

# Automatic Fault Localization in Concurrent Programs using Noising and Search Strategies

by

Luisa Fernanda Rojas Garcia

A thesis submitted to the  
School of Graduate and Postdoctoral Studies in partial  
fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

Faculty of Science

University of Ontario Institute of Technology (Ontario Tech University)

Oshawa, Ontario, Canada

May 2020

© Luisa Fernanda Rojas Garcia, 2020

# Thesis Examination Information

Submitted by: **Luisa Fernanda Rojas Garcia**

**Master of Science in Computer Science**

**Thesis title:** Automatic Fault Localization in Concurrent Programs using Noising and Search Strategies

An oral defence of this thesis took place on May 26, 2020 in front of the following examining committee:

## **Examining Committee:**

Chair of Examining Committee

KQ Pu

Research Supervisor

Jeremy S. Bradbury

Supervisory Committee Member

João Lourenço

Thesis Examiner

Heidar Davoudi

The above committee determined that the thesis is acceptable in form and content and that a satisfactory knowledge of the field covered by the thesis was demonstrated by the candidate during an oral examination. A signed copy of the Certificate of Approval is available from the School of Graduate and Postdoctoral Studies.

# Abstract

Multithreaded programs can have considerable performance benefits over sequential programs. However, these advantages often come at a cost with respect to program understandability as well as testing and debugging challenges. To address these challenges we have developed CFLASH (Concurrency Faults Localized Automatically using Search Heuristics), an automatic fault localization tool for multithreaded Java programs. CFLASH utilizes a combination of noise-based code injection and a heuristic search algorithm to identify potentially faulty code sections containing concurrency bugs. We demonstrated the effectiveness of CFLASH by localizing concurrency faults in a set of benchmarked concurrent programs as well as student programs collected at Ontario Tech University.

**Keywords:** concurrency; Java; bug; noise; fault localization

# Author's Declaration

I hereby declare that this thesis consists of original work of which I have authored. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize the University of Ontario Institute of Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research. I further authorize University of Ontario Institute of Technology to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research. I understand that my thesis will be made electronically available to the public.

The research work in this thesis was performed in compliance with the regulations of Ontario Tech's Research Ethics Board under REB Certificate number 15672.

---

**Luisa Fernanda Rojas Garcia**

# Statement of Contributions

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication. I have used standard referencing practices to acknowledge ideas, research techniques, or other materials that belong to others. Furthermore, I hereby certify that I am the sole source of the creative works and/or inventive knowledge described in this thesis.

# Acknowledgements

First and foremost, I would like to thank my family for their unconditional love, support, and opportunities that resulted in me being where I am today.

I wish to also express my gratitude to my boyfriend, Alexandar Mihaylov, for keeping me sane and for his constant encouragement to achieve my goals and dreams.

Special thanks to my supervisor, Jeremy S. Bradbury, for his mentorship and guidance throughout the development of this work. Similarly, I want to thank professor João Lourenço for his invaluable ideas and contributions. Both of their support was key in the completion of this thesis.

Moreover, thank you to Heidar Davoudi, my thesis examiner, for his questions and insights during my defense.

Lastly, I also want to acknowledge my friends and colleagues at Ontario Tech who have, in one way or another, also contributed to this accomplishment.

# Table of Contents

<b>Abstract</b>	<b>iii</b>
<b>Author’s Declaration</b>	<b>iv</b>
<b>Statement of Contributions</b>	<b>v</b>
<b>Acknowledgements</b>	<b>vi</b>
<b>Table of Contents</b>	<b>vii</b>
<b>List of Tables</b>	<b>x</b>
<b>List of Figures</b>	<b>xi</b>
<b>Listings</b>	<b>xii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation and Problem Definition . . . . .	1
1.2 Thesis Statement . . . . .	3
1.3 Contributions . . . . .	4
1.4 Thesis Organization . . . . .	5

<b>2</b>	<b>Background and Related Work</b>	<b>7</b>
2.1	Concurrency . . . . .	7
2.1.1	Definition . . . . .	7
2.1.2	Concurrency Bugs . . . . .	9
2.1.3	Bug-Detection Techniques . . . . .	15
2.2	Java Concurrency . . . . .	19
2.2.1	Threads . . . . .	19
2.2.2	Concurrency Mechanisms . . . . .	21
<b>3</b>	<b>System Implementation</b>	<b>25</b>
3.1	Overview and Architecture . . . . .	25
3.2	User Input . . . . .	27
3.2.1	Configuration File . . . . .	27
3.2.2	Script Execution . . . . .	29
3.3	Annotation Phase . . . . .	30
3.3.1	Annotation Targets . . . . .	32
3.4	Search Phase . . . . .	35
3.4.1	Thresholds . . . . .	37
3.4.2	Noise . . . . .	37
3.4.3	Test Phase . . . . .	42
3.5	Output Logging . . . . .	43
<b>4</b>	<b>Evaluation</b>	<b>45</b>
4.1	Experimental Setup . . . . .	45
4.2	Search Correctness . . . . .	49

4.3	Threats to Validity . . . . .	51
<b>5</b>	<b>Conclusion</b>	<b>52</b>
5.1	Summary . . . . .	52
5.2	Limitations . . . . .	53
5.3	Future Work . . . . .	54
5.3.1	Further Experimentation . . . . .	54
5.3.2	Optimization . . . . .	55
5.3.3	Machine Learning . . . . .	55
5.3.4	Automatic Test Generation . . . . .	56
<b>A</b>	<b>Command-Line User Input Flags</b>	<b>67</b>
<b>B</b>	<b>Binary Search Algorithm</b>	<b>68</b>
<b>C</b>	<b>Research Ethics Board Approval</b>	<b>70</b>
<b>D</b>	<b>University Course Laboratory Description</b>	<b>72</b>
<b>E</b>	<b>ConMAN Concurrency Mutation Operators for Java</b>	<b>74</b>
<b>F</b>	<b>Sample CFLASH Output</b>	<b>76</b>

# List of Tables

4.1	CFLASH Evaluation Data . . . . .	46
4.2	Concurrency Bug Patterns vs. Concurrency Mutation Operators [7] .	47
4.3	Subset of ConMAN Mutation Operators [7] . . . . .	48
4.4	CFLASH Evaluation Results . . . . .	49
4.5	CFLASH Evaluation Results for Non-Buggy Projects . . . . .	51
E.1	ConMAN Mutation Operators [7] . . . . .	75

# List of Figures

2.1	Time Slicing in Concurrency . . . . .	8
2.2	Threads Shared Memory . . . . .	11
3.1	CFLASH System Architecture . . . . .	26
3.2	Visualization of Search Strategy . . . . .	36

# Listings

2.1	AccountManage.java thread class from <b>account</b> project . . . . .	14
2.2	Account.java class from <b>account</b> project . . . . .	14
2.3	Java Synchronized Method Syntax . . . . .	22
2.4	Java Synchronized Block Syntax . . . . .	22
3.1	Sample User Configuration File . . . . .	28
3.2	TXL Rule for Annotation of Java Synchronized Methods . . . . .	31
3.3	Annotated Synchronized Method Example . . . . .	33
3.4	Annotated Synchronized Block Example . . . . .	34
3.5	Annotated Statement Accessing Global Variables . . . . .	34
3.6	Annotated Statement Accessing Object Parameter . . . . .	35
3.7	Noised Synchronized Method Example . . . . .	39
3.8	Noised Synchronized Block Example . . . . .	41
3.9	Noised Statement Example . . . . .	42
F.1	Sample CFLASH Output . . . . .	76

# Chapter 1

## Introduction

### 1.1 Motivation and Problem Definition

With the introduction of multi-core processors into our everyday devices, concurrency has become a crucial programming paradigm in the development of new software. Concurrency and parallelism can be very effective at leveraging the performance benefits of multi-core processors by enabling programs to execute various actions simultaneously on different cores on the same machine [10]. This concept of parallelism affects every aspect of modern programming; for example, it is given rise to the ability of having multiple computers on a network, running multiple applications on a single computer, or even web sites managing many various users at once [29].

Nonetheless, the performance benefits of concurrency come at a cost. Developing concurrent programs can be very challenging because of their non-deterministic nature — the output of a program depends not only on its inputs but also on the thread interleavings that dictate the order in which certain tasks are executed [29, 46, 62]. In turn, developers must consider areas in the program where data is shared between

threads, as well as the order in which it is accessed.

Managing such a wide range of variables in execution scenarios can also make the testing and debugging of processes a difficult task [12]. Concurrency bugs such as data races [18], deadlocks [4] and atomicity violations [45] are, unfortunately, easily introduced but are very difficult to discover and fix.

Often times, concurrency-related bugs result in **latent** bugs, which are not immediately visible to users when triggered; instead, they silently corrupt the internal data structures of the system, and only become externally visible much later, usually in a different environment than the one it was developed in [25]. Developers are in need of reliable tools they can use to effectively detect and localize concurrency bugs.

In this thesis, we propose an approach that could support developers working on concurrent programs by automatically detecting and localizing concurrency bugs. To accomplish this we developed a tool, CFLASH, that takes in a concurrent Java program as one of its inputs and is able to locate the code section where a potential bug may be present. CFLASH is composed of a series of modules that employ noising and search heuristics to determine an appropriate output.

First, the Java program received as input is annotated by automatically identifying all concurrency-related mechanisms (e.g. synchronized blocks and methods) as well as unsafe data access statements; then, such annotations are replaced by thread delays at the source code level. Finally, the transformed program is tested against a user-provided test suite in order to narrow down the location of the concurrency bug in question using a binary search approach.

By noising the program using thread delays, we are more likely to explore more diverse thread interleavings — some of which may cause the bug to occur. More-

over, given the modular architecture of CFLASH, the tool could be adjusted and be applied to different programming languages by using its corresponding grammar instead and leaving the noise and search heuristics unchanged.

## 1.2 Thesis Statement

**Thesis Statement:** *Using automated noising and search heuristics is an effective method in localizing concurrency bugs in multithreaded Java programs.*

This thesis presents a tool that capable of localizing concurrency bugs by identifying points in a program where a thread is likely to be interrupted by another; when such action occurs, it is known as a **context switch**. By introducing thread delays, or **noise**, around areas where context switches are likely, we can explore different thread schedules by executing the same program with the same tests multiple times and considering the schedules that cause a bug to surface. Finally, a **search heuristic** based on a binary search strategy is used to narrow down the location of a concurrency bug in the program by considering the failure rate when testing with noise at different context switches.

We measure the effectiveness of our approach by evaluating it using programs from several sources, including the IBM Concurrency Benchmark [20–22, 35], textbook examples [30, 31], the Software Infrastructure Repository (SIR) [14] and student programs from the Massively Parallel Computing course at Ontario Tech University.

## 1.3 Contributions

The goal of this research is to develop a tool that can automatically noise, test and search a given Java program with the objective of discovering and localizing unintended concurrency faults.

This thesis has four main contributions:

1. **CFLASH**, a tool that performs automatic concurrent fault localization on multithreaded Java programs. The project aims to address the challenges that come with the development of concurrent programming, including difficulty testing and debugging due to the non-deterministic nature of concurrent programs. This thesis describes the overall architecture of **CFLASH** and the parts it is composed of, such as: annotating (Section 3.3), noising (Section 3.4.2) and searching (Section 3.4).
2. A **bug localization algorithm** that utilizes a binary search strategy — the most robust apparatus within **CFLASH**. It was designed to systematically inject annotations, noise and test the program in question to narrow down the likelihood of a concurrency bug being located in a given scope. This approach is based on previous findings by Ben-Asher et al. and Edelstein et al. regarding the optimal placement of noise that causes a concurrent bug to manifest [3, 16].
3. As part of **CFLASH**, we also introduce a new **noising tool** that targets critical regions and unsafe shared variable access in concurrent Java programs. Annotation and noising targets four specific types of structures: synchronized methods, synchronized blocks, potentially unsafe statements that access a global variable,

and potentially unsafe methods that access a local object variable passed to a method as a parameter. This noising task can be easily adapted to other programming languages with very little changes to the core of CFLASH.

4. A new **data set** for evaluating tools that support concurrent program development. The data consists of a set of 31 programs that were collected, parsed and anonymized for use in this research. These programs were submitted as part of the course “Massively Parallel Computing” at Ontario Tech University and will be made public with the authors’ consent.

## 1.4 Thesis Organization

The organization of the remainder of the thesis is as follows:

- **Chapter 2:** Provides the background required to understand the basis on which this project stands. Such context information includes the fundamental concepts of concurrency: what it means and how it works. Moreover, we discuss some of the most commonly occurring concurrency bugs and a few tools that have been developed in the past to help avoid or repair them.
- **Chapter 3:** Describes the development and internal architecture of the tool implemented: CFLASH. We examine each of the main components of CFLASH in detail and explain the noise approach taken. Finally, we introduce the binary search algorithm designed to uncover and localize bugs in a concurrent Java program (appendix B).
- **Chapter 4:** In this section, we provide insights into the data used for the

experimentation portion of this research, together with the experimental set up used to test the tool developed. We, then, present the experiment results and discuss CFLASH’s capabilities and effectiveness in localizing different types of bugs across projects selected.

- **Chapter 5:** The concluding chapter summarizes the research presented and considers the possible limitations related to it. Lastly, we discuss different areas for future work, including further evaluation in multiple bug localization, overall optimization of the tool, and opportunities in machine learning.

# Chapter 2

## Background and Related Work

This chapter covers an overview of the background required to understand the work presented in this thesis. First, we present a synopsis of general concurrency concepts; that is, a formal definition of concurrency behaviour, its applications and the types of bugs that are typically exhibited in concurrent applications. We also discuss past work that has focused on concurrency-related bug detection. Then, we focus on concurrency mechanisms in Java, including synchronized blocks and methods.

### 2.1 Concurrency

#### 2.1.1 Definition

Complex activities in the real world can generally be broken into smaller, relatively more simple tasks. Consider the somewhat complex act of riding a bicycle, which involves the more elementary tasks of pedalling and steering. Performing both of these tasks at the same time is necessary in order to perform the more complex task

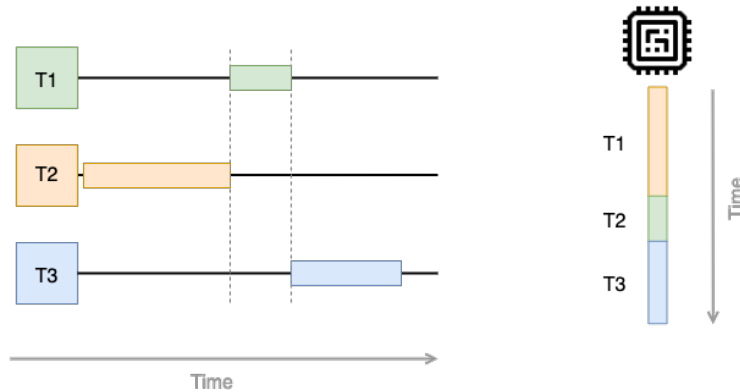


Figure 2.1: Time Slicing in Concurrency

of riding the bicycle. The simultaneous execution of the pedalling and steering tasks is said to be done in parallel, or more broadly, concurrently.

Computer programs are analogous in that they are complex activities that consist of simpler tasks that work together in an effort to make progress towards an objective. A sequential program has a single thread that executes tasks in sequence; concurrent programs, in contrast, consist of two or more threads that make progress within the same time frame [15, 29]. This does not imply parallelism, which states that both threads are executing at the same time.

A single-core processor is still able to achieve concurrency by context switching between threads, thus giving the illusion of parallel computation. For example, Figure 2.1 illustrates a system with one core and three threads. On the left, we can observe how the core switches between threads, while the right-hand image shows the time during which each thread gets CPU time to run. This naturally incurs some overhead, consequently diminishing some of the advantages offered by concurrency. Luckily, modern processors come equipped with multiple cores that enable processes and threads to perform in true parallel fashion, enabling users to gain performance

boosts. Naturally, multithreaded programs allow for concurrency, hence for the purposes of this work, both terms will be used interchangeably.

By increasing the parallelism in a system, the performance throughput of a concurrent program can be increased in proportion to the number of processors. However, there does exist an upper-bound maximum speedup that can be achieved by executing a program in multiple threads simultaneously, since programs will always include portions that cannot be parallelized and must be executed serially.

Amdahl’s law is an approximation used to model the maximum speedup that can happen when a serial program is modified to run in parallel. The expected speedup of parallel code over sequential code on a machine with  $n$  cores available is determined by the proportion of the program that *can* be parallelized,  $P$  and the proportion that cannot  $(1 - P)$  [11, 23]:

$$S(n) = \frac{1}{(1 - P) + \frac{P}{n}} \quad (2.1)$$

While there are seemingly inherent benefits resulting from the use of concurrency, they come along with a heavy cost for software developers and testers which will be discussed in section 2.1.2.

### 2.1.2 Concurrency Bugs

Ensuring the correctness of sequential software is in itself a great challenge. Ensuring the correctness of concurrent software is an even greater undertaking and requires extensive knowledge. The complexity and difficulty in analyzing the correctness of concurrency programs are, in large part, attributed to their non-deterministic nature; that is, the notion that two executions of the same software may behave differently despite having the same inputs [53]. The reason for the non-deterministic behaviour

is the large number of possible interleavings between threads (i.e. schedules), which often lead to a different sequence of executions at the hardware level — contrary to sequential programs, which tend to be deterministic and to always have the same sequence of low-level instructions given the same input.

The interleaving space of a concurrent program is far too large, making it unfeasible to explore all possible thread interleavings during testing. Also, a small subset of thread schedules can produce incorrect results or have undesired side effects, despite the greater part of the interleaving space producing correct results. Consequently, testing and uncovering concurrency bugs (or faults) are also challenging tasks during which bugs go undetected.

The case of Therac-25 exemplifies the importance of software quality, testing, and the implications resulting from concurrency bugs. Therac-25 was a computer-controlled radiation therapy machine that ultimately overdosed six people with large amounts of radiation [41] and, in some cases, caused death [42]. The machine worked with the principle of using high-energy beams to remove unwanted tumour cells and keep the surrounding healthy tissue undamaged. The rotating turntable piece of the device could be in three different positions, where two were used as therapy modes while the other was merely used to correctly position the patient without the use of any type of beam [41]. Unfortunately, concurrency bug(s) persisted through previous iterations of the Therac machines and had gone undetected, resulting in unsafe access to a shared variable that was to track the completion of the technician’s initial machine setup. A specific sequence of commands exposed the race condition, which caused the machine mode configurations to not take place, or to only be fulfilled partially, despite the UI giving the illusion of doing so successfully. Without the correct

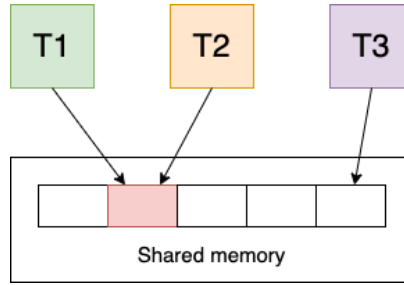


Figure 2.2: Threads Shared Memory

settings in place, the patients receiving radiation treatment were exposed to tremendous overdoses of radiation, resulting in four deaths. Investigations concluded that “the software allows concurrent access to shared memory” and that “race conditions resulting from this implementation of multitasking played an important part of the accidents” [42].

Concurrency is built into most modern-day programming languages, including Java. These languages provide a diverse range of tools that permit programmers to write concurrent code and greatly mitigate unwarranted bugs. Testing the correctness of a concurrent program is also a challenge since bugs can surface very unpredictably and often in critical situations, like in production or heavy load [28]. In order to effectively mitigate concurrency bugs, developers must first thoroughly understand the various types of bugs that may arise and that should actively be avoided. Some of the most common ones are explained below.

- **Data race:** A race condition, or data race, occurs when the data is corrupted due to inappropriate protection of data that is accessed by multiple threads and modified by at least one. For example, in Figure 2.2 two threads, 1 and 2, share the piece of data in red. Under these circumstances, the final result for each thread may depend on the order in which tasks were executed [32], as well as if

the threads interrupted one another during the process. Moreover, the *type* of data access is also relevant; for instance, if all threads involved are only reading from the shared memory location, then there should be no concerns. However, if at least one of the threads is modifying the data being accessed by others, then the output could be impacted.

Listings 2.1 and 2.2 contain a modified version of the program `account`, a banking system simulation from the IBM Benchmark [20–22, 35]. The program is mainly comprised of two classes: `Account` and `ManageAccount`. The latter, `ManageAccount`, extends the `Thread` class and is instantiated from the main method as a thread. The `run` method in this class defines the behaviour for each thread that will be created; in this case, the task is to perform three transactions on the `Account` they are to manage: (1) deposit, (2) transfer to another account, and (3) withdraw.

The flaw lies in the `transfer` method under the `Account` class in Listing 2.2. This method consists of two crucial steps: removing the funds from the outgoing account and adding them to the outgoing account. The `transfer` is synchronized on the current `Account` object; nevertheless, this is not sufficient, since there are **two** objects involved in the transaction and in need of a second synchronization mechanism (see Section 2.2.2.1 for more details) to protect against other threads attempting to access an account in the middle of a transfer.

- **Deadlock:** Happens when the execution of a program results in a state in which no thread or component is able to make any progress. This behaviour is typically due to multiple threads waiting for resources that other threads hold, while also holding resources needed by those threads. A deadlock occurs when

the Coffman conditions are met [32]:

- Only one thread can access a shared resource at a time.
  - A thread has the lock for a resource and it is requesting the lock for another resource. While waiting, it does not release any resources.
  - The resources can only be released by the threads that hold them.
  - There is a circular wait, each thread involved is waiting on a resource held by another and thread  $n$  would be waiting for a resource being held by thread 1.
- **Livelock:** A system is livelocked when two or more threads repeatedly change their state due to the actions of another. This behaviour can result in an infinite loop of state changes [30]. Livelocked systems are similar to deadlocked ones in that they get stuck and fail to make any progress; however, threads in a livelocked system may appear to be active and progressing externally, while they are failing internally [53].
  - **Starvation:** Resource starvation arises when a thread in the system never gets access to a resource that it needs to continue with its execution. Often times, this is due to an unfair algorithm used in selecting the next thread to acquire a resource once it has been released [32].

Concurrency provides major performance advantages, but also provides major responsibilities for software testers and developers to be accountable for ensuring edge cases do not go undetected, like with Therac-25. It is important that we not only educate developers on coding practices and synchronization techniques but also on

supplementary tools that are able to aid in identifying potential concurrency bugs that may otherwise be missed. This thesis aims to achieve the latter by highlighting potentially problematic areas of code, giving developers greater certitude in the intended operation of a system.

---

Listing 2.1: AccountManage.java thread class from account project

---

```
1 public class ManageAccount extends Thread {
2     private Account[] bank;
3     private Account account;
4
5     public ManageAccount(Account account, Account[] bank) {
6         super("T" + account.name);
7         this.account = account;
8         this.bank = bank;
9     }
10
11    public void run() {
12        int currIndex = -1;
13        for (int i = 0; i < bank.length; i++) {
14            if (bank[i] == this.account) {
15                currIndex = i;
16                break;
17            }
18        }
19
20        if (currIndex < 0) System.exit(-1);
21
22        this.account.deposit(220);
23        this.account.transfer(bank[(currIndex+1)%this.bank.length], 20);
24        this.account.withdraw(20);
25    }
26 }
```

---

---

Listing 2.2: Account.java class from account project

---

```
1 public class Account {
2     public String name;
3     public int number;
4     public double balance;
5 }
```

```

6     public Account(String name, int number, double balance) {
7         this.name = name;
8         this.number = number;
9         this.balance = balance;
10    }
11
12    synchronized void deposit(double amount) { this.balance += amount; }
13    synchronized void withdraw (double amount) { this.balance -= amount; }
14    synchronized void transfer(Account toAccount, double amount) {
15        if (toAccount == this) return;
16        this.balance -= amount;
17        toAccount.balance += amount; // Dangerous operation
18    }
19 }

```

---

### 2.1.3 Bug-Detection Techniques

We identified, at least, two major categories of tools related to debugging and testing of multithreaded programs. The first category aims to *detect* whether there is a fault in a program, while the second one also has the intention of *localizing* the origin of the fault. The latter would typically indicate where the suspected buggy code is within the source code in order to support developers in the implementation process. This research work focuses primarily on bug localization.

There has been considerable research done on the analysis of multithreaded programs aiming to detect both sequential and concurrency bugs in different programming languages; however, there is a greater focus in Java, likely given to its presence in industry and academia. The types of tools that have been developed for bug localization in concurrent programs can be broadly classified into static and dynamic techniques.

### 2.1.3.1 Static Bug Localization

When analysis is performed in a static manner, this indicates that the goal is to examine it in the absence of any input and without actually executing the program [2]. A static analysis approach trades off false positives for more scalability.

RELAY [60] is a static data race detection tool that focuses on low-level C programs — however, the authors consider it unsound because the tool ignores certain reads and writes in assembly code, does not consider corner pointer cases correctly, and uses a per-file analysis. Additionally, it is possible that the filters used to categorize likely races remove real races.

In an earlier work, Naik, Aiken, and Whaley developed CHORD [49] (originally named JCHORD), considered to be a pioneer static race detector for Java. Their approach makes use of call-graph construction, alias analysis, thread-escape analysis, and lock analysis to find pairs of memory accesses potentially involved in a race. It is worth noting that the project is no longer maintained.

Another static analysis publication is RACERX [19], a tool developed by Stanford University that gathers program information such as the locks and the operations it protects, multithreaded contexts, and dangerous shared accesses. By tracking these features, it can identify data races and deadlocks in C and sort the errors found from most to least severe.

FINDBUGS is one of the most popular tools developed for bug detection on sequential Java programs. FINDBUGS searches for suspicious patterns and performs instrumentation at the bytecode level. Like JCHORD, this tool is no longer maintained.

Examples of more recent research include RACERD [5] and SWORD [43]. RAC-

ERD was developed as a data race detector for Java that prioritized speed and scalability and depends on human annotations to detect bugs in code segments. RACERD was used also in migrating Facebook’s main Android app from a single-threaded to multi-threaded architecture. Finally, SWORD is one of the most recent static data race detection tools for Java, presented in the form of a plugin for the Eclipse IDE. Starting from an entry point, the tool traverses all reachable methods in the program’s call graph and records the events in its own static “happens-before”<sup>1</sup> (SHB) graph. SWORD, then, checks all shared memory accesses registered to determine if any data races exist. The authors of SWORD have reported that the tool has comparable performance with RACERD for most benchmarks evaluated on, as well as more precise detection results at the whole-program level.

### 2.1.3.2 Dynamic Bug Localization

Dynamic tools are able to visit only feasible paths (i.e. actual execution paths), therefore having a more accurate view of the value and variable relations in a program; however, the trade-off is a very high computational price. It can be very time consuming to run test cases and nearly impossible to use on programs that have timing limitations or requirements [19].

Lockset algorithms are commonly used in dynamic analysis techniques. They compute the set of locks that are held per thread, per variable in memory. Then, the sets are used to derive the set of locks that were held by determining if the set intersection is empty or not [6, 39, 60]. Nonetheless, the disadvantage in this

---

<sup>1</sup>A happens-before relationship is an assurance that an action performed by a statement is visible from another statement. In the context of multithreaded programs, this means that the consequences of a writing action by one thread are guaranteed to be visible to a read operation by another thread *if* the initial write operation *happens-before* the read operation [33, 37].

approach and others like it is that it often suffers from false positives compared to other methods [61].

The lockset algorithm is the strategy used by ERASER[54], one of the best known dynamic tools for data race detection. ERASER dynamically tracks the set of locks held during execution and uses them to compute their intersection when accessing a shared state. Since it was published, many other research works have used it and incorporated it into their own tools, like JAVA PATHFINDER (JPF) [8, 34] did. JPF is considered an explicit state model checker that detects deadlocks and exception violations by default, but custom properties can be defined manually by the user. The tool operates at the bytecode level and contains a run-time analysis mode that utilizes the ERASER algorithm to check whether locks have been taken in different orders [47]. Compared to static approaches, model checking can be more precise but does not scale well to larger programs [36].

Another tool based on dynamic techniques is RACEFUZZER [56], which proposes an approach to leverage information obtained from an existing hybrid-analysis tool [50] to, first, *create* a race condition and then reveal it using a random thread scheduler. One year later, THREADSANITIZER [57] introduced a tool for data races in multithreaded C/C++ code using a hybrid algorithm based on happens-before and locksets. Finally, the FASTTRACK [24] tool re-implemented ERASER and achieved better performance by tracking less information and adapting the original algorithm’s representation of the happens-before relation based on patterns pertaining to memory access.

## 2.2 Java Concurrency

The Java platform is designed to support concurrent programming through Java class libraries and basic concurrency support in the Java programming language, such as high-level concurrency APIs [59]. This section will discuss some of the Java mechanisms considered in this research.

### 2.2.1 Threads

Multithreaded programs contain two or more sections or tasks that can be executed concurrently, where each part of such a program is referred to as a **thread**. Each thread is considered to be the smallest unit of dispatchable code within the program, which also defines different execution paths within it [55].

There are several states a thread can be in; it can be ready to execute as soon as there is CPU time available, or it could be running, or a running thread could be suspended — which would later be resumed. Moreover, a running thread could be blocked when waiting for a resource, or it could be terminated, in which case its execution finishes permanently [55].

The Java multithreaded mechanisms are built on the **Thread** class and the **Runnable** interface, either of which can be used to create a new thread object. The **Thread** class contains various methods that can be used to manage multithreaded operations<sup>2</sup>; for example:

- **void start()**: Triggers the thread execution; the Java Virtual Machine (JVM) calls the **run()** method for this thread object (see below). A thread can only

---

<sup>2</sup>As described on the `java.lang.Thread` documentation: <https://docs.oracle.com/javase/8/docs/api/index.html>.

be started once [38].

- **void run():** Method that is called and its contents executed when a thread is started. Is to be overridden by the developer upon extension of class **Thread** or implementation of interface **Runnable**.
- **static void sleep(long milliseconds):** Causes the thread being executed currently to temporarily cease execution for the amount of time specified [38].
- **final void wait():** Causes the thread being executed currently to wait until another thread makes a method call to **notify()** or **notifyAll()** [38] (see below).
- **final void notifyAll()** and **final void notifyAll():** The first wakes up one thread that is waiting on this object's monitor — if more than one thread is waiting, then one of them is chosen arbitrarily to be awakened. In contrast, the latter wakes up *all* threads that are waiting on this object's monitor [38].
- **final void join():** Waits for a given thread to terminate its execution [38].

Threads generally share state and memory, making it more effortless to switch between them compared to switching between processes [9]. The primary means of communication threads utilize is the sharing of access to fields and their references. While this type of communication is quite effective, it is also at risk of two types of errors possible: thread interference<sup>3</sup> and memory consistency<sup>4</sup> [59]. For this rea-

---

<sup>3</sup>**Thread interference** accounts for errors that are introduced as a result of multiple threads accessing shared data [59].

<sup>4</sup>**Memory inconsistency** characterizes errors that result from inconsistent views of shared memory [59].

son, it is necessary to manually manage shared resources amongst threads by using synchronization mechanisms.

## 2.2.2 Concurrency Mechanisms

### 2.2.2.1 Synchronization

In Java, the `synchronized` keyword is used as a method modifier or block type to mark a critical section in a multithreaded program to avoid undesired data access amongst multiple threads by using a **monitor**. Monitors (also referred to as *intrinsic locks*) are the central concept that synchronization in Java is based on; they control access to an object and work by implementing the idea of a lock [55].

When an object is locked by a thread, no other thread can access that object at that moment. Only when the current thread exits, the object's monitor becomes unlocked and is available for other threads to use [46, 55]. It is important to note that, since monitors are a part of every object in Java, then it is possible to synchronize on any object [55].

**Synchronized Methods** The general syntax for synchronized method is shown in Listing 2.3. In this example, a method with name `myMethod` is using the `synchronized` keyword as a modifier in its declaration. The parameters being passed to this particular method are not currently relevant.

When a thread calls a synchronized method it will automatically acquire the monitor for that method's object and release it when the method returns, including the case of an uncaught exception [59]. In the case of a static method, the thread would acquire the monitor for the Class object instead.

Listing 2.3: Java Synchronized Method Syntax

---

```
1 // Method declaration
2 [method modifiers] synchronized myMethod ([parameters]) {
3     // Safe method body
4 }
```

---

Listing 2.4: Java Synchronized Block Syntax

---

```
1 synchronized ([object]) {
2     // Safe code block
3 }
```

---

**Synchronized Blocks** Unlike synchronized methods, synchronized blocks require that the object providing the monitor be specified as a parameter [59].

The general syntax for synchronized method is illustrated in Listing 2.4. Here, `[object]` is a reference to an object whose monitor is being used to lock the section of code contained in the synchronized block.

Locks can be quite computationally expensive, considering the delays and overhead they create when waiting or blocking occurs. For this reason, it is a best practice to design a multithreaded program with as few critical sections as possible [46]. Moreover, it is typically preferred to use synchronized blocks as opposed to synchronized methods — their flexibility makes it much easier to manipulate locking scopes, so they can be as small as possible.

#### 2.2.2.2 `java.util.concurrent`

Some of the concurrency mechanisms provided by Java under the `java.util.concurrent.locks` package are outlined below. All descriptions and definitions are taken from the Java Platform Standard Ed. 8 API Documentation [38].

- **Explicit Lock:** By using the `ReentrantLock` class a similar behaviour to synchronized sections (see below) can be achieved. However, additional features such as timeouts are available.
  - `lock()` Acquire a lock.
  - `unlock()` Attempt to release a lock.
  - `isLocked()`: Check if the lock is currently being held by a thread.
- **Semaphore** Also known as a counting semaphore or a general semaphore, this tool is to maintain the count of the number of available permits. This indicates that the number of threads that are able to access a semaphore depends on the number of permits available.
  - `acquire()`: Acquires a permit from the current semaphore. If none are available, then this action is blocked.
  - `release()`: Releases a permit and returns it to the semaphore.
- **Latch** Allows one or more threads to wait until a set of tasks being performed in other threads complete. The latch object of type `CountDownLatch` is initialized with a count, which is decreased due to invocations of the `countDown()` method. All waiting threads are released when the count reaches zero.
  - `getCount()`: Gets the current latch count.
  - `countDown()`: Decrements the current count of the latch, and releases all waiting threads when reaches zero.
- **Barrier** Through the `CyclicBarrier` class, this aid allows a set of threads to wait for each other to reach a common point.

- `await()`: Waits until all threads have invoked `await` on this barrier.
  - `getNumberWaiting()`: Returns the number of threads that are waiting at the barrier point.
  - `reset()`: Resets the barrier object to its initial state.
- **Concurrent Data Structures and Atomic Variables** Through the Java Collections Framework, various additions are made available to developers, such as **concurrent data structures** (e.g. `BlockingQueue`, `ConcurrentMap`, `ConcurrentHashMap`) and **atomic variables** for atomic operations on single variables (e.g. `AtomicInteger`, `AtomicLong`). These collections help reduce overhead and avoid memory consistency errors [59].
  - **Synchronization**: The Java `synchronized` keyword delimits a method or a code block as a critical section. This means that only one thread at a time may hold the lock for a given code section in order to access it. Details on synchronized **blocks** and in Section 2.2.2.1.

# Chapter 3

## System Implementation

### 3.1 Overview and Architecture

In order to aid programmers with arduous debugging tasks that come with multi-threaded programs, we developed CFLASH, a tool that is able to automatically detect and localize concurrency faults. To accomplish this, CFLASH takes in a concurrent Java program as one of its inputs, automatically identifies, annotates and noises critical code regions in which data may be shared. Then, it proceeds to narrow down the localization of a bug utilizing a test suite designed to ensure program correctness. This test suite is also provided as input. We leverage heuristic noise injection such that we can explore varying scheduling scenarios of threads in the input program.

Figure 3.1 displays the overall architecture of CFLASH. It is divided into three major stages:

1. Extract and parse settings set by the user;

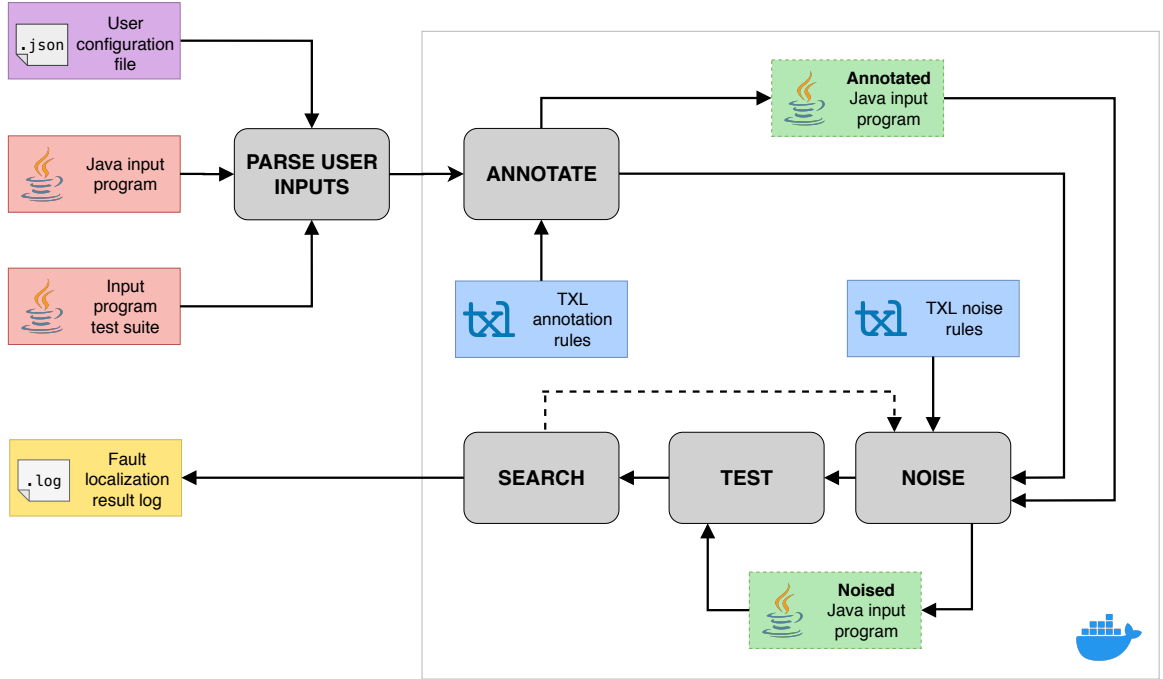


Figure 3.1: CFLASH System Architecture

2. Identify seemingly vulnerable areas in the program and introduce annotations;
3. Search for the code segment that is the most likely to have a bug in it by systematically noising and testing different sections of the program in question.

Most of the tools, libraries and frameworks used to develop CFLASH do not need to be installed on the host machine in order for it to run. Instead, we use a Docker image<sup>1</sup> where all the needed dependencies are installed: TXL<sup>2</sup>, Java, JUnit, Python, Pandas, pytz and regex. The only two programs needed on the host machine are Docker and Python 3.6 or newer. Furthermore, the modular architecture of CFLASH allows the tool to be adjusted to accommodate other programming languages by using

<sup>1</sup>Available for download from the Docker Hub at <https://hub.docker.com/repository/docker/lrogar/cflash>.

<sup>2</sup>Available at <http://txl.ca>.

their corresponding grammar instead, as well adapt the TXL noising rules to match that change.

User input, annotation, search and noising are the primary components that comprise CFLASH and will be described in detail in Sections 3.2, 3.3, and 3.4.

## 3.2 User Input

There are two ways in which the user can customize the use of CFLASH; through a JSON configuration file and through command-line flags and parameters. The user configuration file, discussed in Section 3.2.1, should contain the paths that CFLASH will need to locate the input Java project, as well as the thresholds and limits used in the bug localization search algorithm (see Section 3.4). On the other hand, command-line parameters will accept a series of optional flags in the interest of running time and high customization, in addition to the configuration file.

### 3.2.1 Configuration File

The first required user input is the JSON configuration file. While there is no particular location in which it should be kept, the path to it will be needed upon running CFLASH<sup>3</sup>

The configuration file must follow a pre-defined structure, which includes the required variables and values to customize the tool.

A sample configuration file with fake data is shown in Listing 3.1.

---

<sup>3</sup>For the sake of consistency and organization, it is strongly suggested this file is kept together with the Java project to be parsed and analyzed.

Listing 3.1: Sample User Configuration File

---

```
1 {
2   "root": {
3     "path": "/path/to/root/"
4   },
5   "src": {
6     "path": "path/to/src/"
7   },
8   "test": {
9     "test_suite": {
10       "path": "path/to/test/suite/",
11       "entrypoint": "test/suite/entrypoint"
12     },
13     "num_test_runs": 500,
14     "diff_threshold": 0.05
15   }
16 }
```

---

Below, we break down each of the configuration file parameters, including the expected type and its purpose.

- `root.path` (`str`): Path to the root directory containing the source code and test suite of the input project to be analyzed and provided by the user. Can be either absolute or relative to the CFLASH entrypoint.
- `src.path` (`str`): Path to the directory containing the main source code for the input project. **Relative to attribute** `root_path` above.
- `test.test_suite.path` (`str`): Path to the directory containing the set of unit tests for the input project. **Relative to** `root_path` above.
- `test.test_suite.entrypoint` (`str`): Name of the Java file that contains the main method in order to compile and run the test suite of interest.

- `test.num_test_runs (int)`: Number of times to run the given test suite. The average of such test runs are used to calculate the failure rate of a given search scope. The recommended amount is  $\geq 500$ , as suggested by Ben-Asher et al. and [3] Edelstein et al. on their tool ConTest [16] after careful testing and analysis.
- `test.diff_threshold (int)` Minimum difference between two failure rates corresponding to two separate search scopes in order to be considered different enough. In the case they *are* considered to be distinct, then one of them will be discarded; otherwise, both scopes will continue to be explored simultaneously. This value must be within the range  $[0, 1]$ .

### 3.2.2 Script Execution

Currently, CFLASH must be executed from the terminal using command-line arguments. The tool entry-point is the Python script `cflash.py`, which can be found at the root directory of the project.

```
$ python cflash.py [-h] [--debug] [--syncmethods] [--globalvars]
  [--objectvars] [--noisethreads] CONFIG_FILE_PATH
```

`CONFIG_FILE_PATH` is a required argument and corresponds to the relative or absolute path to the user configuration file (Section 3.2.1). The remainder of the flags are considered optional (denoted by the `[ ]` brackets) and correspond to the four noising targets available (Section 3.3.1); however, the user must specify at least one. See Appendix A for full details on all of the optional and required flags.

### 3.3 Annotation Phase

The annotation process is the first step in localizing faults CFLASH. The process statically analyzes the input program and identifies the most relevant targets for annotations in order to later noise them during the search step (Section 3.4). Each annotation follows the structure `@Noise@ID`, where “ID” is a unique identifying number for each added annotation. This will be key in the search and noising steps that are to follow.

There are four types of annotation targets that are considered:

1. **Synchronized methods:** Methods that use the `synchronized` keyword as a modifier in order to lock on that method’s object when executing the statements in the method body.
2. **Synchronized blocks:** A type of statement that wraps around a critical section using any locked object, which is to be passed as a parameter.
3. **Statements accessing global variables:** Any type of statement that accesses a global variable to read or to write.
4. **Statements accessing objects passed as parameters:** Any type of statement that accesses an object variable that has been passed as an argument to the current method to read or write.

TXL is a rule-based, pattern-matching language that is designed to allow explicit programmer control over transformation rules [13]. In this phase, the TXL rules written for CFLASH parse Java input and apply source code transformations on the

Listing 3.2: TXL Rule for Annotation of Java Synchronized Methods

---

```
1 rule annotateSyncMethods
2   replace $ [method_declaration]
3     METHOD_DECLARATION [method_declaration]
4   deconstruct METHOD_DECLARATION
5     OPT_NOISE_ANNOTATION [opt_noise_annotation] REPEAT_MODIFIER
6       [repeat_modifier] GENERIC_PARAMETER [opt
7         generic_parameter] TYPE_SPECIFIER [type_specifier]
8     METHOD_DECLARATOR [method_declarator] THROWS [opt
9       throws] METHOD_BODY [method_body]
10
11   where
12     REPEAT_MODIFIER [isSynchronized]
13   construct NOISE_ANNOTATION [modifier]
14     '@Noise '@0
15   by
16     NOISE_ANNOTATION
17     REPEAT_MODIFIER
18     GENERIC_PARAMETER
19     TYPE_SPECIFIER
20     METHOD_DECLARATOR
21     THROWS
22     METHOD_BODY
23 end rule
```

---

targets mentioned above to generate the annotations required. This version is not intended for compilation.

For example, the TXL rule shown in listing 3.2 is designed to search for all synchronized methods (lines 2-3) and replace them with a new one that includes a noise annotation (lines 8-17) of ID 0. In a separate rule, annotation IDs are assigned to a unique numerical value across all Java files in the project.

### 3.3.1 Annotation Targets

As mentioned previously, CFLASH has four types of annotation targets that the user can select from, which correspond to either synchronization events, or unsafe statements potentially accessing shared memory. Though alterable, the recommended default is to include all four so as to ensure the highest accuracy possible. If the user opts for less, they must include at least one. Let us further detail each below.

#### 3.3.1.1 Synchronized Methods

The Java 8 concurrency API encompasses a series of synchronization mechanisms that allow for the definition of critical sections and task synchronization at specific points. The **synchronized** keyword is one of the most important mechanisms of this type, allowing for the definition of a critical region around a specific code block or, in this case, encompassing all contents of a method [31].

In CFLASH, a synchronized method is simply one that contains the **synchronized** keyword as part of that method’s modifier list. This type of task synchronization falls under the “mutex” mechanism, which also encompasses the **ReentrantLock** and the **Semaphore** class [31]. Note that when the **synchronized** keyword is used with a method, the lock object reference is implicitly the object or class that the method belongs to [30].

Listing 3.3 shows the general syntax of a synchronized method in Java 8, as well as a sample annotation that would be introduced by CFLASH. In this example, an annotation with ID 1 is introduced and placed as a modifier right before the rest of the synchronized method declaration. The noise annotation syntax resembles that of the native Java annotations.

Listing 3.3: Annotated Synchronized Method Example

---

```
1 public class classA {  
2  
3     @Noise@1 synchronized void methodA() {  
4         // Critical section code  
5     }  
6 }
```

---

### 3.3.1.2 Synchronized Blocks

Like synchronized methods, synchronized blocks also fall under the “mutex” mechanism provided by the concurrency API. However, one of the most prominent differences is that in the case of synchronized blocks, an object reference is passed as a parameter to act as a lock. This means that no more than one execution thread is able to access any critical regions with this same lock on it.

Given that description, by adding annotations and noise around synchronized blocks, CFLASH aims to target concurrency bugs where the wrong objects are acquired as locks, or where they are acquired in the wrong order.

Listing 3.4 illustrates the syntax of what a synchronized block in Java 8 would look like after an annotation with ID 2 is introduced by CFLASH. Similarly to the syntax followed in annotating synchronized methods, the noise annotation is automatically placed before the synchronized block declaration. In this particular example, we can observe the noise annotation belonging to that synchronized block has an ID of 2.

### 3.3.1.3 Global Variables

The other two annotation targets only pertain to single statements, as opposed to methods or code blocks. With these types of targets, we want to localize bugs that

Listing 3.4: Annotated Synchronized Block Example

---

```
1 public class classB {
2
3     void methodB() {
4
5         @Noise02 synchronized (lock) {
6             // Critical section code
7         }
8     }
9 }
```

---

Listing 3.5: Annotated Statement Accessing Global Variables

---

```
1 public class classC {
2
3     public int globalVariable;
4
5     void methodC() {
6
7         @Noise03 globalVariable += 1; // Unsafe access to global variable
8     }
9 }
```

---

result from a lack of any locking mechanism around potential access to shared data.

The first type of statement targets is looking to find instances where a global class member is being accessed to be either read or modified. This includes global class members in the current class or in external classes, as well as static variables.

The following types are considered: expression statements, if statements, switch statements, while statements, do-while statements, for loops, and return statements. They correspond to all statements in Java available through TXL that could be affected by a bug due to shared data or lock order.

Listing 3.6: Annotated Statement Accessing Object Parameter

---

```
1 public class classD {  
2  
3     void methodD(ObjectClass variable) {  
4  
5         @Noise@4 variable.data += 1; // Unsafe access to object variable  
6     }  
7 }
```

---

#### 3.3.1.4 Shared Objects

In Java, when an object is passed to a method as a parameter, the contents of the original object can be accessed and modified from within it [44]. However, this is not the case for primitive types, where only a copy of the variable is passed.

For this reason, it is important to also consider non-primitive variables passed as method arguments as our fourth target. An example of an annotated statement of this type is shown in Listing 3.6. Like global variable statement annotations, the purpose of shared object statement annotations is to localize potential no-lock bugs.

The following statement types where object variables are accessed are annotated: expression statements, if statements, switch statements, while statements, do-while statements, for loops, and return statements. As is the case of global variable statements (Section 3.3.1.3), these correspond to all statements in Java available through TXL that could be affected by a bug due to shared data or lock order.

## 3.4 Search Phase

The Search phase of CFLASH uses a binary-search approach to narrow down the location of the concurrent bugs found — if any. This phase iteratively noises and tests

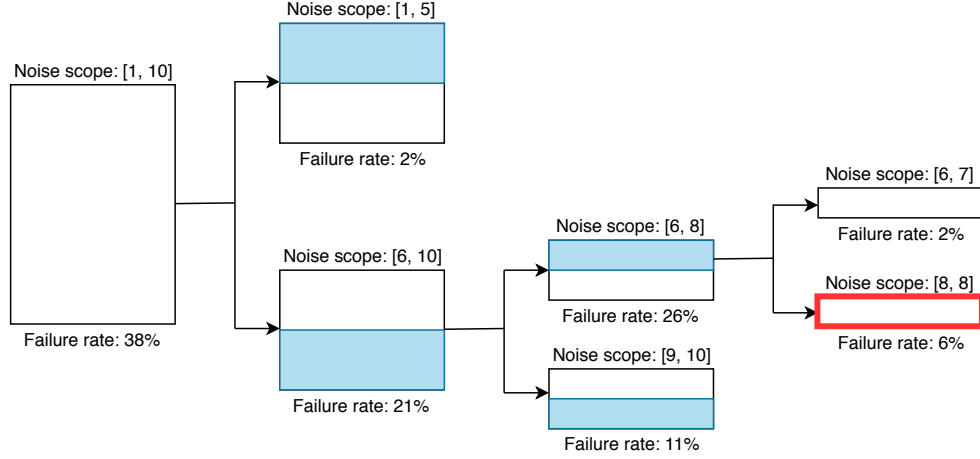


Figure 3.2: Visualization of Search Strategy

the input program to determine the search scope at each step of the binary search process (Appendix B). The stopping condition for the search phase is when either a single annotation has been reached or when the scopes being compared return the same (or similar) failure rates after a set number of iterations.

Figure 3.2 is a visual depiction of the search algorithm implemented. The search starts by dividing the program into two halves using the annotation IDs that had been previously assigned when identified as a relevant target. After, each half is noised and tested independently, which returns a failure rate corresponding to the number of unsuccessful test runs. For example, if there are 10 annotations in total, then 5 annotations are noised and tested first, a failure rate is calculated, and then the other 5 are noised and tested as well.

Finally, after determining if the failure rates calculated are relevant enough for consideration, they are compared with each other. If the failure rates are significantly different, the scope with the highest failure rate is explored further while the other is permanently discarded; otherwise, the algorithm will proceed to further explore both

simultaneously until the maximum number of iterations is reached.

The number of test runs to be performed by CFLASH is also specified by the user-supplied configuration file. While this value can vary greatly, the higher the value is, the more accurate the results will be since more data points will be available to the tool for analysis. However, consider the fact that a higher number of test runs will result in a longer program run time.

### 3.4.1 Thresholds

To dictate whether a failure rate outcome is *relevant* or not, CFLASH makes use of a **relevancy threshold**. This corresponds to the failure rate calculated by testing the project with all its targets noised. If a given failure rate exceeds this threshold value, then it is considered pertinent enough and is further explored. If two given failure rates exceed the threshold, then they are compared so as to decide which scope to follow, or if both should be explored.

On the other hand, for the purpose of *comparing* two failure rates, CFLASH makes use of a **difference threshold**. This value is specified by the user and supplied through the configuration file discussed previously in Section 3.2.1. This measure is an indication of the difference between two failure rates and is used to establish whether two scopes are notably distinct or not. As mentioned previously, it is possible to pursue both in the case where they are not deemed distinct enough.

### 3.4.2 Noise

Noising the input program involves iterating over the annotations present and replacing them with random thread delays. Each of the annotation targets in Section 3.3.1

is noised differently to ensure context switches are covered and to avoid over-noising, considering that some types of targets are used much more frequently in code. This strategy was put in place to improve the probability that a given concurrency bug will be exposed during testing.

Recall the four different types of annotation targets that are considered in CFLASH: synchronized methods, synchronized blocks, statements accessing global variables and statements accessing object variables that have been passed as a method parameter. However, for noising purposes, these types can be categorized into three levels: synchronized methods, synchronized blocks and statements.

Experiments performed in the past, such as IBM’s ConTest, have demonstrated that Java’s `Thread.sleep()` method works better as a noising mechanism than the alternative of using the `yield()` method [16, 17, 40]. Thus, `sleep` is the noise method used in our heuristics for all annotation targets. This will likely force a context switch where introduced, allowing for interesting thread interleavings to take place. By exploring different interleavings in the program, we are likely to find the order of concurrent events that cause a bug to manifest.

Moreover, it has been discovered that the optimal location for noise in the program is code segments that access shared variables as well as those that make use of synchronization primitives [3, 16, 17, 40, 48, 56]. It has been shown that it is adequate to introduce delays before synchronization operations. This is the approach that CFLASH takes on its noising heuristics. As done in the Annotation phase (Section 3.3), TXL [13] is used as a source transformation tool to introduce noise.

Listing 3.7: Noised Synchronized Method Example

---

```
1 public class classA {
2
3     void methodA() { // Wrapper method
4
5         try { // Try-finally
6
7             try {
8                 Thread.sleep(java.util.concurrent.ThreadLocalRandom.current()
9                     .nextInt(100, 301)); } catch (Exception __) { } // Noise
10
11                 methodA___sync___(); // Call to original method
12             } finally {
13
14                 try {
15                     Thread.sleep(java.util.concurrent.ThreadLocalRandom.current()
16                         .nextInt(100, 301)); } catch (Exception __) { } // Noise
17                 }
18             }
19         } synchronized void methodA___sync___() {
20             // Original method body
21         }
22     }
```

---

#### 3.4.2.1 Annotated Synchronized Methods

The first target to be noised, synchronized methods, are arguably the most intricate of the three to execute. As mentioned in Section 3.4.2, the optimal positioning of the thread delays is before and after a critical section or context switch for the purpose of exposing a potential concurrency bug. This means that the noise should be injected before and after a synchronized method is called.

However, to parse the entire program and inject noise before and after every single synchronized method call is not efficient. Instead, CFLASH identifies the

synchronized methods and generates a noised wrapper method for each and replaces the synchronized method call to a call to its corresponding wrapper, so as to ensure noise is correctly introduced. This change will propagate to the rest of the method calls in the program, without the need of noising each one individually.

For instance, Listing 3.7 contains an example noised method based on the annotated method shown previously in Listing 3.3. First, the user-defined `methodA` will be renamed to `methodA___sync___` with no other modifications. Secondly, a new method with an identical name to the original, `methodA`, will be automatically generated with the sole purpose of the `methodA___sync___` method. Hence, we term it a wrapper method.

Finally, noise is introduced before and after the call to the original method within the wrapper method. Now, every call to `methodA` in the program will be noised before and after its critical region. It is worth noting that this approach works the same for synchronized methods that contain parameters or a non-void return type, in which case the call to the original method would be a `return` statement instead.

The duration of the delay was set based on ConTest, a previous work on noising tools, but also taking into consideration our hardware limitations.

### 3.4.2.2 Annotated Synchronized Blocks

Synchronized blocks that have been annotated will be adapted to include a thread delay before and after the critical section as well. In this case, the noise can easily be injected around the code block without the need for any other code structures, as expressed in Listing 3.8.

Listing 3.8: Noised Synchronized Block Example

---

```
1 public class classB {
2
3     void methodB() {
4
5         try { // Try-finally
6
7             try {
8                 Thread.sleep(java.util.concurrent.ThreadLocalRandom.current()
9                     .nextInt(100, 301)); } catch (Exception __) { } // Noise
10
11             synchronized (lock) {
12                 // Critical computation
13             }
14         } finally {
15
16             try {
17                 Thread.sleep(java.util.concurrent.ThreadLocalRandom.current()
18                     .nextInt(100, 301)); } catch (Exception __) { } // Noise
19             }
20 }
```

---

### 3.4.2.3 Annotated Statements

Both annotation targets: statements accessing global variables and statements accessing local object variables, discussed in Section 3.3.1.4, fall under this category in terms of noising strategies.

For potentially unsafe statements like these, it would also be ideal to noise before and after the code segment in question. However, we found that, often, there is a high number of statements to noise in the input program. This leads to over-noising and considerable performance issues for the total run time of CFLASH. Consequently, we decided to only noise before the targeted statements, which resulted in unaltered

Listing 3.9: Noised Statement Example

---

```
1 public class classC {  
2  
3     public int globalVariable;  
4  
5     void methodC() {  
6  
7         try { Thread.sleep(java.util.concurrent.ThreadLocalRandom.current()  
8             .nextInt(100, 301)); } catch (Exception __) { } // Noise  
9         globalVariable += 1; // Unsafe access to global variable  
10  
11     }  
12 }
```

---

accuracy according to preliminary results. This strategy is employed to help mitigate the otherwise unnecessary computational burden of noising in too many locations.

Listing 3.9 shows an example of a noised statement that unsafely attempts to access and modify a global variable. The body of method `methodC` contains a statement that uses the `+=` operator to add 1 to the current value of `globalVariable`; a non-atomic operation lacking any concurrency safety mechanisms. This example is based on the annotation example discussed earlier in Listing 3.5.

### 3.4.3 Test Phase

Testing of concurrent programs is a challenging task due to the numerous thread interleavings possible. These interleavings depend on other events and the environment they are a part of; for example, other programs running on the same machine, network traffic, scheduling set by the operating system, etc. For this reason, it's very challenging to find bugs, like race conditions using testing — and when found, it is difficult to reproduce the bug and localize it [17].

Such types of bugs are called **heisenbugs** which refer to faults that are non-deterministic and hard to reproduce [29]. While the noise-injection technique used in CFLASH helps expand the interleavings to be investigated, it must be executed multiple times in order to retrieve meaningful results and find heisenbugs.

After the input program has been annotated and the most interesting code sections of the scope have been noised, the last step consists of testing the noised program  $n$  times as specified in the user previously. We opted to use 500 as our number of tests to run, following the findings of Ben-Asher et al. [3] on the subject. In addition, we also noticed this to be the minimum number of test runs to perform using ConTest [16] in order to uncover the concurrency bugs present in the benchmark used.

When completed, the testing module will return a rate corresponding to the percentage of runs that failed. This is the value that is used to narrow down the search scope at every step, which is then fed back into the noising component to repeat the process.

### 3.5 Output Logging

After every successful CFLASH run, the program will output a summary log of the findings. A successful run is defined as a CFLASH execution that is completed without syntax error or fatal exceptions, whether concurrency bugs were found and localized or not. The result logs are precise and detailed, such that the user can locate the potentially affected area in their project.

The data outlined includes general project and pre-run information for reference purposes such as the project name, time of the run, and the number of test runs used. Additionally, the log is also comprised of run-time data pertaining to the findings

uncovered by CFLASH, like the number of annotations introduced, the threshold values used, and the failure rate calculated for each step of the Search stage. Note that some of this information may only be available in debugging mode.

A sample output log is available in Appendix F.

# Chapter 4

## Evaluation

### 4.1 Experimental Setup

In order to evaluate CFLASH’s ability to localize concurrency bugs, we used 9 program variations based on 5 programs from the IBM Concurrency Benchmark [20–22, 35], the Software Infrastructure Repository (SIR) [14], textbooks [30, 31], and student submissions from the “Massively Parallel Programming” course at Ontario Tech University<sup>1</sup>. For each program used in the evaluation, a program variant with no known concurrency bugs was also used on CFLASH as a sanity check.

The program variants were selected based on the type of concurrency mechanisms used. For example, programs that utilized explicit locks or semaphores were excluded from the evaluation since CFLASH was designed to work with synchronized methods and synchronized statements. Moreover, the majority of the initial programs gathered did not include unit tests, and they would have had to be modified extensively in order to test them adequately. Extensive modification to the programs or their tests

---

<sup>1</sup>Student programs were used and made public with the authors’ consent.

<b>Program</b>	<b>LOC</b>	<b># Classes</b>	<b># Total Statements</b>	<b>Source</b>
<code>account</code>	98	3	63	IBM Benchmark [20–22, 35]
<code>airplane-ticketing</code>	93	3	44	Ontario Tech
<code>dining-philosophers</code>	91	3	29	SIR [14]
<code>parking</code>	145	4	69	Textbook [30]
<code>taxi-dispatcher</code>	123	3	76	Ontario Tech

Table 4.1: CFLASH Evaluation Data

would potentially introduce bias and violate our requirement to only evaluate with third-party programs. An overview of the programs used is depicted in Table 4.1.

The original versions of the programs were tested for bugs using noise multiple times, between 100 and 500 iterations, to ensure no bugs were present. If a faulty program was found during this testing process, then it was fixed manually and tested again in the same manner until the number of failed tests was zero. While most of the programs used from established benchmarks already contained concurrency bugs, those collected from students did not. For instance, the `account` and `parking` programs had pre-existing concurrency bugs, while the `airplane-ticketing` and `taxi-dispatcher`, student programs, did not.

Next, we used ConMAN<sup>2</sup> [7], a mutation testing tool, on the non-buggy program variants as a mutation tool to automatically and systemically generate a set of mutated program, each with a different type of concurrency bug inserted through source-code transformation techniques. The ConMAN tool is able to introduce 24

---

<sup>2</sup>Available at <https://github.com/sqrlab/ConMAN>.

Concurrency Bug Pattern	Mutation Operators
Nonatomic operations assumed to be atomic bug pattern	RVK, EAN
Two-stage access bug pattern	SPCR
Wrong lock or no lock bug pattern	<b>MSP</b> , ESP, EELO, SHCR, SKCR, EXCR, <b>RSB</b> , <b>RSK</b> , ASTK, RSTK, RCXC, RXO
Double-checked locking bug pattern	-
The sleep() bug pattern	MXT, RJS, RTXC
Losing a notify bug pattern	RTXC, RCXC
Notify instead of notify all bug pattern	RNA
Other missing or nonexistent signals bug pattern	MXC, MBR, RCXC
A “blocking” critical section bug pattern	RFU, RCXC
The orphaned thread bug pattern	-
The interference bug pattern	MXT, RTXC, RCXC
The deadlock (deadly embrace) bug pattern	ESP, EXCR, EELO, RXO
Starvation bug pattern	MSP, ELPA
Resource exhaustion bug pattern	MXC
Incorrect count initialization bug pattern	MXC

Table 4.2: Concurrency Bug Patterns vs. Concurrency Mutation Operators [7]

different types of concurrency bugs into a given program; however, only a small subset is relevant to the kind of synchronization mechanisms that are being considered for this research, namely, synchronized methods and synchronized blocks. ConMAN was used for all programs except `dining-philosophers`, which contained a deadlock. Given the mechanisms involved and the conditions in the code to lead to this bug, a ConMAN mutation operator was not possible. Instead, the bug was introduced manually following the author’s program description.

Data races and deadlocks were prioritized when evaluating CFLASH’s correctness

<b>Operator</b>	<b>Description</b>
RSK	Remove Synchronized Keyword from Method
RSB	Remove Synchronized Block
MSP	Modify Synchronized Block Parameter

Table 4.3: Subset of ConMAN Mutation Operators [7]

since it has been found they are two of the most common concurrency bugs [1]. This choice means that only the types of mutations that lead to these types of bugs were considered for evaluation. Table 4.2 outlines the different mutation operators available on ConMAN, as well as the concurrency bug pattern that each operator may induce. The ConMAN operators used to mutate our evaluation data are listed in Table 4.3, while a complete list of all ConMAN operators is listed in Appendix E. Note that the total number of statements for the program variations generated may vary depending on the mutation operator used. For example, operator RSK would reduce the number of statements by removing a synchronized block.

Lastly, due to time limitations and the high number of evaluation runs executed, experiments were performed simultaneously. Unless otherwise specified, all experimentation runs were performed on one of the following machines, where the machine with more capacity was prioritized for executing larger programs:

- Linux PC with a 2.80GHz processor, 8 gigabytes of RAM running Ubuntu 4.4.0, 4 cores with 2 threads per core.
- Linux PC with a 2.40GHz processor, 64 gigabytes of RAM running Ubuntu 4.15.0, 16 cores with 2 threads per core.

Project	Mutation Operator	# Annotations Added	# Test Runs	# Relevancy Threshold	Statement Distance (Program %)
account	MSP (v1)	4	500	0.000	0 (1.59%)
	MSP (v2)	8	500	0.002	1 (3.17%)
	RSB	7	500	0.000	3 (6.45%)
airplane-ticketing	RSK	4	500	0.576	0 (2.27%)
dining-philosophers	<i>Manual</i>	2	500	0.512	0 (3.57%)
parking	RSB (v1)	16	100	1.000	16 (25.00%)
	RSB (v2)	15	100	1.000	2 (4.41%)
	RSK	15	100	0.220	12 (18.84%)
taxi-dispatcher	RSB	6	100	0.450	0 (1.33%)

Table 4.4: CFLASH Evaluation Results

## 4.2 Search Correctness

The objective of CFLASH’s noising strategy is to increase the probability of a concurrency bug manifesting itself during testing. In addition, noising is used in conjunction with a binary search approach to ultimately localize a concurrency bug in a given program.

In order to quantify the correctness of the bug localization task, we have manually measured the number of statements between the point at which the bug occurs and the noise location that CFLASH has identified as causing the most buggy executions. The results are presented in Table 4.4. For consistency, we measured distance using the number of statements instead of the number of lines of code (LOC) to eliminate variations due to code style. We considered statements to be those defined in TXL and the Java grammar, including expressions, if statements, switch statements, while

loops, do-while loops, for loops, breaks, return statements, and try-catch blocks.

All buggy program variations in Table 4.4, as well as their correct counterparts in Table 4.5, were analyzed by CFLASH using the following parameters:

- **Number of test runs:** Between 100 and 500, depending on the program’s runtime and the computing resources available to analyze it. While Ben-Asher et al. and Edelstein et al. used 500 test runs with satisfactory results, we found that, for some programs, using fewer test runs (i.e. 100) resulted in drastic improvements in performance while still maintaining high accuracy.
- **Minimum and maximum noise delay:** During testing, the random noise delay is between 100 and 300 milliseconds. A maximum noise delay of 300 milliseconds was selected because it provided high quality results with improved performance when compared to an upper bound of 500 and 700 milliseconds.
- **Distance threshold:** The default value of 0 was used for all experiments.

The results of the experiments indicate that CFLASH was able to narrow the location of a concurrency bug. The largest distance recorded from the real bug statement to the estimated location by CFLASH was 16, corresponding to 25% of the total statements in that program.

In most cases, CFLASH was able to localize the bug with much higher accuracy, narrowing down the bug’s location to less than 10% of the program statements. Moreover, we observed that CFLASH correctly identified the exact point in the program where the bug was present in 4 of the program variations used.

In Table 4.5 we outline the results for the program variations containing no known bugs. As expected, CFLASH labelled all of these as having no bugs based on the

<b>Project</b>	<b># Annotations Added</b>	<b># Test Runs</b>	<b># Relevancy Threshold</b>
account	8	500	0
airplane-ticketing	3	500	0
dining-philosophers	2	500	0
parking	15	500	0
taxi-dispatcher	6	500	0

Table 4.5: CFLASH Evaluation Results for Non-Buggy Projects

test suites that were provided, and thus identified no false positives.

Throughout the fine-tuning process of the CFLASH algorithm and its parameters, approximately 115 evaluation runs were executed using different programs and bug types. However, evaluation runs performed after the CFLASH algorithms were finalized resulted in no false negatives.

### 4.3 Threats to Validity

The primary threat to the validity of this work is *external* in nature: the generalization of the results. The set of programs used in our experiments is small and rather short in length, which may not be representative of concurrency bugs in programs of a larger scale, both in terms of size (number of statements) and number of threads. This threat can be addressed by conducting further experiments with larger concurrent software systems.

# Chapter 5

## Conclusion

### 5.1 Summary

This thesis presents CFLASH, a tool that combines static and dynamic analysis techniques to automatically localize concurrency bugs in Java programs. To achieve this, CFLASH first strategically adds annotations to an input program at all concurrency-related mechanisms (e.g. synchronized blocks and methods) as well as unprotected access to potentially shared data. Next, the annotations are replaced by thread delays (noise) at the source code level using transformation rules in TXL [13]. Finally, the transformed program is tested against a user-provided test suite in order to narrow down the source code location of the concurrency bug in question using a binary search approach.

We evaluated CFLASH on a set of programs from different sources, including the IBM Concurrency Benchmark and student programs a parallel programming course at Ontario Tech University. The results demonstrate that a noise and search heuristic approach is feasible for localizing concurrency faults in Java programs.

## 5.2 Limitations

There are several known limitations of CFLASH’s design and performance:

- CFLASH has proven to produce the best results when the number of threads used in a given program is higher than the number of physical cores available on the machine it is being executed on. This is likely because there is less opportunity of threads waiting and interleaving than if there is a relatively high number of threads running simultaneously. While this is not a direct limitation of the tool, it certainly affects its performance.
- The number of times the user chooses to run and test on CFLASH is a very important setting and it is recommended that this value is set to be the highest possible, based on the architecture and capacity of the machine being used. A higher number of runs will result in a more accurate result by CFLASH. However, this can be a drawback because the number of runs can also significantly slow down the execution time of the tool.
- We have observed in our evaluation that CFLASH is able to obtain the most accurate results when the relevancy threshold calculated is less than 1. Recall, the relevancy threshold corresponds to the average failure rate calculated when the project is tested with all of its targets noised (see Section 3.4.1). While the tool can narrow down the buggy code vicinity with a high threshold, the localization is more precise when the relevancy threshold is *not* close to 1 — even a relevancy threshold of 0 performed satisfactorily when tested.

## 5.3 Future Work

In the future, we would be interested in exploring the following improvements or extensions to CFLASH.

### 5.3.1 Further Experimentation

The evaluation of CFLASH as a bug localization tool was focused mainly on its ability to identify and localize a single concurrency fault. Nonetheless, CFLASH was built with additional built-in features that, unfortunately, were not evaluated due to time limitations.

For example, the tool has the ability of exploring multiple search scopes simultaneously, targeting programs that contain more than one concurrency bug to consider. This behaviour would be triggered when there are two failure rates corresponding to different scopes that do not differ from each other enough; once the maximum number of iterations has been exhausted, CFLASH would opt to explore them both further.

Moreover, another feature worthy of in-depth evaluation is the difference threshold. The threshold is a value provided by the user as a configuration setting and is used to determine whether two failure rates are sufficiently distinct to firmly discard one of the two scopes in question. For example, if the difference threshold is set to be 0.05, then two failure rates of 0.45 and 0.49 would be considered effectively the same and would be run again. We would like to investigate the effect of varying the threshold would have on bug localization within CFLASH.

Finally, the third characteristic of the tool that was kept constant in this research work was the use of different noise targets. Upon executing a CFLASH run, the user is also able to specify what kind of target to noise in the input program

through command-line flags. The available targets to select from are: (1) synchronized methods, (2) synchronized blocks, (3) unsafe statements where global variables are accessed, and (4) unsafe statements where object variables passed as method parameters are accessed. This could be particularly useful if the user has an idea of the *type* of bug that is contained in the program. For example, if the behaviour is incorrect output, then the bug is likely a data race, in which case it may be useful to focus on examining unsafe statements. By selecting fewer targets, CFLASH is able to run a lot faster — however, the correctness of varying target selection has not been evaluated fully at this time.

### 5.3.2 Optimization

One of the most relevant limitations of CFLASH is its trade-off between the run time and number of test case executions, which could be interpreted as accuracy given their direct relationship. The tool’s modular architecture is quite flexible and a distributed test execution system could be implemented for CFLASH without the complication of having to adapt other parts of the tool substantially. With a distributed implementation, we could, for example, divide 500 test runs across the number of nodes available, speeding up the tool when compared to a single-machine run.

### 5.3.3 Machine Learning

Another approach to improving CFLASH is using by machine learning. Given enough data, the implementation of a machine learning method within CFLASH could replace the current noising annotation approach by identifying more efficient

and effective annotation targets on the input program. In turn, the binary search approach would also be enhanced by narrowing down the search space to the most relevant areas. While this may not necessarily improve runtime, it is possible that the noised areas are more likely to be relevant to the set objective.

### 5.3.4 Automatic Test Generation

In an effort to lessen the amount of manual work that is required from the user, this research could be extended further to incorporate automatic test generation. This is a well-researched topic, particularly for sequential and concurrency programs written in Java. For instance, some of the most relevant work includes CONSUITE [58], Java String Testing (JST) [27] and EVOSUITE [26].

The CONSUITE prototype takes a coverage-driven approach to test generation by presenting a search-based technique that derives concurrent test cases by statically analyzing the interleaving coverage criteria for a given class. The tool then applies a genetic algorithm method to generate the tests. On the other hand, JST is based on the JAVA PATHFINDER [8, 34] model checker as well as its symbolic execution extension: Symbolic PathFinder [51, 52]. The authors of this work not only have used JAVA PATHFINDER as their underlying platform but have also extended it to support all Java primitive types, Strings, and to also address some of the bottlenecks of using the tool. Finally, EVOSUITE uses a hybrid search and dynamic symbolic execution to automatically generate test cases with assertions for sequential Java classes. In order to maximize the number of seeded defects in a class, this tool also applies a mutation testing method.

# References

- [1] Sara Asadollah Abbaspour, Daniel Sundmark, Sigrid Eldh, and Hans Hansson. “Concurrency Bugs in Open Source Software: A Case Study”. In: *Journal of Internet Services and Applications*. Vol. 8. 1. Springer, 2017, p. 4.
- [2] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. “Using Static Analysis to Find Bugs”. In: *IEEE Software*. Vol. 25. 5. IEEE, 2008, pp. 22–29.
- [3] Yosi Ben-Asher, Yaniv Eytani, Eitan Farchi, and Shmuel Ur. “Noise Makers Need to Know Where to be Silent - Producing Schedules That Find Bugs”. In: *Proceedings of the 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2006)*. 2006, pp. 458–465.
- [4] Saddek Bensalem and Klaus Havelund. “Dynamic Deadlock Analysis of Multi-threaded Programs”. In: *Proceedings of the 1st International Haifa Verification Conference on Hardware and Software, Verification and Testing*. Springer Berlin Heidelberg, 2006, pp. 208–223.
- [5] Sam Blackshear, Nikos Gorogiannis, Peter W. O’Hearn, and Ilya Sergey. “RacerD: Compositional Static Race Detection”. In: *Proceedings of the ACM on*

- Programming Languages*. Vol. 2. Association for Computing Machinery, Oct. 2018.
- [6] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. “PACER: Proportional Detection of Data Races”. In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. Vol. 45. 6. Association for Computing Machinery, 2010, pp. 255–268.
  - [7] Jeremy S. Bradbury, James R. Cordy, and Juergen Dingel. “Mutation Operators for Concurrent Java (J2SE 5.0)”. In: *Proceedings of the 2nd Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*. 2006. ISBN: 076952897X.
  - [8] Guillaume Brat, Klaus Havelund, Seungjoon Park, and Willem Visser. “Java PathFinder – Second Generation of a Java Model Checker”. In: *Proceedings of the 2000 Workshop on Advances in Verification*. 2000.
  - [9] Phillip J. Brooke and Richard F. Paige. “Chapter 4 - Concurrency in Operating Systems”. In: *Practical Distributed Processing*. Springer Science & Business Media, 2007.
  - [10] David W. Bustard. *Concepts of Concurrent Programming — Curriculum Module*. Carnegie Mellon University, Software Engineering Institute, Apr. 1990.
  - [11] João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz. “Chapter 2 - High-performance embedded computing”. In: *Embedded Computing for High Performance*. Boston: Morgan Kaufmann, 2017, pp. 17–56. ISBN: 978-0-12-804189-5.

- [12] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O’Callahan, Vivek Sarkar, and Manu Sridharan. “Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs”. In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. PLDI ’02. Berlin, Germany: Association for Computing Machinery, 2002, pp. 258–269. ISBN: 1581134630.
- [13] James R. Cordy. “TXL - A Language for Programming Language Tools and Applications”. In: *Proceedings of the 4th Workshop on Language Descriptions, Tools, and Applications*. Vol. 110. Dec. 2004, pp. 3–31.
- [14] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. “Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact.” In: *Empirical Software Engineering: An International Journal*. Vol. 10. 4. 2005, pp. 405–435.
- [15] Oracle Documentation. *Chapter 1 Covering Multithreading Basics & Defining Multithreading Terms*. 2010. URL: <https://docs.oracle.com/cd/E19455-01/806-5257/6je9h032b/index.html> (visited on 03/09/2020).
- [16] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. “Framework for Testing Multi-Threaded Java Programs”. In: *Concurrency and Computation: Practice and Experience*. Vol. 15. 3-5. Feb. 2003, pp. 485–499.
- [17] Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby, and Shmuel Ur. “Multithreaded Java Program Test Generation”. In: *IBM Systems Journal*. Vol. 41. 1. 2002, pp. 111–125.

- [18] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. “Goldilocks: A Race and Transaction-Aware Java Runtime”. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 245–255. ISBN: 9781595936332.
- [19] Dawson Engler and Ken Ashcraft. “RacerX: Effective, Static Detection of Race Conditions and Deadlocks”. In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. SOSP ’03. Bolton Landing, NY, USA: Association for Computing Machinery, 2003, pp. 237–252. ISBN: 1581137575.
- [20] Yaniv Eytani, Klaus Havelund, Scott D. Stoller, and Shmuel Ur. “Towards a Framework and a Benchmark for Testing Tools for Multi-threaded Programs”. In: *Concurrency and Computation: Practice and Experience*. Vol. 19. 3. 2007, pp. 267–279.
- [21] Yaniv Eytani, Rachel Tzoref, and Shmuel Ur. “Experience with a Concurrency Bugs Benchmark”. In: *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*. 2008, pp. 379–384.
- [22] Yaniv Eytani and Shmuel Ur. “Compiling a Benchmark of Documented Multi-threaded Bugs”. In: *Proceedings of the 18th International Parallel and Distributed Processing Symposium*. IEEE. 2004, pp. 266–273. ISBN: 0-7695-2132-0.
- [23] Rob Farber. “Chapter 1 - First Programs and How to Think in CUDA”. In: *CUDA Application Design and Development*. Morgan Kaufmann, 2012, pp. 1–31.

- [24] Cormac Flanagan and Stephen N. Freund. “FastTrack: Efficient and Precise Dynamic Race Detection”. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’09. Dublin, Ireland: Association for Computing Machinery, 2009, pp. 121–133. ISBN: 9781605583921.
- [25] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. “A study of the internal and external effects of concurrency bugs”. In: *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*. 2010, pp. 221–230.
- [26] Gordon Fraser and Andrea Arcuri. “Evosuite: Automatic Test Suite Generation for Object-Oriented Software”. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. 2011, pp. 416–419.
- [27] Indradeep Ghosh, Nastaran Shafiei, Guodong Li, and Wei-Fan Chiang. “JST: An Automatic Test Generation Tool for Industrial Java Applications with Strings”. In: *Proceedings of the 2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 992–1001.
- [28] Brian Goetz, Tim Peierls, Doug Lea, Joshua Bloch, Joseph Bowbeer, and David Holmes. *Java Concurrency in Practice*. Pearson Education, 2006.
- [29] Max Goldman and Rob Miller. *6.031: Software Construction, Reading 19: Concurrency*. Massachusetts Institute of Technology (MIT), 2020. URL: <http://web.mit.edu/6.031/www/sp20/>.
- [30] Javier Fernández González. “Java 9 Concurrency Cookbook - Second Edition”. In: Packt Publishing, Apr. 2017. ISBN: 9781787124417.

- [31] Javier Fernández González. “Mastering Concurrency Programming with Java 8”. In: Packt Publishing, Feb. 2016. ISBN: 9781785886126.
- [32] Javier Fernández González. *Mastering Concurrency Programming with Java 9*. Packt Publishing Ltd, 2017. ISBN: 9781785887949.
- [33] *Happens-before Order*. URL: <https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.5> (visited on 03/09/2020).
- [34] Klaus Havelund and Thomas Pressburger. “Model checking Java programs using Java PathFinder”. In: *International Journal on Software Tools for Technology Transfer*. Vol. 2. 4. 2000, pp. 366–381.
- [35] Klaus Havelund, Scott D. Stoller, and Shmuel Ur. “Benchmark and Framework for Encouraging Research on Multi-threaded Testing Tools”. In: *Proceedings International Parallel and Distributed Processing Symposium*. 2003, 8–pp.
- [36] Vendula Hrubá, Bohuslav Křena, Zdeněk Letko, Hana Pluháčková, and Tomáš Vojnar. “Multi-objective Genetic Optimization for Noise-Based Testing of Concurrent Software”. In: *Proceedings of the 6th International Symposium on Search-Based Software Engineering*. Ed. by Claire Le Goues and Shin Yoo. 2014, pp. 107–122.
- [37] *Java Concurrency - Happens-before Relationship*. URL: <https://gerardnico.com/lang/java/concurrency/happens-before> (visited on 03/09/2020).
- [38] *Java Platform, Standard Edition 8 API Specification*. URL: <https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html> (visited on 03/11/2020).

- [39] Bohuslav Křena, Zdeněk Letko, and Tomáš Vojnar. “Coverage Metrics for Saturation-Based and Search-Based Testing of Concurrent Software”. In: *Proceedings of the 6th International Conference on Runtime Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 177–192.
- [40] Bohuslav Křena, Zdeněk Letko, and Tomáš Vojnar. “Noise Injection Heuristics for Concurrency Testing”. In: *Proceedings of the 7th International Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*. Vol. 7119. 2012, pp. 123–135. ISBN: 9783642259289.
- [41] Nancy G. Leveson et al. “Medical Devices: The Therac-25”. In: *Appendix of: Safeware: System Safety and Computers*. Citeseer, 1995.
- [42] Nancy G. Leveson and Clark S. Turner. “An Investigation of the Therac-25 Accidents”. In: *Computer*. Vol. 26. 7. IEEE, 1993, pp. 18–41.
- [43] Yanze Li, Bozhen Liu, and Jeff Huang. “SWORD: A Scalable Whole Program Race Detector for Java”. In: *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2019, pp. 75–78.
- [44] Robert Liguori and Patricia Liguori. “Chapter 4 - Reference Types”. In: *Java Pocket Guide: Instant Help for Java Programmers*. 4th edition. O’Reilly Media, Inc., 2017.
- [45] Shan Lu, Weihang Jiang, and Yuanyuan Zhou. “A Study of Interleaving Coverage Criteria”. In: *Proceedings of the 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*. ESEC-FSE companion ’07.

- Dubrovnik, Croatia: Association for Computing Machinery, 2007, pp. 533–536. ISBN: 9781595938121.
- [46] Robert C. Martin. “Chapter 13 - Concurrency”. In: *Clean Code: Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.
  - [47] Peter Mehlitz, Corina Pasareanu, Dimitra Giannakopoulou, and Masoud Mansouri-Samani. *Java PathFinder: A Model Checker for Java Programs*. 2012. URL: <https://ti.arc.nasa.gov/tech/rse/vandv/jpf/> (visited on 03/24/2020).
  - [48] Madanlal Musuvathi and Shaz Qadeer. “Iterative Context Bounding for Systematic Testing of Multithreaded Programs”. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 446–455. ISBN: 9781595936332.
  - [49] Mayur Naik, Alex Aiken, and John Whaley. “Effective Static Race Detection for Java”. In: *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation - PLDI ’06*. 2006, p. 308. ISBN: 1595933204.
  - [50] Robert O’Callahan and Jong-Deok Choi. “Hybrid Dynamic Data Race Detection”. In: *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP ’03. San Diego, California, USA: Association for Computing Machinery, 2003, pp. 167–178. ISBN: 1581135882.
  - [51] Corina S Păsăreanu, Peter C Mehlitz, David H Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. “Combining Unit-Level Symbolic Execution and System-Level Concrete Execution for Testing NASA Soft-

- ware”. In: *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. 2008, pp. 15–26.
- [52] Corina S Păsăreanu and Neha Rungta. “Symbolic PathFinder: Symbolic Execution of Java Bytecode”. In: *Proceedings of the 2010 IEEE/ACM International Conference on Automated Software Engineering*. 2010, pp. 179–180.
  - [53] Bill Roscoe. *The Theory and Practice of Concurrency*. Pearson, 1998, p. 3.
  - [54] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. “Eraser: A Dynamic Data Race Detector for Multithreaded Programs”. In: *ACM Transactions on Computer Systems*. Vol. 15. 4. New York, NY, USA: Association for Computing Machinery, Nov. 1997, pp. 391–411.
  - [55] Herbert Schildt. “Chapter 11 - Multithreaded Programming”. In: *Java: A Beginner’s Guide*. 8th edition. McGraw-Hill, 2018.
  - [56] Koushik Sen. “Race Directed Random Testing of Concurrent Programs”. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’08. Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 11–21. ISBN: 9781595938602.
  - [57] Konstantin Serebryany and Timur Iskhodzhanov. “ThreadSanitizer: Data Race Detection in Practice”. In: *Proceedings of the Workshop on Binary Instrumentation and Applications*. WBIA ’09. New York, New York, USA: Association for Computing Machinery, 2009, pp. 62–71. ISBN: 9781605587936.
  - [58] Sebastian Steenbuck and Gordon Fraser. “Generating Unit Tests for Concurrent Classes”. In: *Proceedings of the 2013 IEEE 6th International Conference on Software Testing, Verification and Validation*. 2013, pp. 144–153.

- [59] *The Java Tutorials — Lesson: Concurrency*. URL: <https://docs.oracle.com/javase/tutorial/essential/concurrency/> (visited on 03/11/2020).
- [60] Jan Wen Voun, Ranjit Jhala, and Sorin Lerner. “RELAY: Static Race Detection on Millions of Lines of Code”. In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC-FSE ’07. Dubrovnik, Croatia: Association for Computing Machinery, 2007, pp. 205–214. ISBN: 9781595938114.
- [61] Zan Wang, Yingquan Zhao, Shuang Liu, Jun Sun, Xiang Chen, and Huarui Lin. “MAP-Coverage: A Novel Coverage Criterion for Testing Thread-Safe Classes”. In: *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, pp. 722–734.
- [62] Junfeng Yang, Heming Cui, Jingyue Wu, Yang Tang, and Gang Hu. “Making Parallel Programs Reliable with Stable Multithreading”. In: *Communications of the ACM*. Vol. 57. 3. 2014, pp. 58–69.

# Appendix A

## Command-Line User Input Flags

- `--help` or `-h`: Displays a `man` help message with details on execution parameters.
- `--debug` or `-d`: Enable debugging mode. Detailed information is accessible to the user, such as verbose execution, access to annotated and noised source files via Docker volumes, and a log with descriptive data on the input program.
- `--syncmethods` or `-m`: Annotates and noises **synchronized methods** found.
- `--syncblocks` or `-b`: Annotates and noises **synchronized blocks** found.
- `--globalvars` or `-g`: Annotates and noises **statements** that access **global variables**.
- `--objectvars` or `-o`: Annotates and noises **statements** that access **object variables** that have been passed to the wrapping method.
- `--noisethreads` or `-t`: Annotates and noises any targets mentioned above that are contained inside a `run()` method contained in a threaded class. CFLASH would otherwise skip these.

# Appendix B

## Binary Search Algorithm

- *lower* = Lower-bound for the current search scope. Refers to an annotation ID.
- *upper* = Upper-bound for the current search scope. Refers to an annotation ID.
- *highestfailrate* = The highest failure rate that has been recorded at the moment.
- *firstfrate* = The failure rate corresponding to the program where it's **first** half was noised.
- *secondfrate* = The failure rate corresponding to the program where it's **second** half was noised.
- *itercount* = Maximum number of times a program can be tested until it is considered relevant or not.

---

**Algorithm B.1** CFLASH Search Algorithm

---

```
1: function BINARY(lower, upper, highestfailrate)
2:   if lower == upper and highestfailrate  $\geq$  relevancethres then
3:     Add ID range to potential top bugs
4:   end if
5:   while lower < upper and itercount  $\leq$  maxiter do
6:     middle  $\leftarrow$  (lower + upper)/2
7:     NOISE(lower, middle) ▷ Noise first half
8:     firstfrate  $\leftarrow$  failure rate of tested noised program ▷ Test
9:     NOISE(middle + 1, upper) ▷ Noise second half
10:    secondfrate  $\leftarrow$  failure rate of tested noised program ▷ Test
11:    if firstfrate  $\geq$  relevancethres or secondfrate  $\geq$  relevancethres then
12:      if firstfrate > 0 and firstfrate  $\geq$  highestfailrate then
13:        highestfailrate  $\leftarrow$  first half's results
14:      end if
15:      if secondfrate > 0 and secondfrate  $\geq$  highestfailrate then
16:        highestfailrate  $\leftarrow$  second half's results
17:      end if
18:      if abs(firstfrate - secondfrate)  $\geq$  differenthres then
19:        if firstfrate > secondfrate then
20:          upper  $\leftarrow$  middle
21:        else if firstfrate < secondfrate then
22:          lower  $\leftarrow$  middle + 1
23:        end if
24:        Update current highestfailrate
25:      end if
26:    else ▷ Rates are too similar
27:      if itercount  $\leq$  maxiter then
28:        continue
29:      else if (lower  $\neq$  middle or middle + 1  $\neq$  upper) and (firstfrate > 0
or secondfrate > 0) then
30:        BINARY(lower, middle, firstfrate) ▷ Recursive call
31:        BINARY(middle + 1, upper, secondfrate)
32:        break
33:      end if
34:    end if
35:  end while
36: end function
```

---

## **Appendix C**

### **Research Ethics Board Approval**

**From:** researchethics@uoit.ca  
**Subject:** Approval Notice - REB File #15672  
**Date:** November 21, 2019 at 3:45 PM  
**To:** Bradbury Jeremy(Primary Investigator) jeremy.bradbury@uoit.ca  
**Cc:** researchethics@uoit.ca



**Date:** November 21, 2019  
**To:** Jeremy Bradbury  
**From:** Paul Yelder, REB Vice-Chair  
**File # & Title:** 15672 - An Evaluation of An Automatic Localization Tool for Concurrency Bugs  
**Status:** **APPROVED**  
**Current** **November 01, 2020**  
**Expiry:**

Notwithstanding this approval, you are required to obtain/submit, to Ontario Tech Research Ethics Board, any relevant approvals/permissions required, prior to commencement of this project.

The Ontario Tech Research Ethics Board (REB) has reviewed and approved the research study named above to ensure compliance with the Tri-Council Policy Statement: Ethical Conduct for Research Involving Humans (TCPS2 2014), the Ontario Tech Research Ethics Policy and Procedures and associated regulations. As the Principal Investigator (PI), you are required to adhere to the research protocol described in the REB application as last reviewed and approved by the REB. In addition, you are responsible for obtaining any further approvals that might be required to complete your project.

Under the Tri-Council Policy Statement 2, the PI is responsible for complying with the continuing research ethics reviews requirements listed below:

**Renewal Request Form:** All approved projects are subject to an annual renewal process. Projects must be renewed or closed by the expiry date indicated above ("Current Expiry"). Projects not renewed 30 days post expiry date will be automatically suspended by the REB; projects not renewed 60 days post expiry date will be automatically closed by the REB. Once your file has been formally closed, a new submission will be required to open a new file.

**Change Request Form:** If the research plan, methods, and/or recruitment methods should change, please submit a change request application to the REB for review and approval prior to implementing the changes.

**Adverse or Unexpected Events Form:** Events must be reported to the REB within 72 hours after the event occurred with an indication of how these events affect (in the view of the Principal Investigator) the safety of the participants and the continuation of the protocol (i.e. un-anticipated or un-mitigated physical, social or psychological harm to a participant).

**Research Project Completion Form:** This form must be completed when the research study is concluded.

Always quote your REB file number (**15672**) on future correspondence. We wish you success with your study.

Sincerely,

Dr. Paul Yelder  
REB Vice-Chair  
paul.yelder@uoit.ca

Emma Markoff  
Research Ethics Assistant  
researchethics@uoit.ca

*NOTE: If you are a student researcher, your supervisor has been copied on this message.*

# Appendix D

## University Course Laboratory Description

**CSCI 4060U — Laboratory #7**  
**Java Threads Applications**  
Lab Due: 11pm, Mar. 24, 2019 (Blackboard)

**Introduction**

The main goal of this lab is to apply what has been learned in previous activities about C pthreads to Java Concurrency.

**Activity #1**

Select an application or topic from the below list and implement it using Java threads.

- Sort
- Search
- Transactional System (e.g. banking)
- Producer/Consumer
- Reader/Writer (e.g. basic message board)
- Genetic Algorithm
- Taxi Dispatcher
- Number Generation (e.g. Fibonacci)

**Activity #2**

Write a set of unit tests that assert the correctness of your program; make sure they pertain to the data being accessed and modified by threads.

**Submission**

You should submit all your commented source files through the lab drop box in Blackboard.

## Appendix E

# ConMAAn Concurrency Mutation Operators for Java

Operator Category	Concurrency Mutation Operators for Java (J2SE 5.0)
Modify Parameters of Concurrent Methods	<b>MXT</b> - Modify Method-X Time ( <i>wait()</i> , <i>sleep()</i> , <i>join()</i> , and <i>wait()</i> method calls)
	<b>MSP</b> - Modify Synchronized Block Parameter
	<b>ESP</b> - Exchange Synchronized Block Parameters
	<b>MSF</b> - Modify Semaphore Fairness
	<b>MXC</b> - Modify Permit Count in Semaphore and Modify Thread Count in Latches and Barriers
	<b>MBR</b> - Modify Barrier Runnable Parameter
Modify the Occurrence of Concurrency Method Calls	<b>RTXC</b> – Remove Thread Method-X Call ( <i>wait()</i> , <i>join()</i> , <i>sleep()</i> , <i>yield()</i> , <i>notify()</i> , <i>notifyAll()</i> Methods)
	<b>RCXC</b> – Remove Concurrency Mechanism Method-X Call ( <i>methods in Locks, Semaphores, Latches, Barriers, etc.</i> )
	<b>RNA</b> - Replace NotifyAll() with Notify()
	<b>RJS</b> - Replace Join() with Sleep()
	<b>ELPA</b> - Exchange Lock/Permit Acquisition
	<b>EAN</b> - Exchange Atomic Call with Non-Atomic
Modify Keyword	<b>ASTK</b> – Add Static Keyword to Method
	<b>RSTK</b> – Remove Static Keyword from Method
	<b>RSK</b> - Remove Synchronized Keyword from Method
	<b>RSB</b> - Remove Synchronized Block
	<b>RVK</b> - Remove Volatile Keyword
	<b>RFU</b> - Remove Finally Around Unlock
Switch Concurrent Objects	<b>RXO</b> - Replace One Concurrency Mechanism-X with Another (Locks, Semaphores, etc.)
	<b>EELO</b> - Exchange Explicit Lock Objects
Modify Critical Region	<b>SHCR</b> - Shift Critical Region
	<b>SKCR</b> - Shrink Critical Region
	<b>EXCR</b> – Expand Critical Region
	<b>SPCR</b> - Split Critical Region

Table E.1: ConMAN Mutation Operators [7]

# Appendix F

## Sample CFLASH Output

Listing F.1: Sample CFLASH Output

```
1 =====
2 Project name: Taxi Dispatcher (Remove synchronized block - Version 1)
3 Date: Monday, February 17, 2020
4 Time: 06:27:50 PM EST
5
6 * Project root: ../Data-Testing-Review/data/taxi-dispatcher/RSB/v1
7 * Source code: src
8 * Project test suite: test
9 * Number of test runs: 100
10 =====
11
12 2020-02-17 23:27:51,015 [INFO] Synchronized methods
13 2020-02-17 23:27:51,046 [INFO] Statements accessing global variables
14 2020-02-17 23:27:51,369 [INFO] Statements accessing object variables
15 2020-02-17 23:27:51,407 [INFO] Synchronized blocks
16 2020-02-17 23:27:51,436 [INFO] Remove annotations in threaded class methods
17 2020-02-17 23:27:51,508 [DEBUG] Introduced a total of 6 annotations
18 2020-02-17 23:27:51,508 [DEBUG] Output files saved at 'core/annotate/out'
19 2020-02-18 00:34:29,540 [DEBUG] Completed in 3998.03135 seconds
20 2020-02-18 00:34:29,541 [DEBUG] Project details by annotation ID stored
    ↪ at:
    ↪ 'core/volumes/log/details/v1__100__20200217182750990209__annotation-
    ↪ details.log'
```

```

21 2020-02-18 00:34:29,541 [INFO] Using binary search
22 2020-02-18 00:34:29,541 [DEBUG] Using 100 sample test run(s)
23 2020-02-18 00:34:29,541 [INFO] Difference threshold entered: 0
24 2020-02-18 00:34:29,541 [INFO] Calculate relevance threshold
25 2020-02-18 00:52:54,723 [INFO] Relevancy threshold computed: 0.45
26 2020-02-18 00:52:54,724 [DEBUG] Current highest failure rate: [1, 6]: 45.0%
27 2020-02-18 00:52:54,724 [INFO] Noise and test first half: [1, 3]
28 2020-02-18 01:04:42,493 [DEBUG] Failure rate: 0.23
29 2020-02-18 01:04:42,494 [INFO] Noise and test second half: [4, 6]
30 2020-02-18 01:20:57,050 [DEBUG] Failure rate: 0.5
31 2020-02-18 01:20:57,051 [DEBUG] Proceed with [4, 6], discard [1, 3]
32 2020-02-18 01:20:57,051 [DEBUG] Current highest failure rate: [4, 6]: 50.0%
33 2020-02-18 01:20:57,051 [INFO] Noise and test first half: [4, 5]
34 2020-02-18 01:34:36,737 [DEBUG] Failure rate: 0.48
35 2020-02-18 01:34:36,737 [INFO] Noise and test second half: [6, 6]
36 2020-02-18 01:46:45,066 [DEBUG] Failure rate: 0.18
37 2020-02-18 01:46:45,066 [DEBUG] Proceed with [4, 5], discard [6, 6]
38 2020-02-18 01:46:45,066 [DEBUG] Current highest failure rate: [4, 6]: 50.0%
39 2020-02-18 01:46:45,067 [INFO] Noise and test first half: [4, 4]
40 2020-02-18 01:57:26,219 [DEBUG] Failure rate: 0.69
41 2020-02-18 01:57:26,219 [INFO] Noise and test second half: [5, 5]
42 2020-02-18 02:10:01,082 [DEBUG] Failure rate: 0.62
43 2020-02-18 02:10:01,082 [DEBUG] Proceed with [4, 4], discard [5, 5]
44 2020-02-18 02:10:01,083 [INFO] Parse and store container runtime metrics
45
46 Average CPU usage: 6.38%
47 Average memory usage: 0.27%
48 Average number of PIDs: 49.56
49
50 2020-02-18 02:10:01,475 [DEBUG] Exhaustive approach finished in 3998.03135
    ↪ seconds
51 2020-02-18 02:10:01,475 [INFO] CFLASH finished in 9730.092166 seconds
52 2020-02-18 02:10:01,475 [DEBUG] Annotation ID ranges and failure rates for
    ↪ buggy vicinities found:
53 2020-02-18 02:10:01,475 [DEBUG] [4 ,4]: 69.0%
54 2020-02-18 02:10:01,475 [DEBUG] [4 ,6]: 50.0%
55 2020-02-18 02:10:01,475 [DEBUG] [4 ,6]: 50.0%
56 2020-02-18 02:10:01,475 [INFO] Potentially buggy code found in 3
    ↪ vicinities:
57 2020-02-18 02:10:01,476 [INFO]
58 --> Annotation ID: 4 (69.00%)
59 Type: statement
60 File: Dispatcher.java

```

```
61 Class: Dispatcher
62 Method: public Customer dispatchResp ()
63 Statement content: assignedCustomer = customers.remove (0);
64
65 2020-02-18 02:10:01,476 [INFO]
66 --> Annotation ID: 6 (50.00%)
67 Type: statement
68 File: Taxi.java
69 Class: Taxi
70 Method: private void gotoLocation (int dest)
71 Statement content: this.location = dest
72
73 2020-02-18 02:10:01,476 [INFO]
74 --> Annotation ID: 5 (50.00%)
75 Type: block
76 File: Dispatcher.java
77 Class: Dispatcher
78 Method: public boolean checkCustomers ()
```