseconds on average. *RacerD* detects all conflicting accesses in the *IntSetLinkedList* example but misses some accesses in the other examples and has an average runtime of 44 milliseconds. *RacerD* required a `@ThreadSafe` annotation for the IntSet interface, an annotation for the PrimeFinder class and an annotation for each of the internal classes.

The runtime of our tool is up to 4.46 times slower than *RacerD* in the *IntJavaConcurrentHashSet* class. However, *RacerD* misses possible data races in the other classes *InSetHash* and *IntSetSkipList*. In *IntSetSkipList RacerD* does not report the write access to the `m_level` field inside the method *add* at line 112 as a conflicting access, neither reports the read accesses to the `m_forward` field at lines 27 and 31 in the methods *getLevel* and *setForward*.

RacerD also misses a write access to the field `size` in the method *postInsertHook* at line 124 of the *IntSetHash* class. In this same class, 7 read accesses are missed. In method *postInsertHook* none of the reads at lines 120 or 124 to field `free` is reported and none of the reads to field `size` at lines 124 or 129 is reported. The read access to field `set` in the method *add* at line 31 and the read access to field `states` at line 32 were not reported. In the method *remove* the read accesses to fields `states` and `size` at lines 48 and 49 respectively were not reported. The indirect read to the field `states` in method *contains* through the private method *index* was not reported.

In general, it seems that *RacerD* does not consider the reads to array fields where the program writes to an index of the array. The tool also seems not to report read accesses for fields with postfix increment or decrement operations and seems to not report read or write accesses to fields with prefix increment or decrement operations. We are not sure of why the read access to the field `m_level` of class *IntSetSkipList* at line 115 is not reported by *RacerD*.

In the *IntSetSkipList* class our tool detected 1 false write on the constant `s_random` of the class due to a flaw that causes field declarations with initialization after a method *m* to be considered as a field update being done by *m*. While this flaw does not affect the results in this class since it occurs on a constant, it can affect the analysis in other cases. This means that, while our tool can already detect all of the field accesses, it still needs to be improved to be usable in any real case scenarios.

5

Conclusions

In this chapter we expose our conclusions derived from the results of our solution in section 5.1 and talk about the work that should be done in the future in section 5.2.

5.1 Conclusions

While there is still an issue that can cause false accesses to be detected in certain cases, we managed to detect all the conflicting accesses and data races in the fields of tested examples. Access detection still needs to be improved to distinguish accesses to different indexes. Despite the lack of concurrency and the limitations in the context-sensitivity of our implementation, the average runtime was found to be reasonable.

We therefore conclude that it is possible to use static analysis to perform PTA for detecting data races without the need for annotations in the code. While the runtime on small examples is acceptable, unfortunately, we did not have the time to test our solution against bigger examples to find out how much the lack of concurrency affects the runtime in larger code bases. Despite the imperfections of this implementation, it has the potential to help programmers make their concurrent programs work correctly.

5.2 Future Work

The tool we created can help find data races, but the issue with false accesses needs to be fixed, and the tool needs to be optimized. While we have yet to test this implementation with larger code examples, in theory, it should be multi-threaded scale properly for such examples.

Our plan for the future, once we have improved this implementation of our PTA, is to fully integrate it with the *AtomiS* [23] tool in such a way that we will be able to both analyse, infer and inject the concurrency control code into Java programs that do not have it.

Bibliography

[1] M. Abadi, C. Flanagan, and S. N. Freund. "Types for safe locking: Static race detection for Java". In: *ACM Trans. Program. Lang. Syst.* 28.2 (2006), pp. 207–255. DOI: 10.1145/1119479.1119480 (cit. on pp. 12, 24, 25).

[2] R. H. Arpaci-Dusseau. "Operating Systems: Three Easy Pieces". In: *login Usenix Mag.* 42.1 (2017) (cit. on p. 2).

[3] S. Blackshear et al. "RacerD: compositional static race detection". In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018), 144:1–144:28. DOI: 10.1145/3276514 (cit. on pp. 22, 24, 25, 64).

[4] M. Christiaens and K. D. Bosschere. "TRaDe: Data Race Detection for Java". In: *Computational Science - ICCS 2001, International Conference, San Francisco, CA, USA, May 28-30, 2001. Proceedings, Part II*. Ed. by V. N. Alexandrov et al. Vol. 2074. Lecture Notes in Computer Science. Springer, 2001, pp. 761–770. DOI: 10.1007/3-540-45718-6_81 (cit. on p. 9).

[5] P. Cousot. *An Informal Overview of Abstract Interpretation*. 2005. URL: https://web.mit.edu/afs/athena.mit.edu/course/16/16.399/www/lecture_01-intro/Cousot_MIT_2005_Course_01_4-1.pdf (visited on 2023-07-08) (cit. on p. 7).

[6] D. Detlefs, G. Nelson, and J. B. Saxe. "Simplify: a theorem prover for program checking". In: *J. ACM* 52.3 (2005), pp. 365–473. DOI: 10.1145/1066100.1066102 (cit. on p. 14).

[7] *Eclipse Java development tools (JDT)*. URL: https://eclipse.dev/jdt/ (visited on 2023-07-12) (cit. on p. 60).

[8] D. R. Engler and K. Ashcraft. "RacerX: effective, static detection of race conditions and deadlocks". In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*. Ed. by M. L. Scott and L. L. Peterson. ACM, 2003, pp. 237–252. DOI: 10.1145/945445.945468 (cit. on pp. 8, 15, 24, 25).

[9] *facebook/infer GitHub*. 2019. URL: https://github.com/facebook/infer/blob/main/infer/tests/codetoanalyze/java/racerd/GuardedByTests.java (visited on 2023-07-12) (cit. on p. 23).

[10] C. Flanagan and S. N. Freund. "Detecting race conditions in large programs". In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001*. Ed. by J. Field and G. Snelting. ACM, 2001, pp. 90–96. DOI: 10.1145/379605.379687 (cit. on pp. 8, 12, 13, 25).

[11] C. Flanagan and S. N. Freund. "Type-based race detection for Java". In: *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*. Ed. by M. S. Lam. ACM, 2000, pp. 219–232. DOI: 10.1145/349299.349328 (cit. on pp. 8, 11, 14, 16, 25).

[12] C. Flanagan, S. N. Freund, and S. Qadeer. "Thread-Modular Verification for Shared-Memory Programs". In: *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*. Ed. by D. L. Métayer. Vol. 2305. Lecture Notes in Computer Science. Springer, 2002, pp. 262–277. DOI: 10.1007/3-540-45927-8_19 (cit. on pp. 8, 13, 15, 25).

[13] C. Flanagan et al. "Modular verification of multithreaded programs". In: *Theor. Comput. Sci.* 338.1-3 (2005), pp. 153–183. DOI: 10.1016/j.tcs.2004.12.006 (cit. on pp. 14, 25).

[14] V. Kahlon et al. "Static data race detection for concurrent programs with asynchronous calls". In: *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*. Ed. by H. van Vliet and V. Issarny. ACM, 2009, pp. 13–22. DOI: 10.1145/1595696.1595701 (cit. on pp. 8, 17, 25).

[15] G. Korland, N. Shavit, and P. Felber. "Noninvasive concurrency with Java STM". In: *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*. 2010 (cit. on p. 65).

[16] A. van der Lee. *Race condition vs. Data Race: the differences explained*. 2021. URL: https://www.avanderlee.com/swift/race-condition-vs-data-race/ (visited on 2023-04-13) (cit. on p. 3).

[17] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. ii).

[18] S. Lu et al. "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics". In: (2008). Ed. by S. J. Eggers and J. R. Larus, pp. 329–339. DOI: 10.1145/1346281.1346323 (cit. on p. 4).

[19] D. Marino, M. Musuvathi, and S. Narayanasamy. "LiteRace: effective sampling for lightweight data-race detection". In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. Ed. by M. Hind and A. Diwan. ACM, 2009, pp. 134–143. DOI: 10.1145/1542476.1542491 (cit. on pp. 9, 10).

[20] M. Naik, A. Aiken, and J. Whaley. "Effective static race detection for Java". In: *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*. Ed. by M. I. Schwartzbach and T. Ball. ACM, 2006, pp. 308–319. DOI: 10.1145/1133981.1134018 (cit. on pp. 8, 16, 25).

[21] D. Neves and H. Paulino. "Condition-based synchronization in data-centric concurrency control". In: *SAC '22: The 37th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, April 25 - 29, 2022*. Ed. by J. Hong et al. ACM, 2022, pp. 1268–1275. DOI: 10.1145/3477314.3507120 (cit. on p. 6).

[22] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999. ISBN: 978-3-540-65410-0. DOI: 10.1007/978-3-662-03811-6 (cit. on p. 7).

[23] H. Paulino et al. "AtomiS: Data-Centric Synchronization Made Practical". In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (2023), pp. 116–145. DOI: 10.1145/3622801 (cit. on pp. 6, 60, 71).

[24] H. Paulino et al. "From atomic variables to data-centric concurrency control". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*. Ed. by S. Ossowski. ACM, 2016, pp. 1806–1811. DOI: 10.1145/2851613.2851734 (cit. on p. 6).

[25] H. Paulino et al. "Sound Atomicity Inference for Data-Centric Synchronization". In: *CoRR* abs/2309.05483 (2023). DOI: 10.48550/ARXIV.2309.05483. arXiv: 2309.05483 (cit. on p. 6).

[26] Q. Phan, P. Malacaria, and C. S. Pasareanu. "Concurrent Bounded Model Checking". In: *ACM SIGSOFT Softw. Eng. Notes* 40.1 (2015), pp. 1–5. DOI: 10.1145/2693208.2693240 (cit. on pp. 8, 20, 25).

[27] C. Radoi and D. Dig. "Practical static race detection for Java parallel loops". In: *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*. Ed. by M. Pezzè and M. Harman. ACM, 2013, pp. 178–190. URL: https://doi.org/10.1145/2483760.2483765 (cit. on pp. 8, 19, 25).

[28] J. Regehr. *Race Condition vs. Data Race*. 2011. URL: https://blog.regehr.org/archives/490 (visited on 2023-04-13) (cit. on pp. 3, 4).

[29] D. Rhodes, C. Flanagan, and S. N. Freund. "BigFoot: static check placement for dynamic race detection". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by A. Cohen and M. T. Vechev. ACM, 2017, pp. 141–156. DOI: 10.1145/3062341.3062350 (cit. on p. 11).

[30] H. K. Swamy. "Structured parallel programming patterns for efficient computation by Michael McCool, Arch D. Robison and James Reinders". In: *ACM SIGSOFT Softw. Eng. Notes* 37.6 (2012), p. 43. DOI: 10.1145/2382756.2382773 (cit. on pp. 1, 2, 5).

[31] *WALA Wiki*. 2019. URL: https://wala.sourceforge.net/wiki/index.php/Main_Page (visited on 2023-07-12) (cit. on p. 20).

[32] *What are Microservices?* URL: https://aws.amazon.com/microservices/ (visited on 2023-04-19) (cit. on p. 2).

[33] *What is Microservices Architecture?* URL: https://cloud.google.com/learn/what-is-microservices-architecture (visited on 2023-04-19) (cit. on p. 2).

[34] Y. Yu, T. Rodeheffer, and W. Chen. "RaceTrack: efficient detection of data race conditions via adaptive tracking". In: *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*. Ed. by A. Herbert and K. P. Birman. ACM, 2005, pp. 221–234. DOI: 10.1145/1095810.1095832 (cit. on pp. 7, 9, 10).

I

Auxiliary Points-To Analysis Functions

The *visitDoWhileLoop* function in Algorithm 26 is very similar to the *visitWhileLoop* function in algorithm 6 and only differs on the value the *doLoop* boolean initially takes at line 8. Since *doLoop* takes into account the values of both the *blockCondition* and *i*, it needs to be updated at lines 13 and 20 regardless of the resource values of *p2aData* having changed or not.

Algorithm 27 shows the auxiliary function used to visit for loop blocks. It begins by retrieving the statements and the condition at the head of the block *b* at lines 2 and 6, then visits them at lines 4 and 7. Afterward, the function retrieves the block body and the operation at the head of the block at lines 9 and 10, before preparing to visit the block body statements. As the body is visited at each iteration, besides checking for meaningful changes in the *p2aData* struct, the function also visits the operation at line 23 and reevaluates the condition at line 24.

Algorithm 26 Visiting Do-While Loop Blocks

```
1: function visitDoWhileLoop(p2aData : dataStruct, b : doWhileLoop)
2:     ce ← conditionExpression(b)
3:     blockCondition ← visitE(p2aData, ce)
4:     L ← getVarsMap(p2aData)
5:     body ← blockBody(b)
6:     pushCondition(p2aData, blockCondition)
7:     oldData ← copyOf(p2aData)
8:     doLoop ← true
9:     i ← 0
10:    while doLoop do
11:        L ← L ∪ visitBlock(p2aData, body)
12:        if isEquivalent(p2aData, oldData) then
13:            doLoop ← mayBeTrue(blockCondition) and i < 10
14:            i ← i + 1
15:        else
16:            oldData ← copyOf(p2aData)
17:            blockCondition ← visitE(p2aData, ce)
18:            popCondition(p2aData)
19:            pushCondition(p2aData, blockCondition)
20:            doLoop ← mayBeTrue(blockCondition) and i < 10
21:            i ← 0
22:        end if
23:    end while
24:    popCondition(p2aData)
25:    return L
26: end function
```

Algorithm 27 Visiting For Loop Blocks

```
1: function visitForLoop(p2aData : dataStruct, b : forLoop)
2:     sl ← getStatements(b)
3:     for s ∈ sl do
4:         visitS(p2aData, s)
5:     end for
6:     ce ← conditionExpression(b)
7:     blockCondition ← visitE(p2aData, ce)
8:     L ← getVarsMap(p2aData)
9:     body ← blockBody(b)
10:    operation ← blockOp(b)
11:    doLoop ← mayBeTrue(blockCondition)
12:    i ← 0
13:    while doLoop and i < 10 do
14:        pushCondition(p2aData, blockCondition)
15:        oldData ← copyOf(p2aData)
16:        L ← L ∪ visitBlock(p2aData, body)
17:        if isEquivalent(p2aData, oldData) then
18:            i ← i + 1
19:        else
20:            i ← 0
21:        end if
22:        popCondition(p2aData)
23:        visitE(p2Data, operation)
24:        blockCondition ← visitE(p2aData, ce)
25:        doLoop ← mayBeTrue(blockCondition)
26:    end while
27:    return L
28: end function
```