Assessing Tools for Finding Bugs in Concurrent Java

Alexandros Spathoulas

Master of Science
Computer Science
School of Informatics
University of Edinburgh
2014

Abstract

Nowadays, concurrent programming is used more and more in order to make multicore systems achieve the best possible performance. However, the usage of concurrent programming introduces more errors (usually more difficult to be identified) than sequential programming and for this reason it is very important to have tools that can identify such errors. The most used technique to identify errors in multi-threaded environment is code analysis; it consists of static and dynamic analysis. Code analysis can be done manually or by using automated tools and it happens at an early stage in the software development life cycle.

The number of static analysis tools has increased rapidly in the last decade. In order to choose the appropriate tool each developer must identify what is really needed, learn about the strengths and weaknesses of the existing tools and then choose the most suitable. In this project, we focus on helping programmers to learn how to evaluate properly a tool and choose the right one between free open-source (FindBugs), commercial (ThreadSafe) and tools that appeal mostly to experienced developers (Chord).

Moreover, we do a theoretical and experimental analysis of three static analysis tools (ThreadSafe, FindBugs and Chord) that are used for detecting a variety of Java concurrency bugs. The tools are evaluated by their accuracy, performance, usability and the ability to detect obfuscated programs. Moreover, for the evaluation, we use a test suite (by combining reliable resources like IBM Haifa Research Lab and CERT) containing Java programs with bugs and see what errors can be identified and explain why some of them cannot be captured.

Acknowledgements

I would like to express my deepest appreciation to my supervisor, Professor Don Sannella, who recognised my capabilities, trusted me and was always there to help me. Without his help and support it would be impossible to finish the project.

Furthermore, I would like to thank Contemplate Ltd for providing a licence for ThreadSafe until the end of August. However, the licence would be useless without the help in order to set up and configure the tool.

Last but not least, I would like to thank my family for encouraging me and giving me the ability to study in Edinburgh and learn so many things.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Alexandros Spathoulas*)

Table of Contents

1 Introduction **1**

1.1 Overview . . . 1

1.2 Motivation . . . 2

1.3 Approach . . . 2

1.4 Thesis outline . . . 3

2 Background **5**

2.1 Concurrent programming . . . 5

2.1.1 Overview . . . 5

2.1.2 Advantages and Disadvantages . . . 6

2.1.3 Problems in concurrent programming . . . 6

2.2 Code analysis . . . 7

2.2.1 Static analysis . . . 8

2.2.2 Dynamic analysis . . . 8

2.2.3 Static analysis vs Dynamic Analysis . . . 9

2.2.4 False Positives and False Negatives . . . 9

2.3 Static analysis tools . . . 10

2.4 Previous Evaluations of Static Analysis Tools . . . 10

3 Tools and Configuration **13**

3.1 Tools . . . 13

3.1.1 ThreadSafe . . . 13

3.1.1.1 Overview . . . 13

3.1.1.2 Requirements . . . 13

3.1.2 FindBugs . . . 14

3.1.2.1 Overview . . . 14

3.1.2.2 Requirements . . . 15

3.1.3 Chord . . . 15

3.1.3.1 Overview . . . 15

3.1.3.2 Requirements . . . 15
3.2 Test Suite . . . 16
3.3 Configuration . . . 20
3.3.1 System configuration . . . 20
3.3.2 Tools installment and configuration . . . 20
3.3.2.1 ThreadSafe . . . 20
3.3.2.2 FindBugs . . . 23
3.3.2.3 Chord . . . 25

4 Evaluation of the tools **27**
4.1 Usability . . . 27
4.1.1 ThreadSafe . . . 27
4.1.2 FindBugs . . . 28
4.1.3 Chord . . . 30
4.1.4 Summary . . . 30
4.2 Accuracy . . . 31
4.2.1 1a - Java Concurrency Guidelines . . . 31
4.2.1.1 Results and Discussion . . . 31
4.2.1.2 False Positive/Negatives . . . 35
4.2.2 1b - Eytani Benchmark . . . 36
4.2.3 Part 2 - Web . . . 38
4.2.4 Part 3 - Open-Source Projects . . . 39
4.3 Performance . . . 40
4.3.1 Time . . . 40
4.3.2 Bug representation . . . 44
4.4 Obfuscation . . . 48
4.5 Summary . . . 53

5 Conclusion and future work **55**
5.1 Conclusion . . . 55
5.2 Future Work . . . 56

Bibliography **59**

List of Figures

3.1 Example Locking Bug . . . 17
3.2 Example Benchmark Bug . . . 18
3.3 Example Common Deadlock Bug . . . 19
3.4 ThreadSafe Eclipse plug-in . . . 21
3.5 ThreadSafe Command Line Configuration . . . 22
3.6 ThreadSafe Configuration . . . 22
3.7 FindBugs Plug-in Reporter Configuration . . . 23
3.8 FindBugs Plug-in Detector Configuration . . . 24
3.9 FindBugs GUI New Project . . . 25
3.10 FindBugs GUI Results . . . 25
3.11 Chord Properties . . . 26
3.12 Chord Execution . . . 26

4.1 ThreadSafe Plug-in . . . 27
4.2 ThreadSafe CommandLine . . . 28
4.3 FindBugs GUI . . . 29
4.4 FindBugs CommandLine . . . 29
4.5 Chord CommandLine . . . 30
4.6 Java Concurrency Guidelines Graph . . . 35
4.7 Deadlock and Race condition graph . . . 39
4.8 ThreadSafe Command Line Results Representation 1 . . . 45
4.9 ThreadSafe Command Line Results Representation 2 . . . 45
4.10 ThreadSafe Accesses . . . 46
4.11 Chord Results Representation . . . 46
4.12 FindBugs Plug-in Results Representation . . . 47
4.13 FindBugs GUI Results Representation . . . 47
4.14 Obfuscation Pattern 1 Example . . . 49
4.15 Obfuscation Method 2 Example . . . 51

List of Tables

2.1 False positives/negatives . . . 9
2.2 Static Analysis Tools . . . 10

3.1 Java Concurrency Guidelines . . . 16
3.2 Eytani Benchmark LOC . . . 17
3.3 Common Bugs . . . 18
3.4 Open-Source Projects . . . 19
3.5 Final Test Suite table . . . 20

4.1 Visibility and Atomicity Bugs . . . 31
4.2 Explanation of VNA Bugs . . . 32
4.3 Locking Bugs . . . 32
4.4 Explanation of LCK Bugs . . . 33
4.5 Thread APIs Bugs . . . 33
4.6 Explanation of THI Bugs . . . 34
4.7 Thread-Safety Miscellaneous Bugs . . . 34
4.8 Explanation of TSM Bugs . . . 34
4.9 Guards Bugs . . . 35
4.10 Recall Graph . . . 36
4.12 IBM benchmark Bugs . . . 37
4.13 Deadlocks Bugs . . . 38
4.14 Race Condition Bugs . . . 38
4.15 Open Source Projects Bugs . . . 39
4.17 FindBugs Execution Times . . . 40
4.19 ThreadSafe Execution Times . . . 40
4.20 FindBugs and ThreadSafe Execution Times Graph . . . 41
4.22 Time Performance Table . . . 42
4.23 Graph for ThreadSafe and FindBugs for 60 to 600 lines of code . . . 42
4.24 Graph for ThreadSafe and FindBugs for 60 to 600 lines of code for Chord . . . 42

4.25 Chord Deadlock-analysis Outliers 43
4.26 Open Source Projects Time Performance 44
4.27 Graph for ThreadSafe and FindBugs for JOscar and K9Mail 44
4.29 Benchmark Obfuscated Programs 48
4.30 Finbugs Results on LCK after Obfuscation with Pattern 1 49
4.31 Benchmark Pattern 1 Results 50
4.32 FindBugs Results on LCK after Obfuscation with Pattern 2 51
4.33 FindBugs Results on THI after Obfuscation with Pattern 2 52
4.34 Benchmark Pattern Results 52
4.35 Obfuscation Results 53

Chapter 1

Introduction

1.1 Overview

Concurrency is the idea of making multiple things happening in parallel. In real life, many things like traffic movements, bank transactions and office tasks are happening concurrently. In order to be able to model or control these types of operations, it was very important to develop new programming models and more specifically concurrent programming.

On the one hand, concurrent programs comes with many benefits, such as the execution of more than one task in a certain time period and interaction between tasks. On the other hand, writing correct concurrent programs can be difficult and because of that the programmer must take into consideration issues like safety, liveness and fairness.

Java is one of the first programming languages that had the feature of concurrency designed into the language from the very beginning. Concurrent Java is used widely and nowadays programs are becoming larger and more complex. Additionally, the difficulty in reproducing a concurrency bug and the fact that concurrency bugs can be intermittent make it particularly difficult to find and fix bugs. Apart from the bugs that exist in sequential programs, in concurrent Java there are special bugs like deadlocks and race conditions that are explained in Chapter 2.

Code analysis is a widely-used technique to detect bugs in most programming languages. One of the areas that code analysis finds errors is concurrency. Especially in Java a lot of work has been done to develop tools and find ways to improve the analysis. There are two types of analysis: static and dynamic. The main difference between them is that static analysis [38] can be performed to a program without executing it and dynamic analysis involves executing the program [32]. Between these two options, static analysis is the one that can be used before program is completely finished and accounts to exhaustive testing [38].

Even though the existing static analysis tools use some of the same basic techniques for

finding concurrency bugs there are crucial differences in their accuracy, performance and usability in detecting bugs. Another very important difference between static analysis tools is their ability to avoid false alarms (false positives) and missed bugs (false negatives). Both of them are dangerous, but false negatives are more dangerous, because these are bugs that can remain undetected until they arise in production without warning.

1.2 Motivation

Concurrency bugs are a very special type of bugs in programming. The detection of them is very difficult and it becomes even harder when a programmer tries to detect them manually. This happens because the events in the threads of a concurrent program can be executed in a different order each time the program runs. As a result, bugs can evade even the most rigorous testing regimes and it is rarely possible to reproduce an error.

Developers of programming tools have tried to meet the challenge of finding concurrency bugs by implementing new code analysis tools or by developing new features for the existing tools. On the one hand, there are commercial tools like Coverity Static Analysis or ThreadSafe and on the other hand free open-source tools like FindBugs, Chord and JLint. In the last decades, the number of static analysis tools has increased rapidly and it is difficult for a programmer or a company to choose the most appropriate tool. When a tool is needed the user must take into consideration his requirements from it and think about performance and scalability.

The specific research and evaluation of the tools provided by this project can be used as a guide for a programmer to understand what really matters in choosing the right static analysis tool that matches with his needs and how it can provide the best possible results.

1.3 Approach

This project is based on the idea of choosing three static analysis tools and evaluating them according to their usability, accuracy and performance.

The first step was to study concurrent Java in order to understand how threads work and how they are used. Then, research on concurrency bugs was important in order to find the most common bugs and to understand why they lead to errors. The third step was to study how code analysis and especially static analysis tools work to discover concurrency bugs. Finally, three tools were chosen in order to make the evaluation.

The most extensive part of the tool evaluation included the creation of a test suite with

programs that include bugs followed by comparison of the accuracy of the tools on this set of tests. The test suite is separated into parts that involve the same type of bugs. Furthermore, the performance of the tools is judged by the time they need to find a bug or not and the information that is given with the alert that is produced when a bug is found. It is also important to evaluate the tools according to usability and how easy is to configure and use them.

A special part of the project that is added to the evaluation of the tools is obfuscation. More specifically, we try to ”hide” bugs in source code and examine how easily each tool can be thwarted.

Finally, the conclusion of the project is a final evaluation of the tools that are used and generally the efficiency of static analysis.

1.4 Thesis outline

This dissertation contains 5 chapters. After the *Introduction* there are:

Chapter 2 provides a background on concurrent programming and bugs that exist in this specific domain. Furthermore, code analysis and its types (static and dynamic) are explained and information about certain related published papers are provided.

Chapter 3 is about tools that exist for finding concurrency bugs and especially those that are used in this specific project. Furthermore, the test suite that is used for the evaluation of the tools is presented with details about every part of it. The final part of the chapter is about the configuration of the system and the tools.

Chapter 4 presents the evaluation of the tools according to the results from the test suite. The chapter contains 4 sections and a summary. Each section contains a different evaluation of the tools according to usability, accuracy, performance and a special part that evaluates the tools according to their ability to detect bugs in obfuscated programs.

Chapter 5 summarizes the whole project in a few paragraphs and suggests future work that can be done.

Chapter 2

Background

2.1 Concurrent programming

2.1.1 Overview

The first computers had a single CPU processor and were able to execute only one program at a time. However, with the development of new computer systems, it was very important to improve the processors' performance. Computer multitasking is the scenario of sharing the resources of a single processor in order to interleave the execution of multiple jobs. Otherwise computer resources are wasted while waiting for input. As new systems were introduced, the challenges in programming were increased and this led to the evolution of multithreading [5].

Multithreading is the ability of a single program to contain multiple threads that can run simultaneously. The main difference with multitasking is that threads are part of a single program that can be coordinated in order to achieve work together. Later, the development of multicore processors enabled threads to run on separate processors.

Along with others, from the very beginning Java was one of the first languages that offered multithreading capabilities to developers.

There are two basic units of execution, processes and threads [12]. They are both very important in computers with multiple processors, however they are active in single execution core systems as well. **Processes** are execution environments which are included in programs or applications. **Threads** are lightweight processes that exist within a process and need fewer run-time resources than a process. The main difference between processes and threads is that processes have their own address space and threads share the resources that belong to the process in which they are contained, like memory, CPU and files [12]. The memory that is shared between threads is used to establish communication between them and provide synchronization for their activities. Java is able to use both processes and threads, however features for threads are built directly into the language.

2.1.2 Advantages and Disadvantages

The advantages of concurrent programming are [3]:

⇒ Resources can be used in a better way by complex programs in architectures with multiple cores.
⇒ Short-running tasks are not delayed by long-running tasks.
⇒ Tasks are controlled in a better way when they need resources in order to continue with their execution.
⇒ There is more protection because of the isolation of the activities in threads.

The disadvantages are [2, 3]:

⇒ Danger of special bugs like deadlock and race condition.
⇒ Threads can be expensive because of scheduling and synchronization.
⇒ The state of program should not be corrupted by the tasks.
⇒ Concurrent programs might need more time to run than their corresponding sequential version (use of one single thread) [2] even with multiple processors, if the task is inherently sequential.

2.1.3 Problems in concurrent programming

Generally, it is difficult to write and understand concurrent programs because of the potentially different orderings of events of the several threads and the absence of proper timing over the order. As far as concerns concurrency bugs, they are harder to be detected in comparison with usual bugs, as it is extremely hard to reproduce them. The main reason for this is that there are many possible interleavings between thread's operations and it is not under the programmer's control which of them is chosen.

In a single-threaded program usually debugging is not very hard for developers. In concurrent programs it is more difficult because of the several synchronizations that might exist and the communication and the interaction between the threads. Basically, there are two kinds of issues: **safety** and **liveness** issues [11].

On the one hand, a safety property ensures that nothing bad will happen [39]. The main idea behind safety is correctness [40]. Correctness means that each class of the program follows its specifications. An example of a safety property is lack of race condition. On the other hand, a liveness property asserts that something good eventually happens. It is extremely significant for all the threads in a concurrent program to have at least a chance to execute. Examples of liveness are fairness and lack of deadlock or livelock [29].

The most common categories of problems in concurrent programming are:

Deadlock

A deadlock is a situation where multiple threads wait forever because of a cyclic locking dependency [40]. More specifically, a deadlock occurs when a thread is in a waiting state because it needs resource A that is held by another thread, which also needs another resource, held by the first thread, in order to provide resource A to the first thread. When the thread is not able to change its state, then we say that the system is in a deadlock [43, 10].

Livelock

Livelock is similar to deadlock. The main difference is that in livelock threads act but there is no progress to reach the final goal of the program. In this scenario two or more processes continually change their state in response to changes in other processes. As a result, none of the processes complete.

A simple example used to describe livelock problem is a scenario where two people meet in a path and each one tries to step around the other. However, they finally end up changing from side to side getting in each other's way [43].

Starvation-Fairness

A thread can starve to death when it is not granted CPU time because other threads are granted too much time. The solution to this is fairness, which is the idea of granting all the threads a chance to execute.

Starvation can happen because of not setting priorities wisely or incorrect synchronization. Fairness can be achieved by using locks properly and synchronizing [5].

Race Condition

A race condition is a situation where two or more threads that have access to the same data try to change it at the same time [10]. The main reason for a race condition is a flaw in the ordering or timing of the events. In other words, a race condition happens when the correctness of the computation of a program depends on timing [40].

2.2 Code analysis

As mentioned previously, there are two kinds of code analysis: **static** and **dynamic**. Simply, static analysis is about gathering information about a program from the source code and dynamic analysis from its execution.

2.2.1 Static analysis

Static analysis is about detecting errors in source code without executing it [39]. It is performed in a special phase of the Security Development Lifecycle called Code Review (or White-box testing) [25]. For this project, the specific process deals with understanding the code structure and providing information about how code adheres with industry standards. Static analysis can be done manually (called ”program understanding” or ”program comprehension”), however it is very time consuming so mostly it is done by automated tools.

Static analysis tools track vulnerabilities in non-running code by using the following two techniques:

Data Flow Analysis is used to gather data about the dynamic behaviour of a program in a static way.

Taint Analysis identifies variables that have been tainted (potentially equal to invalid values) by the users that can lead to vulnerabilities. There are languages like Ruby and Perl that have taint checking built into them [16].

The advantages of static analysis are:

- It is fast when automated tools are used.
- Entire codes bases can be scanned.
- Provides the ability of finding defects early in the development (even before the code is finished).

The disadvantages of static analysis are:

- Manual static analysis is time consuming.
- Not all languages are supported by the tools.
- False positives (false alarms) and false negatives (bugs not-found) are produced because of high complexity or size of a program.
- Vulnerabilities that are introduced in the runtime environment are not found.

2.2.2 Dynamic analysis

Dynamic analysis is the technique of analysing a program during its execution time. The process contains steps like preparation of input data, run of the program and analysis of the output. The goal is to find bugs in a program while it is running and not by examining the code [22]. Besides Java, dynamic analysis can be used in languages like C/C++, C#, Python, PHP etc.

Some of the advantages [19] of dynamic analysis are:

- Identifies problems in runtime environment.
- Can be used against any application.
- Tools are flexible.
- Is able to collect temporal information.
- Has access to actual running situation.

There are also disadvantages [19] like:

- Detects errors only on the route defined by the input data.
- Only one path can be checked at a time and the developer must run many tests to check all the paths.
- A lot of effort is required by the user.
- When the test runs on a real processor, execution of incorrect code may lead to undesirable situations.
- The automated tools that exist produce false positives and false negatives.

Some of the most used existing dynamic analysis tools are Avalanche, Parasoft Jtest, Purify and Gcov.

2.2.3 Static analysis vs Dynamic Analysis

On the one hand, static analysis refers to strategies used to analyse code for all the possible paths before the execution. On the other hand, dynamic analysis is used to verify the properties of the system for one execution trace. The analysis of all the paths makes static analysis the most desirable technique [26]. However, an advantage of dynamic analysis with respect to static is that certain information is available only at runtime [31]. Moreover, the system's behaviours might depend on the environment where the program is executed.

However, the two techniques can be combined and the user can benefit from the "soundness" of static and of the efficiency of dynamic analysis. Soundness is that properties hold for all the paths [31].

2.2.4 False Positives and False Negatives

Below there is a table that characterises false negatives and false positives:

		FOUND ???	
		YES	**NO**
ARE THEY	**YES**	True Positive	False Negative
BUGS ???	**NO**	False Positive	True Negative

Table 2.1: False positives/negatives

A false positive is a warning produced by the tool but is not an actual bug. A false negative is a bug that exists in the program but the tool did not manage to find it. The main reason false positives arise is that the detected source code has many similarities with known malware codes. False Negatives are bugs that the tool was not able to detect. As it can be understood, it is very important for an analysis tool to minimise as much as possible the amount of false positives and false negatives and identify only the real bugs. By achieving this, the tool will finally provide the user the ability to work only on the real problems.

2.3 Static analysis tools

A table of static analysis tools that exist for Java follows:

Open-Source	**Commercial**
FindBugs	ThreadSafe (Contemplate)
Chord	Security Advisor (Coverity)
PMD	IntelliJ (JetBrains)
JLint	Source Patrol (Pentest)
Java PathFinder	Source Code Analysis (HP/Fortify)
SWAAT Project (OWASP)	Parasoft Test/JTest (Parasoft)
VCG	

Table 2.2: Static Analysis Tools

2.4 Previous Evaluations of Static Analysis Tools

This project is closely related to a paper by D.Kester, M. Mwebesa and J.S. Bradbury [32], which analysed and tested three open-source static analysis tools (FindBugs, Chord and JLint). The test suite included 12 programs that had bugs such as deadlocks and race conditions. The main difference is that we have focused on more tools and we have used more test programs and more kinds of concurrency bugs.

Another similar project is called ”Comparing Four Static Analysis Tools for Java Concurrency Bugs” [39]. Their research is about Coverity, Jtest, FindBugs and JLint tools. These tools are compared by using tests that are included in a published benchmark [27]. The evaluation is based on common bug patterns like deadlock and data race.

Another study that is related to the project is from the University of Queensland by Brad Long, Paul Strooper and Luke Wildman [34]. This project contains a theoretical analysis of common concurrency problems like livelock, deadlock and starvation. Furthermore, there is a comparison between static and dynamic analysis tools and an evaluation of FindBugs and JLint. However, the project focuses on the failures of the tool and not on the domains that a tool is helpful.

Another paper that tried to deal with the same problem is called "Effective Static Analysis to Find Concurrency Bugs In Java" [38]. The research was done by the IBM Group and they managed to create a concurrency bug detector based on the Eclipse and the WALA toolkit. Moreover, they used it on bugs, evaluated it and, finally, compared it with FindBugs. Even though their study is far from just evaluating tools, there is a very interesting analysis of concurrency bug patterns.

Another interesting study was published in 2008 by Shan Lu, Soyeon Park, Eunsoo Seo and Yuanyuan Zhou [37]. Even if there are not any tools involved into the research there is a very helpful theoretical study on bugs and their characteristics with respect to open-source bug sources like Apache, MySQL and Mozilla.

FindBugs and PMD are involved in a paper [44] published in 2008. The study does not focus on concurrency bugs, but evaluates the tools according to the bug patterns they are able to detect. With respect to efficiency, the tool calculates costs and does a complete analysis of problems as far as concern bugs and patterns.

A similar project with the previous one is "A Comparison of Bug Finding Tools for Java" [41]. The tool that are used are five: Bandera, ESC/Java 2, FindBugs, JLint and PMD. The interesting part of the evaluation of the tools according to the number of warnings is that the test suite includes open-source projects with 30.000 lines of code and uses the time needed in order to evaluate the performance as well. Even if it not based on concurrency errors there is a special section, which analyses them.

"Finding Bugs is Easy" [30] includes only FindBugs and is not based on concurrency, however the techniques and the strategies of the tool are explained and the bug patterns, that contain concurrency, are analysed with respect to the tool.

Another paper [21] which contains only FindBugs and evaluates it with not only concurrency bugs is by the University of Maryland and Google Inc. The errors that were used in the evaluation are presented theoretically and practically (with examples) and the conclusion contains very significant points.

Two papers [26, 28] describe how the benchmark used in the evaluation process was created and what it contains. They were able to analyse why the specific programs were included in the benchmark and what was the experience that the developers had.

Finally, obfuscation is a very important part of the project and in order to have a guide

on how to attempt to thwart the tools a paper called ”Program Obfuscation: A Quantitative Approach” [20] was used. There is not a tool evaluation in it, although the metrics and the techniques like control flow and data are presented with details in a way that they are easily understood.

Chapter 3

Tools and Configuration

3.1 Tools

3.1.1 ThreadSafe

3.1.1.1 Overview

ThreadSafe is a commercial static analysis tool for finding concurrency bugs in Java code. ThreadSafe can find underlying intentions in design and warn the programmers about problems. The techniques [17] that the tool uses are:

Context-sensitive inter-procedural analysis ThreadSafe is able to track synchronizations on locks across method calls, but it is also able to track references to shared data. This is extremely significant, because more defects are found and the false positives are reduced.

Inter-class analysis There is also a tracking of the interactions of classes in order to avoid bugs that might arise from mismatches between the assumptions of the programmers of the different classes.

Aggregation and filtering of results Filtering is also important to reduce the number of false positives in the results of the analysis.

Straightforward presentation of results When a bug is detected, important information is presented to the programmer, like location and details about the defect.

ThreadSafe is capable of finding all kinds of concurrency bugs like race conditions, inconsistent synchronization, atomicity problems and deadlocks.

3.1.1.2 Requirements

ThreadSafe supports the following [17]:
Platforms:

- Windows XP or later
- Mac OS X 10.6 or later
- Linux (glibc 2.3 or later)

Further requirements are:

- Java SE 6 Update 10 or later
- Eclipse 3.6 or later (ThreadSafe for Eclipse)
- SonarQube 3.5 or later (ThreadSafe for SonarQube)

3.1.2 FindBugs

3.1.2.1 Overview

FindBugs is an open-source static analysis tool from the University of Maryland used for detection of bug patterns in bytecode. Java classes are analysed and the bugs that are found are matched with source code. Byte Code Engineering Library and ASM bytecode framework are used in order to do this [6]. The bugs that can be detected by FindBugs can be typographic faults, non-proper use of language or mistakes during maintenance. Even if the most common use is as a plugin in Eclipse it can also be used as a command-line or GUI tool or as a plugin in other platforms like NetBeans and Hudson.

The strategies of the tool can be separated in the following categories [30]:

Linear Code Scan Detectors that scan linearly bytecode.

Class Structure Detectors that just look in the structure without looking at the code.

Control Sensitive Detectors the use control flow graph to analyse methods.

Dataflow Control and Data flow graphs are used for taking both control and data into account.

The detectors that FindBugs uses belong to one or more of the categories: Single-threaded, Thread/Synchronization correctness, Performance Issue and Security and Vulnerability to detect malicious untrusted code.

As the developers of the tool claim it is possible sometimes for the tool to make mistakes which lead to false-positives. However, the number of them has decreased as new releases are implemented [6]. They also claim that even a project with many lines of code can be analysed in no more than a few minutes.

3.1.2.2 Requirements

FindBugs is a tool that is platform independent so it can definitely be used in platforms like GNU/Linux, Windows, and MacOS X platforms. Also, in order to use FindBugs a runtime environment compatible with Java 2 Standard Edition or later is required.

Furthermore, at least 512 MB of memory are needed for the analysis and for larger projects it is possible to need more.

The latest release (FindBugs 3.0.0), that is used in the specific project, requires at least Java 7 and uses ASM 5 which means that the detectors are upgraded [6] to work for the new version of Java.

3.1.3 Chord

3.1.3.1 Overview

Chord is a relatively new open-source program analysis platform which was developed in order to detect bugs in concurrent Java. The project was started by Mayur Naik and Alex Aiken at Georgia Institute of Technology int 2010. Since then many people have contributed with their applications and algorithms. At present, the source code is maintained by Mayur Naik and Ariel Rabkin [15].

The platform is used to allow users to design and implement static and dynamic program analyses written in Java or in Datalog.

Four kinds of static analysis methods are used by Chord:

Call-graph analysis Representation of relationships of sub-routines in the program.

Thread-escape analysis If an object can be restricted to a thread without escaping to another.

Lock analysis Analysis of locks in order to make sure that locking has been done correctly.

Alias analysis If an object is accessed by multiple threads.

The reason that the specific techniques are used is that they are good with scalability and precision and able to help in the detection of bugs in concurrent Java [32]. However, it is significant to determine the analysis scope to use the techniques. Analysis scope shows the reachable methods and classes. For the specific research this happens statically, although it can also be determined dynamically.

3.1.3.2 Requirements

Chord can be used on many platforms, including Windows/Cygwin, MacOS and Linux. However, as we can see below for the specific research we used the version that exists for Linux.

Furthermore, Joeq (Java compiler framework), Javassist (Java bytecode manipulation framework) and bddbdd (BDD-based Datalog server) are required.

3.2 Test Suite

One of the most important parts of the projects was to create the test suite to evaluate the tools as thoroughly and as fairly as possible. In order to make a clear evaluation, the suite is divided into 3 parts. The resources that were used are the following:

Part 1a - Java Concurrency Guidelines

Java Concurrency Guidelines [35] was developed as a part of The CERT Oracle Secure Coding Standard for Java [36]. Even though the programs are not more than 40 lines there is a big variety of categories. The categories that are about concurrent Java and are used in the project are **Visibility and Atomicity**, **Locking**, **Thread APIs** and **Thread-Safety Miscellaneous** guidelines. Each one of the categories contains from 4 to 12 types of bugs and for every type there are short examples that contain the bug(non-compliant) and ways to fix it (compliant). For every type of bugs both programs (compliant and non-compliant) were used in order to track false alarms (false positives) that the program might find. The number of the tests is provided below:

Category	Number of programs
Visibility and Atomicity	7
Locking	12
Thread APIs	6
Thread-Safety Miscellaneous	4
Total	**29**

Table 3.1: Java Concurrency Guidelines

Below there is an example-program of the Guidelines:

```
//LCK01-J. Do not synchronize on objects that may be reused

package locking;

public class SynchronizedReused {
    private final Boolean initialized = Boolean.FALSE;

    private void doSomething() {
            synchronized (initialized) {
            }

    }

}
```

Figure 3.1: Example Locking Bug

Part 1b - Eytani et al Benchmark

The Eytani et al Benchmark [27] was developed by the IBM Haifa Research Lab in association with the University of Haifa. The repository that was created contains 40 programs with bugs that do not belong to many different types of bugs. However, the benchmark is helpful as far as concerns scalability. The following table presents a categorisation according to the number of lines of code of the programs:

Programs	Lines of code
7	<100
20	100-300
6	300-500
3	500-1000
3	1000-3000

Table 3.2: Eytani Benchmark LOC

The project uses 30 of the programs which is the number of the programs that were provided. The benchmark has one more program called JOscar that contains more than 15000 lines of code, that is used in the final part of the suite.

Below there is an example-program of the benchmark as it was in Eclipse:

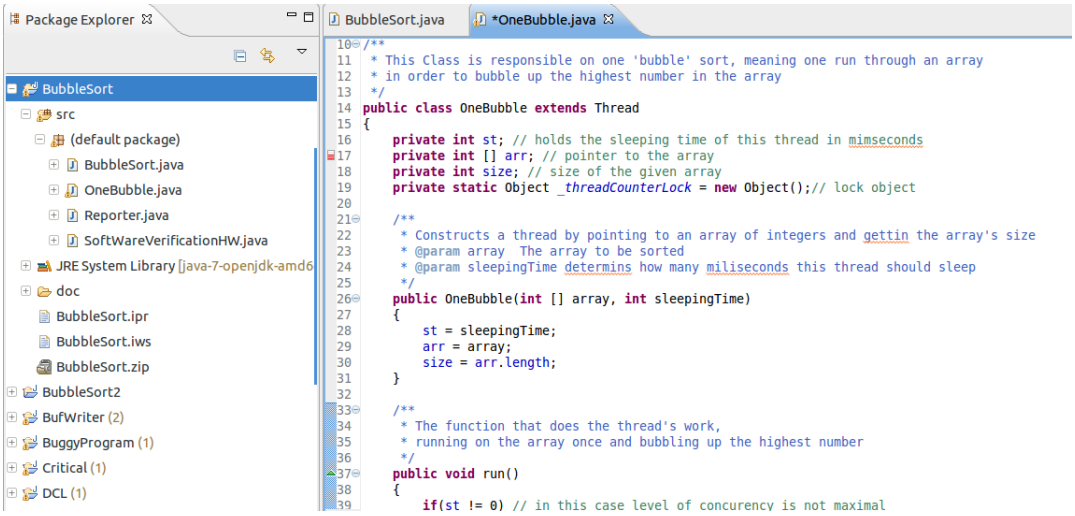

Figure 3.2: Example Benchmark Bug

Part 2 - Web (Common bugs)

Deadlock and Race Condition are the most common types of bugs in concurrent programming. These tests contain programs that were found from reliable resources [40, 8, 7, 13, 18] on the web and were used in order to evaluate how the tools work on simple examples.

Bug	Number of Programs
Deadlock	6
Race Condition	7
Total	**13**

Table 3.3: Common Bugs

Below there is an example-program of the common bugs:

```
public class Dead5 {

    String str1 = "Java";
    String str2 = "UNIX";

    Thread trd1 = new Thread("My Thread 1"){
        public void run(){
            while(true){
                synchronized(str1){
                    synchronized(str2){
                        System.out.println(str1 + str2);
                    }
                }
            }
        }
    };
    Thread trd2 = new Thread("My Thread 2"){
        public void run(){
            while(true){
                synchronized(str2){
                    synchronized(str1){
                        System.out.println(str2 + str1);
                    }
                }
            }
        }
    };

    public static void main(String a[]){
        Dead5 mdl = new Dead5();
        mdl.trd1.start();
        mdl.trd2.start();
```

Figure 3.3: Example Common Deadlock Bug

Part 3 - Open Source Projects

This part of the test suite is concerned with how well the tools work with large projects. So, after testing many open source projects we decided to use three of them that contain more than 15000 lines of code. The specific projects were chosen, because they are easy to build and contain concurrency.

Open-Source Project	Version	Description
k9Mail	1.0	Open-source email client based application
JOscar	0.9	Robust library for connecting to AOL and ICQ
ArgoUml	0.34	Open source UML modeling tool

Table 3.4: Open-Source Projects

ThreadSafe and FindBugs were tested on all test programs. However, Chord needs a main method in order to analyse a program and it does not work on programs with many lines of code. So, it could not be tested on many of our example programs. In order to summarise the tests that were done we created the following table:

Title	Programs	Tools
Part 1a - CERT	29	ThreadSafe, FindBugs
Part 2b - Eytani Benchmark	30	ThreadSafe, FindBugs,Chord
Part 3 - Web	13	ThreadSafe, FindBugs,Chord
Part 4 - Open Source Projects	3	ThreadSafe, FindBugs
Total	**75**	

Table 3.5: Final Test Suite table

3.3 Configuration

3.3.1 System configuration

For the project one computer system was used, which includes:

- Software
  - Linux Operating System
  - Ubuntu Desktop version 13.10
- Hardware
  - Processor: Intel Core i5-4200U CPU @ 1.60GHz x4
  - Memory: 5.4 GiB
  - OS type: 64-bit

3.3.2 Tools installment and configuration

3.3.2.1 ThreadSafe

ThreadSafe is a commercial product that was used for the project with a licence that professor Sannella was able to provide. The version that was used is ThreadSafe 1.3.1. There are three releases that are developed:

- Command Line
- Plug-in for Eclipse
- Plug-in for SonarQube

For the first tests that were executed we used the plug-in for Eclipse. However, for most of the tests the release for Command Line was used. It was preferred from the plug-in, because it provided more configuration options. The analysis component is the same for both versions.

Eclipse plug-in

As far as concerns the plug-in, the figure below presents the rules that can be included in the analysis of the program. The rules that exist are more than 20 and there are details for each one. For the testing of the tool all the rules were included.

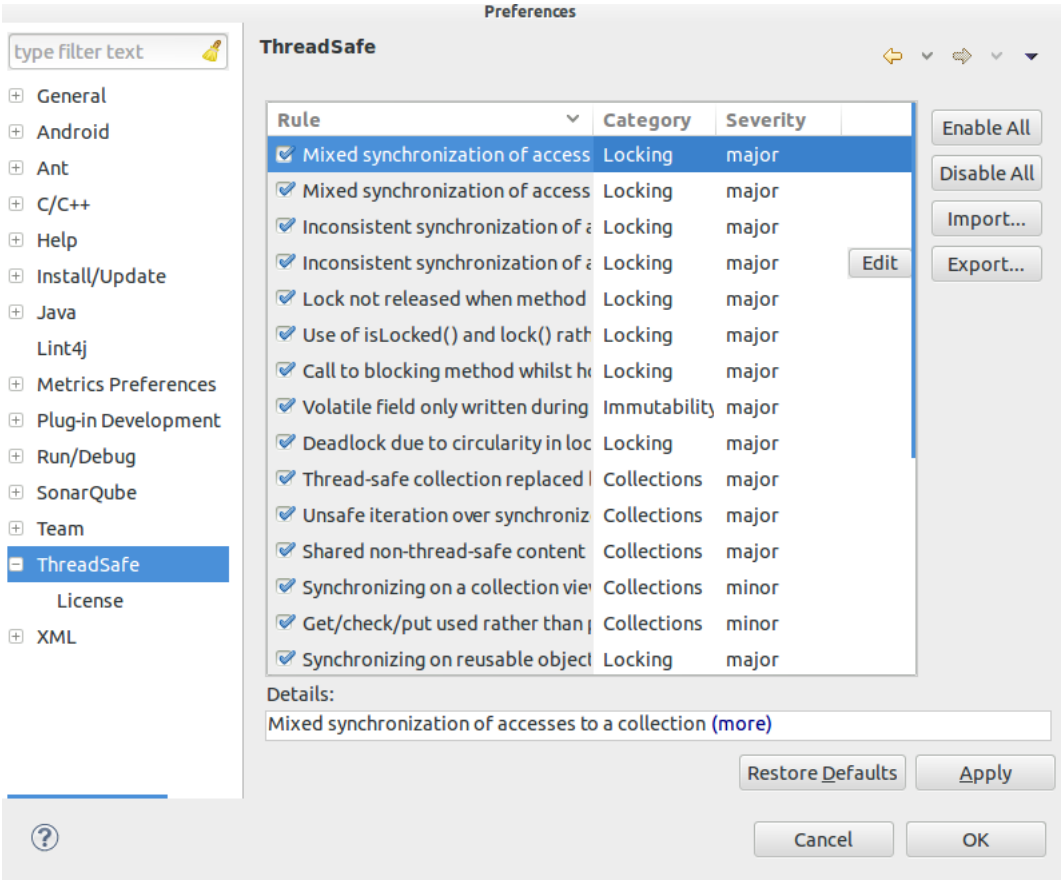

Figure 3.4: ThreadSafe Eclipse plug-in

Command Line

For the command line tool there are more configurations that could be made. In order to analyse a program it is important to store a txt file outside the project directory called ”project-config.properties” that contains information about the execution of ThreadSafe. There are mandatory and optional configurations that can be made. The choices that we have are provided in the user guide of ThreadSafe [17]. They are provided below:

Property	Type	Description
Mandatory:		
`binaries`	Paths	Directories/JARs containing bytecode to be analyzed.
`sources`	Paths	Directories containing sources of code to be analyzed.
`outputDirectory`	Path	Output directory for HTML report.
Optional:		
`baseDirectory`	Path	Project root directory.
`libraries`	Paths	Directories/JARs required on the project classpath.
`projectName`	String	Project name to show in the report.
`rulesFile`	Path	Rule configuration file.

Figure 3.5: ThreadSafe Command Line Configuration

When the configuration file is ready then it is time to run ThreadSafe. For the project all the configurations that exist were tried out. However, the configuration that was used for the final evaluation of ThreadSafe contained four of them: projectName, sources, binaries and outputDirectory(Figure 3.6).

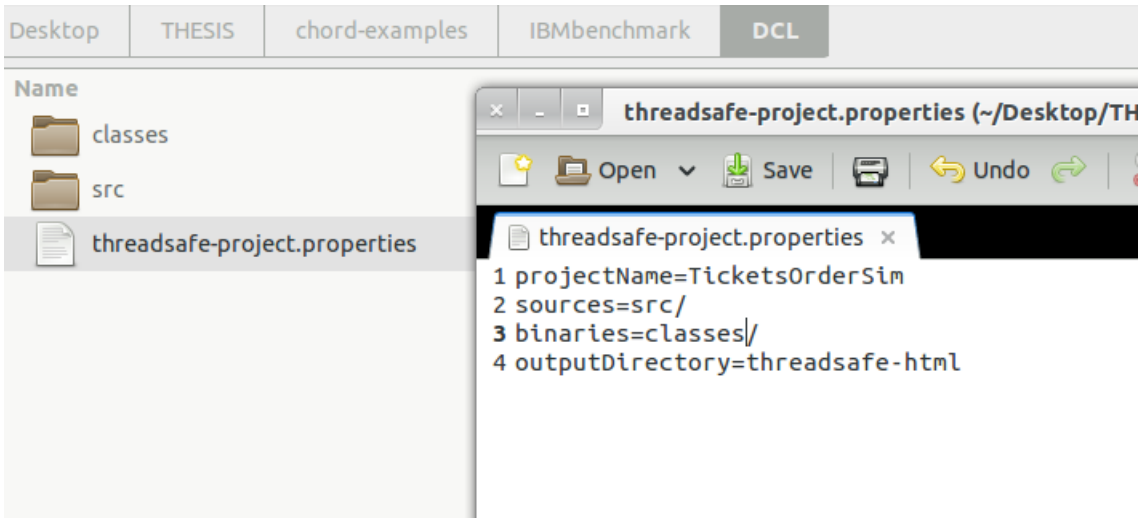

Figure 3.6: ThreadSafe Configuration

Then, in order to run the tool there are two steps:

1. Move to the directory of the project
   cd <project-root-directory>
2. Run the command for ThreadSafe
   java -jar <installation-directory>/threadsafe.jar

The results are provided in an html file.

3.3.2.2 FindBugs

FindBugs 3.0.0 is a free open-source tool that has three releases: Command Line, plug-in for Eclipse and GUI. Even if all of them were used the project focused more on the plug-in and the GUI. Furthermore, the analysis component is the same in all of them.

Eclipse Plug-in

The preferences that could be set for FindBugs are shown below.

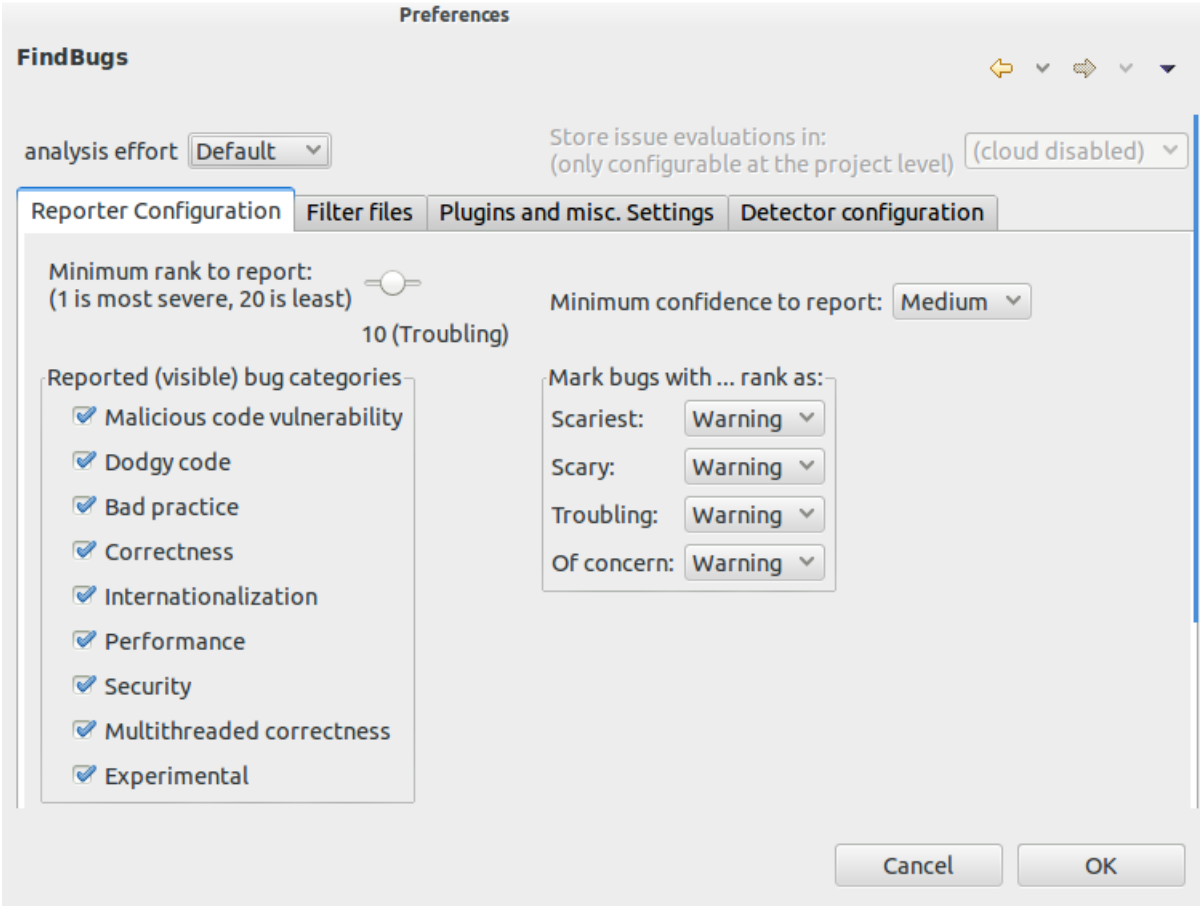

Figure 3.7: FindBugs Plug-in Reporter Configuration

Some important observations from the configuration are:

- The user is able to choose the types of bugs that the tool must detect.
- There is a minimum rank of bugs to report, which is between 1(scariest) and 20(of Concern).

For the project the only category that was needed was the one for multithreaded programs and the minimum rank was set to 20 in order to detect all the bugs that exist in the program.

Furthermore, filter files can be used and plug-ins can be installed. Below there is a figure for the tab ”Detector configuration”.

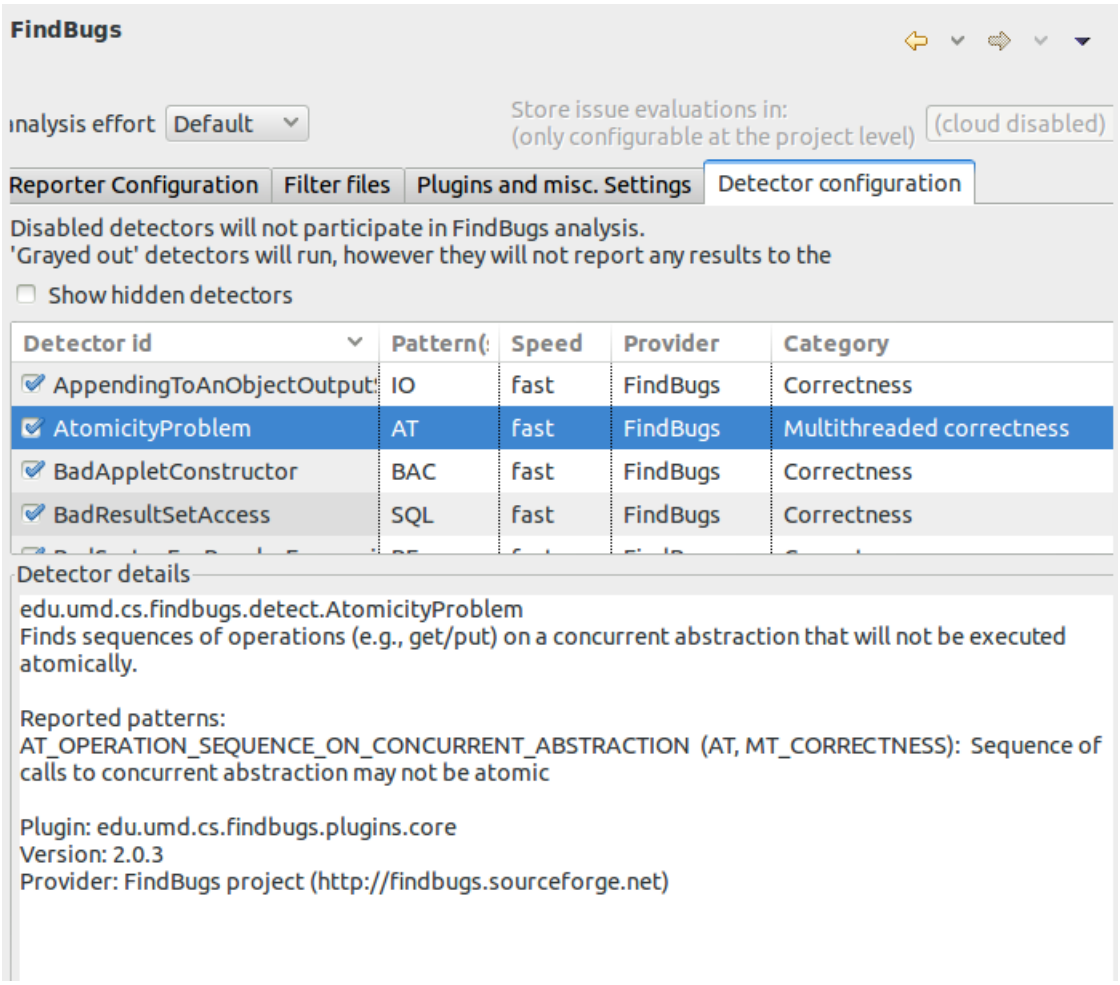

Figure 3.8: FindBugs Plug-in Detector Configuration

In this tab we can see that FindBugs and ThreadSafe are quite similar in configuration. For each bug-type there are exact categories that can be chosen. Furthermore, information for each type is provided. For the project only the types that concern concurrency bugs are used.

Graphical User Interface(GUI)

In order to analyse a program a new project has to be created:

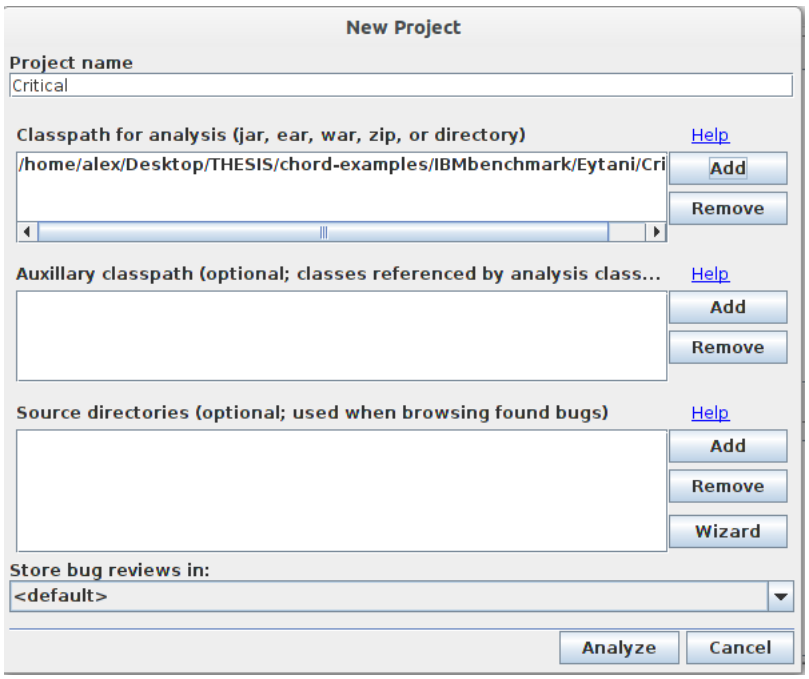

Figure 3.9: FindBugs GUI New Project

Then the program is analysed and finally the tool presents the results.

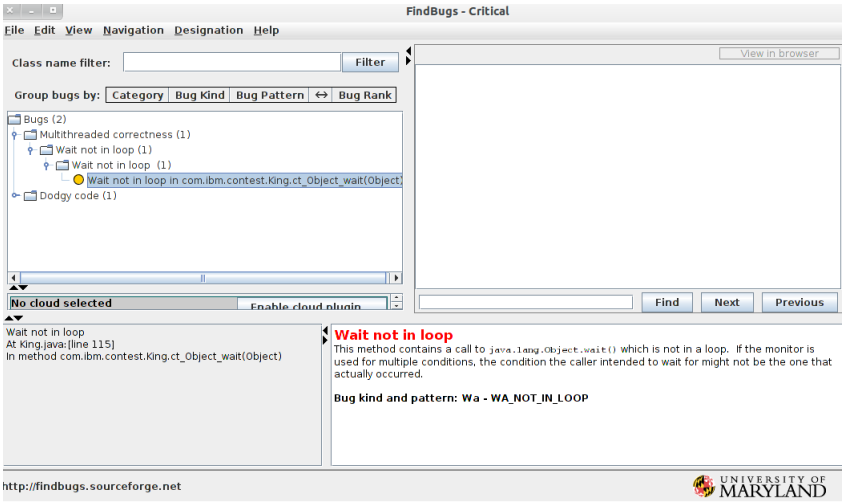

Figure 3.10: FindBugs GUI Results

Something important that should be mentioned is that FindBugs GUI can analyse programs by their .class files. As a result, in order to use the specific interface to detect bugs in a program, we have to compile it before.

3.3.2.3 Chord

Chord is developed to be used only from the command line. The first step in order to run chord is to create a file called ”chord.properties” in the project direction. There are many prop-

erties that could be used [1]. The properties that were used for the project are the following:

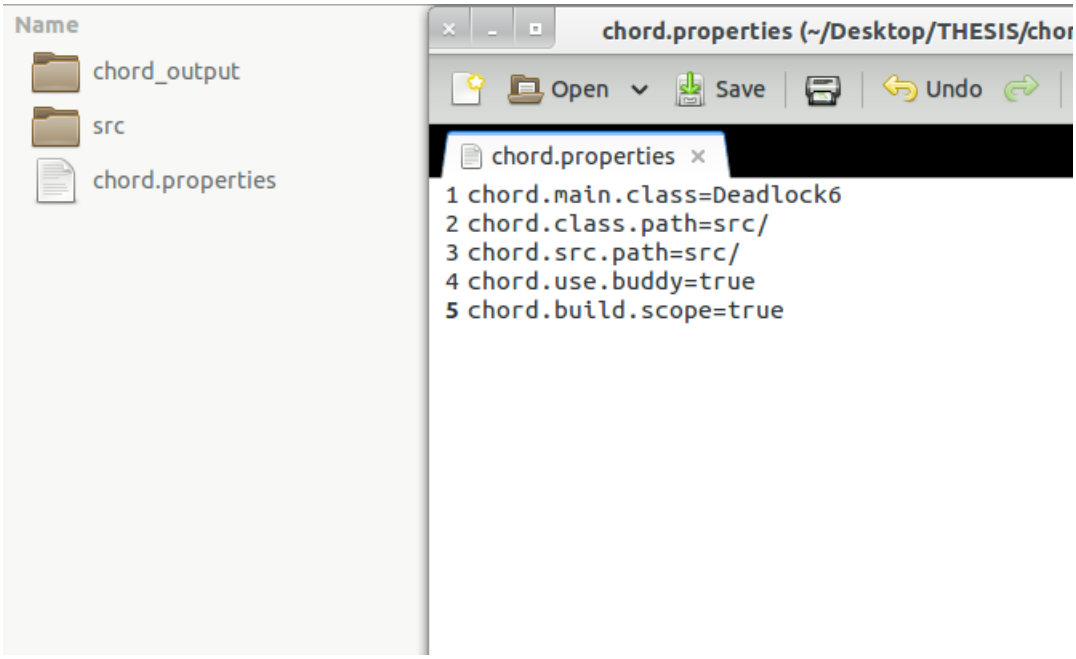

Figure 3.11: Chord Properties

The chord.use.buddy property sets that the BDD library will be used in the analysis and the chord.build.scope that the scope of the Java program will be computed.

In order to run the tool it is important to set the project directory and configure the ”chord.run.analyses”. There are two analyses that were used: ”deadlock-analyses” and ”datarace-analyses”. Each one of them follows a different technique in detecting errors. For every test both of them were used in order to have as much information as possible. The command for a deadlock-analysis is the following:

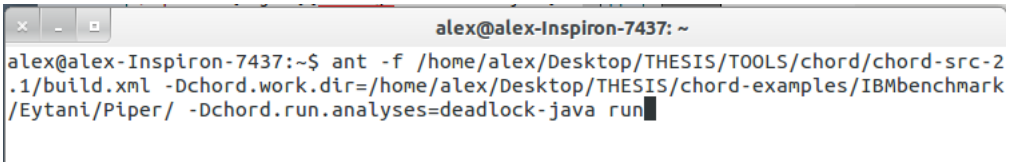

Figure 3.12: Chord Execution

Chapter 4

Evaluation of the tools

4.1 Usability

This section is about evaluating the tools according to their usability. Even this is extremely subjective there is an attempt to make objective points about each one of them.

4.1.1 ThreadSafe

Eclipse plug-in

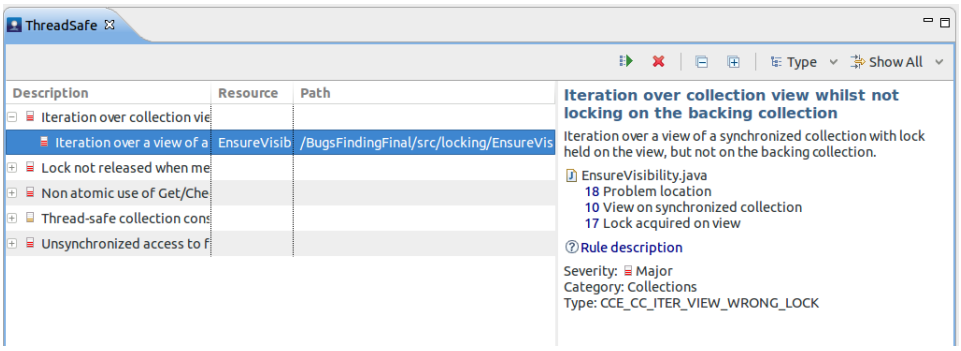

Figure 4.1: ThreadSafe Plug-in

The ThreadSafe plug-in for Eclipse has a lot in common with the FindBugs plug-in. Two similarities are that both of them are able to scan just a class or a package and not only the whole project and that the user can configure the categories that the tool must focus in each scanning.

Furthermore, after the analysis of the desirable piece of code there are marks that present the bugs to the user by showing the exact lines of code that the errors occur. In addition to FindBugs, the window that presents the bugs is better organised with less, but more precise

information.

Command Line

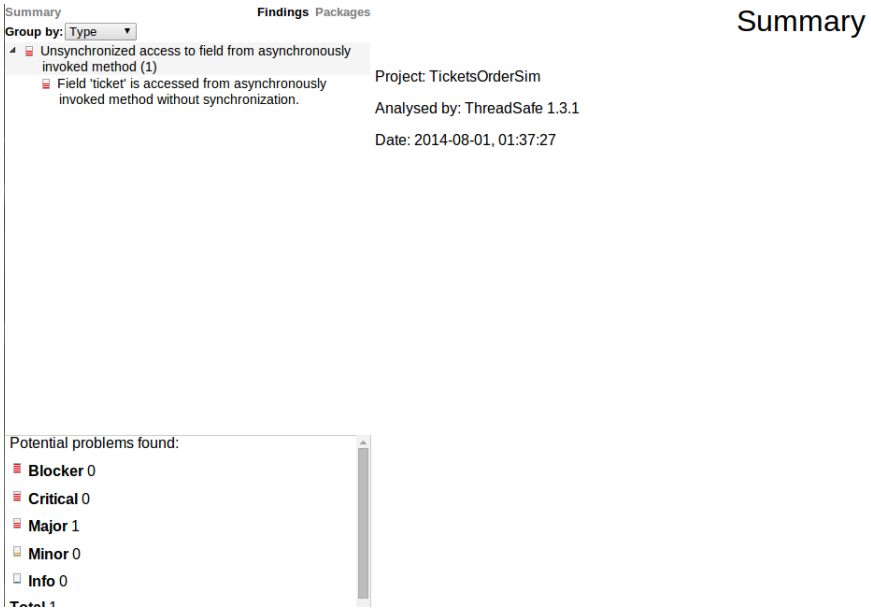

Figure 4.2: ThreadSafe CommandLine

Even though the plug-in for Eclipse is useful in presenting the bugs to the user, the most valuable release for the specific tool is the command line version. The specific version works like Chord. Even if the user must set the properties manually the output of the program presents the bugs in a way that helps the user understand the problem.

4.1.2 FindBugs

Graphical User Interface

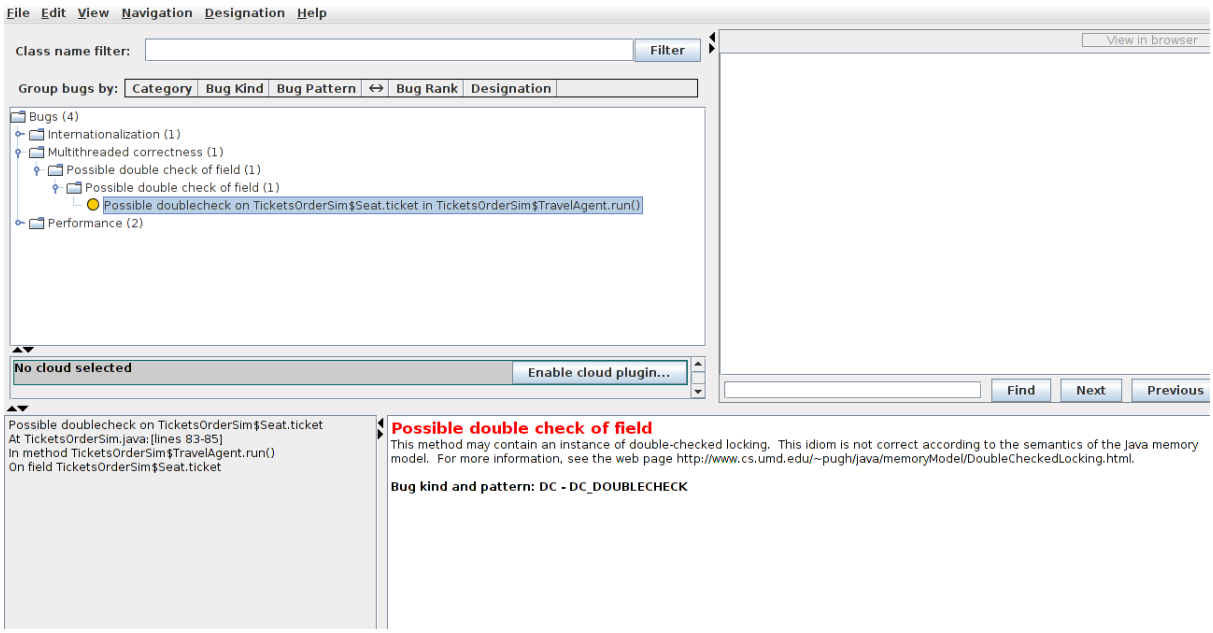

Figure 4.3: FindBugs GUI

In the GUI release of the tool the bugs are presented in two ways. On the one hand, there is a window which shows the bugs that were detected divided in eight categories (Multithreaded Correctness,Performance, Internationalization, etc.). On the other hand, there is another window which shows detailed information (what and where is the error exactly) about each defect.

The user of the GUI is able to configure the bugs that the analysis must focus on choosing the appropriate categories or the bugs that affect the code the most (Rank 1-4:scariest, etc.) in order to save time and memory.

Eclipse plug-in

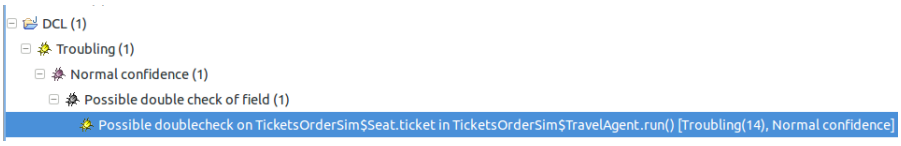

Figure 4.4: FindBugs CommandLine

The plug-in is probably the most useful for someone who needs to scan large projects. The tool can analyse a specific class or package by running the analyser only for it. There is also the ability to configure the analysis with the same preferences that are provided by the GUI (categories of bugs, ranking).

After the analysis ”a small bug” is used to show the presence and the exact position (line of code) of the error. This is very useful, because despite the window that presents all the bugs the user can see immediately the position of the bug in the code.

Command Line

The release for command line is the hardest to use of the three of them. This happens because the user must set the options of the program like arguments, home directory and effort in order to increase or decrease precision in the detection of bugs.

4.1.3 Chord

Command Line

Datarace Reports (Grouped By Field)

Details	Trace 1		Trace 2	
	Thread	Memory Access	Thread	Memory Access
1. Dataraces on TicketsOrderSim.bug_accured				
1.1	TicketsOrderSim$TravelAgent.run()	TicketsOrderSim.check_ticket_details(int) (Wr)	TicketsOrderSim.main(java.lang.String[])	TicketsOrderSim.main(java.lang.String[]) (Rd)
1.2	TicketsOrderSim$TravelAgent.run()	TicketsOrderSim.check_ticket_details(int) (Wr)	TicketsOrderSim.main(java.lang.String[])	TicketsOrderSim.main(java.lang.String[]) (Rd)
1.3	TicketsOrderSim$TravelAgent.run()	TicketsOrderSim.check_ticket_details(int) (Wr)	TicketsOrderSim.main(java.lang.String[])	TicketsOrderSim.main(java.lang.String[]) (Rd)
1.4	TicketsOrderSim$TravelAgent.run()	TicketsOrderSim.check_ticket_details(int) (Wr)	TicketsOrderSim.main(java.lang.String[])	TicketsOrderSim.main(java.lang.String[]) (Rd)
2. Dataraces on TicketsOrderSim.bug_count				
2.1	TicketsOrderSim$TravelAgent.run()	TicketsOrderSim.check_ticket_details(int) (Wr)	TicketsOrderSim.main(java.lang.String[])	TicketsOrderSim.main(java.lang.String[]) (Rd)
2.2	TicketsOrderSim$TravelAgent.run()	TicketsOrderSim.check_ticket_details(int) (Wr)	TicketsOrderSim.main(java.lang.String[])	TicketsOrderSim.main(java.lang.String[]) (Rd)

Figure 4.5: Chord CommandLine

Chord is released only for use in command line. As far as concerns the user, the number of the properties that can be used in the configuration is extremely high and the user must takes into consideration what are the needs for each analysis in order to find a way to save time, and most important, memory. In the first tests of the tool for the specific project when a variety of properties were used there were times that the tool crashed because of not enough memory or needed more that 7 minutes to finish with a simple analysis.

Another observation that was made is that in order to run the tool for a project there were properties like ”chord.run.analyses” that are crucial for the detection of bugs of types like deadlock or race condition that were not so obvious in the documentation of the tool.

4.1.4 Summary

In order to summarize for this section of the evaluation, we could say that FindBugs is the most easy of the three tools to use because of the well-developed GUI and the plug-in that provides many conveniences to the user. ThreadSafe is better in presenting the bugs to the user but the command line version is not so easily used. Finally, Chord is the hardest to be used efficiently because of many different configurations that provides to the user that must be chosen wisely.

4.2 Accuracy

4.2.1 1a - Java Concurrency Guidelines

As we said in chapter 3 the tests are separated in 3 parts. The first part contains ”Java Concurrency Guidelines” [36]. There are four categories. For each test there is a brief explanation and a degree of priority, which was assigned using a measure based on critical analysis [33]. P1 is the bug with the highest priority and P9 with the lowest. The programs that are used are all non-compliant. However, the compliant programs were used as well. The results from them are in section 4.2.1.2.

4.2.1.1 Results and Discussion

The first one is Visibility and Atomicity. The results for the 7 programs are the following:

	Priority of Bugs	ThreadSafe	FindBugs
VNA00-J	P8	✗	✗
VNA01-J	P4	✗	✗
VNA02-J	P8	✗	✗
VNA03-J	P4	✓	✗
VNA04-J	P4	✗	✗
VNA05-J	P2	✗	✗
VNA06-J	P8	✓	✓
Total		**2/7**	**1/7**

Table 4.1: Visibility and Atomicity Bugs

Discussion: In the specific section of CERT concurrency guidelines it is obvious that ThreadSafe detects 28.5% of the bugs and FindBugs 16.6% of them. It is also important to mention that ThreadSafe detects a medium and low-priority bug (VNA03-J and VNA06-J). Below there is a brief explanation of what the VNA bugs are about [35]:

Bug	Explanation
VNA00-J	Ensure visibility when accessing shared primitive variables.
VNA01-J	Ensure visibility of shared references to immutable objects.
VNA02-J	Ensure that compound operations on shared variables are atomic.
VNA03-J	Do not assume that a group of calls to independently atomic methods is atomic.
VNA04-J	Ensure that calls to chained methods are atomic.
VNA05-J	Ensure atomicity when reading and writing 64-bit values.
VNA06-J	Do not assume that declaring an object reference volatile guarantees.

Table 4.2: Explanation of VNA Bugs

The next category of Concurrency Guidelines is Locking. The results are in the following table:

	Priority of Bug	ThreadSafe	FindBugs
LCK00-J	P4	✗	✓
LCK01-J	P8	✗	✓
LCK02-J	P8	✗	✓
LCK03-J	P8	✗	✓
LCK04-J	P4	✓	✗
LCK05-J	P4	✗	✗
LCK06-J	P8	✗	✗
LCK07-J	P3	✗	✗
LCK08-J	P9	✓	✓
LCK09-J	P2	✓	✓
LCK10-J	P4	✓	✓
LCK11-J	P4	✗	✗
Total		**4/12**	**7/12**

Table 4.3: Locking Bugs

Discussion: In this part of the CERT Guidelines FindBugs has a much better accuracy ratio than in the previous part. On the one hand, ThreadSafe detects less than 40% and on the other hand, FindBugs detects 58.3%. As far as concerns priority, both tools detects bugs from all the levels. Table 4.4 contains an explanation of the bugs in the LCK category.

Bug	Explanation
LCK00-J	Use private final lock objects to synchronize classes
LCK01-J	Do not synchronize on objects that may be reused
LCK02-J	Do not synchronize on the class object returned by getClass()
LCK03-J	Do not synchronize on the intrinsic locks of high-level concurrency objects
LCK04-J	Do not synchronize on a collection view if the backing collection is accessible
LCK05-J	Synchronize access to static fields that can be modified by untrusted code
LCK06-J	Do not use an instance lock to protect shared static data
LCK07-J	Avoid deadlock by requesting and releasing locks in the same order
LCK08-J	Ensure actively held locks are released on exceptional conditions
LCK09-J	Do not perform operations that can block while holding a lock
LCK10-J	Do not use incorrect forms of the double-checked locking idiom
LCK11-J	Avoid client-side locking when classes that do not commit to their locking strategy

Table 4.4: Explanation of LCK Bugs

The next category is about Thread APIs.

	Priority of Bug	ThreadSafe	FindBugs
THI00-J	P4	✗	✓
THI01-J	P4	✗	✗
THI02-J	P2	✗	✗
THI03-J	P2	✗	✓
THI04-J	P4	✗	✗
THI05-J	P4	✓	✗
Total		**1/6**	**2/6**

Table 4.5: Thread APIs Bugs

Discussion: The accuracy for both of the tools is less than 50%. However, it is significant to mention that FindBugs detects a bug with a high degree of priority (THI03-J) that ThreadSafe is not capable of finding. As table 4.6 shows, the specific bug is about invoking the wait() method within a loop.

Bug	Explanation
THI00-J	Do not invoke Thread.run()
THI01-J	Do not invoke ThreadGroup methods
THI02-J	Notify all waiting threads rather than a single thread
THI03-J	Always invoke wait() and await() methods inside a loop
THI04-J	Ensure that threads performing blocking operations can be terminated
THI05-J	Do not use Thread.stop() to terminate threads

Table 4.6: Explanation of THI Bugs

The final part of Java Concurrency Guidelines contains several tests about Thread-Safety.

	Priority of Bug	ThreadSafe	FindBugs
TSM00-J	P4	✗	✗
TSM01-J	P4	✓	✗
TSM02-J	P2	✓	✗
TSM03-J	P8	✗	✗
Total		**2/4**	**0/4**

Table 4.7: Thread-Safety Miscellaneous Bugs

Discussion:As the table above shows the ThreadSafe succeed 50% and FindBugs 0%. It is also important that it detects the bug with the highest priority (TSM02-J) that is about class initialization.

Bug	Explanation
TSM00-J	Do not override thread-safe methods with methods that are not thread-safe
TSM01-J	Do not let the this reference escape during object construction
TSM02-J	Do not use background threads during class initialization
TSM03-J	Do not publish partially initialized objects

Table 4.8: Explanation of TSM Bugs

Below in the graph we can see the summary for of the Java Concurrency Guidelines.

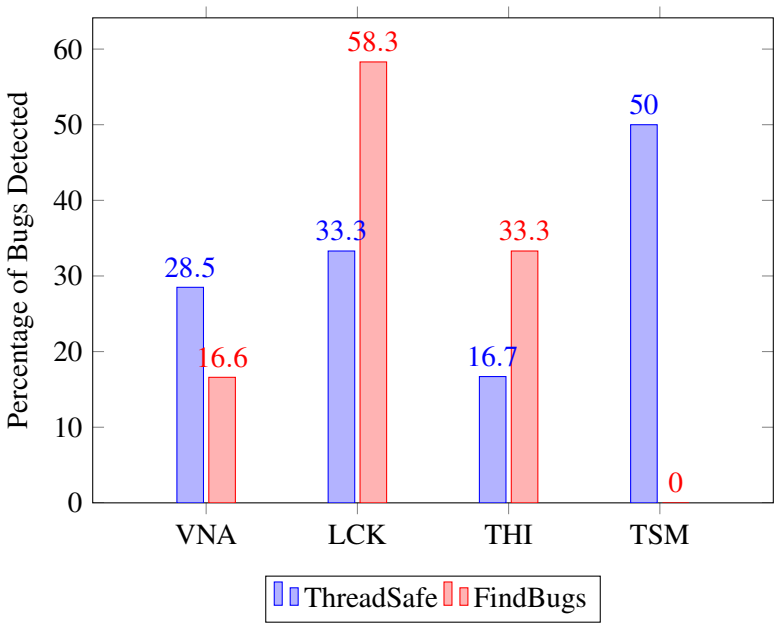

Figure 4.6: Java Concurrency Guidelines Graph

Discussion: The two tools have a similar degree of accuracy in total. However, it is important to mention that there is not much overlap in the results and each one of them identifies different types of bugs.

4.2.1.2 False Positive/Negatives

The decrease of the number of false positives and false negatives is one of the biggest concerns of each tool's developers. Two metrics that can present how good a static analysis tool is, are precision and recall. Precision refers to the proportion of retrieved set that are in fact bugs and recall refers to the fraction of all bugs that were found. In other words:

$P = \frac{TP}{TP+FP}$ and

$R = \frac{TP}{TP+FN}$

In order to calculate the number of false positives, we used the compliant program of each type of bug. The result is that none of the tools had any false positive in the CERT concurrency guidelines. As a result P = 1 for both of the tools.

As far as concerns recall, both of the tools had false negatives to show, which are the bugs that were not detected by the tools. On the one hand ThreadSafe had 20 false negatives and 9 True Positives and FindBugs 19 false negatives and 10 true positives. So we could say that:

	Precision	Recall
ThreadSafe	100 %	9/29 or 31%
FindBugs	100 %	10/29 or 34%

Table 4.9: Guards Bugs

The percentages are extremely close so we cannot come to a safe conclusion. However, we can notice that FindBugs has a higher degree of recall.

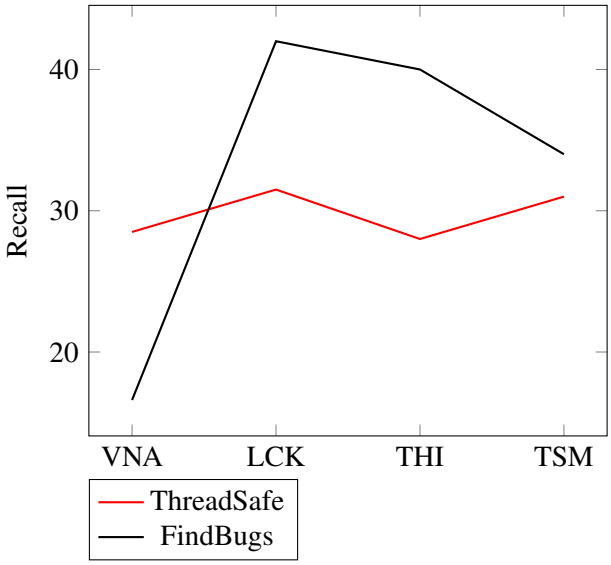

Table 4.10: Recall Graph

Above we can see a graph that represents the recall for ThreadSafe and FindBugs. At the beginning, ThreadSafe succeeds a higher degree of Recall, but as tests are scanned, FindBugs's recall increases. Finally, the two tools succeed degrees of recall with a small difference.

4.2.2 1b - Eytani Benchmark

In this part all tools are used (ThreadSafe, FindBugs and Chord).

Program Name	**Type of Bug** [23]	**ThreadSafe**	**FindBugs**	**Chord**
ABpushpop	Stack Overflow	✗	✗	✗
Account	Wrong lock/no lock	✗	✓	✓
AirlinesTickets	Not-atomic	✗	✓	✗
AllocationVector	Weak-reality	✗	✗	✓
BoundedBuffer	Notify instead of NotifyAll	✓	✓	✓
BubbleSort	Orphaned-Thread	✓	✓	✓
BubbleSort2	Initialization-sleep pattern	✗	✓	✓
BufWriter	Wrong Lock/No lock	✓	✗	✓
BuggyProgram	Non-atomic, wrong lock	✓	✗	✗
Critical	Non-atomic	✗	✗	✗

DCL	Double Checked Locking	✓	✓	✓
Deadlock	Deadlock	✗	✓	✓
DeadlockException	Deadlock	✗	✗	✓
FileWriter	Non-atomic	✗	✓	✗
FtpServer	Liveness	✓	✓	✓
GarageManager	Deadlock	✓	✗	✓
HierarchyExample	Race Condition	✓	✗	✓
JCuteExamples	Race Condition	✓	✓	✓
LinkedList	Non-atomic	✗	✗	✓
Liveness	Liveness	✗	✗	✓
Lottery	Wrong lock/no lock	✓	✓	✗
Manager	Non-atomic	✓	✗	✓
MergeSort	Non-atomic	✗	✓	✓
MergeSortBug	Non-atomic	✗	✗	✗
PingPong	Non-atomic	✗	✗	✗
Piper	Wait not in loop	✗	✓	✓
ProductConsumer	Orphaned-Thread	✗	✓	✓
Shop	Sleep,weak reality	✓	✗	✓
SunsAccount	Non-atomic	✗	✗	✓
XtangoAnimator	Deadlock	✓	✗	✗(NoMem)
Total		**13/30**	**15/30**	**21/30**

Table 4.12: IBM benchmark Bugs

Discussion: First of all it is important to mention that Chord was not able to detect the bugs for ”XtangoAnimator”, because it could not finish because of ”Out of Memory” problem of the system. This shows that Chord can be used only for programs with not many lines of code.It is worth mentioning that the specific program contains 2088 lines of code.

It is obvious that Chord was the tool that found most of the bugs. Although, as we can see ThreadSafe is good on finding bugs that belong to the Wrong Lock/No Lock category and FindBugs is good non-atomic errors. However, as a conclusion we could say that Chord is really good and the other two are almost similar with accuracy around 43 to 50%.

4.2.3 Part 2 - Web

Part 2 is about testing simple programs with the two most common types of bugs (deadlock and race condition). For this part all the tools are used (ThreadSafe, FindBugs and Chord). In the tables below we can see the results for both types.

	ThreadSafe	FindBugs	Chord
Deadlock1	✓	✗	✓
Deadlock2	✓	✓	✗
Deadlock3	✗	✓	✓
Deadlock4	✗	✓	✗
Deadlock5	✓	✓	✓
Deadlock6	✓	✗	✗
Total	**4/6**	**4/6**	**3/6**

Table 4.13: Deadlocks Bugs

	ThreadSafe	FindBugs	Chord
RaceCondition1	✗	✗	✓
RaceCondition2	✗	✗	✓
RaceCondition3	✓	✗	✓
RaceCondition4	✓	✗	✓
RaceCondition5	✗	✓	✗
RaceCondition6	✗	✗	✓
RaceCondition7	✓	✗	✓
Total	**4/7**	**1/7**	**6/7**

Table 4.14: Race Condition Bugs

Discussion: On the 6 programs with deadlocks that are used, ThreadSafe and FindBugs detect 66.7% of the bugs and Chord detects one less so 50% of them. However, there is not any program that all the tools have the same results (all or none of them detect the same bug). This shows that even though they are all static analysis tools the differences between them and especially the way they analyse code are crucial.

ThreadSafe detects 57% of the race conditions and FindBugs only 14%. Although, Chord is very good at this part of testing and succeeded in finding 85% of them.

From the results we can say that Chord had a very good degree of accuracy (9/13). Furthermore, ThreadSafe's accuracy was also high (8/13). However, FindBugs's accuracy was poor.

Below there is a graph to represent the results.

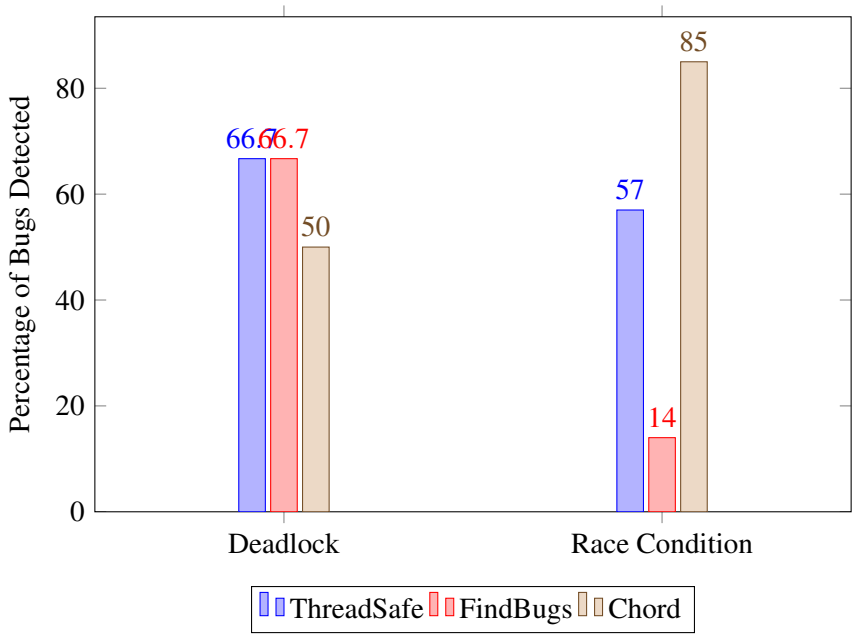

Figure 4.7: Deadlock and Race condition graph

4.2.4 Part 3 - Open-Source Projects

The final part is about testing on very large open source projects. In this part ThreadSafe and FindBugs are used. Chord is not used because after several attempts to analyse the projects there were ”run out of memory” problems.

It would be pointless to just check if the tools find bugs or not, because the projects have so many lines of code that every one of them will detect bugs. Instead, we decided to count how many bugs each tool detects and identify which one finds the most important ones.

	ThreadSafe	FindBugs
K9Mail	28	23
JOscar	58	16
ArgoUml	31	14

Table 4.15: Open Source Projects Bugs

Discussion: As far as concerns K9Mail, ThreadSafe found 28 bugs and FindBugs found 23 with most of them being inconsistent synchronizations of accesses to a field. The results were quite similar and they both needed 15 to 20 seconds to analyse the whole project.

However, in JOscar the results had important differences. First of all as we can see ThreadSafe found more bugs than FindBugs. As a matter of fact, ThreadSafe identified 49 inconsistent

synchronizations in addition to FindBugs, which found less than 5. We can say that ThreadSafe works better on large projects and is able to detect bugs that other tools cannot.

Finally, in ArgoUml ThreadSafe detects 31 bugs and FindBugs 14. Both of them detect same categories of bugs (for example inconsistent synchronizations, unsafe operations and non-atomic uses), but ThreadSafe finds more bugs in each category than FindBugs.

4.3 Performance

4.3.1 Time

In order to check the performance of the tools according to the time they need we decided to use the benchmark. Some assumptions that were made are:

- In order to make a fair comparison the command line release was used for all the tools.
- The times that are used are the average score between 3 execution for each program.
- The lines of code for each program were counted by a tool called EclipseMetrics [4].
- An observation that was made was that FindBugs need more time in the first executions and that after a few it needs less time. With respect to fairness, the times that were used were after 5 executions.

In order to prove the last point the times of 10 executions of FindBugs are provided for ”ABpushpop” program. As we can see, there is a convergence after the fifth execution of the program.

Executions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time in Seconds	5.70	1.82	1.10	0.84	0.77	0.69	0.65	0.56	0.58	0.69

Table 4.17: FindBugs Execution Times

As far as concerns ThreadSafe, it needs two executions in order to reach convergence. Below we can see the times of ThreadSafe for the same program:

Executions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time in Seconds	4.73	1.45	1.42	1.4	1.47	1.46	1.38	1.4	1.44	1.47

Table 4.19: ThreadSafe Execution Times

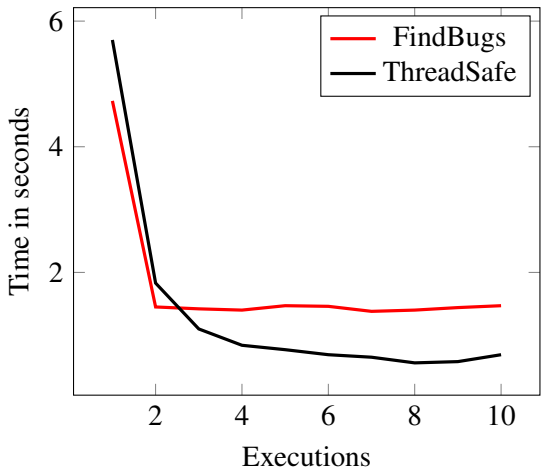

Table 4.20: FindBugs and ThreadSafe Execution Times Graph

For Chord the first time refers to datarace-analyses and the second to deadlock-analyses. LOC means Lines Of Code, TS means ThreadSade and FB means FindBugs. The following table shows the information that will be needed:

Program Name	**LOC**	**TS**	**FB**	**Chord**
ABpushpop	317	1.44s	0.71s	57s / 53s
Account	155	1.15s	0.29s	1m 31s / 1m 34s
AirlinesTickets	61	1.03s	0.56s	1m 37s / 1m 33s
AllocationVector	286	1.23s	0.26s	1m 6s / 55s
BoundedBuffer	536	1.48s	0.48s	1m 7s / 59s
BubbleSort	236	1.34s	0.29s	1m 34s / 1m 29s
BubbleSort2	130	1.10s	0.20s	1m 14s / 1m 19s
BufWriter	255	1.27s	0.25s	1m 36s /1m 30s
BuggyProgram	359	1.35s	0.46s	1m 33s / 1m 31s
Critical	73	1.12s	0.18s	1m 4s / 59s
DCL	138	1.17s	0.24s	1m 35s / 1m 32s
Deadlock	95	1.2s	0.34s	1m 41s / 4m 44s
DeadlockException	149	1.34s	0.43s	1m 40s / 1m 38s
FileWriter	311	1.2s	0.28s	1m 42s / 1m 40s
FtpServer	1200	4.2s	4.08s	1m 44s / 5m 32s
GarageManager	584	1.44s	0.46s	1m 43s / 5m 48s
HierarchyExample	88	1.19s	0.46s	59s / 51s
JCuteExamples	280	2.7s	3.02s	1m 21s / 1m 22s
LinkedList	416	1.36s	0.24s	1m 31s /1m 30s
Liveness	161	1.18s	0.26s	1m 6s / 54s

Lottery	359	1.72s	1.02s	1m27s / 1m 26s
Manager	236	1.34s	0.26s	1m 6s / 57s
MergeSort	375	1.36s	0.8s	1m 30s / 4m 45s
MergeSortBug	275	1.3s	0.40s	1m 31s / 1m30s
PingPong	126	1.25s	0.26s	52s / 59s
Piper	116	1.35s	0.26s	1m 28s / 4m 35s
ProductConsumer	203	1.21s	0.4s	1m 28s / 4m 51s
Shop	273	1.48s	0.26s	1m 5s / 56s
SunsAccount	66	1.10s	0.24s	1m 8s / 59s
XtangoAnimator	2088	2.5s	1.37s	Out of Memory

Table 4.22: Time Performance Table

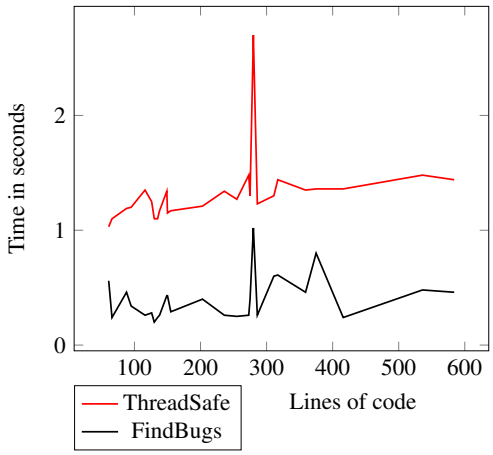

Table 4.23: Graph for ThreadSafe and FindBugs for 60 to 600 lines of code

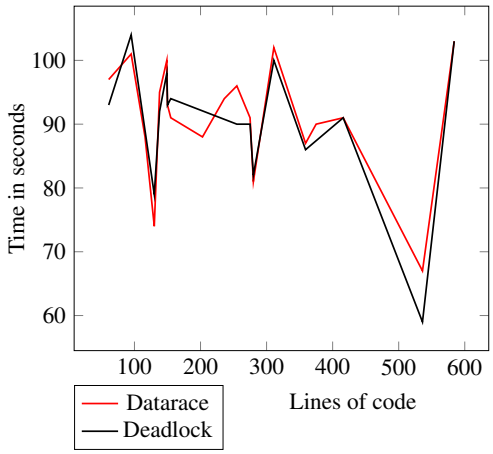

Table 4.24: Graph for ThreadSafe and FindBugs for 60 to 600 lines of code for Chord

Discussion: From the table and the graphs above we can make some points for the time performance of the tools. First of all, it is obvious that ThreadSafe and FindBugs can analyse programs with many lines of code in a few seconds. FindBugs is a little faster, but we should mention that it needs to run a few times before it has the best possible time performance. Furthermore, there are occasions, where a program with less lines than another needs more time (e.g. ”Deadlock” with 95 lines of code needs 0.34 seconds and ”FileWriter” with 311 lines needs 0.28 seconds). This proves that the besides the lines of code, complexity is a very important parameter in the scanning of a program.

In addition to these two, Chord has a worst time performance and needs more than a minute even for programs that have less than 150 lines of code. The two analyses that were used in Chord (deadlock and datarace analysis) generally need same amount of time for the same program. However, there are many ups and downs, which proves that complexity is also very important for each analysis. Furthermore, there are occasions, where deadlock-analyses needs much more time than datarace. For example, in ”Piper” the time for datarace is 1 minute and 28 seconds and deadlock 4 minutes and 31 seconds. Outliers like this are not included in the graphs because they are special cases and one of them is examined below.

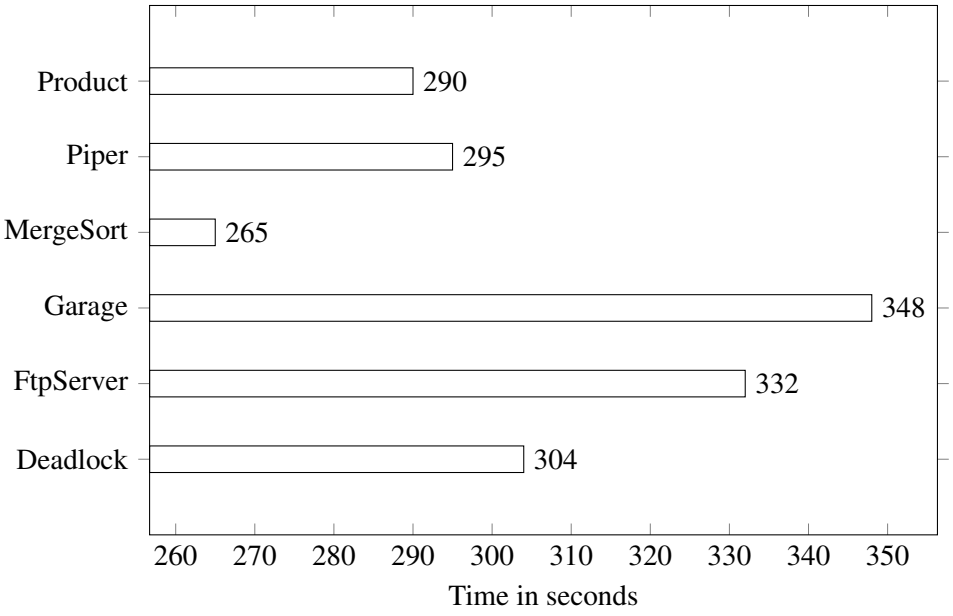

Table 4.25: Chord Deadlock-analysis Outliers

In ”Piper” the analysis that identifies the bug is deadlock-analyses. As we can see above it needs more than 4 minutes. If we examine the output of Chord we can see that the traces and the details of the bugs are contained in an xml file that is more than 5 MB. As a result, we can say that the number of the lines of code is a very important parameter, but the complexity and the time needed to produce the output are significant as well.

The graphs above show us how the three tools work on not very large projects that do not

exceed 1200 lines of code. As far as concerns the open source projects the results are below:

Project Name	LOC	ThreadSafe	FindBugs
JOscar	17.000	5.6s	7.09s
K9Mail	>30.000	12.62s	19.62s
ArgoUml	>100.000	205s	255s

Table 4.26: Open Source Projects Time Performance

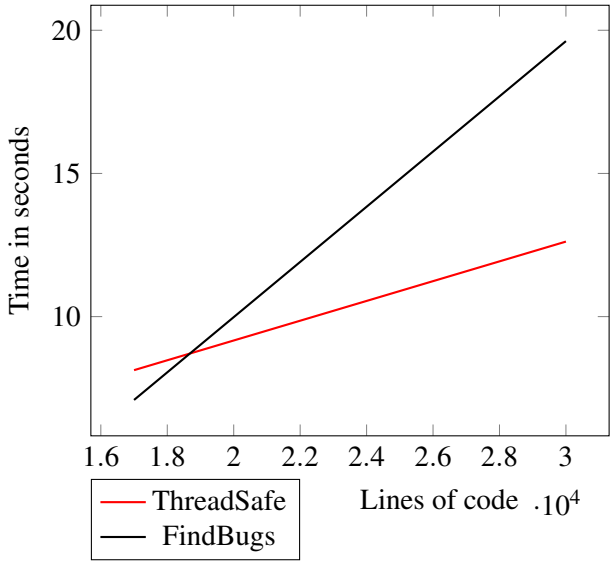

Table 4.27: Graph for ThreadSafe and FindBugs for JOscar and K9Mail

As the number of lines is low (less than 600 lines), ThreadSafe and FindBugs need less than two seconds to finish scanning. Even though FindBugs needs less time than ThreadSafe, as programs become larger FindBugs needs more time than FindBugs. In other words, as the number of lines of code increases FindBugs has a more rapid increase in time needed than ThreadSafe. For both benchmark and large open source projects, FindBugs has a more wide range of time (from 0.28 to 20 seconds) than ThreadSafe (1 to 12.62 seconds).

Furthermore, in even larger projects (like ArgoUml) FindBugs is still slower than ThreadSafe.

4.3.2 Bug representation

In order to evaluate the tools for the way they represent the bugs we will examine the output for the same test. The program that we will use is ”DCL” from the Eytani benchmark. The type of the bug in the specific piece of code is double-checked locking.

ThreadSafe (CLI)

First of all we will examine the output of ThreadSafe (CLI). Below there are two figures with the results.

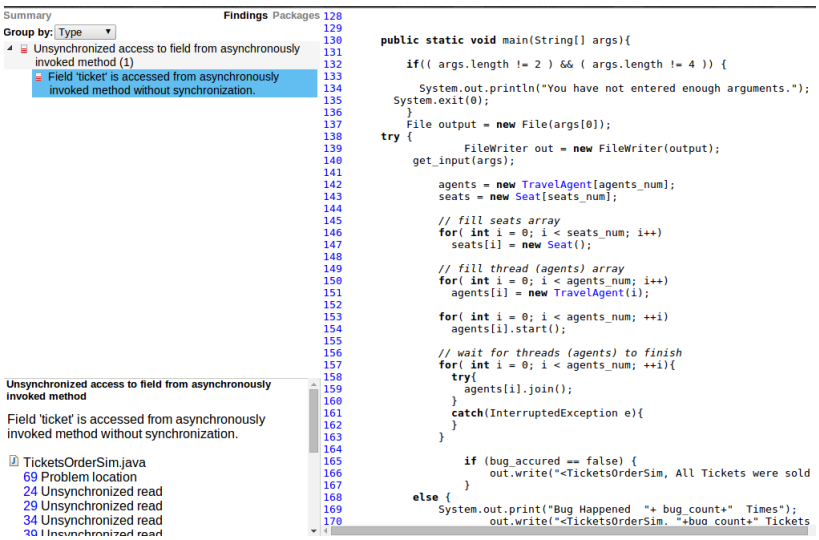

Figure 4.8: ThreadSafe Command Line Results Representation 1

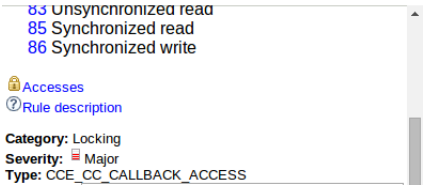

Figure 4.9: ThreadSafe Command Line Results Representation 2

Discussion:As we can see, the output is clear about the bugs with enough information that helps the user find and fix the bugs. There are 3 main windows in the html file. Firstly, there is a upper left window where there is a list with the program's bugs that can be presented according to their type, severity, class or category. Secondly, in the down left window information about the bug is provided (type, category and severity). Furthermore, unsynchronized read and writes are presented with the lines of code they occur. Finally, in the right window the code of the program is given in order to help the user identify the bug.

Furthermore, for bugs that come from inconsistent synchronization ThreadSafe provides additional information in a special section called "Accesses". Below there is n example for a race condition bug:

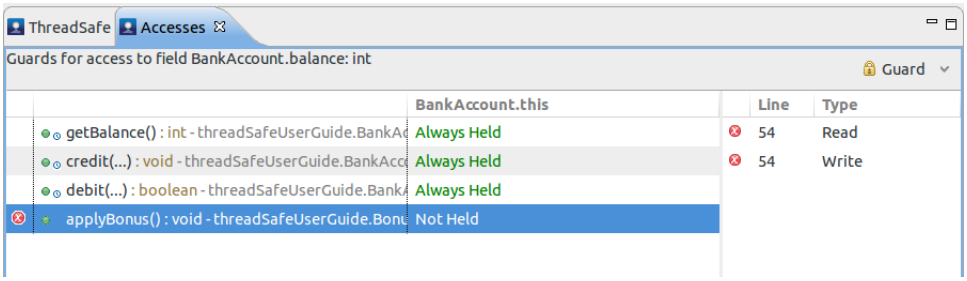

Figure 4.10: ThreadSafe Accesses

As we can see, for each method that there is no consistent synchronization there is information provided for where exactly in the program a read or a write happens.

Chord (CLI)

Below there is a figure for the output of the tool for a datarace analyses.

Datarace Reports (Grouped By Field)

Details	Trace 1	
	Thread	Memory Access
1. Dataraces on TicketsOrderSim.bug_accured		
1.1	TicketsOrderSim$TravelAgent.run()	TicketsOrderSim.check_ticket_details(int) (Wr)
1.2	TicketsOrderSim$TravelAgent.run()	TicketsOrderSim.check_ticket_details(int) (Wr)
1.3	TicketsOrderSim$TravelAgent.run()	TicketsOrderSim.check_ticket_details(int) (Wr)
1.4	TicketsOrderSim$TravelAgent.run()	TicketsOrderSim.check_ticket_details(int) (Wr)
2. Dataraces on TicketsOrderSim.bug_count		
2.1	TicketsOrderSim$TravelAgent.run()	TicketsOrderSim.check_ticket_details(int) (Wr)
2.2	TicketsOrderSim$TravelAgent.run()	TicketsOrderSim.check_ticket_details(int) (Wr)

Trace 2	
Thread	Memory Access
TicketsOrderSim.main(java.lang.String[])	TicketsOrderSim.main(java.lang.String[]) (Rd)
TicketsOrderSim.main(java.lang.String[])	TicketsOrderSim.main(java.lang.String[]) (Rd)
TicketsOrderSim.main(java.lang.String[])	TicketsOrderSim.main(java.lang.String[]) (Rd)
TicketsOrderSim.main(java.lang.String[])	TicketsOrderSim.main(java.lang.String[]) (Rd)
TicketsOrderSim.main(java.lang.String[])	TicketsOrderSim.main(java.lang.String[]) (Rd)
TicketsOrderSim.main(java.lang.String[])	TicketsOrderSim.main(java.lang.String[]) (Rd)

Figure 4.11: Chord Results Representation

Discussion: As we can see the output of Chord is not exactly what the other tools provide. After a datarace-analysis the report presents the traces of the bug. This means that the user can see where exactly an unsynchronised ”read” or ”write” happens where the error arises. We can clearly see that Chord is developed for experienced developers that can examine the details, track the defect and fix the problem. A suggestion for improvement is that in the output folder there could be a file that lists the bugs in a way that inexperienced developers can understand and fix.

FindBugs (GUI - Eclipse plug-in)

The two figures below present FindBugs's output for the graphical user interface and the plug-in.

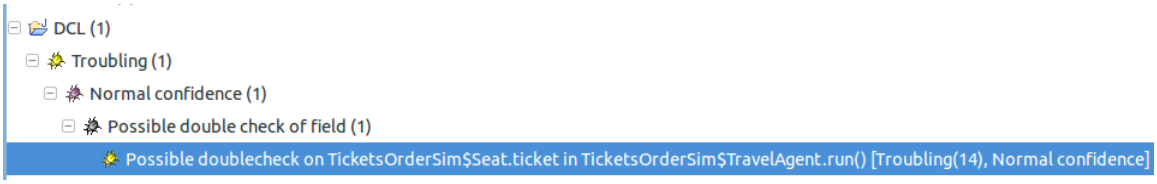

Figure 4.12: FindBugs Plug-in Results Representation

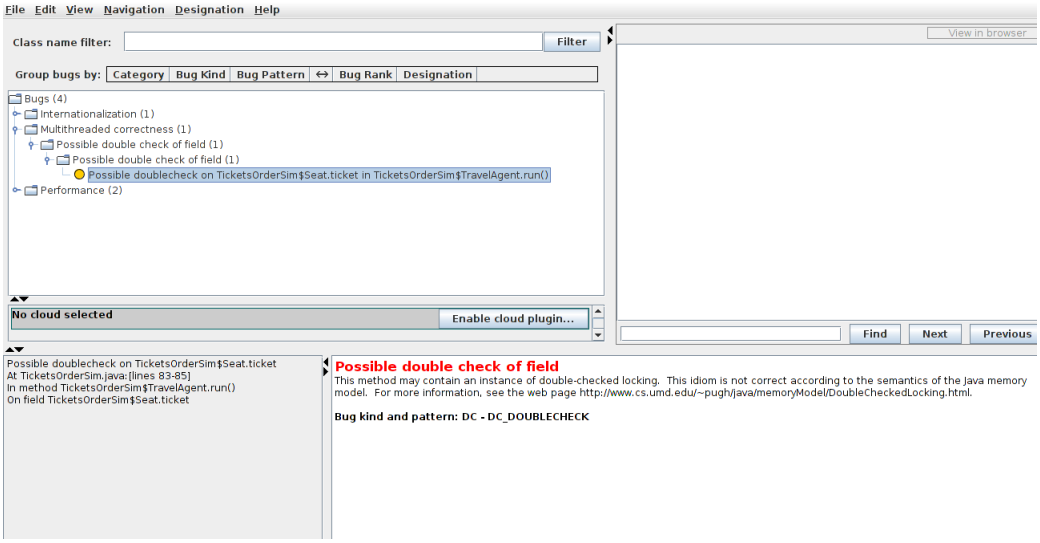

Figure 4.13: FindBugs GUI Results Representation

Discussion: The GUI has a lot in common with the html output of ThreadSafe. The user is able to see the bugs that are identified and details for each one of them. However, a disadvantage is that, even if the line of code that the bug occurs is given, the tool does not have a feature to show code in order to help the developer find where the problem is exactly.

Summary

In order to summarize, even if the Eclipse plug-ins (ThreadSafe and FindBugs) are the easiest to use, ThreadSafe CLI is the tool that provides the most clear and precise information needed for the user. As far as concerns FindBugs, the GUI and the plug-in provide the same information, but not as clear as ThreadSafe. The interface can be improved by implementing features like ways to fix a bug.

The hardest tool to use is Chord, because when the developer needs to track and fix a bug he must follow some specific steps. First, the user must choose the appropriate analysis for

the program and the bugs that may exist. Secondly, it is important to examine the traces and locate where the bugs is exactly and what are the required fixes and, finally, the repair must take place taking into consideration all the previous steps. An improvement that can be done is an implementation that produces an html or xml page with more clear results.

4.4 Obfuscation

Obfuscation is the act of turning source code, using several transformations, to a form that a human cannot easily understand [20]. The main reason that obfuscation is used is to provide security in order to prevent attacks like tampering [24].

Obfuscation includes [9]:

Name Obfuscation Renaming methods, classes etc.
Code and Data Flow Obfuscation Alternative logic
Other String Encryption, Unnecessary Code Injection, Remove information from debugging

Obfuscation is used in order to show how easily a tool can be thwarted and not detect a bug that was detected in the original program. The idea is to use a number of patterns in a subset of the programs and identify when exactly each tool does not detect mistakes in the program (false negatives) [42].

From the test suite we used the Java Concurrency Guidelines, Common Bugs and the following 10 programs from the benchmark:

Program Name	Type of Bugs	Tool that detected the bugs
BoundedBuffer	Notify instead of NotifyAll	ThreadSafe, FindBugs, Chord
BubbleSort	Orphaned-Thread	ThreadSafe, FindBugs, Chord
BuggyProgram	Wrong Lock	ThreadSafe
DCL	Double-Checked Locking	ThreadSafe, FindBugs, Chord
FileWriter	Non-atomic	FindBugs
FtpServer	Liveness	ThreadSafe, FindBugs, Chord
JCuteExamples	Race Condition	ThreadSafe, FindBugs, Chord
Manager	Non-atomic	ThreadSafe, Chord
Piper	Wait not in loop	FindBugs, Chord
ProductConsumer	Orphaned-Thread	Chord

Table 4.29: Benchmark Obfuscated Programs

In order to choose the programs from the benchmark, we decided to include the programs that contain bugs that were detected by every tool. However, in order to have a more complete subset of the benchmark, we included five more that help us have a variety of types of bugs.

Pattern 1: Call the buggy part of the program through another method

For this pattern the idea is to create another method and call the buggy part from it. Pattern 1 increases the degree of the complexity of the tool's analysis by adding one more method to call. Below there is an example of calling the buggy part through another method:

```
3  package locking;
4  class oneObject {
5
6      public synchronized void changeValue() {
7          // ...
8      }
9  }
10 public class SomeObject{
11     // Untrusted code
12     public void something(){
13         oneObject someObject = new oneObject();
14         synchronized (someObject) {
15             while (true) {
16                 //  Indefinitely delay someObject
17                 try {
18                     Thread.sleep(Integer.MAX_VALUE);
19                 } catch (InterruptedException e) {
20                     //  TODO Auto-generated catch block
21                     e.printStackTrace();
22                 }
23             }
24         }
25     }
26 }
```

```
3  package locking;
4  class oneObject {
5
6      public synchronized void changeValue() {
7          // ...
8      }
9  }
10 public class SomeObject{
11     private void anotherMethod(){
12         try {
13             Thread.sleep(Integer.MAX_VALUE);
14         } catch (InterruptedException e) {
15             //  TODO Auto-generated catch block
16             e.printStackTrace();
17         }
18     }
19     public void something(){
20         oneObject someObject = new oneObject();
21         synchronized (someObject) {
22             while (true) {
23                 anotherMethod();
24             }
25         }
26     }
```

Figure 4.14: Obfuscation Pattern 1 Example

Java Concurrency Guidelines-Visibility and Atomicity

ThreadSafe and FindBugs were tested in the VNA part of the CERT Guidelines [35]. As we can see in section 4.2 only ThreadSafe was able to detect 2 of the bugs. After the obfuscation ThreadSafe was still able to detect them. FindBugs detected only one bug, but after applying pattern 1 it was not able to detect it.

Java Concurrency Guidelines-Locking

In the LCK part ThreadSafe was again able to detect the bugs that it could detect before. However, FindBugs did not detect the three following bugs:

Bug	Summary
LCK00-J	Use private final lock objects when interacting with untrusted code
LCK08-J	Ensure actively held locks are released on exceptional conditions
LCK09-J	Do not perform operations that can block while holding a lock

Table 4.30: Finbugs Results on LCK after Obfuscation with Pattern 1

Java Concurrency Guidelines-Thread-APIs

In this part of the Guidelines after the obfuscation both of the tools were still able to detect the bug (one for ThreadSafe and two for FindBugs).

Java Concurrency Guidelines-Thread-Safety Miscellaneous

There was only two bugs (TSM02-J and TSM02-J) detected (out of four) for ThreadSafe. After calling the buggy part through another method it was still able to detect the bugs.

Eytani Benchmark

In order to check how the tools work in obfuscated programs of the benchmark, as mentioned before we used some of the programs.

	Was it Thwarted ?		
Program Name	ThreadSafe	FindBugs	Chord
BoundedBuffer	✗	✗	✗
BubbleSort	✗	✓	✗
BuggyProgram	✗	———-	———-
DCL	✗	✗	✗
FileWriter	———-	✓	———-
FtpServer	✗	✗	✗
JCuteExamples	✗	✗	✗
Manager	✗	———-	✗
Piper	———-	✗	✗
ProductConsumer	———-	———-	✗

Table 4.31: Benchmark Pattern 1 Results

As a result, the specific pattern was not able to thwart ThreadSafe and Chord. However, FindBugs had again false-negatives (bugs not found).

Common Bugs

In common bugs the accuracy for deadlocks was 66% for both ThreadSafe and Finbugs and for Chord it was 50%. After the obfuscation ThreadSafe was still able to detect the bugs, but FindBugs detected only 25% of the 4 bugs that it detected before the obfuscation. Furthermore, Chord was still able to detect the bugs. This shows the analyses that Chord does is extremely powerful and it is difficult to thwart the tool.

As far as concerns the race conditions, even if ThreadSafe detected the bugs, FindBugs did not detect the one bug that it was able to detect before. Chord had a very good degree of accuracy in common race conditions (85%), which remained the same after obfuscating the programs.

Pattern 2 - Include the buggy part into if-else statement

For this pattern the idea is to insert the buggy part of the program into an if-else statement. The buggy part will be executed in both blocks (if and else). Below there is an example of the technique:

```
public void acquireLock(){
// Untrusted code
    synchronized (SomeObject.class) {
        try {Thread.sleep(Integer.MAX_VALUE);}
        catch (InterruptedException e) {e.printStackTrace();}
    }
}
private int flag=0;
public void acquireLock(){
// Untrusted code
    synchronized (SomeObject.class) {
        if(flag==0){
            try {Thread.sleep(Integer.MAX_VALUE);}
            catch (InterruptedException e) {e.printStackTrace();}
        }
        else{
            try {Thread.sleep(Integer.MAX_VALUE);}
            catch (InterruptedException e) {e.printStackTrace();}
        }
    }
}
```

Figure 4.15: Obfuscation Method 2 Example

Discussion of the example: As we can see from the example, the buggy part(try-catch) is included in an if-else statement. Even if the bug is the same, it exists two times, so the tool must identify both of them.

Java Concurrency Guidelines-Visibility and Atomicity

In the fist category of the guidelines (Visibility and Atomicity), ThreadSafe was able to detect 2/6 bugs. After the obfuscation with the specific method ThreadSafe was still able to detect the bug. Furthermore, FindBugs could also detect the bug it could detect from the beginning.

Java Concurrency Guidelines-Locking

For the specific type of bug ThreadSafe had 33.3 % and FindBugs 58.3 %. After the code obfuscation by using method 2, ThreadSafe had the same accuracy. However, FindBugs was thwarted in 4 bugs out of 7 it was able to detect before. The bugs that it was unable to detect are the following:

Bug	Summary
LCK00-J	Use private final lock objects when interacting with untrusted code
LCK01-J	Do not synchronize on objects that may be reused
LCK08-J	Ensure actively held locks are released on exceptional conditions
LCK09-J	Do not perform operations that can block while holding a lock

Table 4.32: FindBugs Results on LCK after Obfuscation with Pattern 2

Java Concurrency Guidelines-Thread-APIs

In addition to pattern 1, FindBugs was thwarted for both of the bugs that it was able to detect before.

Bug	Summary
THI00-J	Do not invoke Thread.run()
THI03-J	Always invoke wait() and await() methods inside a loop

Table 4.33: FindBugs Results on THI after Obfuscation with Pattern 2

Java Concurrency Guidelines-Thread-Safety Miscellaneous

In TSM there was two bugs detected by ThreadSafe which are TSM01-J (Do not let the this reference escape during object construction) and TSM02-J (Do not use background threads during class initialization). After including the buggy part in an if-else statement the tool was still able to detect it.

Eytani Benchmark

In order to check how the tools work in obfuscated programs of the benchmark, as mentioned before we used some of the programs.

	Was it Thwarted ?		
Program Name	ThreadSafe	FindBugs	Chord
BoundedBuffer	✓	✓	✗
BubbleSort	✓	✓	✗
BuggyProgram	✓	———-	———-
DCL	✗	✓	✗
FileWriter	———-	✗	———-
FtpServer	✗	✓	✗
JCuteExamples	✗	✓	✗
Manager	✗	———-	✗
Piper	———-	✓	✗
ProductConsumer	———-	———-	✗

Table 4.34: Benchmark Pattern Results

As a result, pattern 2 succeed to thwart ThreadSafe for ”BuggyProgram”. However, Chord was able to detect the bugs after the obfuscation and FindBugs could detect one of them.

Common Bugs

In deadlocks, another time ThreadSafe and Chord were not thwarted. However, FindBugs couldn't detect any of the bugs that it was able to detect before obfuscation.

As far as concerns race conditions, FindBugs could not detect the bug in both blocks of code (if and else blocks) in the one program it could before. ThreadSafe and Chord could not be thwarted.

4.5 Summary

In order to summarize the 4.4 section we could make some very helpful observations. First of all, as we can see FindBugs was thwarted in many occasions which shows as that it cannot be trusted for complicated programs, where bugs might be hidden within dead code or statements and method calls. ThreadSafe had a good degree of accuracy as it was only thwarted in one program of the benchmark. Finally, with the specific test suite and the specific patterns it was impossible to thwart Chord as it could detect all the bugs in the programs that were obfuscated. The results are presented in the below table. The number correspond to the obfuscated programs that managed to thwart the tools.

	Pattern 1			Pattern 2		
	ThreadSafe	FindBugs	Chord	ThreadSafe	FindBugs	Chord
Guidelines	0/9	4/10	———	0/9	6/10	———
Common Bugs	0/8	4/5	0/9	0/8	5/5	0/9
Benchmark	0/7	2/7	0/8	3/7	6/7	0/8
Summary	**0/24**	**9/22**	**0/17**	**3/24**	**17/22**	**0/17**

Table 4.35: Obfuscation Results

Chapter 5

Conclusion and future work

5.1 Conclusion

In this project, we described the process of finding bugs in concurrent Java by static analysis tools. We used a commercial (ThreadSafe) and two free open-source tools (FindBugs and Chord). In order to evaluate and compare them, we used a test suite that contains more than 70 programs coming from the CERT Concurrency Guidelines; a benchmark developed mainly from the IBM Research Lab with the support of other open source web communities. Except for Chord, the other two tools provide several functionalities (GUI, Command Line and Plug-ins). Even though we worked with each of them, we focused on the GUI and Command Line versions for most of the tests. The used test suite is published for future experiments on the homepage of Professor Don Sannella: *http://homepages.inf.ed.ac.uk/dts/other-students.html*

As far as concerns static analysis tools, generally they are at a good level in finding concurrency bugs. However, even if in small programs they are able to detect most of the bugs, in large projects they have false negatives and sometimes the bugs are lost in false positives.

Moreover, we focused on evaluating the mentioned tools in four categories (Usability, Accuracy, Performance and Obfuscation). As far as concern usability, FindBugs and ThreadSafe have quite a lot of similarities. We consider FindBugs to be slightly more useful because of the GUI, but ThreadSafe is better in the way the bugs are presented to the user. Finally, we believe that Chord is the least useful; because of the many configurations the user must set for every scan.

With respect to accuracy, for tests on which Chord could be used, the degree of its accuracy is dominant over the others. Thus, it is significant, that it cannot be tested on every program, because not all of them are complete (with main method) and because Chord does not scale to use with large programs. The other two have an almost identical accuracy, but it is significant to mention that there is not much overlap in the results and the two tools (ThreadSafe and

FindBugs) are able to detect different bugs.

As far as concerns performance, there are some important observations that were made. Firstly, it was not possible to include Chord in large-project testing, because it is not able to run programs with thousands lines of code. In small programs (60 to 600 lines of code) ThreadSafe and FindBugs needed only a few seconds to scan them, on contrast to Chord that needed from 50 seconds to 5 minutes. Even though FindBugs needed a smaller amount of time than ThreadSafe, in large-scale projects (more than 15000 lines of code) ThreadSafe needed less time than FindBugs.

The final way to evaluate the tools was obfuscation. Two simple methods were used in order to change the control or data flow of a program and attempt to thwart the tools. FindBugs was extremely easy to thwart and there were many bugs that it was not able to detect. ThreadSafe was a little stronger, as it was more difficult to hide bugs from it. Finally, it was not possible with the specific methods used to thwart Chord which makes it the most powerful in this respect.

Summarising, we could say that Chord is the best tool to use for complete small programs, even if it needs more time than the other two and it is not extremely usable. The other two tools have an almost similar behaviour with ThreadSafe being more precise and powerful as the project becomes larger.

5.2 Future Work

There were several ideas that came to mind while the project was in progress. Something important is the implementation of a benchmark. The IBM benchmark is useful but it should be extended to be more applicable for scalability and size. On the one hand, scalability can be tested by including programs with different sizes, which are measured with lines of code. On the other hand size can be tested by including large-scale projects with more than 30.000 lines of code.

From the three tools we used, we believe that Chord was the one that needed more time to get familiar with and to understand how it exactly works. A project could be entirely focused on Chord and it techniques of analyses. Furthermore, the differences between the two types of analysis can be pointed and an improvement in order to decrease the time for deadlock-analysis can be suggested.

Another aspect of the project was obfuscation. Obfuscation is a very special act in programming that is used mostly for security reasons. A project about obfuscation can focus on more patterns than the two that were used in the project and on the combination of them in order to attempt to thwart static analysis tools.

Finally, the novelty of our work comes partly from the fact that the specific three tools have not been compared before. However, as we can see in the section ”2.4 Previous Evaluations of Static Analysis Tools” there are many studies about comparisons of static analysis tools. The exact same ideas of the specific project could be used for the evaluation of Dynamic Analysis tools, like Parasoft JTest, Purify and Avalanche.

Bibliography

[1] Chord @ georgia tech [online] 2013 , http://pag-www.gtisc.gatech.edu/chord/.

[2] Concepts of concurrency, lecture 1: Introduction, chris greenhalgh,[online] 2007, school of computer science, http://www.cs.nott.ac.uk/.

[3] Concurrent programming actors and coordination abstractions,[online] march 2007, http://www.cs.rpi.edu/academics/courses/spring07/dci.

[4] Eclipsemetrics @ state of flow [online] 2006, http://www.stateofflow.com /projects/16/eclipsemetrics.

[5] Efficiency matters @ jenkov.com [online], http://www.jenkov.com/.

[6] Findbugs - find bugs in java programs. findbugs. [online] 21 december 2011 http://findbugs.sourceforge.net/.

[7] How to do in java [online] 2014, http://howtodoinjava.com/.

[8] Java examples @ java2notice [online] 2014, http://www.java2novice.com/.

[9] Java obfuscation tutorial with zelix klassmaster @ netspi [online] july 2013 , https://www.netspi.com/blog/entryid/184/java-obfuscation-tutorial-with-zelix-klassmaster.

[10] Microsoft support @ microsoft. [online] june 2012 http://support.microsoft.com /kb/317723.

[11] Openstax cnx @ cnx. [online] http://www.cnx.org/.

[12] Oracle [online], http://www.oracle.com/.

[13] Oreilly examples [online], http://examples.oreilly.com/9781565923713/.

[14] Palizine information security intelligence @ plynt [online] http://palizine.plynt.com/issues/2005oct/hiding-control-flows/.

[15] Program analysis group @ georgia tech [online] http://pag.gatech.edu/home.

[16] Static code analysis @ owasp [online], https://www.owasp.org.

[17] Threadsafe @ contemplate [online] http://www.contemplateltd.com/threadsafe.

[18] Tutorials point [online] 2014, http://www.tutorialspoint.com/java/.

[19] What is dynamic code analysis @ stackoverflow [online] september 2008, http://stackoverflow.com/questions/49937/what-is-dynamic-code-analysis.

[20] B. Anckaert, M. Madou, B. D. Sutter, B. D. Bus, K. D. Bosschere, and B. Preneel. Program obfuscation: A quantitative approach, 2007.

[21] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 1–8, New York, NY, USA, 2007. ACM.

[22] T. Ball. The concept of dynamic analysis. In *Software EngineeringESEC/FSE99*, pages 216–234. Springer, 1999.

[23] J. S. Bradbury, I. Segall, E. Farchi, K. Jalbert, and D. Kelk. Using combinatorial benchmark construction to improve the assessment of concurrency bug detection tools. In *Proceedings of the 2012 Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, pages 25–35. ACM, 2012.

[24] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations, 1997.

[25] N. Davis. Secure software development life cycle processes: A technology scouting report. Technical report, DTIC Document, 2005.

[26] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur. Toward a framework and benchmark for testing tools for multi-threaded programs. *Concurrency and Computation: Practice & Experience*, 19(3):267–279, 2006.

[27] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur. Toward a framework and benchmark for testing tools for multi-threaded programs. *Concurrency and Computation: Practice & Experience*, 19(3):267–279, 2006.

[28] Y. Eytani, R. Tzoref, and S. Ur. Experience with a concurrency bugs benchmark. In *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, ICSTW '08, pages 379–384, Washington, DC, USA, 2008. IEEE Computer Society.

[29] K. Havelund, S. D. Stoller, and S. Ur. Benchmark and framework for encouraging research on multi-threaded testing tools. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 8–pp. IEEE, 2003.

[30] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, Dec. 2004.

[31] M. Ishrat, M. Saxena, and M. Alamgir. Comparison of static and dynamic analysis for runtime monitoring. *International Journal of Computer Science & Communication Networks*, 2(5), 2012.

[32] D. Kester, M. Mwebesa, and J. Bradbury. How good is static analysis at finding concurrency bugs? In *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on*, pages 115–124, Sept 2010.

[33] S. A. Limited, I. E. Commission, et al. *Analysis Techniques for System Reliability: Procedure for Failure Mode and Effects Analysis (FMEA).* Standards Australia, 2008.

[34] B. Long, P. Strooper, and L. Wildman. A method for verifying concurrent java components based on an analysis of concurrency failures: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):281–294, Mar. 2007.

[35] F. Long. *Java Concurrency Guidelines*. Technical report. Carnegie Mellon University, Software Engineering Institute, 2010.

[36] F. Long, C.-M. U. C. C. Center, D. Mohindra, and R. Seacord. *The CERT Oracle Secure Coding Standard for Java*. SEI series in software engineering. Addison-Wesley, 2011.

[37] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM.

[38] Z. D. Luo, L. Hillis, R. Das, and Y. Qi. Effective static analysis to find concurrency bugs in java. In *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on*, pages 135–144, Sept 2010.

[39] M. A. A. Mamun, A. Khanam, H. Grahn, and R. Feldt. Comparing four static analysis tools for java concurrency bugs. In *Proc. of the Third Swedish Workshop on Multi-Core Computing (MCC-10)*, pages 143–146, 2010.

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

[41] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, ISSRE '04, pages 245–256, Washington, DC, USA, 2004. IEEE Computer Society.

[42] S. Schrittwieser and S. Katzenbeisser. Code obfuscation against static and dynamic reverse engineering. In *Proceedings of the 13th International Conference on Information Hiding*, IH'11, pages 270–284, Berlin, Heidelberg, 2011. Springer-Verlag.

[43] A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Principles, 7TH ED*. Wiley student edition. Wiley India Pvt. Limited, 2006.

[44] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb. An evaluation of two bug pattern tools for java. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 248–257, April 2008.