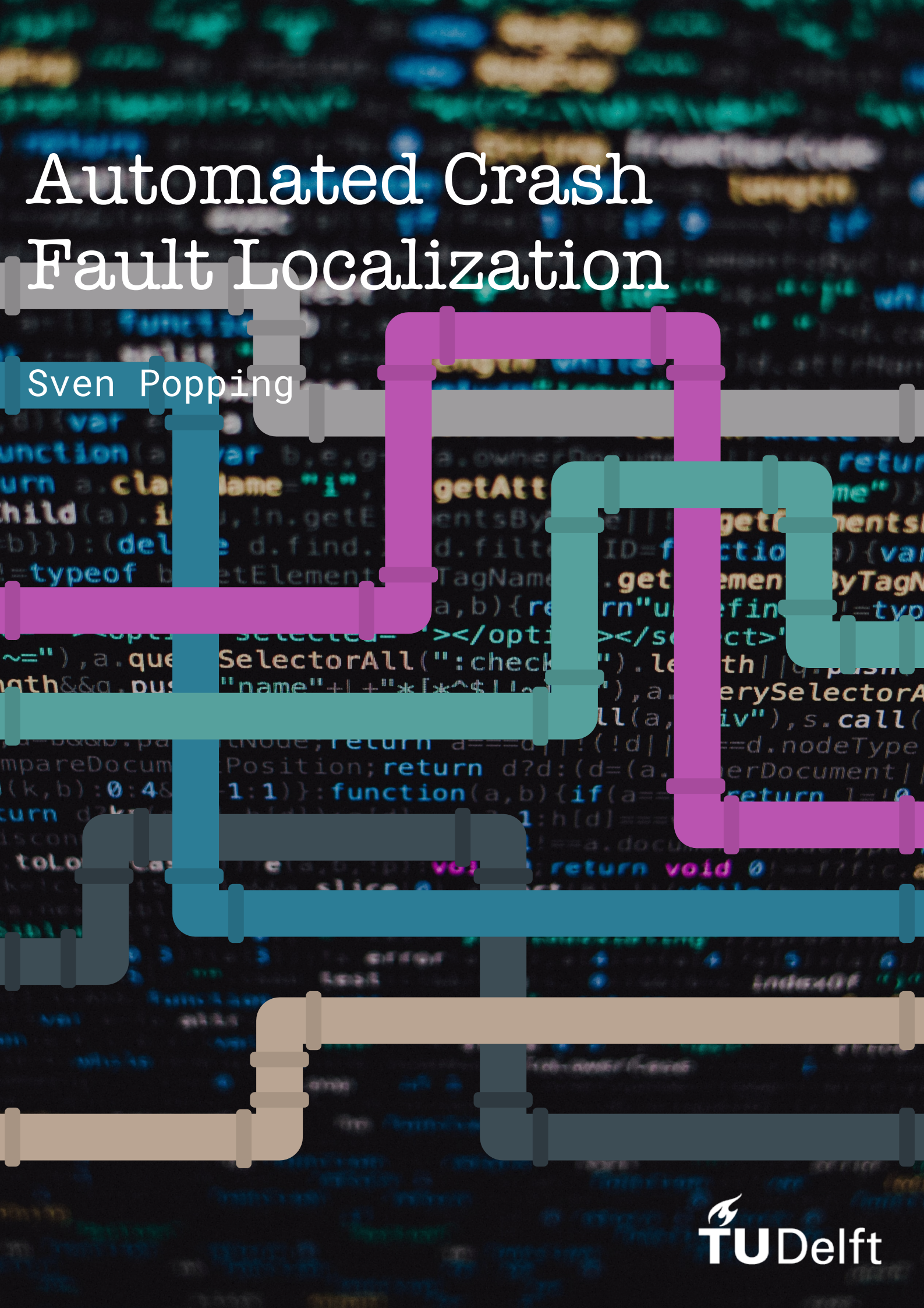
Automated Crash Fault Localization

Sven Popping



TU Delft

Automated Crash Fault Localization

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Sven Popping
born in Drachten, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Automated Crash Fault Localization

Author: Sven Popping
Student id: 4289455
Email: s.popping@student.tudelft.nl

Abstract

Debugging application crashes is an expensive and time-taking process, relying on the developer's expertise, and requiring knowledge about the system. Over the years, the research community has developed several automated approaches to ease debugging. Among those approaches, search-based crash reproduction, which tries to generate a test case capable of reproducing a given crash to make it observable to the developers, solely based on the stack trace included in the crash report. We believe that this makes crash reproduction the perfect candidate to achieve end-to-end crash fault localization. In this thesis, we explore and empirically evaluate the usage of search-based crash reproduction combined with spectrum-based fault localization on 50 real-world crashes. Starting from a crash report, we generate crash-reproducing test cases and use them in conjunction with the existing or an automatically generated unit test suite as input for spectrum-based fault localization. Our results show that, although, hand-written test cases remain the most efficient in the general scenario, automatically generated crash-reproducing test cases still reduce the number of statements to be investigated by developers. Additionally, when considering the best-case scenario where only crash-reproducing test cases covering the fault are evaluated, we observe no statistically significant difference between the accuracy of fault localization when using hand-written or automatically generated test cases. Our results confirm the feasibility of end-to-end automated crash fault localization. The results also identify new challenges for both automated test case generation and fault localization, as well as when they are combined.

Thesis Committee:

Chair:	Prof.dr. A. van Deursen, Faculty EEMCS, TU Delft
Daily supervisor:	Dr. X. Devroey, Faculty EEMCS, TU Delft
Committee Member:	Dr. A. Panichella, Faculty EEMCS, TU Delft
	Dr. M.M. de Weerdt, Faculty EEMCS, TU Delft
	P. Derakhshanfar MSc., Faculty EEMCS, TU Delft

In front of you lays the thesis entitled *Automated Crash Fault Localization*. The research is the first step towards end-to-end crash fault localization and automated program repair. The manuscript you are reading describes the work that has been conducted over the previous months, between September 2019 and June 2020. It has been written to obtain the degree of Master of Science at the Delft University of Technology within the Computer Science program.

From the moment Xavier pitched the project description to me, I was hooked, and combining two research fields would be an incredible opportunity. After a long process of diving into existing literature, figuring out how to link the components together, fixing bugs, re-running the entire evaluation several times, accidentally deleting parts of the research, and understanding the results, I am thrilled, and proud of, the final result.

First of all, I would like to thank Xavier, Arie, and Pouria for their assistance, support, valuable time, critical notes, and feedback during these nine months. Furthermore, I am proud of the (online) collaboration that has led to the paper we have submitted to the ASE2020 conference in a short time. It has been an honor to work with all of you!

With this work, I also conclude my time as a student at the Delft University of Technology. I would like to thank all my friends for the support during but also after studying. A special note to Ruth, who has been nothing but supportive during the previous nine months. Last but not surely least, I like to thank my parents for their endless support and unconditional love. *"Chefke komt er wel!"*

I want to thank everyone that is taking the time to read this thesis. I wish you a pleasant reading and hopefully you will have as much fun as I had writing this thesis.

Sven Popping
Delft, the Netherlands
June 2020

In memoriam of Oma † 01 February 2020

Contents

List of Figures

Chapter 1

Introduction

Since the invention of software in the 19th century, the usages have been ever-increasing. Nowadays, software programs are a fundamental part of our daily lives, as it is not only employed in but also critical to many systems and facilities around the world.

Despite the significant effort spent on software testing and verification, software systems still fail. A report by the Austrian software testing firm Tricentis estimated that software errors in 2016 cost the world economy $1.1 trillion dollars and affected 4.4 billion people worldwide [2]. To put that into perspective, software errors have affected more than half of the world's population [1], and the financial losses are more than the Netherlands' GDP in 2019 [3].

When a failure is detected, for example, due to faulty behavior or an error message, a developer has to debug the code and repair the program. Debugging the fault can be a time-consuming and complicated task because often little information is available (*e.g.,* a stack trace, a core dump, or an error description by the end-user).

Often, when a failure is reported, the first step for a developer is to write a *test case for debugging* [90]. These test cases reproduce the crash to make it *observable* to the developer and check whether it is *fixed* after patching the source code. With the crash observable, the second step is identifying the location of the underlying fault, also called *fault localization* [85]. Fault localization has been a complicated, tedious, and prohibitively expensive manual task [76], and given the size of today's software systems, the job has become even more tedious and time-consuming. The time spent, and thus the effectiveness of fault localization, depends primarily on the developer's experience and familiarity with the software program. Over the years, researchers have produced various automated approaches to assist developers in their debugging practices, including but not limited to *search-based test case generation* [24, 25] and *automated fault localization* [65, 85].

When writing software programs in Java, the developer typically writes unit test cases to verify the intended behavior of the methods and to detect faults do to changes in the software. Instead of tediously writing unit test cases manually, researchers have developed search-based test case generation approaches to generate them automatically.

These approaches often rely on a meta-heuristic optimizing algorithm [53] because of

the complexity of the search problem (*i.e.,* the search space may contain infinitely many test cases that could be generated). For instance, white-box search-based test case generation approaches [33, 36, 60] utilize evolutionary algorithms. Commonly, these approaches have the goal of maximizing the coverage of the generated tests suite, for example, line or branch coverage [35], or the weak mutation score [36]. Furthermore, studies have shown that the generated test suites using search-based test case generation, can reduce the debugging effort [62] and can be used for real-bug detection [12].

In recent years, researchers developed other approaches focused on the debugging of software failures. Among them, *search-based crash reproduction* approaches [30, 71] aim to generate one or more test cases that can reproduce a crash. These test cases serve as a basis for the debugging process, eliminating the step of creating a test case for debugging. As these test cases are generated automatically, they reduce the debugging effort for the developer.

Unlike other search-based test case generation approaches, crash reproduction does not aim to maximize the coverage or mutation score. Instead, it aims at reproducing the sequence of executions that led to the crash. Empirical evaluation with developers demonstrated the usefulness of crash-reproducing test cases to facilitate the debugging process [71], resulting in more fixes in a shorter time. We argue that the debugging effort could be further reduced by combining search-based crash reproduction with automated fault localization.

The primary goal of automated fault localization is to reduce the debugging effort by identifying (potentially) faulty statements (or *defective statements*) in a program [85]. Over the years, many approaches have been developed, of which the most popular technique is *spectrum-based fault localization* [85]. Spectrum-based fault localization relies on coverage information (the *program spectrum*) constructed from a set of *passing* and *failing* test cases. Based on the program spectrum, the approach identifies potentially *suspicious* statements that cause the underlying fault.

For the correct behavior of spectrum-based fault localization, the program spectrum must contain at least one failing test case. Otherwise, there is no fault to detect, so applying spectrum-based fault localization would be pointless. When a new failure is reported, it is can be assumed that there is no failing test case available to detect the underlying fault. Therefore, the developer first has to write a test case for debugging and thus has to understand the failure to expose it, reducing the benefits of automated fault localization. In this thesis, we use search-based crash reproduction to automate the definition of the test case for debugging, restoring the full benefit of automated fault localization.

To determine the *suspiciousness* of a statement, spectrum-based fault localization is predicated on a *similarity coefficient* to measure how close the execution pattern of a statement is to the execution pattern of the failure (approximated by the failing test case(s)). A higher similarity denotes a higher suspiciousness level for the statement.

Several similarity coefficients have been developed and evaluated in the field of automated fault localization [8, 9, 44, 84]. The advantage of spectrum-based fault localization is

that it only relies on the coverage information of the available passing and failing test cases, making it easy to integrate into existing testing and debugging infrastructures. The downside is the limited diagnostic accuracy as no optimum similarity coefficient can outperform all the others [8, 89].

Recently, automated fault localization has also been used as a preliminary step of many automated program repair approaches. The automated fault localization is used to identify (potentially) faulty statements to repair [41, 58]. In such a setting, the developer typically has to write a test case, which makes the fault observable before applying the program repair approach. We suggest that this step can also be automated using crash reproduction and propose to automatically locate the fault that causes the crash, opening a new path to the full automation of end-to-end program repair [13].

This thesis marks the first step toward the full automation of end-to-end program repair [13], by empirically evaluating the potential of combining search-based crash reproduction with spectrum-based fault localization, and we name it *Automated Crash Fault Localization*.

For our evaluation, we use BOTSING [30], a search-based crash reproduction framework, EVOSUITE [35], a search-based unit test generation framework, and GZOLTAR [21], a spectrum-based fault localization framework. Following best practices [63], we use 50 *real-world faults* from DEFECTS4J [47] manifesting as crashes and previously used for crash reproduction [72]. In particular, we (i) compare the performance of crash fault localization using hand-written and automatically generated test cases; (ii) investigate if particular combinations of written and automatically generated test cases provide higher accuracy when used with different coefficients for fault localization; and (iii) manually analyze and compare hand-written and automatically generated test cases to identify factors influencing spectrum-based fault localization accuracy.

The remaining structure of this thesis is as follows. In Chapter 2, we describe the background and related work on the topics: *fault-localization*, *automated unit test case generation*, and *automated crash reproduction*. Continuing on the state-of-the-art research, we define and motivate our automated crash fault localization approach and describe our implementation of this approach, in Chapter 3.

Subsequently, in Chapter 4, we describe the evaluation methodology to assess the effectiveness of our approach described in the previous chapter. To conduct the evaluation, we constructed components which assist us in obtaining the required datasets, in Chapter 5

In Chapter 6, we present the results from the evaluations that we have conducted on the automated crash fault localization approach to assess its feasibility. Chapter 7 is dedicated to discussing the results on our work, and we review the threats today validity. In Chapter 8, we provide a summarizing conclusion and present our view on interesting future research on the matter.

Chapter 2

Background and related work

In this chapter, we introduce and discuss the research efforts that have been made within the fields of *fault localization*, *automated unit test case generation*, and *automated crash reproduction*.

2.1 Fault localization

Fault localization is the task of identifying the location of faults (*i.e.,* localizing the defective statement) in software programs. Traditional and manual techniques used for fault localization are (i) *program logging*, (ii) *assertions*, (iii) *breakpoints*, and (iv) *profiling*.

When using program logging, a developer inserts log statements into the code to monitor variable values and other state information [32]. The information can be examined by the developer to diagnose the underlying fault by determining where the state of the program reaches an unexpected value.

Assertions are predicates that have to be true during the correct execution of the program. Developers add these predicates in the program as conditional statements that kill the execution if the criteria are not met. The assertions can be used to detect incorrect program behavior at runtime. [67]

To further inspect the program, breakpoints can be added to pause the program at a specific point for an examination of the current state. The developer can gradually add breakpoints, step the program line-by-line, or step into a method to further localize the fault. Early studies use this approach to assist developers in localizing the bug while the program is executed under the control of a symbolic debugger [29, 43].

Profiling is a form of runtime analysis to determine the performance of the program, using metrics such as execution time, memory usage, and method call counting. Commonly, profiling is used to optimize the performance of the program. However, it can also be leveraged for fault localization, *e.g.,* by detecting unexpected frequencies of different methods [17], identifying memory leaks or code performance drops [42], and examining the side effects of lazy evaluation [69].

The techniques described above are commonly manual and rely on the developer's expertise. Therefore, over the years, researchers have developed various approaches to automate the task of fault localization. The primary goal of automated fault localization is to reduce the debugging effort by identifying (potentially) faulty statements in the software program [85]. This reduction is achieved by ranking the statements in the software program, based on the suspicion that a statement is involved in the fault.

Wong *et al.* [85] classified these approaches into categories, including but not limited to (i) *spectrum-based*, (ii) *slice-based*, (iii) *state-based*, and (iv) *machine learning-based* approaches

Spectrum-based approaches

Spectrum-based fault localization, used in the approach proposed in this thesis, relies on coverage information (a *program spectrum*) of a set of *passing* and *failing* test cases. A program spectrum details the execution information of a program from a particular perspective. A widely used program spectrum is the *Executable Statement Hit Spectrum (ESHS)*, describing which statements have been executed at runtime. Usually, within spectrum-based fault localization, the program spectrum is based on the execution information of the program's test suite. Depending on the approach, the program spectra are based on passing test cases, failing test cases, or both.

Collofello *et al.* [28] suggested that the program spectra could be used for software fault localization. The program spectrum can be used by comparing the spectrum of failed test cases with the spectrum of successful test cases and analyzing the differences [6].

The goal of spectrum-based fault localization is to identify the statement with an execution pattern that is *as close as possible* to the failure pattern of all test cases [85]. Intuitively, a statement is more suspicious when the execution pattern of the statement is more similar to the failure pattern of all the failing test cases. For example, a statement that is only executed by a failing test case is likely to be part of the underlying fault. Conversely, if the execution pattern of the statement is similar to the execution pattern of all successful test cases, the statement is less likely to contain the bug.

The closeness level is quantified by a similarity coefficient, and the degree can be interpreted as the suspiciousness level of the statement. Over the years, many similarity coefficients have been proposed [8, 9, 44, 84]. Table 2.1 shows four of the best performing and well-studied coefficients, together with their algebraic form. Pearson *et al.* [63] showed that none of these four similarity coefficients provide a statistically significant difference over the others when used for spectrum-based fault localization.

To better understand the concept of spectrum-based fault localization, we look at the example, as shown in Table 2.2. The code snippet in column two contains a bug at Line 6, which should be `d = a / b` (instead of `d = b / a`). Assume there exists two passing test cases t_1 and t_2 with inputs $a = 1, b = 1$ and $a = 2, b = 1$, respectively, and one failing test case t_3 with the input $a = 1, b = 2$. Columns three to five contain the statement coverage for each of

Table 2.1: Similarity coefficients. For a given statement σ, N_F is the total number of failing tests, N_S is the total number of passing tests, N_{CF} is the number of failing tests covering σ, N_{UF} is the number of failing tests not covering σ, N_{CS} is the number of passing tests covering σ, and N_{US} is the number of passing tests not covering σ.

Coefficient	**Algebraic Form**
DSTAR* [84]	$S = \frac{N_{CF}^*}{N_{UF} + N_{CS}}$
OCHIAI [8]	$S = \frac{N_{CF}}{\sqrt{N_F \times (N_{CF} + N_{CS})}}$
BARINEL [9]	$S = 1 - \frac{N_{CS}}{N_{CS} + N_{CF}}$
TARANTULA [44]	$S = \frac{N_{CF}/N_F}{N_{CF}/N_F + N_{CS}/N_S}$

$\star$ - variable $* > 0$, we use $* = 2$ as it is the most explored [63].

the four test cases. The last column shows the suspiciousness level per statement calculated using the TARANTULA coefficient.

The result of the spectrum-based fault localization is a sorted list of statements based on the suspiciousness level in the last column. In this case, Line 5 and Line 6 are marked as the most suspicious, which would be correct as the bug is caused by a fault in Line 6.

Table 2.2: Example of spectrum-based fault localization, where S is calculated using the TARANTULA similarity coefficient (• = statement is executed by a test case)

#	Statements	Test t_1 $a=1,b=1$	Test t_2 $a=2,b=1$	Test t_3 $a=1,b=2$	N_{CF}	N_{CS}	S
1.	`if (a > b)`	•	•	•	1	2	0.50
2.	`c = a + b`		•		0	1	0.00
3.	`d = a * b`		•		0	1	0.00
4.	`else`				0	0	0.00
5.	`c = a - b`	•		•	1	1	0.67
6.	`d = b / a`	•		•	1	1	0.67
7.	`return c + d`	•	•	•	1	2	0.50
		Passing	Passing	Failing			

To apply spectrum-based fault localization to a program, the test suite for that program must have the property that $N_F > 0$ and $N_S > 0$. Intuitively, if $N_F = 0$, there is no fault in the program. And if $N_S = 0$, all the statements executed by the failing test case will be marked suspicious, adding no additional information for the developer.

Likewise, we can assume that for the defective statement holds that $N_{CF} > 0$, as a defective statement should at least be executed by one failing test case. OCHIAI, BARINEL, and TARANTULA are based on the fact that $N_{CS} + N_{CF} > 0$, which holds as $N_{CF} > 0$. Similarly, DSTAR assumes that for a defective statement $N_{CS} + N_{UF} > 0$. However, this assumption does not always hold, as all the failing test cases might cover a statement ($N_{UF} = 0$), and in the same time, the statement is not covered by any passing test case ($N_{CS} = 0$) (*e.g.,* due to the lack of test coverage). Intuitively, when $N_{CF} > 0$ and $N_{CS} + N_{UF} = 0$, then the statement should be the most suspicious of all. Nevertheless, in this case, DSTAR results in an undefined suspiciousness level due to a division by zero. In the original paper [84], no additional procedure has been defined to address this problem. In this thesis we will assume that $N_{CS} + N_{UF} \geq 0$ and when $N_{CS} + N_{UF} = 0$ that the DSTAR coefficient is equal to zero.

Slice-based approaches

Program slicing is used to localize the fault by reducing the size of the program by increasingly removing irrelevant parts of the program, such that the resulting program slice has the same behavior as the original program concerning the fault. A program slice is created by reducing the program to a *slicing criterion*, specifying a statement and a set of variables at interest. A slicing criterion is a pair $\langle s, V \rangle$, where s is a statement in the program, and V is a subset of the program variables [77]. The program slice, given the slicing criterion $\langle s, V \rangle$, consists of the set of statements and predicates that directly or indirectly affect the variables in V before the execution of s.

Weiser [77] proposed the first implementation of program slicing in 1978. The primary purpose of program slicing is to reduce the program's size, such that developers, unfamiliar with the program, do not have to go through the whole program to patch the fault. The concept is based on the assumption that a failure is due to an incorrect variable assignment, and the fault must be found in the slice associated with the statements affecting the variable value. Within the field of program slicing, there are three types of approaches: *static slicing*, *dynamic slicing*, and *execution slicing* [85].

Static slicing, proposed by Weiser [77], is computed without considering a program's input [20]. A static slice contains the minimal set of the executable statements that affect the variables in V up to the execution of s.

The problem of static slicing is that computing the minimal subset of statements is undecidable [77]. In the original approach, the program slice is approximated by using the program's control-flow graph (CFG). Researchers have ever since been trying to improve the approximation using different techniques. Ottenstein *et al.* [59] introduced static slicing using the program dependence graph (PDG), leveraging the graph reachability. Recent static slicing approaches have been applied to reduce the size of binary executable files [49] and to optimize type checkers [74].

A disadvantage of static slicing is that the slice contains all the executable statements that could affect the outcome, resulting in a slice containing statements that may be irrelevant to the fault. Rather than reasoning which statements affect the variables at interest, dy-

namic slicing utilizes the program's execution behavior (typically using a failing execution). Therefore, a developer does not only know what could have happened but can also see what has happened. Dynamic slicing contains all the statements that affect the program outcome concerning a particular input.

Korel *et al.* [50] proposed the notion of dynamic slicing, using dynamic analysis to identify the statements that affect the variables of interest in a particular program execution. As only one execution is taken into account, dynamic analysis can be considerably reduced the size of the slice, making it easier to localize the faults. Over the years, additional approaches have been proposed, including multiple-points dynamic slicing [92], scenario-oriented dynamic slicing [66], and relevant slicing [40].

As collecting dynamic slices may consume a lot of time and space, execution slicing can be used. Execution slicing is based on the data-flow of test cases, which can be used to locate program faults [11]. An execution slice for a given test case contains the set of statements executed by this test case. In other words, it extracts all the distinct statements in the execution path of a given test case.

Various execution slicing debugging tools have been developed, for example, eXVantage [79] and xSuds [10]. Agrawal *et al.* [11] used execution slicing by comparing the execution slice of one failed and one passing test case. This approach has been extended by using multiple passing and failing test cases [45, 82].

To compare the different slice-based approaches, Table 2.3 depicts an example program that contains a bug at Line 7 (assuming that the line should be `y = y * 2 * i`). The slicing criterion is defined as $\langle 7, y \rangle$. The static slice contains all the statements, which can affect the outcome of the variable `y`. The dynamic slice contains the statements affecting the variable `y` for a given test case (in the example $a = 3$). Finally, the execution slice contains the statements executed by a given test case, so in this case, the statement executed for $a = 3$.

State-based approaches

The state of a program contains the variables and values at a particular point at runtime, which can be utilized for fault localization. One way is through *relative debugging* [4], where the current state is compared to a *reference* state that is known to be correct. Another way is by comparing the program state of a passing test case with the program state of a failing test case.

Relative debugging starts with the developer, who formulates a set of assertions about critical data structures in the reference state and the execution state. These assertions specify locations at which the state of both programs should be the same. The relative debugger is then responsible for managing the execution of the two programs, validating the assertions, and reporting any differences [4]. If a difference is reported, the developer isolates the faulty code by recursively refining the assertions. Once the size of the isolated code is small enough, manual debugging techniques can be used to locate the underlying fault. Research has shown that relative debugging is a beneficial technique that allows developers to locate faults in programs that have been modified or ported to other languages [5].

Table 2.3: The difference between static, dynamic and execution slicing for the program with the bug `y = y * 2 * i` (● = statement is included in the slice). [85]

		Static slice for y	Dynamic slice for y with $a = 3$	Execution slice for y with $a = 3$
1.	`input(a)`	●	●	●
2.	`int i = 2`	●	●	●
3.	`int x = 0`			●
4.	`int y = 1`	●		●
5.	`if (i < a)`	●	●	●
6.	`  x = x + i`			●
7.	`  y = y * i //bug`	●	●	●
8.	`else`	●		
9.	`  x = x - i`			
10.	`  y = y / i`	●		
11.	`return y`	●	●	●

Zeller *et al.* [91] proposed a technique called *delta debugging*, which compares the states of passing and failing test cases. Instead of ranking the statement, delta debugging determines the suspiciousness of variables in the program. The suspiciousness of a variable is determined by replacing the value of a passing test case with the value of a failing test case and checking whether it changes the outcome of the test case. Unless the same failure is encountered, the variable is no longer considered suspicious.

Machine learning-based approaches

In the context of fault localization, machine learning is used to learn or deduce the location of the bug based on the available data (*i.e.,* the source code, code coverage, and the execution results of test cases). The machine learning-based fault localization approaches [16, 37, 81, 83] have the same key processing steps, with possible additional pre- and post-processing. Therefore, the machine learning-based fault localization approaches can be generalized as follows:

1. *Collect training dataset.* The first step is to collect the needed training dataset. Commonly, the training dataset consists of the program spectrum for each test case and the corresponding outcome (which is similar to spectrum-based fault localization). Depending on the approach, the training dataset is enriched with additional information, such as the relation between test cases.

2. *Train the machine learning model.* Using the training dataset, the machine learning model is trained and optimized, such that the model can predict the outcome of a test case. For example, based on the statements executed by the test case.

3. *Rank suspicious statements.* Based on the trained model, the statements in the program are ranking, considering the suspiciousness. The suspiciousness is calculated based on the results from the trained model.

Wong and Qi [81] proposed a *back-propagation neural network* (BPNN) fault localization technique. Back-propagation neural networks have a simple structure, are easy to implement, and have been proven to be suitable for approximating complex non-linear functions [81]. The input of the neural network is the executable statement hit spectrum, as used in spectrum-based fault localization, and the output is whether the executed statement should cause a failure or not. In other words, the BPNN is trying to estimate the execution outcome (passing or failing) based on the statements that are executed. After the model training, the suspicious level for each statement is calculated using *virtual test cases* (*i.e.,* test cases which execute only one statement).

When the program becomes larger, the number of statements involved in the fault localization process increases. It is resulting in a large BPNN, which has been proven to be less effective as well as computationally expensive [81]. To remedy this problem Wong and Qi [81] added execution slicing to reduce the program size.

Originally the proposed method has been created for the programming language C. However, the approach is independent of the development paradigm, so back-propagation also works on Object-Oriented Programming Languages, such as Java [16]. Furthermore, Ascari *et al.* [16] proposed an approach that uses Support Vector Machines with the same goal and methodology.

In 2011, Wong *et al.* [83] proposed another approach based on *Radial Basis Function* (RBF) networks, which uses the same input as output as the back-propagation neural network. The advantage of the RBF is that it has a faster learning rate and is less prone to paralysis and local minima.

Evaluation

Generally, a fault localization approach F takes as input a program P and a test suite T with at least one failing test case. It produces a sorted list of statements, with $s_1 \geq \ldots \geq s_N$, where s_i denotes the suspicious level of statement i and N is the number of executable statements in the program P. The effectiveness of the approach F is based on the percentage of the program P that is needed to be examined by a developer before the defective statement d is identified [85].

Researchers have developed multiple evaluation metrics to determine the diagnostic accuracy, these metrics are the EXPENSE score [44], the EXAM score [78], T-SCORE [51] and the LIL [55].

The EXPENSE score, shown in Equation 2.1, calculates the percentage of the program that does not need to be examined when the statements are ranked according to their sus-

piciousness level. In Equation 2.1, n is the rank of the defective statement d in the fault localization report and N is total number of statements in the program P.

$$\text{Expense} = \frac{N-n}{N} \tag{2.1}$$

Wong *et al.* [78] proposed the most commonly used and accepted evaluation metric, the EXAM score, which is also used in this thesis. The EXAM score, shown in Equation 2.2, is similar to the EXPENSE score, the difference is that it calculates the percentage of code which *needs to* be examined by the developer. In Equation 2.2, n is the rank of the defective statement d in the fault localization report and N is total number of statements in the program P.

$$\text{EXAM} = \frac{n}{N} \tag{2.2}$$

The T-SCORE, see Equation 2.3, is designed for non-statistical fault localization methods, which produce a small set of suspicious statements in a program. It estimates the percentage of code a developer does not have to examine before identifying the defective statement. The metric uses the program dependency graph (*PDG*) to compute a set of vertices in the graph that must be examined to reach the defective statement. The smallest set of vertices that includes the defective statement is called the dependency sphere (*DS*).

$$\text{T-Score} = 1 - \frac{|DS|}{|PDG|} \tag{2.3}$$

Locality Information Loss (LIL) [55] is an alternative evaluation metric, which uses a measure of distribution divergence between the suspiciousness level's distribution and the perfect expected distribution. The advantage of the LIL metric is that it does not depend on a ranked list of statements and can be applied to non-statistical models, making it suitable for determining the effectiveness of a fault localization technique for automated program repair [55].

2.2 Automated unit test case generation

Unit test cases are a type of test cases used in software testing. The primary goal of unit test cases is to ensure that a component (*i.e.,* a section of the program) meets its design, behaves as intended, and to discover fault caused by future program changes.

Furthermore, the generated unit test cases can be used for debugging. When the underlying fault of a crash is identified, the unit test cases allow the developer to eliminate components by checking that the component is functioning as intended, concerning the crash. The higher the code coverage, the quicker the developer can eliminate parts of the code.

Researchers have introduced several automated test generation approaches, which allows automatic generation of unit test cases according to predefined criteria [53]. For instance, white-box search based test case generation approaches [33, 36, 60] rely on evolutionary

algorithms to generate unit test cases. EVOSUITE [33] is a white-box search-based test case generation approach for unit test case generation. The input for EVOSUITE is one or more target classes in the program, and the output is a test suite maximizing a given test criterion (*e.g.,* line coverage, branch coverage, or weak mutation coverage).

Previous studies confirmed that EVOSUITE can generate test cases with high code coverage [36, 61], real-bug detection power [12], reduce debugging costs [62], and complement hand-written test cases [25]. Besides that, EVOSUITE has been used in the context of automated fault localization [22, 64]. Campos *et al.* [22] proposed a search-based algorithm to reduce the entropy of the diagnostics ranking. They empirically established that the approach reduced the number of statements marked as potentially suspicious (hence, the developer has to examine less code).

Perez *et al.* [64] showed that the diagnostic accuracy of the automated fault localization approach depends on the *quality* of the test suite. Therefore, Perez *et al.* introduced a metric [65], called DDU, to quantify the quality of the test suite based on the *density*, *diversity*, and *uniqueness*. The higher the DDU metric, the better the diagnostic accuracy of the automated fault localization approach.

2.3 Automated crash reproduction

As mentioned before, the first step for the developer is usually to write a test case for debugging when a new crash is reported (or at least, derive the input that reproduces the crash). Finding these inputs can be a time consuming and tedious task [52]. Researchers have proposed various automated techniques to generate tests that reproduce the target crashes. The collective name for these techniques is called automated crash reproduction.

To generate these test cases, crash reproduction approaches leverage either *runtime data* [15, 18, 23, 27, 38, 56, 68, 73] or *stack traces* [19, 26, 57, 71, 88], and either use a *record-replay* or a *post-failure* approach.

Record-replay approaches

Record-replay crash reproduction uses software or hardware instrumentation and monitoring to record and observe the runtime data at the moment of failure. The developer can replay the recorded data and so be used to identify the underlying fault [15, 23, 56]. This makes these approaches efficient and easy to use, as the developer can rollback to the state of the program before and after the crash. Therefore, the developer can quickly observe the crash.

Over the years many record-replay approaches have been developed, including RECRASH [15], BUGNET [56], JRAPTURE [73], SYMCRASH [23], and MOTIF [38]. These approaches all record runtime data to enable the developer to replay the state of the program. Compared to the other approaches, SYMCRASH uses a selective recording approach that only monitors the important and complex methods to reduce instrumentation overhead. Moreover, MOTIF uses crowdsourced data to analyze recurrent patterns in crash data, mak-

ing it more efficient [38], under the assumption that crashes from different programs have the same characteristics.

However, the disadvantages of record-replay approaches are the decrease in the overall performance [68], due to the overhead caused by the instrumentation, and the privacy risk due to the possible collection of sensitive data [26].

Post-failure approaches

Since using runtime data raises multiple difficulties, post-failure approaches are more suitable for general use. Post-failure approaches replicate the crash by utilizing data that is available after the crash. The most commonly available data are stack traces, as they are easily collected using bug tracking systems (*e.g.,* JIRA or trac) or can be extracted from log files. However, most post-failure approaches require additional data as input besides the stack traces (*e.g.,* core dumps, software models, existing test suite, or class invariants) [26]. Examples of post-failure approaches are STAR [26], JCHARMING [57], and MUCRASH [88]. STAR and JCHARMING both use a combination of stack traces and model checking to generate a crash-reproducing test case, whereas MUCRASH generates a test case by mutating existing test case until the crash is detected.

Stack trace-based post-failure approaches are a subset of post-failure approaches that solely rely on stack traces as a source of information. This advantage makes these approaches easy to integrate into existing software systems and do not acquire additional steps for the developer (*i.e.,* no additional data is required). Soltani *et al.* [71] showed that search-based crash reproduction approaches relying on a single objective guided genetic algorithm, outperform all other stack trace-based approaches in terms of crash reproduction ratio (*i.e.,* can reproduce more crashes). In addition, Soltani *et al.* [71] confirmed that the generated test cases are useful for automated debugging and manual debugging.

In the guided genetic algorithm, a population of candidate test cases evolves towards a crash-reproducing test case. Each candidate test case is a sequence of executable statements in the program under test. The initial population of test cases is created by generating random test cases, which at least call one of the methods in the stack trace. The evolution is an iterative process of which the population in each iteration is called a generation. In each generation, the fitness of each test case is evaluated using a *fitness function*. A fitness function quantifies how close a given candidate is to the global optimal. In the case of crash reproduction, the fitness function defines the similarity of the stack trace generated by the test case compared to the original stack trace. The fitness function can be extended with additional heuristics to improve the quality of the test cases. The fitter candidates are stochastically selected and recombined to generate new candidate solutions (called off-spring) and form the new generation. The new generation is then used in the next iteration of the genetic algorithm. Commonly, the algorithm is terminated when either a satisfactory fitness is reached or after a maximum number of iterations.

A more extensive evaluation by Soltani *et al.* [72], demonstrated the ability of search-based crash reproduction approaches to reproduce complex crashes. The approach defines

a fitness function (called *Crash Distance*) using three heuristics to guide the search process: (i) *line coverage* checks whether the generated test case covers the line of code where the exception is propagated; (ii) *exception coverage* indicates if the generated test case throws the same type of exception as the given stack trace; and (iii) *stack trace similarity* compares the similarity of the original stack trace with the one produced by the test.

BOTSING [31] is an open-source and well-tested framework for search-based crash reproduction, and it implements the Crash Distance fitness function, along with other new techniques improving it. The input of BOTSING is a stack trace and one of the *frames* in the stack trace (*i.e.,* one of the lines indicating the class and method where the exception propagated) as the *target frame*. Then, it generates one or multiple unit test cases, which tests the class in the target frame and (when executed) causes a crash producing the identical stack trace. For example, providing the stack trace of Listing 2.1 and target frame 3 (Line 3) to BOTSING, it generates a unit test for the class `DocumentContentDisplayer` throwing a `NullPointerException` propagating trough the same first three frames of the stack trace.

One of the crash reproduction approaches implemented in BOTSING uses two helper objectives. Along with Crash Distance, the algorithm uses the heuristics *Method Sequence Diversity* and *Test Length*, transforming it into a multi-object search problem. The Method Sequence Diversity heuristic enables BOTSING to reproduce multiple crash-reproducing test cases, which are diverse in the method calls sequences. Moreover, the Test Length ensures that BOTSING does not generate huge test cases.

A recent study [31] shows that algorithms relying on the two helper objectives increase the ability to reproduce crashes, hence increasing the crash reproduction ratio. Moreover, by utilizing this multi-objective approach, BOTSING can produce multiple crash-reproducing test cases, which are diverse in the method calls sequences, for a single given crash.

Listing 2.1: XWIKI-13303 crash stack trace [72]

```
0 java.lang.NullPointerException
1     at [...].XWikiDocument.getXDOM(...)
2     at [...].DocumentContentDisplayer.getContent(...)
3     at [...].DocumentContentDisplayer.display([...]:248)
4     at [...].DocumentContentDisplayer.display([...])
5     at [...].DefaultDocumentDisplayer.display([...])
6     ...
```

Test cases generated by BOTSING can be directly used for automated fault localization, as the generated test case ensures a crash will happen that includes the target frame in its stack trace. However, the choice of the target frame for crash reproduction can impact the outcome of automated fault localization.

For example, assume that the defective statements are located in frame 2 of the stack trace in Listing 2.1 (*i.e.,* the fault is in `getDisplayName`). When reproducing this crash with the target frame set to 1, BOTSING will generate a unit test for the class `XWikiDocument` that throws a `NullPointerException` with a stack containing only the first frame (from line 0 to 1 in Listing 2.1). Although this generated test case fails (*e.g.,* if it does not respect a precondition of the method `getXDOM`), it might not cover the fault. In practice,

the developer cannot know beforehand if a crash-reproducing test case covers a fault or not, as it would mean that the developer already knows what the fault is and would not need fault localization in the first place. For this reason, previous researches recommended reproducing higher frames to cover the underlying fault [26, 71].

Chapter 3

Automated crash fault localization

In this chapter, we present a motivational example demonstrating the usefulness of automated crash fault localization. Furthermore, we provide the details of our approach.

3.1 Motivation

The program under test, depicted in Table 3.1, shows a code snippet of the method `countMatches` with the coverage of the different test cases $\langle t_1, \ldots, t_4 \rangle$ that are written by the developer. The method counts the occurrence of a `letter` in the a `sentence` while ignoring the case. However, the method contains a bug: the counter is incremented by 2 if the character in the string is the uppercase equivalent of the search character (see Line 7). Line 7 should then be `count += 1`.

Of the four hand-written test cases $\langle t_1, \ldots, t_4 \rangle$, t_4 is the only failing test cases, as it executed the defective statement on Line 7. To localize the fault, the developer could apply spectrum-based fault localization, since there is a failing test case. When applying spectrum-based fault localization, it does indeed identify the defective statement, as Line 7 is the only statement executed by a failing test case. Using this information, the developer can quickly find the defective statement and patch the buggy method.

With the program patched, the test suite green, it is time to ship the code into production. Once in production, a user enters the input combination `(null, 'a')` which causes a `NullPointerException`, thrown at Line 3. By logging the exceptions, the developer gains insight into the fact that there is an error in the program, due to unexpected input. Spectrum-based fault localization cannot be applied to debug the fault because there is no failing test case among the existing test cases.

To apply spectrum-based fault localization, the developer has to write a *debugging test case* exposing the crash. This requires the developer to understand the underlying cause of the crash and, therefore, find the bug first. Consequently, spectrum-based fault localization loses its primary purpose, which is to reduce the debugging effort, when applied to a newly reported crash.

Table 3.1: Example program with coverage of the hand-written test cases ($\langle t_1, \ldots, t_4 \rangle$) and a crash-reproducing test case t' (● = covered by the test case)

#	Statements	t_1	t_2	t_3	t_4	t'
1.	`countMatches(String sentence, char letter)`					
2.	`  int count = 0;`	●	●	●	●	●
3.	`  for (char idx : sentence.toArray())`	●	●	●	●	●
4.	`    if (idx.isCapital())`	●	●	●	●	
5.	`      idx = idx.toLowerCase();`			●	●	
6.	`      if (letter == idx)`			●	●	
7.	`        count += 2 //bug`				●	
8.	`    else`	●	●			
9.	`      if (letter == idx)`	●	●			
10.	`        count += 1;`	●				
11.	`  return count;`	●	●	●	●	

To restore the primary purpose, we argue that search-based crash reproduction (simply called crash reproduction) could automate the task of writing the debugging test case. The debugging test case would fail, by reproducing the crash, on the current codebase. In our example of Table 3.1, the crash-reproducing test case t' has been generated using crash reproduction. With the failing crash-reproducing test case, spectrum-based fault localization can be used again. In this case, applying spectrum-based fault localization with the additional crash-reproducing test case would identify Line 2 and 3 as potentially faulty. We consider this an accurate diagnosis because the patch would be a *null-check* at the beginning of the method.

3.2 Approach

Figure 3.1 depicts an overview of the Automated Crash Fault Localization (ACFL) approach we propose. The approach consist of three components: *crash reproduction*, *unit test generation* and *fault localization*, of which unit test generation is an optional component.

On the left side of the diagram the required input is shown, *i.e.,* the *program*, the *test-suite*, and *crash report*. The primary reason for these inputs is that they are easy to obtain and do not require additional effort from the developer (*i.e.,* it is not necessary to define additional information).

The program is the code of the software program where the crash originated from. Depending on the techniques used for crash reproduction, fault localization and unit test generation, the program should be the source code, the compiled code, or both. If there is a test suite for the program, then this test suite is input for the automated fault localization process. For non-static approaches, the test suite should most likely be provided in compiled form. The crash report contains a description of the crash, depending on the approach, the

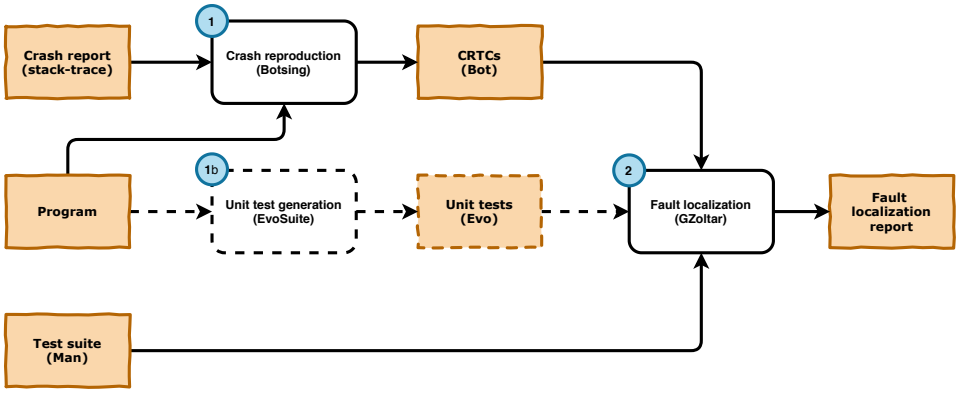


Figure 3.1: Automated Crash Fault Localization (ACFL) approach

format of the crash report may differ (*e.g.,* a core dump or a stack trace).

The output of the approach is a *fault localization report*, depicted on the right side of the diagram. Generally, the fault localization report ranks the statements in the program under test based on their suspiciousness towards the crash fault.

In the first step ①, the crash report is combined with the program under test to generate one or multiple crash-reproducing test cases. The generated test cases should fail on execution, as most fault localization approaches [85] depend on failing test cases. In the second step ②, the failing crash-reproducing test cases and the existing test suite are combined in the fault localization process to write the fault localization report. As an alternative to the existing test suite, generated test cases can be used, adding the step ⑯ to the approach. This is relevant when, for example, specific parts of the program are not sufficiently tested. In this case, it is recommended to generate a test suite with a sufficient level of coverage, as Perez *et al.* [64] showed that the density of the test suite influences the diagnostic accuracy of fault localization.

3.3 Implementation

In the previous section, we defined the general approach of automated crash fault localization. We follow this by describing our implementation of the ACFL approach.

In the first step ①, crash-reproducing test cases should be generated. Researchers have developed various techniques depending either on runtime data [15, 18, 23, 27, 38, 56, 68, 73] or stack traces [19, 26, 57, 71, 88].

To reduce the debugging effort as much as possible, we decided to use a crash reproduction approach that solely employs stack traces, as stack traces can be easily obtained and do not cause a decrease in performance or are a privacy risk.

Many approaches solely rely on stack traces, as described in Section 2.3. Soltani *et al.* [71] showed that the search-based crash reproduction approach using the Crash Distance

fitness function outperforms all other stack trace-based methods in terms of crash reproduction ratio. The test cases generated by this approach are also useful for manual debugging, allowing the developer to identify the defective statement with the fault localization report. Therefore, we decided to use BOTSING [31], a search-based crash reproduction framework which implements the Crash Distance fitness function, to generate the crash-reproducing test cases.

An automated fault localization approach is required in step ② to generate the fault localization report. As shown by Wong *et al.* [85], there are many different approaches, including the slicing-based, spectrum-based, state-based, and machine learning-based approaches.

Out of these four approaches, we deem slicing-based and state-based fault localization unfit. Slicing-based approaches, because most slices are lengthy and hard to understand [80], making them inefficient for fault localization. State-based approaches, because it requires a correct reference state [4] or it requires iterative feedback by the developer [91].

Both spectrum-based and machine learning-based approaches fit our requirements for the automated crash fault localization approach. However, we decided to use spectrum-based fault localization as more research has been conducted in that field than in the field of machine learning-based fault localization [85].

Spectrum-based fault localization approaches rely on a similarity coefficient. Pearson *et al.* [63] showed no significant difference between the best-studied coefficients. Also, these similarity coefficients have not been tested using automated crash-reproducing test cases. Therefore, GZOLTAR [21] will be used as the spectrum-based fault localization framework, as it includes over 20 similarity coefficient.

The unit test generation in step ⑪, optionally used when there is little or no test coverage available, is performed by EVOSUITE [33], as it is the best open-source search-based unit test generation tool available [48, 54, 61].

Additionally, previous studies confirmed that EVOSUITE can generate test cases with high code coverage [36, 61], real-bug detection [12], and reduce debugging costs [62]. Besides that, EVOSUITE has been used in the context of automated fault localization [22, 64].

To sum up, we rely on BOTSING [31], an open-source crash reproduction framework, for generating the crash-reproducing test cases. The fault localization is executed by GZOLTAR [21], an open-source spectrum-based fault localization tool used in other evaluations providing over 20 similarity coefficients [63]. The alternative step to generate unit test cases relies on EVOSUITE [33], the best open-source search-based unit test generation tool available [48, 54, 61].

Chapter 4

Evaluation

In this chapter, we define the research questions and the methodology to evaluate the proposed automated crash fault localization approach in Chapter 3. We start with defining the research questions and their aim. We then describe the used evaluation metrics, the dataset and the test suites used, after which we will focus on the data analysis to evaluate the results.

4.1 Research questions

To evaluate the effectiveness of automated crash fault localization compared to the manual crash fault localization using hand-written test cases for debugging. In this case, we assume that the performance of the manual localization of the crash error is the thing to improve or at least match. Therefore, we formulate the following research questions:

RQ1: *How does automated crash fault localization perform in comparison to manual crash fault localization in terms of accuracy when* BOTSING *generates one or more crash-reproducing test cases?*

With **RQ1**, we aim at evaluating to what extent automated crash fault localization is feasible in the *general scenario* when using automated crash reproduction compared to hand-written test cases. In the general scenario, we compare all the crash-reproducing test cases generated by BOTSING.

However, as mentioned in Section 2.3, BOTSING requires a target frame for which the test case is generated. The crash-reproducing test case will simulate the propagation of the exception, as in the given stack trace up to that target frame level. There is no guarantee that this propagation is not an expected behavior of the (target) class (*e.g.,* if an (implicit) precondition is not respected). From a fault localization perspective, this means that the (failing) crash-reproducing test cases introduce noise in the program spectra. These test cases do not fail due to the underlying bug. Ideally, in such cases, BOTSING should be

aimed at generating crash-reproducing tests for a higher frame. In practice, however, it is not known in advance which frames should be targeted.

In our second research question, we address this concern by looking at the *best-case scenario* in which BOTSING only generates a crash-reproducing test case that triggers the underlying bug.

RQ2: *How does automated crash fault localization perform in comparison to manual crash fault localization in terms of accuracy when* BOTSING *generates one or more crash-reproducing test cases covering the fault?*

Finally, our last research question aims at understanding the opportunities and challenges of automated crash fault localization and identify the strengths and weaknesses of the automatically generated crash-reproducing test cases.

RQ3: *Which factors influence the performance and applicability of automated crash fault localization?*

4.2 Evaluation metrics

In literature, most automated fault localization techniques are evaluated on artificial faults created by mutating the source code of a program. Generally, only one statement in the code is modified to create a bug in the code. Therefore, the standard technique to evaluate an automated fault localization approach F is based on a program P with one defective statement d [63] (see Section 2.1). The fault localization technique T outputs a ranking of the statements in P ordered by their suspiciousness level towards the crash. The diagnostic accuracy is then computed based on the rank of the defective statement d. The most popular evaluation metric is the EXAM score [78] (see Equation 2.2).

However, to apply this metric to real-world faults, we should extend the standard methodology as proposed by Pearson *et al.* [63]. As we perform our evaluation on real-world faults we have to account for *ties in the suspiciousness level*, *multi-statement faults*, and *faults of omission*.

Ties in suspiciousness level The standard methodology assumes that each statement in the ranking has a unique suspiciousness level [63]. However, in real-world problems, the possibility exists that two separate code blocks or statements have the same suspiciousness level. We assume that the sort function used to order statement arbitrarily breaks these ties. Therefore, when multiple statements have the same suspiciousness level, the statements are given the same rank.

Multi-statement faults A multi-statement fault is a fault for which the fix spans multiple statements. It is estimated that 76% [47] of the real-world faults are multi-statement faults. In this case, the program P contains multiple defective statements d.

We assume that any of the defective statements needs to be localized to understand the underlying cause of the crash. Therefore, if a program *P* contains multiple defective statements, then the EXAM score is based on the defective statement with the highest suspiciousness level.

Faults of omission A fault of omission is a fault for which the patch consists of adding new statements rather than modifying or deleting existing ones. So, the faulty program *P* contains no defective statement *d*, but some statements are missing. In this case, it is not straightforward which statement should be identified to localize the fault.

Pearson *et al.* [63] has defined a methodology and dataset to address this problem and determine which statements should be reported. In case of a fault of omission, we do not speak of the *defective statement*, but instead, call them *candidate statements* because these statements are not necessarily defective.

As a general rule, the candidate statements for a fault of omission are the executable declarations before and after the location where the new statements are to be inserted. For example, in Listing 4.1, the candidate statements are the statements that can be executed directly before and directly after the location of the patch.

Listing 4.1: Candidate statements in case of if-else [63]

```
0 public void exampleIfElse() {
1   if (expression) {
2     beforeStatement();                    < candidate statement
3   } else {
4     beforeStatement();                    < candidate statement
5   }
6   // patch
7   afterStatement();                       < candidate statement
8 }
```

However, in some cases, additional statements are included, for example in Listing 4.2. In this case, all the possible statements before and after the patch localization are included in the set of candidate statements because it is not possible to determine in advance which of these statements will influence the program outcome.

Listing 4.2: Candidate statements in case of initializer block [63]

```
0 public void exampleInitializerBlock() {
1   map = new HashMap();                    < candidate statement
2   map.put(k1, v1);                        < candidate statement
3   // patch
4   map.put(k3, v3);                        < candidate statement
5   map.put(k4, v4);                        < candidate statement
6   return map;                             < candidate statement
7 }
```

4.3 Dataset

We make use of the DEFECTS4J [47] dataset (version 1.5.0), which consists out of 435 real faults from 6 open source projects: JFREECHART, GOOGLE CLOSURE, APACHE COMMONS LANG, APACHE COMMONS MATH, MOCKITO and JODA-TIME. For each fault, DEFECTS4J provides a faulty and a fixed version of the program together with a minimized patch that represents the isolated bug fix. The minimized patch indicates which statements in the code should be reported by the fault localization approach. Besides the different versions of the program, there is also a hand-written test suite available including a bug triggering test case (that fails in the faulty version).

For the evaluation, we use 50 real-world faults from the DEFECTS4J [47], which have previously been used for crash reproduction [72]. In addition, we use the JCRASHPACK [72] dataset and the fault localization dataset by Pearson *et al.* [63].

The JCRASHPACK dataset extends the DEFECTS4J dataset by supplying the required data needed for crash reproduction, including the crash stack traces and a list of reproducible frames. Pearson *et al.*'s fault localization dataset contains additional information needed that is for fault localization, such as a list of defective and candidate statements and the *statement-lines-of-code* for each of the DEFECTS4J programs.

4.4 Test suites

For each target frame of each crash, we consider five different test suites: (i) the hand-written passing test suite (Man_{pass}^{+}), provided by DEFECTS4J; (ii) the hand-written failing test suite (Man_{fail}^{+}), also provided by DEFECTS4J; (iii) a test suite containing a single crash-reproducing test case (Bot_{fail}^{1}), generated by BOTSING; (iv) a test suite containing multiple crash-reproducing test cases (Bot_{fail}^{+}), also generated by BOTSING; and (v) a generated test suite (Evo_{pass}^{+}) with unit test cases generated by EVOSUITE. Figure 4.1 depicts an overview of the considered test suites for the evaluation.

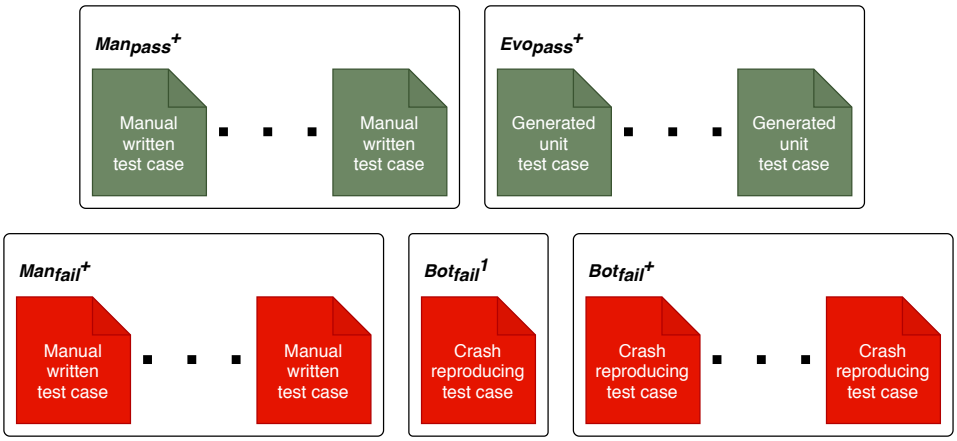


Figure 4.1: Test suites used for the evaluation (the color green indicates that a test case is passing, red indicates a failing test case).

Hand-written test suites (Man_{pass}^{+} and Man_{fail}^{+})

The hand-written test suites are extracted from the DEFECTS4J dataset. For the evaluation, we split the test suite for each program in a test suite containing the *relevant passing test cases* (Man_{pass}^{+}) and a test suite comprising the *relevant failing test cases* (Man_{fail}^{+}).

To avoid the executing of the whole test suite, we filter the existing test suite and only use the *relevant test cases*. A test case is relevant for a fault if it executes at least one class listed in the crash stack trace.

DEFECTS4J also provides a list of relevant test cases, which are the test cases that execute at least one of the modified classes (*i.e.,* classes that are changed in the patch). However, we argue that the provided list is not exhaustive and cannot be used as-is. First, because it only contains the test cases of the modified classes and no unit test cases for all the classes listed in the stack trace. And second, because in practice, the set of modified classes are unknown in advance.

Therefore, we define the following methodology for obtaining the relevant test cases from the DEFECTS4J whole test suite:

(i) Each test case in the test suite is executed.

(ii) For each of the test cases, the `JVM ClassLoader` is used to determine which classes are loaded and, therefore, executed by the test case.

(iii) If one of the loaded classes is in the crash stack trace, the test case will be marked as a relevant test case.

After that, each test case is executed on the faulty version of the DEFECTS4J program to obtain the relevant passing test cases from the relevant test cases.

Obtaining the relevant failing test cases is a lot easier, as DEFECTS4J provides a list of fault triggering test cases. These test cases have the properties that they fail before the fix and pass after the repair, and the test case failure is not random and does not dependent on the test execution order.

Crash-reproducing test cases (Bot_{fail}^{1} and Bot_{fail}^{+})

To generate the crash-reproducing test cases, we executed BOTSING with a time budget of 3 minutes, three search objectives, and using the *SPEA2* multi-objective evolutionary algorithm [31]. The algorithm uses the main objective *Crash Distance* and the two secondary objectives *Test Length* and *Method Sequence Diversity*. All the other parameters are left to their default value.

For each frame of each stack trace, we ran two rounds of BOTSING, producing two test suites Bot_{fail}^{1} and Bot_{fail}^{+}. In the former, BOTSING stops after finding the first crash-reproducing test case (*i.e.,* the test suite consists out of a single crash-reproducing test case). In the latter, BOTSING continues the search process after the first crash-reproducing test case and stops after the time budget runs out (*i.e.,* the test suite consists out of multiple

crash-reproducing test cases). BOTSING is based on a meta-heuristic optimizing algorithm, so there is no guarantee that a crash-reproducing test case is generated on the first run. Therefore, we give BOTSING a maximum of 15 attempts to generate a crash-reproducing test case.

Automatically generated unit test suites (Evo_{pass}^{+})

For the generation of the unit test cases, we use EVOSUITE with the default parameters, also used in a previous study by Shamshiri *et al.* [70]. Shamshiri *et al.* used EVOSUITE with the *Whole Test Suite* fitness function [34] and a time budget of 1 minute.

Similar to the Man_{pass}^{+}, we only generate unit test cases for the *relevant classes*. So, for each crash, we target all the distinct classes appearing in the stack trace, resulting in a test suite per crash.

Non-deterministic test cases

The test suites generated by BOTSING and EVOSUITE come with specific scaffolding, which ensures that each test case in the test suite is always executed in the same state to avoid non-deterministic test cases (which could lead to flaky test cases). Unfortunately, due to compatibility issues with GZOLTAR, we had to remove the scaffolding from the generated test cases. To compensate for the possibility that a test case might become non-deterministic, we add a step to identify and remove the non-deterministic test case by comparing the test case's output before and after the scaffolding has been removed. If the output of the test case is different, then the test case will be removed.

Undeclared exception thrown

Furthermore, EVOSUITE can generate passing test cases that create an expected *undeclared exception*. In this case, EVOSUITE surrounds the test case in a try-catch block to ensure that the test case succeeds. However, the possibility exists that a generated test case reproduces the crash [71]. This may cause interference in the automated fault localization approach because a crash triggering execution path will be labeled as passing. For this reason, we have decided to remove all test cases covering an *undeclared exception*.

For example, the test case generated by EVOSUITE for `LANG-16b` shown in Listing 4.3. In this example, the input `"-0x"` generated by EVOSUITE is a possible input that will trigger the underlying bug causing the crash.

Listing 4.3: Undeclared exception generated by EVOSUITE

```
0 @Test(timeout = 4000)
1 public void test052() throws Throwable {
2   // Undeclared exception!
3   try {
4     NumberUtils.createInteger("-0x")
5     fail("Expecting exception: NumberFormatException")
6   } catch (NumberFormatException e) {
7     verifyException("org.apache.commons.lang.math.NumberUtils", e);
8   }
9 }
```

In the same case, BOTSING can generate a crash-reproducing test case that throws an exception. When the thrown exception is not an *undeclared exception*, it is surrounded by a try-catch block making it a passing test case. For example, the test case generated for `LANG-16b` with target frame 1 shown in Listing 4.4. In this case, we assume that the generated test cases should indeed fail. Therefore, we only remove the try-catch block instead of the entire test case.

Listing 4.4: Exception generated by BOTSING for `LANG-16b`

```
0 @Test(timeout = 4000)
1 public void test0() throws Throwable {
2   try {
3     NumberUtils.createNumber("hnQfuK+F\"w>3%jyxr")
4     fail("Expecting exception: NumberFormatException")
5   } catch (NumberFormatException e) {
6     verifyException("org.apache.commons.lang.math.NumberUtils", e);
7   }
8 }
```

4.4.1 Configurations

In our evaluation, we combine the various test suites into five different combinations. Each configuration persists out of one set of failing test cases, which expose the crash, and one set of passing test cases. Table 4.1 depicts an overview of the different configurations.

Table 4.1: Five configurations used for the evaluation.

Configuration	Description
Man_{fail}^{+}-Man_{pass}^{+}	Contains only hand-written test cases, for both the crash exposing test cases as well as the relevant passing test cases.
Bot_{fail}^{1}-Man_{pass}^{+}	Contains one failing generated crash-reproducing test case and the hand-written relevant passing test cases.
Bot_{fail}^{+}-Man_{pass}^{+}	Contains multiple failing generated crash-reproducing test cases and the hand-written relevant passing test cases.
Bot_{fail}^{1}-Evo_{pass}^{+}	Contains one failing crash-reproducing test case and the automatically generated passing unit test cases.
Bot_{fail}^{+}-Evo_{pass}^{+}	Contains multiple failing crash-reproducing test cases and the automatically generated passing unit test cases.

4.5 Data analysis

Since BOTSING can only target one frame at the time and can reproduce multiple frames of some stack traces, it means that BOTSING can make a crash-reproducing test case for 98 frames for the 50 crashes under test. Besides that, we want to compare the four similarity coefficients from Table 2.1. As a result, we end up with $98 \times 5 \times 4 = 1960$ data

points. Hereafter, we focus on the statistical analysis used to answer the different research questions, defined in Section 4.1.

For that, we compare the different combinations based on their EXAM scores. To do so, we use the Vargha-Delaney [75] statistics (Â12) to examine the effect size between two combinations. If the value Â12 is lower than 0.5 for a pair of combinations (A,B), then the combination A reduces the EXAM score, and the opposite applies when the value is higher than 0.5. Additionally, to determine the significance of the effect sizes, we use the non-parametric Wilcoxon Rank Sum test, with $\alpha = 0.05$ for the Type I error.

For **RQ.1**, we compare the diagnostic accuracy of the combinations shown in Table 4.1. Therefore we make a pairwise comparison between the different combinations, following the *tournament ranking* procedure, proposed by Pearson *et al.* [63]. The tournament ranking is calculated by comparing the pairwise EXAM scores, awarding 1 point to the winner if it has statistically significantly better (p-*value* ≤ 0.05 and Â12 < 0.5) EXAM score, and ranking the configurations by the number of points.

For **RQ.2**, we repeat this comparison, but we limit ourselves to the test suites that contain at least one failing test case that covers the fault.

For **RQ.3**, we manually investigate the results of the automated crash fault localization approach to identify the potential factors influencing the diagnostic accuracy. For the analysis, we categorized the different combinations into three sets based on their performance in the automated crash fault localization process.

Category I Executions where crash-reproducing test cases generated by BOTSING performed better than the hand-written test cases exposing the bug, *i.e.,* ($Bot_{fail} > Man_{fail}{}^{+}$)

Category II Executions where the performance of crash-reproducing test cases generated by BOTSING performed similar to the the hand-written test cases exposing the bug, *i.e.,* ($Bot_{fail} = Man_{fail}{}^{+}$)

Category III Executions where the hand-written test cases exposing the bug performed better than the crash-reproducing test cases generated by BOTSING, *i.e.,* ($Bot_{fail} < Man_{fail}{}^{+}$)

Chapter 5

Tooling

In this chapter, we describe the components developed for the evaluation. First, we start with a motivation for the approach used to develop the components, followed by a description of each individual component's functionality. The source code of these components is available at Github[1].

5.1 Motivation

In Chapter 4, we described the evaluation methodology to answer our research questions. To perform the defined evaluations, we developed tools to automate the process and to make it easily reproducible.

First, we build a dataset of the different test suites used for the evaluation, consisting of hand-written test cases ($Man_{pass}{}^{+}$, $Man_{fail}{}^{+}$), crash-reproducing test cases ($Bot_{fail}{}^{1}$, $Bot_{fail}{}^{+}$), and automatically generated unit test cases ($Evo_{pass}{}^{+}$).

For the $Man_{pass}{}^{+}$ and $Man_{fail}{}^{+}$ test suites, we extract the test cases from the DEFECTS4J dataset. Secondly, the $Bot_{fail}{}^{1}$ and $Bot_{fail}{}^{+}$ test suites, we generate the test cases using BOTSING with the defined configuration. Lastly, the $Evo_{pass}{}^{+}$ test suite, we generate the test cases using EVOSUITE with the configurations as defined in Section 4.4.

For each of these tasks, we developed a separate component, in order to keep the code understandable as each component has a single responsibility and to simplify the repetition or to redo parts of the evaluation. Thanks to the modular setup, the research can easily be extended with other test suites or another fault localization approach, as not every part has to be remade.

Figure 5.1 depicts an overview of the components used to obtain the test suites dataset. Three components are used for this task: the DEFECTS4J test case extractor ①, the BOTSING test case generator ②, and the EVOSUITE unit test case generator ③.

[1] https://github.com/svenpopping/acfl-replication-package

The DEFECTS4J test case extractor is used to obtain the relevant passing and failing test cases from a DEFECTS4J project. The test suite and the crash report are used to determine the relevant test cases. The BOTSING test case generator is used to generate the single and multiple crash-reproducing test cases that reproduce the given crash report. The EVOSUITE unit test case generator is used to generate the unit test covering the relevant classes from the crash report.

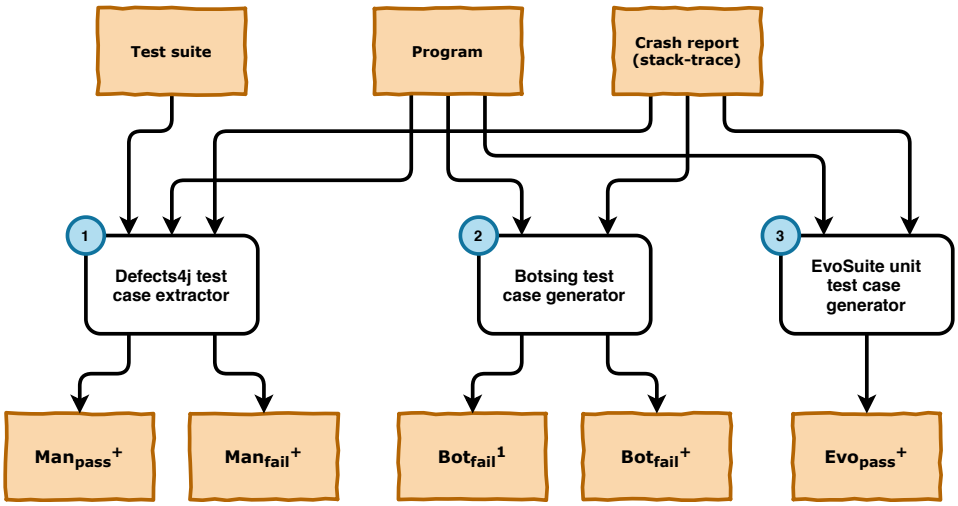


Figure 5.1: Components used to obtain the test suites dataset.

With the dataset complete, we can apply fault localization to each of the five combinations (see Table 4.1). To execute one of the combinations, we use the components as depicted in Figure 5.2. In the first step ①, we run GZOLTAR to apply spectrum-based fault localization, using the different similarity coefficients, to generate the four fault localization reports (*i.e.,* one for each similarity coefficient). In the second step ②, we run post-processing to extract all the required information for the evaluation from the fault localization reports (*e.g.,* exam scores, crash coverage, or program spectra).

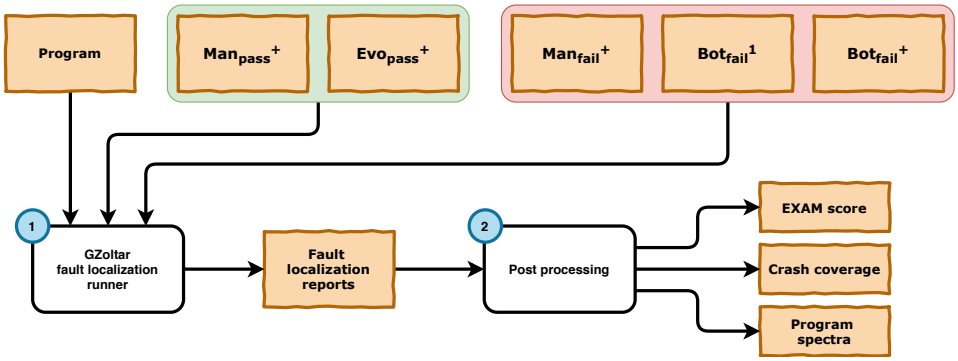


Figure 5.2: Components to obtain the 1960 data points from the dataset.

The execution of the different combinations is orchestrated by a script, which automatically executes all the possibilities to obtain the 1960 data points needed for the data analysis.

5.2 DEFECTS4J test case extractor

The first step in the dataset collection (see Figure 5.1) is to extract the relevant handwritten passing and failing test cases from the DEFECTS4J project under test. For this, we created a DOCKER container called `defects4j-extractor`. The flowchart of the `defects4j-extractor` is shown in Figure 5.3.

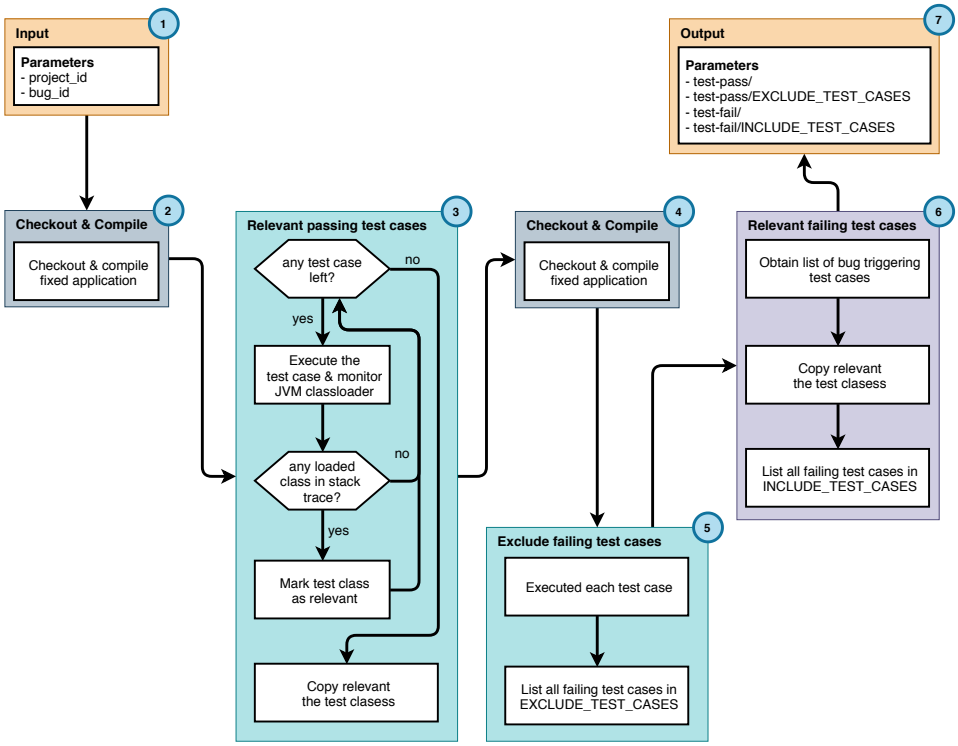


Figure 5.3: Flowchart of the DEFECTS4J test cases extraction process.

The *input* ① of the `defects4j-extractor` is the `project_id` and the `bug_id` of a DEFECTS4J project and the *output* ⑦ is a folder containing all the relevant passing and failing test cases.

In the first *checkout & compile* step ②, the fixed version of the DEFECTS4J project is checked out and compiled using the DEFECTS4J command-line interface (CLI). The fixed version of the project is used to ensure that all the relevant test classes can be executed

without failing in the next phase. Furthermore, it exposes the test cases that fail in the fixed version, which presumably trigger another bug or defect in the project.

In the *relevant passing test cases* step ③, the relevant passing test cases are extracted from the existing test suite. First, the list of all the test cases is generated, using the DEFECTS4J CLI. To determine the relevant test cases, each test case is executed using the `monitor.test` command provided by the DEFECTS4J CLI, which returns the list of classes covered by the test case. If one of the covered classes is in the stack trace corresponding to the crash, the whole test class is listed as relevant, as it is more convenient to copy the whole test class instead of copying a single test case.

After all the test cases are labeled, the relevant passing test classes are copied to the directory `test-pass/`. However, the copied test classes may contain failing test cases. Therefore, in step ④, the faulty version of the project is checked out and compiled. In step ⑤, each test class is executed on the faulty version of the project, and any failing test cases are listed in the `EXCLUDE_TEST_CASES` file. The `EXCLUDE_TEST_CASES` file can later be used to exclude these test cases from the automated fault localization, to ensure that only the passing hand-written test cases are executed.

At last, in step ⑥, the relevant failing test cases are extracted. The DEFECTS4J CLI can export different version-specific properties, including a list of test cases that trigger the bug. The test classes containing these test cases are copied to the `test-fail/` directory. The bug triggering test cases are listed in the `INCLUDE_TEST_CASES` such that only these test cases can be included in the automated fault localization.

To execute the `defects4j-extractor`, there are two required inputs: the `project_id` for a specific project and the `bug_id` for a specific bug of the DEFECTS4J project. An example command to run the `defects4j-extractor` is shown in Listing 5.1, which extracts the relevant passing and failing test cases from `LANG-19b` and stores them in the current directory.

Listing 5.1: Example command to run the `defect4j-extractor` container

```
0 docker run
1   -e PROJECT_ID=Lang
2   -e BUG_ID=9
3   -v $(pwd)/tests-pass:/opt/runner/results/tests-pass
4   -v $(pwd)/tests-fail:/opt/runner/results/tests-fail
5   defect4j-extractor:latest
```

After the successful execution of the `defects4j-extractor`, the results are stored in the following files and directories:

`tests-pass/` containing test classes with at least one relevant passing test case.

`tests-pass/EXCLUDE_TEST_CASES` containing a line for each failing test cases in the format `className#testMethod`. The test cases in this file should be excluded from fault localization to ensure only passing test cases are executed.

`tests-fail/` containing test classes with at least one relevant failing test case.

`tests-fail/INCLUDE_TEST_CASES` containing a line for each bug triggering test cases in the format `className#testMethod`. The test cases in this file should be included in fault localization to ensure that the bug triggering test cases are executed.

5.3 BOTSING test case generator

The purpose of this component is to automate the crash reproduction process for the projects in the DEFECTS4J dataset. The component generates a single crash-reproducing test case or multiple crash-reproducing test cases for a given DEFECTS4J project and a target frame. For this component, we created a DOCKER container called `botsing-generator`, the flowchart of the container is shown in Figure 5.4.

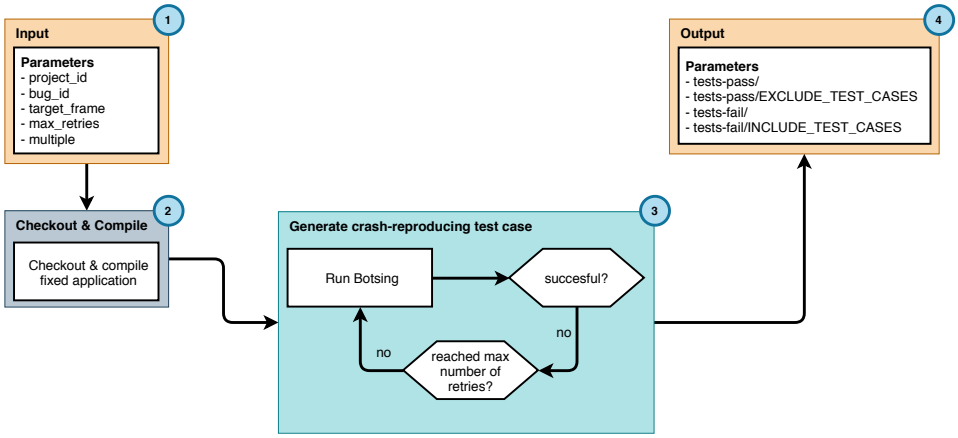


Figure 5.4: Flowchart of the BOTSING test cases generator.

The *input* ① of the `botsing-generator` is the `project_id` and the `bug_id` of a DEFECTS4J project. Together with the `target_frame`, the `max_retries`, indicating the maximum number of retries if botsing fails to generate a test case the first time, and the flag `multiple`, indicating whether or not multiple crash-reproducing test cases should be generated.

The *output* ④ is a folder containing the crash-reproducing test suite and a folder containing the logs of the latest BOTSING execution.

In the *checkout & compile* step ②, the faulty version of the DEFECTS4J project is checked out and compiled using the DEFECTS4J CLI. The faulty version is used to enable BOTSING to generate the crash-reproducing test cases in the next step.

In the *generate crash-reproducing test case* ③ step, BOTSING generates a test class containing one or multiple test cases depending on the sign of the `multiple` flag.

Since BOTSING is based on a meta-heuristic optimization algorithm relying on randomness, it is possible that in the one run, BOTSING can not generate a solution. Still, in another run, BOTSING can find a solution. Therefore, if no solution has been found (determined by the fitness value in the logs), BOTSING is rerun until the maximum number of retries has been reached.

Furthermore, BOTSING relies on the test case minimization algorithm of EVOSUITE. Sometimes it happens that BOTSING generates a crash-reproducing test case, but the test case is minimized to an empty test case. In such a case, BOTSING is also rerun until the maximum number of retries has been reached.

To run the `botsing-generator`, the inputs, as described above, must be provided through the environment variables of the DOCKER container. An example command to run the `botsing-generator` is shown in Listing 5.2, which generates multiple crash-reproducing test cases for `LANG-19b` and target frame 4. The results are stored the results in the current directory.

Listing 5.2: Example command to run the `botsing-generator` container

```
0 docker run
1   -e PROJECT_ID=Lang
2   -e BUG_ID=9
3   -e TARGET_FRAME=4
4   -e MAX_RETRIES=15
5   -e MULTIPLE=true
6   -v $(pwd)/tests-botsing:/opt/runner/results
7   botsing-generator:latest
```

After the successful execution of the `botsing-generator`, the results are stored in the following directories:

`tests-botsing/` containing the crash-reproducing test class with one or multiple test cases, depending on the mode (single or multiple). For each test class, there is a file containing the test classes itself (denoted by `className_ESTest`), and a file containing the EVOSUITE scaffolding (denoted by `className_scaffolding_ESTest`).

`tests-botsing/logs` containing the logs of the latest BOTSING execution.

5.4 EVOSUITE unit test case generator

The purpose of this component is to automate the generation of unit test cases for the projects in the DEFECTS4J dataset. The component generates a test class for each distinct class in the stack trace corresponding to the crash. For this component, we created a DOCKER container called `evosuite-generator`. The flowchart of the container is shown in Figure 5.5.

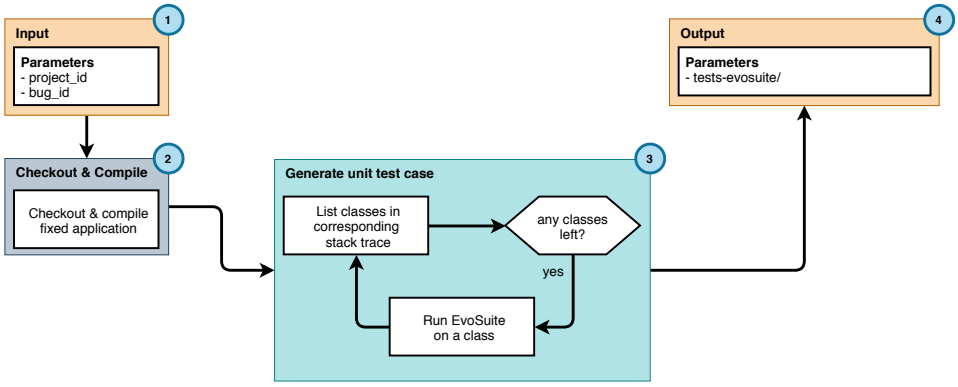


Figure 5.5: Flowchart of the EvoSuite unit test cases generator.

The *input* ① of the `evosuite-generator` is the `project_id` and the `bug_id` of a Defects4J project and the *output* ④ is a folder containing a test class for each distinct class in the corresponding stack trace.

In the *checkout & compile* step ②, the faulty version of the Defects4J project is checked out and compiled using the Defects4J CLI. The faulty version is used, as in practice, this will be the only version available.

In the *generate unit test case* ③ step, a list of relevant classes is constructed for the Defects4J bug. A class is considered relevant if the class is in one of the frames in the stack trace and is part of the Defects4J project. Afterward, EvoSuite is executed to generate a unit test class for each of the classes in the list. EvoSuite is executed using its default settings, as provided by the documentation.

To run the `evosuite-generator`, the inputs, as described above, must be provided through the environment variables of the Docker container. An example command to run the `evosuite-generator` is shown in Listing 5.3, which generates the unit test classes for `LANG-19b` and stores the results in the current directory.

Listing 5.3: Example command to run the `evosuite-generator` container

```
0 docker run
1   -e PROJECT_ID=Lang
2   -e BUG_ID=9
3   -v $(pwd)/tests-evosuite:/opt/runner/results
4   evosuite-generator:latest
```

After the successful execution of the `evosuite-generator`, the results are stored in the following directories:

`tests-evosuite/` containing the unit test classes relevant for the given Defects4J project. For each test class, a file containing the test classes itself (denoted by `className_-`

`ESTest`), and a file containing the EVOSUITE scaffolding (denoted by `className_-scaffolding_ESTest`).

5.5 Test suites post-processing

As mentioned in Section 4.4, the generated test classes by BOTSING and EVOSUITE are not immediately compatible with GZOLTAR, the fault localization framework.

Therefore, we have to prepare the test classes for the fault localization part of the evaluation. This implies that, for the BOTSING test suites, the scaffolding and that any try-catch blocks will be removed from the test suites. Furthermore, the name of the test classes will be renamed such that they include the word BOTSING. This action is taken because the name of the test classes generated by BOTSING and EVOSUITE are identical for the same class, resulting in the fact that the one overrides the other test class when inserted into GZOLTAR.

For the EVOSUITE unit test classes holds that the scaffolding and the test cases throwing an undeclared exception will be removed. Also, the name of the test classes will be renamed, such that the filename includes the word EVOSUITE.

5.6 GZOLTAR fault localization runner

The purpose of this component is to apply fault localization to the different combinations of test suites, as discussed in Section 4.4. As mentioned earlier, we rely on the fault localization framework called GZOLTAR [21].

Figure 5.6 depicts the workflow of the GZOLTAR framework. The workflow consists of four steps, which must be carried out one after the other.

The first step ① is to list all the test cases in the project and determine its type (either JUnit or TestNG). After that, the GZOLTAR instrumentation is added to the project, in step ②. The instrumentation, comprising several APIs, enables GZOLTAR to track the coverage of the test cases throughout the project. Using the list of test cases from step ① and the instrumentation from step ②, in step ③ all test cases are executed in isolation so that the program spectrum can be constructed and saved in a serialized object. In step ④, the fault localization report is constructed using one of the implemented similarity coefficients. Since the program spectrum is saved, it is possible to repeat step ④ using different similarity coefficients, without having to go through steps ①, ②, and ③ again.

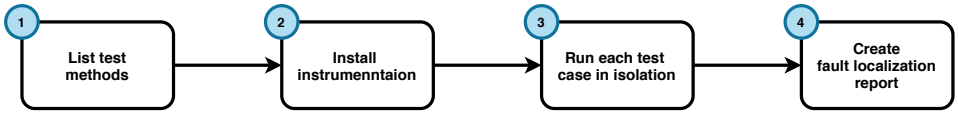


Figure 5.6: Flowchart of the fault localization process using GZOLTAR

After the execution of GZOLTAR, the fault localization reports are stored in the following files:

`<formula>.ranking.csv` containing the ranking of the statements in the project, sorted by their suspiciousness level. `<formula>` is the name of the formula, for example, when using the TARANTULA coefficient, the filename is `tarantula.ranking.csv`.

`matrix.txt` containing the program spectrum matrix, where each row represents a test case and its outcome, and each column represents a statement (1 means that a test case covered a line of code, 0 otherwise).

`tests.csv` a list of all test cases and its outcome, runtime in nanoseconds, and the stack trace (in case of a failing test case).

`spectra.csv` a list of all lines of code executed by a test case. Each row follows the following format: `className#methodName(methodParameters):lineNumber`

To automate the evaluation process, we extended the original workflow and created a DOCKER container around it. For this component, we created a DOCKER container called `gzoltar-runner`, the flowchart of the container is shown in Figure 5.7.

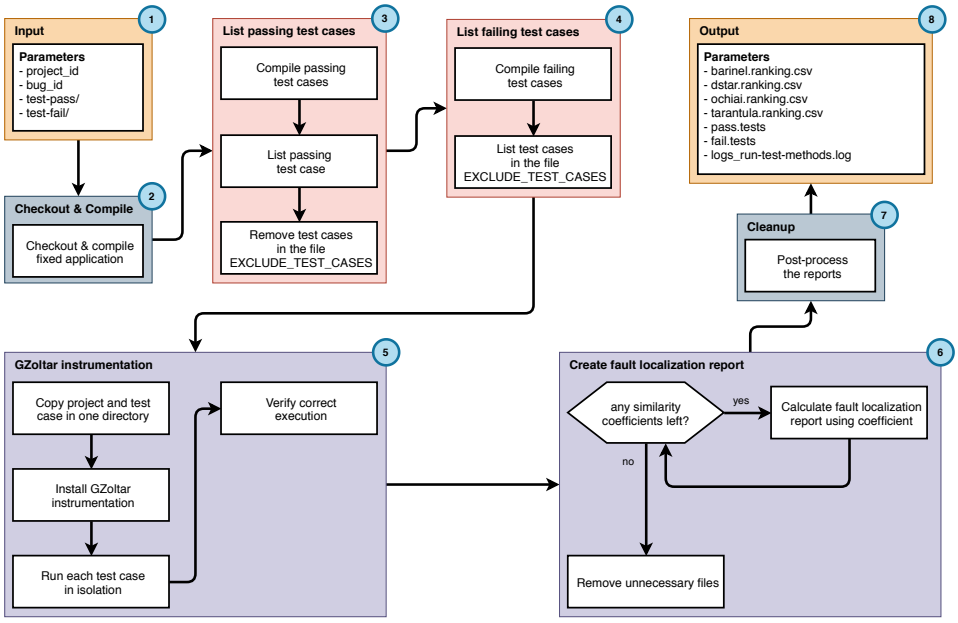


Figure 5.7: Flowchart of the GZOLTAR runner.

The *input* ① is the `project_id` and `bug_id` of a DEFECTS4J project. Together with a directory containing passing test cases (`test-pass/`) and a directory containing failing test cases (`test-fail/`).

In the *checkout & compile* step ②, the faulty version of the DEFECTS4J projects is checked out and compiled using the DEFECTS4J CLI. The faulty version is used, as in practice, this will be the only version available.

Using the faulty version and the directory containing the passing test cases, a list of the passing test cases is generated, in step ③. Similar to the original GZOLTAR workflow, the list is created by compiling the test cases and using the `listTestMethods` command provided in the GZOLTAR CLI. However, only the test cases in the directory `test-pass/` are indexed, and the test cases in the `EXCLUDE_TEST_CASES` are removed from this list.

After that, in step ④, the failing test cases are compiled, and the test cases listed in the `INCLUDE_TEST_CASES` file are added to the list of passing test cases. So there is a list of all the test cases used in the fault localization process.

Then step ⑤, in which, first, the compiled project and the compiled test cases are collected in one directory. Second, the GZOLTAR instrumentation is added to all the classes in the directory, followed by the execution of each test case in isolation (as in the original workflow). In addition to the original workflow, after the execution of the test cases, the logs are checked to ensure everything went as expected (*i.e.,* every passing test case has passed, and every failing test case has failed).

Using the serialized object from step ⑤, the fault localization reports are generated for the four similarity coefficients in Table 2.1, in step ⑥. After the generation of the fault localization reports, some unnecessary files are removed, including the serialized object (which can become quite large).

In the second to last step ⑦, the post-processing of the fault localization reports is executed. As the project and test cases are combined in one directory, the reports also include the test statements (*i.e.,* test case lines of code). The test statements are removed from the reports because they have nothing to do with the crash.

The *output* ⑧, are the fault localization reports (one for each similarity coefficient) and some statistical information about the fault localization process.

To run the `gzoltar-runner`, the inputs, as described above, must be provided through the environment variables of the DOCKER container. An example command to run the `gzoltar-runner` is shown in Listing 5.4, which drafts the fault localizations reports for `LANG-19b`, using the passing test cases from the directory `<path-to-passing-tests>` and the failing test cases from the directory `<path-to-failing-tests>`, and stores the results in the current directory.

Listing 5.4: Example command to run the `gzoltar-runner` container

```
0 docker run
1   -e PROJECT_ID=Lang
2   -e BUG_ID=9
3   -v <path-to-passing-tests>:/opt/runner/tests-pass
4   -v <path-to-failing-tests>:/opt/runner/tests-fail
5   -v $(pwd)/results:/opt/runner/results/sfl/txt
6   gzoltar-runner:latest
```

After the successful execution of the `gzoltar-runner`, the results are stored in the files from the original workflow and in the following files:

`barinel.ranking.csv`, `dstar.ranking.csv`, `ochiai.ranking.csv`,`tarantula.ranking.csv` containing the ranking of the statements in the project, sorted by their suspiciousness level, calculated using the BARINEL, DSTAR, OCHIAI and TARANTULA coefficient, respectively.

`gzoltar.tests` containing a list of all the test cases used in the fault localization process.

`logs_run-test-methods.log` containing the outcome of all the test cases. If a test case fails, the logs contain also the produced stack trace.

5.7 Fault localization post-processing

After all the results have been collected, we applied post-processing on the dataset. To obtain the EXAM scores and to double-verify whether every combination has been executed correctly.

For the EXAM scores, we first constructed the ranking of the statements, based on the extended evaluation methodology described in Section 4.2. We also sanitized the ranking by removing non-executable statements and the statements with a suspiciousness level of zero. With the ranking and the defective statements from the fault localization dataset (by Pearson *et al.* [63]), we computed the EXAM scores for each of the combination of test suites.

To determine whether or not the crash-reproducing test case has covered the defective statement (and therefore covered the fault), we looked at the EXAM score. When the EXAM score could not be calculated, it meant that the defective statement had not been executed. We can say this with certainty because the ranking only contains only the statements that have been executed by at least one failing test case (otherwise, the suspiciousness level would be higher than zero).

Using the logs provided by each component, we could verify that every combination has been executed correctly. We also manually checked that every component had been correctly executed because, in some cases, the failing test case failed (checked in the `gzoltar-runner`), but it is due to another reason than the expected one (*e.g.,* a missing library or the wrong system timezone).

Chapter 6

Results

In this chapter, we discuss the results of the evaluation, following the methodology described in Chapter 4 and using the tooling described in Chapter 5.

Before answering the defined research questions by looking into the general scenario, the best-case scenario, and the influencing factors, we have to look into the crash-reproducing test cases generated by BOTSING. As described in Section 4.5, we used in total 98 frames from 50 DEFECTS4J crashes.

Due to the fact that BOTSING's algorithm relies on randomness, BOTSING was not able to generate non-empty crash-reproducing test suites for all 98 frames in the dataset. Table 6.1 reports how many test suites could be generated out of the 98 target frames from the 50 DEFECTS4J projects under test. For instance, single-BOTSING ($Bot_{fail}{}^{1}$) generated 83 non-empty test suites (*i.e.,* containing one crash-reproducing test case) out of the 98 target frames, and multiple-BOTSING ($Bot_{fail}{}^{+}$) produced 85 non-empty test suites (*i.e.,* containing at least one crash-reproducing test case). After the removal of the flaky test cases (due to the removal of the EVOSUITE scaffolding), we ended up with 80 non-empty $Bot_{fail}{}^{1}$ test suites and 84 non-empty $Bot_{fail}{}^{+}$ test suites.

Out of the 80 non-empty $Bot_{fail}{}^{1}$ test suites, 51 contain a crash-reproducing test case that executes the defective statements. For the 84 non-empty $Bot_{fail}{}^{+}$ test suites holds that 65 contain a crash-reproducing test case that executes the defective statements.

Table 6.1: Number of crashes for which BOTSING could generate a crash-reproducing test suite contain one ($Bot_{fail}{}^{1}$) or multiple ($Bot_{fail}{}^{+}$) crash-reproducing test cases

	$\boldsymbol{Bot_{fail}{}^{1}}$	$\boldsymbol{Bot_{fail}{}^{+}}$
Crashes	50	50
Total number of target frames	98	98
Non-empty test suites generated by BOTSING	83	85
of which contain flaky crash-reproducing tests	3	1
of which execute the defective statement	51	65

Interestingly, the number of non-empty test suites for $Bot_{fail}{}^{1}$ and $Bot_{fail}{}^{+}$ are not equal, which is counter-intuitive. One would expect that if BOTSING can generate a non-empty test suite with multiple crash-reproducing test cases, it would also be possible to generate a non-empty test suite with a single crash-reproducing test case.

However, it most likely has to do with the fact that the multiple-BOTSING does twice as much generation and evaluation within the same search budget of 180 seconds. We have not been able to explain the reason for this adequately, as single- and multiple-BOTSING are executed using the same amount of resources (2 CPUs and 5 GB memory, enforced by the DOCKER runtime options).

6.1 General scenario

After removing the flaky test cases, we have 80 non-empty test suites, common to $Bot_{fail}{}^{1}$ and $Bot_{fail}{}^{+}$, that can be used to fairly compare the results for all the configurations and hence to answer **RQ1**.

RQ1: *How does automated crash fault localization perform in comparison to manual crash fault localization in terms of accuracy when* BOTSING *generates one or more crash-reproducing test cases?*

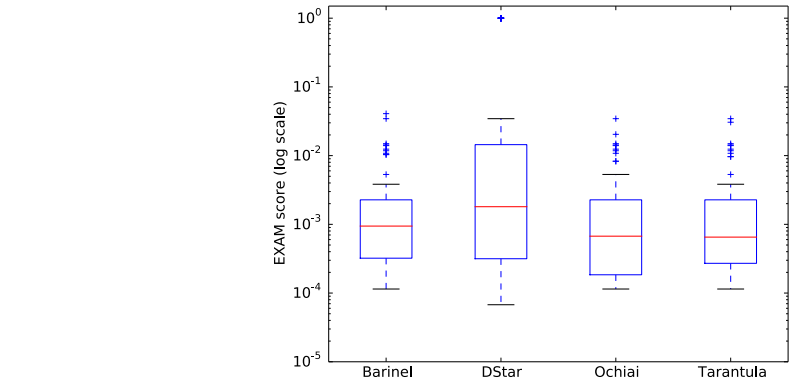
As described in Section 4.2, the evaluation metric used to determine the diagnostics accuracy of a fault localization approaches is the EXAM score. Recall that a lower EXAM score represents better performance because it calculates the percentage of statements before examining the faulty statement.

Figure 6.1 depicts the boxplots for EXAM scores of the 80 test suites for the different configurations and coefficients. The means of the EXAM scores for each of the combinations of configurations and coefficients are reported in Table 6.2.

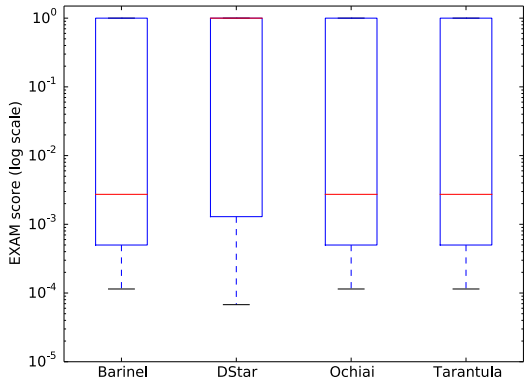
The hand-written test cases ($Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$) achieve the best EXAM score for each of the similarity coefficients compared to the EXAM scores of the automated crash fault localization configurations (see Figure 6.1a and Table 6.2). Overall, the hand-written test cases ($Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$) with the OCHIAI coefficient achieves the best EXAM score (see Table 6.2).

Table 6.2: The mean EXAM of the 80 different test suite combinations, common to $Bot_{fail}{}^{1}$and $Bot_{fail}{}^{+}$.

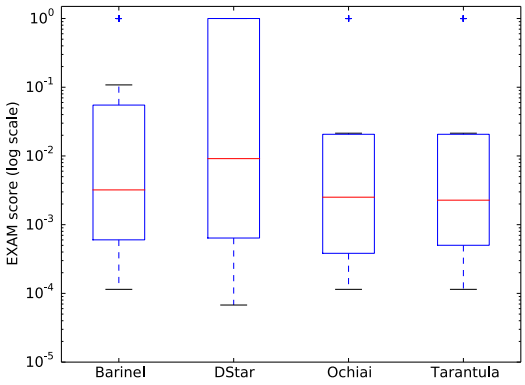
Configurations	BARINEL	DSTAR	OCHIAI	TARANTULA
$Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$	0.0031	0.2149	0.0027	0.0029
$Bot_{fail}{}^{1}$-$Man_{pass}{}^{+}$	0.3647	0.5395	0.3647	0.3647
$Bot_{fail}{}^{+}$-$Man_{pass}{}^{+}$	0.2418	0.3777	0.2402	0.2399
$Bot_{fail}{}^{1}$-$Evo_{pass}{}^{+}$	0.3647	0.5766	0.3647	0.3647
$Bot_{fail}{}^{+}$-$Evo_{pass}{}^{+}$	0.2422	0.4145	0.2399	0.2403



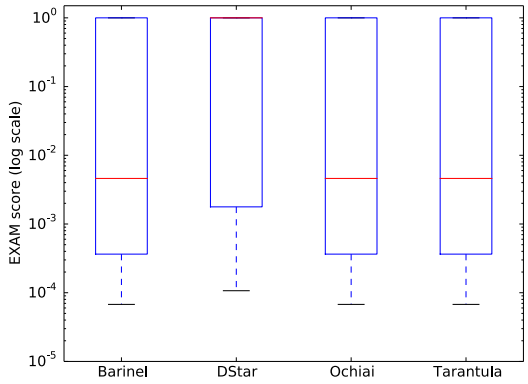
(a) $Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$



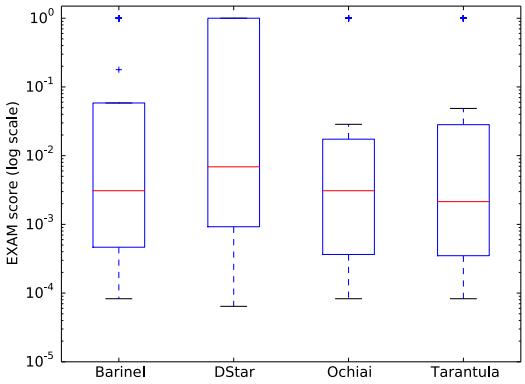
(b) $Bot_{fail}{}^{1}$-$Man_{pass}{}^{+}$



(c) $Bot_{fail}{}^{+}$-$Man_{pass}{}^{+}$



(d) $Bot_{fail}{}^{1}$-$Evo_{pass}{}^{+}$



(e) $Bot_{fail}{}^{+}$-$Evo_{pass}{}^{+}$

Figure 6.1: Representation of the EXAM score for the 80 test suites, common to $Bot_{fail}{}^{1}$ and $Bot_{fail}{}^{+}$, with a failing crash-reproducing test case (general scenario)

Interestingly, the $Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$ configuration with the DSTAR coefficient has quite a poor performance compared to the other similarity coefficients. This decline in performance is caused by the fact that of the 80 executions, 17 achieved an EXAM score of 1.0.

After a manual analysis, we found that this is due to a problem in the way the DSTAR coefficient uses the program spectra, as explained in Section 2.1. That means that the sum of failing test cases not covering the fault and the number of passing test cases covering the fault is zero, making the DSTAR coefficient unsuitable for the test cases used. This problem occurred not only in the hand-written test suite but in all test suites in our dataset. Consequently, the DSTAR coefficient performs the worst of all the similarity coefficients for each of the configurations. We decided to keep the EXAM scores of 1.0 as a penalty for such a case.

From Figure 6.1 and Table 6.2, it can be observed that, for the $Bot_{fail}{}^{1}$-$Man_{pass}{}^{+}$ and the $Bot_{fail}{}^{1}$-$Evo_{pass}{}^{+}$ configuration, the best EXAM score is obtained using the BARINEL, the TARANTULA or the OCHIAI coefficient (see Figures 6.1b and 6.1d). The best EXAM score for the $Bot_{fail}{}^{+}$-$Man_{pass}{}^{+}$ configuration is achieved by the TARANTULA coefficient (see Figure 6.1c), but the differences with the OCHIAI and the BARINEL coefficients are small (see Table 6.2). $Bot_{fail}{}^{+}$-$Evo_{pass}{}^{+}$ has almost the same performance compared to $Bot_{fail}{}^{+}$-$Man_{pass}{}^{+}$, only achieved by using the OCHIAI coefficient (see Figure 6.1e), but the difference with the TARANTULA coefficient is small.

Figures 6.1b, 6.1c, 6.1d, and 6.1e and Table 6.2 indicate that the automated configurations using multiple crash-reproducing test cases ($Bot_{fail}{}^{+}$) perform better than the configurations using a single crash-reproducing test case ($Bot_{fail}{}^{1}$). Besides that, for all the automated crash fault localization configurations holds that the similarity coefficients OCHIAI and TARANTULA give the best result, with BARINEL a close second.

6.1.1 Pairwise ranking

Table 6.3 shows the results of the pairwise tournament ranking of the different combinations of configurations and coefficients, as described in Section 4.5, for the general scenario. The rank is created by comparing the pairwise EXAM scores, awarding 1 point to the winner if it performs statistically significantly better (p-value ≤ 0.05 and $\hat{A}12 < 0.5$).

From Table 6.3, we can conceive that the hand-written test suite combinations ($Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$) win to a large extent. However, the manual combination with DSTAR is better in only half of the cases compared to the other manual combinations.

Furthermore, for the automated crash fault localization approach, we can observe that using multiple crash-reproducing test cases ($Bot_{fail}{}^{+}$) works better than using a single crash-reproducing test case ($Bot_{fail}{}^{1}$). As the $Bot_{fail}{}^{+}$ configurations win in 25 of the executions and the $Bot_{fail}{}^{1}$ only win in 7 executions. This confirms our observation from the boxplots in Figure 6.1 and the results from Table 6.2.

Table 6.3: Ranking of the all 80 different test suite combinations of *Configuration* and *Coefficient*. *Better than* denotes the number of times for which the combination is significantly better than another combinations. The *avg. Â12* gives the average effect size of the configuration.

	Configuration	Coefficient	Better than	avg. Â12
1	$Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$	OCHIAI	16	0.30
2	$Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$	TARANTULA	16	0.30
3	$Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$	BARINEL	16	0.31
4	$Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$	DSTAR	8	0.37
5	$Bot_{fail}{}^{+}$-$Evo_{pass}{}^{+}$	OCHIAI	4	0.37
6	$Bot_{fail}{}^{+}$-$Evo_{pass}{}^{+}$	TARANTULA	4	0.37
7	$Bot_{fail}{}^{+}$-$Man_{pass}{}^{+}$	OCHIAI	3	0.36
8	$Bot_{fail}{}^{+}$-$Man_{pass}{}^{+}$	TARANTULA	3	0.36
9	$Bot_{fail}{}^{+}$-$Evo_{pass}{}^{+}$	BARINEL	2	0.36
10	$Bot_{fail}{}^{+}$-$Man_{pass}{}^{+}$	BARINEL	2	0.37
11	$Bot_{fail}{}^{1}$-$Evo_{pass}{}^{+}$	BARINEL	2	0.39
12	$Bot_{fail}{}^{1}$-$Evo_{pass}{}^{+}$	OCHIAI	2	0.39
13	$Bot_{fail}{}^{1}$-$Evo_{pass}{}^{+}$	TARANTULA	2	0.39
14	$Bot_{fail}{}^{+}$-$Man_{pass}{}^{+}$	BARINEL	2	0.40
15	$Bot_{fail}{}^{+}$-$Man_{pass}{}^{+}$	OCHIAI	2	0.40
16	$Bot_{fail}{}^{+}$-$Man_{pass}{}^{+}$	TARANTULA	2	0.40
17	$Bot_{fail}{}^{+}$-$Man_{pass}{}^{+}$	DSTAR	1	0.41

Also, the observation for the similarity coefficients holds, as OCHIAI and TARANTULA tie in the best performing coefficient for the automated crash fault localization approach. Followed by the BARINEL, which wins at least once for each of the combinations, and at last, DSTAR has only one win for the $Bot_{fail}{}^{+}$-$Man_{pass}{}^{+}$ combination.

Our results confirm that there is no significant difference between the four similarity coefficients when applied to real-world faults using hand-written test cases, as shown by Pearson *et al.* [63]. None of the similarity coefficient using hand-written test suite combination ($Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$), wins against the other hand-written test suite combinations (*i.e.,* p-value > 0.05).

6.1.2 Summary

In general, the hand-written test cases ($Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$) perform better than the automated crash fault localization configurations. One of the reasons is that a generated crash-reproducing test case gives no guarantee that the test case actually triggers the underlying fault. For example, when the bug is located in frames higher than the reproduced frame.

For the automated crash fault localization configurations, the similarity coefficients OCHIAI or TARANTULA in combination with a test suite containing multiple crash-reproducing test cases ($Bot_{fail}{}^{+}$) gives the best results.

6.2 Best-case scenario

As recorded in Table 6.1, BOTSING could generate 51 (out of 80) test suites for $Bot_{fail}{}^{1}$ and 65 (out of 84) test suites for $Bot_{fail}{}^{+}$ for different target frames. For these test suites, at least one crash-reproducing test case covers one of the defective statements. For answering **RQ.2**, we focus on these 51 test suites, common to $Bot_{fail}{}^{1}$and $Bot_{fail}{}^{+}$.

RQ2: *How does automated crash fault localization perform in comparison to manual crash fault localization in terms of accuracy when* BOTSING *generates one or more crash-reproducing test cases covering the fault?*

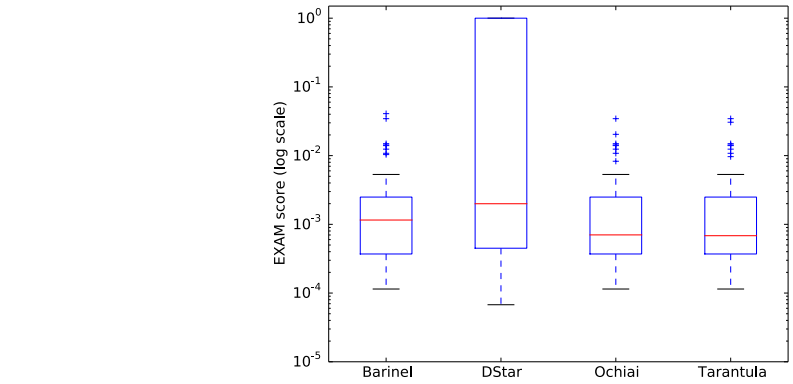
Figure 6.2 depicts the EXAM score of the 51 test suites for the different configurations and coefficient. One thing that springs to mind is the performance of the automated crash fault localization configurations, which improved compared to the performance in the general scenario (see Figure 6.1). As suspected, a part of the poor performance of automated crash fault localization configurations can be explained by the fact that there is no guarantee that the crash-reproducing test cases cover the fault.

On average, OCHIAI is the best performing similarity coefficient for the $Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$ configurations, as shown in Table 6.4. For the $Bot_{fail}{}^{+}$-$Man_{pass}{}^{+}$ configurations, using the TARANTULA or the OCHIAI coefficient, achieve a better mean EXAM score. The best performance is achieved using the $Bot_{fail}{}^{+}$-$Evo_{pass}{}^{+}$ configuration in combination with the OCHIAI similarity coefficient. Similar to the general scenario, for the $Bot_{fail}{}^{1}$-$Man_{pass}{}^{+}$ and the $Bot_{fail}{}^{1}$-$Evo_{pass}{}^{+}$ configurations holds that the best EXAM score is achieved by either using the BARINEL, the OCHIAI or the TARANTULA coefficient.

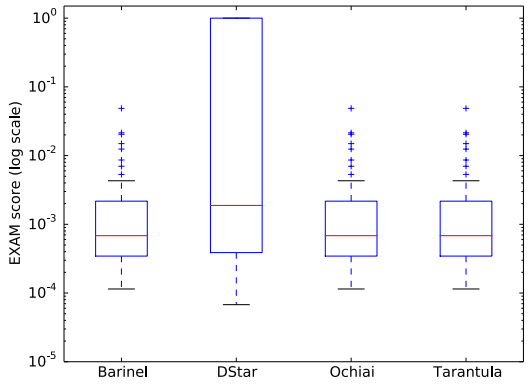
Table 6.4: The average EXAM of the 51 different non-empty test suite combinations.

Configurations	BARINEL	DSTAR	OCHIAI	TARANTULA
$Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$	0.0039	0.2776	0.0034	0.0036
$Bot_{fail}{}^{1}$-$Man_{pass}{}^{+}$	0.0035	0.2776	0.0035	0.0035
$Bot_{fail}{}^{+}$-$Man_{pass}{}^{+}$	0.0064	0.2188	0.0032	0.0031
$Bot_{fail}{}^{1}$-$Evo_{pass}{}^{+}$	0.0035	0.3359	0.0035	0.0035
$Bot_{fail}{}^{+}$-$Evo_{pass}{}^{+}$	0.0071	0.2768	0.0030	0.0038

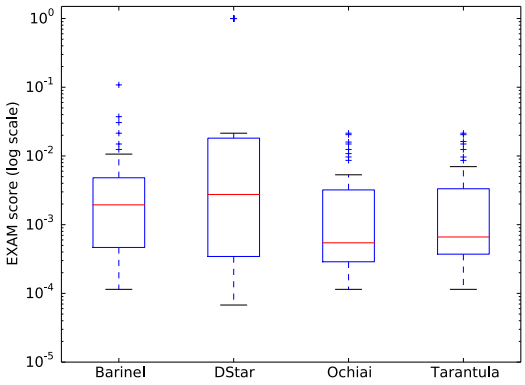
The latter can be explained by looking into the formulas of the three similarity coefficients. For both single crash-reproducing configurations ($Bot_{fail}{}^{1}$), it holds that $N_F = 1$ as there is only the failing crash-reproducing test case. Using the assumption defined in Section 2.1 (for a defective statement $N_{CF} > 0$), we can conclude that for a defective statement, in the single crash-reproducing configurations, it must be that $N_{CF} = 1$ and $N_{UF} = 0$. When we



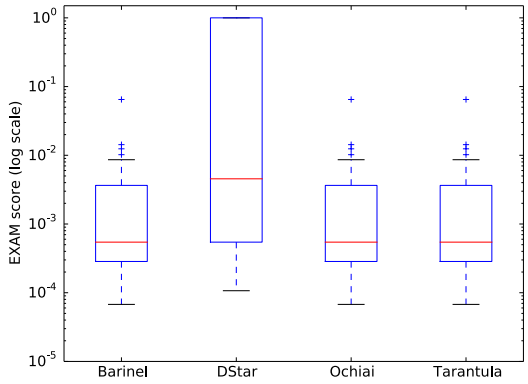
(a) $Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$



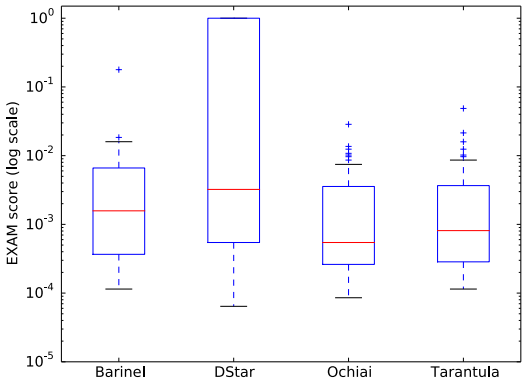
(b) $Bot_{fail}{}^{1}$-$Man_{pass}{}^{+}$



(c) $Bot_{fail}{}^{+}$-$Man_{pass}{}^{+}$



(d) $Bot_{fail}{}^{1}$-$Evo_{pass}{}^{+}$



(e) $Bot_{fail}{}^{+}$-$Evo_{pass}{}^{+}$

Figure 6.2: EXAM score for the 51 test suites with a failing crash-reproducing test case triggering the bug (best-case scenario)

apply the values to the three similarity coefficients, we get that:

$$\text{OCHIAI} = \frac{N_{CF}}{\sqrt{N_F \times (N_{CF} + N_{CS})}} = \frac{1}{\sqrt{1 \times (1 + N_{CS})}} = \frac{1}{\sqrt{1 + N_{CS}}}$$

$$\text{BARINEL} = 1 - \frac{N_{CS}}{N_{CS} + N_{CF}} = 1 - \frac{N_{CS}}{N_{CS} + 1} = \frac{1}{1 + N_{CS}}$$

$$\text{TARANTULA} = \frac{N_{CF}/N_F}{N_{CF}/N_F + N_{CS}/N_S} = \frac{1/1}{1/1 + N_{CS}/N_S} = \frac{1}{1 + N_{CS}/N_S}$$

From this, we can deduce that these three similarity coefficients are mathematically similar and show the same behavior (*i.e.,* all the formulas are only dependent on N_{CS} given that N_S is fixed for all the defective statements) when applied to a program containing a single failing test case. This results in the fact that the ranking of the statements for all these coefficients is the same, and therefore, the EXAM score is the same.

Comparing the different similarity coefficient from Figures 6.2b and 6.2c, we see that using $Man_{pass}{}^{+}$ in combination with $Bot_{fail}{}^{1}$ or $Bot_{fail}{}^{+}$ perform statistically better ($p\text{-}value \leq 0.05$) when using the OCHIAI ($\hat{\text{A}}12 = 0.331$) or TARANTULA ($\hat{\text{A}}12 = 0.342$) coefficient than the DSTAR coefficient.

The BARINEL coefficient, in combination with the single crash-reproducing configurations ($Bot_{fail}{}^{1}$) ($\hat{\text{A}}12 = 0.331$), also achieves a statistically significant difference towards the same configurations using the DSTAR coefficient.

When only considering the non-empty test suites that have the guarantee that they cover the fault, we see that no combination of similarity coefficients outperforms the others, except for the DSTAR coefficient.

Also, we do not discern any statistical difference (*i.e., p-values* > 0.05)) when a given similarity coefficient is used in combination with the hand-written test cases configuration ($Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$) or any other automated crash-reproducing configurations.

6.2.1 Pairwise ranking

Table 6.5 shows the results of the pairwise tournament ranking of the different combinations of configurations and coefficients, as described in Section 4.5, for the best-case scenario.

From Table 6.5, we can observe that the DSTAR coefficient is performing the worst of all the coefficients, as it does not perform better than any of the other combinations.

Furthermore, we can see that no combination significantly outperforms the others (except DSTAR), as all the combinations have the same number of wins. Only the performance of the $Bot_{fail}{}^{+}$-$Man_{pass}{}^{+}$ and $Bot_{fail}{}^{+}$-$Evo_{pass}{}^{+}$ with BARINEL is a bit lower (1 victory vs. 5 victories), but this difference is not statistically significant.

Table 6.5: Ranking of the 51 different test suite combinations of *Configuration* and *Coefficient*. *Better than* denotes the number of times for which the combinations is significantly better than another combinations. The *avg. Â12* gives the average effect size of the configuration.

	Configuration	Coefficient	Better than	avg. Â12
1	$Bot_{fail}{}^{1}$-$Evo_{pass}{}^{+}$	BARINEL	5	0.32
2	$Bot_{fail}{}^{1}$-$Evo_{pass}{}^{+}$	OCHIAI	5	0.32
3	$Bot_{fail}{}^{1}$-$Evo_{pass}{}^{+}$	TARANTULA	5	0.32
4	$Bot_{fail}{}^{+}$-$Evo_{pass}{}^{+}$	OCHIAI	5	0.33
5	$Bot_{fail}{}^{+}$-$Man_{pass}{}^{+}$	OCHIAI	5	0.34
6	$Bot_{fail}{}^{1}$-$Man_{pass}{}^{+}$	BARINEL	5	0.35
7	$Bot_{fail}{}^{1}$-$Man_{pass}{}^{+}$	OCHIAI	5	0.35
8	$Bot_{fail}{}^{1}$-$Man_{pass}{}^{+}$	TARANTULA	5	0.35
9	$Bot_{fail}{}^{+}$-$Evo_{pass}{}^{+}$	TARANTULA	5	0.35
10	$Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$	OCHIAI	5	0.35
11	$Bot_{fail}{}^{+}$-$Man_{pass}{}^{+}$	TARANTULA	5	0.36
12	$Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$	TARANTULA	5	0.36
13	$Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$	BARINEL	5	0.37
14	$Bot_{fail}{}^{+}$-$Evo_{pass}{}^{+}$	BARINEL	1	0.37
15	$Bot_{fail}{}^{+}$-$Man_{pass}{}^{+}$	BARINEL	1	0.38

6.2.2 Summary

In the best-case scenario, we observe no statistically significant difference between the automated crash fault localization configurations ($Bot_{fail}{}^{1}$-$Man_{pass}{}^{+}$, $Bot_{fail}{}^{+}$-$Man_{pass}{}^{+}$, $Bot_{fail}{}^{1}$-$Evo_{pass}{}^{+}$, $Bot_{fail}{}^{+}$-$Evo_{pass}{}^{+}$) and the hand-written test cases ($Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$), which was there in the general scenario. No combination of configuration and similarity coefficient beats the other combinations.

Our findings show that using the OCHIAI or the TARANTULA coefficients results in the best EXAM score. However, when using a single crash-reproducing test case ($Bot_{fail}{}^{1}$), it does not matter which of the coefficients is used (except DSTAR), because in this case, the coefficients are mathematically similar and only dependent on N_{CS}.

Finally, the results suggest that the DSTAR coefficient might not be suitable for the usage in automated crash fault localization.

6.3 Influencing factors

Finally to answer **RQ3**, we identified the factors that influence the diagnostic accuracy of the automated crash fault localization approach, we categorized the executions in $Bot_{fail} > Man_{fail}{}^{+}$, $Bot_{fail} = Man_{fail}{}^{+}$, and $Bot_{fail} < Man_{fail}{}^{+}$, as described in Section 4.5.

RQ3: *Which factors influence the performance and applicability of automated crash fault localization?*

For each category, we manually investigated potential factors influencing spectrum-based fault localization accuracy by identifying the strengths and weaknesses of the automatically generated crash-reproducing test cases. Hereafter, we provide the identified factor for each category with a representative example.

6.3.1 Purified test cases ($Bot_{fail} > Man_{fail}{}^{+}$)

According to existing literature [7, 65, 87], the number of assert statements per test case influences the diagnostic accuracy of spectrum-based fault localization approaches. In particular, when a test case contains multiple assert statements in which the same method is executed multiple times (*e.g.,* Listing 6.1).

Listing 6.1: An example of non-purified test cases

```
0  public void test00() {
1    assertTrue(isValidPhoneNumber("06-12345678"));
2    assertTrue(isValidPhoneNumber("0031612345678"));
3    assertTrue(isValidPhoneNumber("+31612345678"));
4  }
```

If a test case contains multiple assertions and one of them fails, then spectrum-based fault localization cannot distinguish the failing execution path from the non-failing ones (*e.g.,* when the second assert statement fails). This phenomenon, the whole execution path is labeled as failing, causing a decline in diagnostic accuracy. In their work, Xuan and Monperrus [87] showed that *purified test cases*, *i.e.,* test cases containing one assert statement executing one method can positively influence spectrum-based fault localization performance (*e.g.,* Listing 6.2) because the spectrum-based fault localization approach can better distinguish the execution paths (failing or passing). Also, when using purified test cases, all the assertions will be executed instead of the assertions until one fails, which provides more information.

Listing 6.2: An example of purified test cases

```
0  public void test00() {
1    assertTrue(isValidPhoneNumber("06-12345678"));
2  }
3
4  public void test01() {
5    assertTrue(isValidPhoneNumber("0031612345678"));
6  }
7
8  public void test02() {
9    assertTrue(isValidPhoneNumber("+31612345678"));
10 }
```

We observe that for several automated crash fault localization configurations (Bot_{fail}) have a better performance than the hand-written configuration ($Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$) when the hand-written test cases are not purified test cases. In these configurations, the hand-written failing test cases ($Man_{fail}{}^{+}$) contain several assert statements, executing the same method several times (*i.e.,* having the format as shown in Listing 6.1).

For instance, one of the hand-written crash exposing test cases for `LANG-19b` is shown in Listing 6.3. This test case fails because the input used in Line 4 triggers the underlying fault. In this case, execution paths of the first, second, and third execution of the `translate` method will be marked as failing in the program spectrum. However, the inputs used in the first and second executions are valid, which may introduce noise to the fault localization process. Due to the non-purified test cases, the possibility arises that a non-defective statement is innocently suspected of being defective due to a bug triggering execution in the same test case.

Listing 6.3: Handwritten failing test case of `LANG-19b`

```
0 public void testOutOfBounds() {
1   NumericEntityUnescaper neu = new NumericEntityUnescaper();
2   assertEquals("Test &", neu.translate("Test &"));
3   assertEquals("Test &#", neu.translate("Test &#"));
4   assertEquals("Test &#x", neu.translate("Test &#x"));
5   assertEquals("Test &#X", neu.translate("Test &#X"));
6 }
```

This effect (*i.e., non-purified test cases*) does not appear in the crash-reproducing test cases since BOTSING generates only one failing execution per test case, so in a way, there is only one assertion. Therefore, for each crash-reproducing test case, only one execution path is labeled as failing. This leads to the increase of the diagnostic accuracy for the crash-reproducing test cases (Bot_{fail}) compared to the $Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$ configuration.

6.3.2 Input data ($Bot_{fail} = Man_{fail}{}^{+}$)

For 69% of the crashes that produce a small stack trace (≤ 2 frames), the diagnostic accuracy when using Bot_{fail} test suites is equal to or better than when using $Man_{fail}{}^{+}$ test suites. The primary reason seems to be the low number of parameters of the faulty methods because the small crashes generally involve smaller methods (*e.g.,* helper methods). In this case, BOTSING can achieve a high branch coverage (thanks to the additional *method sequence diversity* objective), resulting in more effective test cases for spectrum-based fault localization.

However, after inspecting the Bot_{fail} and the $Man_{fail}{}^{+}$ test suites, we noticed a difference in the input values used to trigger the crash. The smaller crashes are mostly due to unexpected input values that are assumed to never happen by the developers of the method (and yet, it happened due to an edge case in the system).

For example, `LANG-1b` contains the `createNumber()` method, which does not handle large hexadecimal numbers correctly. The method uses the length of the hexadecimal string to determine whether the hexadecimal string should create a `Integer`, a `Long`, or a `BigInteger`. However, the method of using the string length can determine into which object the number should fit is incorrect. As the cut-off point between an `Integer` and a `Long` is between the 8-digit hexadecimal numbers `"0x7FFFFFFF"` and `"0x80000000"`. Therefore, in the faulty

version of the program, the hexadecimal string `"0x80000000"` is parsed into an `Integer` object, which causes the crash.

Looking at the spectrum-based fault localization performances of Bot_{fail}^{+}-Man_{pass}^{+} and Bot_{fail}^{+}-Evo_{pass}^{+} combinations, we see that they are comparable to the performance of Man_{fail}^{+}-Man_{pass}^{+} with an EXAM score of 0.00035. However, when inspecting the crash-reproducing test cases, we observed that the inputs generated by BOTSING are not similar to the ones used in hand-written test cases (*i.e.,* these inputs all start with `"0x"` or `"-0x"`). Most of the inputs generated by BOTSING are random strings such as `"S/VZ9k'&"` and `"4j8cvkguH"`. These random string cause stack trace similar to the stack trace caused by using the input `"0x80000000"`, both inputs cause a parsing exception (whether or not for completely different reasons).

Those results bring an interesting insight into the crash-reproducing test cases generated using BOTSING. Although such test cases with random input strings would appear to be of little help for developers, they can still be used to identify the faulty statement accurately using spectrum-based fault localization.

6.3.3 Target frame selection ($Bot_{fail} < Man_{fail}^{+}$).

In addition to the first two factors, we notice that the target frame of a crash stack trace for which test cases are generated influences the spectrum-based fault localization diagnostic accuracy.

For example, for the crash `TIME-5b` with a stack trace consisting of 3 frames and a fault located in the method of the third frame (*i.e.,* the defective frame is the last frame in the stack trace). From the 3 frames, BOTSING can generate crash-reproducing test cases for the second (frame 2) and the third (frame 3). In general, BOTSING can directly call the target method in the test case (*e.g.,* when the target method is public) and indirectly (*e.g.,* when the target method is not visible or is invoked by another method called in the test case).

When applying BOTSING on `TIME-5b` with the target frame is set to 2, only 80% of the crash-reproducing test cases call the target method of frame 2. The other 20% cover frame 2 by calling the same method as the one indicated in frame 3, so essentially, it is a crash-reproducing test case for target frame 3. These crash-reproducing test cases combined with unit test cases generated by EVOSUITE (Bot_{fail}^{+}-Evo_{pass}^{+}) result in an EXAM score of 0.03329.

However, when directly targeting frame 3, we observe a significant improvement of the EXAM score (0.00022) for the same configuration (*i.e.,* Bot_{fail}^{+}-Evo_{pass}^{+}). The primary factor behind this improvement is the change in the ratio of test cases directly invoking the faulty method of frame 3 (100%). In other words, when targeting frame 2, the test cases directly invoking the target method of frame 2 add noise to the program spectrum. In principle, these crash-reproducing test cases cause execution paths, which are likely to be correct (*i.e.,* BOTSING violates a precondition), to be marked as faulty.

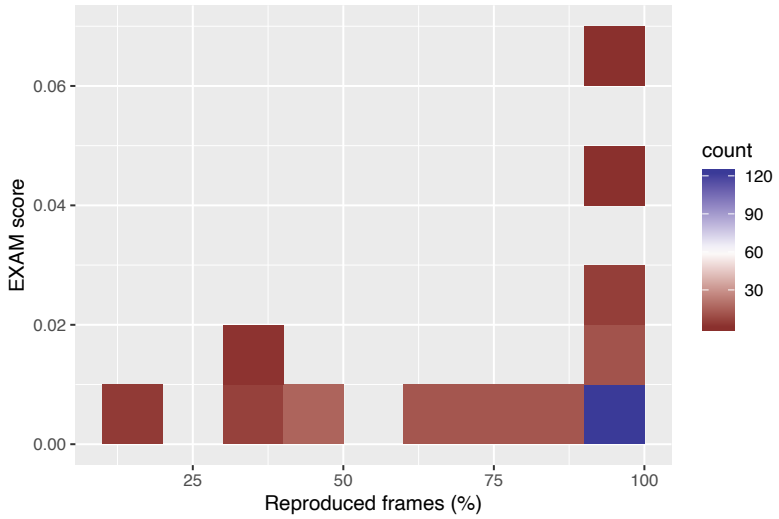


Figure 6.3: Distribution of EXAM score when using the OCHIAI coefficient with the different test suites generated by BOTSING. The reproduced frames indicate the percentage of the frames reproduced by the generated test suite.

This trend is confirmed when looking at the EXAM score of the combination using the Bot_{fail} test suites. Figure 6.3 depicts the distribution of EXAM scores when using the OCHIAI coefficient in combination with the Bot_{fail} test suites. The best EXAM score is accomplished when using test suites generated for the highest frames in the stack trace. Those results suggest that when using BOTSING, one should target the highest possible frame in order to cover as many methods as possible. This confirms previous observations made by related research on manual fault identification using automated crash reproduction [26, 71].

6.3.4 Summary

Various factors linked to the crash-reproducing test cases influence the efficiency of automated fault localization. The *purified* nature of the crash-reproducing tests generated by BOTSING is beneficial as long as the target frame is not below the defective frame. Since this cannot be determined beforehand, one should always aim at the highest frame when using BOTSING. Results show that the form of the input data is not relevant for spectrum-based fault localization as long as it triggers the crash.

Chapter 7

Discussion

In this chapter, we discuss the results found in the empirical evaluation conducted found in the previous chapter using the methodology as described in Chapter 4 and the tooling from Chapter 6. At last, we focus on the threads of validity for our empirical evaluation.

7.1 Similarity coefficients

In the evaluation, we compared four of the best-studied similarity coefficients. Figure 6.2 presents the EXAM score distribution of the corresponding test suites for the different configurations. Among the different coefficients, the DSTAR coefficient performs worst. This contradicts the results reported by Pearson *et al.* [63], showing that there is no significant difference between DSTAR and OCHIAI and TARANTULA when evaluated on real-world faults.

Through manual analysis, we identified that the poor performance could be traced back to an (implicit) assumption made by the researcher behind the DSTAR coefficient (see Section 2.1). If a statement is executed by all the failing test cases ($N_{UF} = 0$), and if there is no passing test case executing the statement ($N_{CS} = 0$), then the DSTAR formula becomes invalid (*i.e.,* DSTAR $= N_{CF}/0$). This occurred in 11 (out of 50) crashes when using hand-written test cases ($Man_{fail}{}^{+}$-$Man_{pass}{}^{+}$).

We conjecture that the different existing similarity coefficients make the same (implicit) assumption that a (potentially) faulty statement is at least covered by one passing test case. We base this conjecture on the dataset used to create and evaluate these similarity coefficients. The four coefficients have been previously evaluated primarily using the Siemens benchmark set [8, 9, 44, 84]. This benchmark set contains test suites which all satisfy the property that each branch in the program is covered by at least 30 test cases [9]. This property supports our hypothesis because each statement in the program is covered at least once by a successful test case.

Currently, a large part of existing similarity coefficients is based on hand-written test cases [85]. Since the automated crash fault localization approach relies on generated test cases,

which have different characteristics compared to hand-written test cases (*e.g.,* the crash-reproducing test cases are *purified*), we believe that it influences the accuracy of automated crash fault localization using the existing similarity coefficients.

In our future work, we plan to investigate this phenomenon primarily by examining the program spectra of crash-reproducing test cases. Secondly, we want to refine the definition of the existing similarity coefficients to take into account the characteristics of automatically generated test suites.

7.2 Program spectra biases

As observed in Section 6.3.3, the choice of a target frame for generating the crash-reproducing test suite can influence the outcome of the fault localization process. In the example, the outcome significantly changed when targeting frame 3, due to the change in the ratio of test cases directly invoking the faulty method. However, the diagnostic accuracy can not only be declined by targeting another frame. It is also possible when targeting the same frame, as the test case generator can favor particular execution paths.

For instance, for the crash `TIME-5b`, when targeting frame 3, BOTSING generated 60 crash-reproducing test cases which have only two different execution paths through the faulty method. At first sight, this does not seem to be a problem. However, a more detailed research shows that the distribution of 60 executions among those two execution paths is not in balance (*i.e.,* 59 for the first path and only 1 for the second path).

In the end, the lack of balance did not affect the diagnostic accuracy for `TIME-5b`, because the defective statement is located in the execution path covered by the 59 crash-reproducing tests. However, it might not always be the case.

In its current implementation, BOTSING generates a test suite containing an unlimited number of crash-reproducing test cases using three objectives: the minimization of the crash distance, the main objective that needs to be achieved to reproduce the crash, and two helper objectives, the maximization of the method sequence diversity and the minimization of the test length to help the search process. Those three objectives ensure in theory that BOTSING produced diverse crash-reproducing test cases. However, diversity in the methods called in the crash-reproducing test cases does not ensure that this diversity will be reflected in the execution paths towards the underlying fault (as it is the case when targeting the third frame of the crash `TIME-5b`).

An example of two test cases generated by BOTSING is shown in Listing 7.1 (assume the bug in located in the `createNumber()` method). The test cases `test01` and `test02` have different execution paths, since `test02` has an additional method call in Line 7. However, when looking at it from the perspective of the underlying fault, the execution paths through the faulty method are the same, because the additional statement does not change the execution path through the defective method.

Listing 7.1: Code example of two test cases that have a diversity in the methods called. However this diversity is not reflected in the execution paths towards the underlying fault (assuming that the bug in located in the `createNumber()` method)

```
0  @Test(timeout = 4000)
1  public void test01() throws Throwable {
2    NumberUtils.createNumber("hnQf");
3  }
4
5  @Test(timeout = 4000)
6  public void test02() {
7    NumberUtils.isParsable(null);
8    NumberUtils.createNumber("hnQf");
9  }
```

One way to balance the execution distribution among the different execution paths trough the faulty method is by using the Density-Diversity-Uniqueness (DDU) metric [64, 65]. The DDU metric quantifies the test suite diagnosability and maximizing the metric increases the effectiveness of spectrum-based fault localization [64]. In their work, Perez *et al.* [65] use DDU as a (primary) search objective to generate unit tests with EVOSUITE. This may, however, not be directly possible for crash reproduction as maximizing the DDU metric might conflict with the crash distance minimization objective (*i.e.,* DDU seeks to cover unique and diverse execution paths while crash-reproducing is only interested in the ones). In our future work, we plan to experiment with search-based crash reproduction with the DDU metric as a primary and secondary objective to improve the effectiveness of the generated tests for fault localization.

7.3 Test case generation

In Section 6.2, we observed that there is no statistical difference in the fault localization diagnostic accuracy when using hand-written (Man_{fail}^{+}) or automatically generated crash-reproducing test cases (Bot_{fail}^{1}or Bot_{fail}^{+}) as long as those test cases cover the defective statement.

Furthermore, as illustrated in Figure 6.3, the best EXAM score is achieved by the crash-reproducing test cases covering a majority of the stack traces (*i.e.,* crash-reproducing test cases generated for higher frames). This suggests that a crash-reproducing test case will be useful for automated crash fault localization as long as it covers enough frames, and the higher, the better. It sends a strong signal to the crash reproduction research community to aim at reproducing the highest frames possible.

However, aiming at higher frames for crash reproduction is easier said than done [72]. Reproducing higher frames requires more advanced search-based crash reproduction approaches [30, 31] and, potentially, a higher computation power that in practice might not always be available in a development environment. Future research should investigate approaches to strike a balance between the computation cost and the frame level to target in order to produce useful crash-reproducing test cases, for instance, using static analysis [86] or predictive models [39] to identify potentially faulty frames.

7.4 Threats to validity

In this section, we describe the threads of the validity of this thesis. Therefore we describe the *internal validity*, the *external validity*, the way to *reproduce* our results.

7.4.1 Internal validity

We selected 50 crashes from DEFECTS4J that have been previously analysed in JCRASHPACK [72] and the fault localization dataset of Pearson *et al.* [63]. We also used BOTSING, EVOSUITE, and GZOLTAR with the configuration described in Chapter 4. We cannot guarantee that those tools are free of defects, but EVOSUITE and GZOLTAR are long-term established state-of-the-art, studied, and used by many users, BOTSING is a fresh and well-tested implementation of the EVOCRASH [71] approach.

Finally, we choose to execute the automated crash fault localization pipeline as a whole (end-to-end). Giving the randomness involved in test case generation, future work should include an extended evaluation of the pipeline by focusing on and repeating the generation of the crash-reproducing test cases. We believe that this will yield more crash-reproducing test cases and increase the statistical significance of our conclusions.

7.4.2 External validity

We cannot guarantee that our results are generalizable to all crashes. However, following Pearson *et al.*'s [63] recommendations, we used crashes from real-world software faults. Of course, considering more crashes would increase confidence in our results, but given the exploratory nature of this study, we believe that using a smaller set of crashes, previously studied both for crash reproduction and fault localization, would provide us more in-depth insights on the results.

7.4.3 Replication of the results

A replication package of our empirical evaluation is available at Github[1]. The repository contains all the tooling created as described in Chapter 5 in the format of a DOCKER container, the scripts to run the automated crash fault localization pipeline, the data described in this thesis including the test case dataset, and the scripts to reproduce the statistical analysis.

For a detailed description of the components used in the evaluation, we refer to the documentation provided in the thesis (see Chapter 5). To execute the DOCKER containers and the scripts used to execute the whole automated crash fault localization pipeline, we refer to the `README.md` in the repository.

[1] https://github.com/svenpopping/acfl-replication-package

Chapter 8

Conclusion

In this thesis, we presented to the best of our knowledge the first automated crash fault localization pipeline, in which we combined search-based crash reproduction and spectrum-based fault localization. The pipeline generates crash-reproducing test cases from the stack trace included in the crash report. These test cases, combined with the existing or an automatically generated test suite, are used as input for the spectrum-based fault localization approach. Using the spectrum-based fault localization approach, the potentially faulty lines of code are identified, with the primary purpose of reducing debugging efforts for developers.

The results of our empirical evaluation of 50 real-world faults show that automatically generated crash-reproducing test cases reduce the number of statements to be investigated by developers. In the general scenario, however, hand-written test cases remain the most efficient because not all crashes can yet be reproduced for a high enough frame (*i.e.,* at least containing the defective frame). After all, when considering the best-case scenario, where the crash-reproducing test cases do indeed cover the fault, we do not observe a statistically significant difference between the fault localization accuracy between hand-written and automatically generated test cases.

In addition, our research into the four similarity coefficients shows that there is no clear winner (no statistically significant difference). However, we can safely claim that DSTAR is not suitable for automatic crash fault localization, because of the assumption that a statement should at least be covered by a successful test case. Furthermore, it does not matter which similarity coefficient (except DSTAR) is applied when using a single crash-reproducing test case, as in this case, the similarity coefficients have a similar mathematical behavior.

Our results confirm the feasibility of the automated crash fault localization pipeline and open up new paths to end-to-end automated crash fault localization and automated program repair.

8.1 Future work

We have shown that our automated crash fault localization approach is feasible. Nonetheless, there are still improvements that can be made, inspired by the manual analyses conducted for the evaluation and the issues we encountered during this research. In this chapter, we will present our suggestions for future work on the matter.

8.1.1 ACFL approach

Potential improvements to the automated crash fault localization pipeline can be obtained by investigating the following alterations:

- The most important aspect of the automated crash fault localization approach is that a crash-reproducing test case is generated, covering a sufficiently high frame level (*i.e.,* at least containing the defective frame). However, the higher the frame level, the more computationally intensive the search process. Therefore, it will be useful to investigate information retrieval approaches to identify an acceptable frame level to target for search-based crash reproduction, balancing computation cost and fault localization accuracy.

- To improve the quality of the crash-reproducing test cases, BOTSING can be extended by including new objectives based on the DDU metric proposed by Perez *et al.* [64]. With this metric, we hope to increase the fault localization accuracy of the crash-reproducing test case.

- In our automated crash fault localization approach, we used spectrum-based fault localization for the fault localization part of the pipeline. However, as stated by Wong *et al.* [85], there are many different automated fault localization approaches. It would be helpful to know which of these approaches is the best suited for the automated crash fault localization approach (*e.g.,* machine learning-based approaches).

- Currently, the fault localization report is exported as a CSV file, containing the sorted list of defective statements. Therefore, it would be useful to extend the automated crash fault localization approach such that in the end, it would visualize the report, for example, by using the visualization technique proposed by Jones *et al.* [46].

- Determining the best method to generate the unit test cases. In our evaluation, the generated test cases also include test cases that might be irrelevant for the underlying bug (*e.g.,* other methods in the class, which are outside of the stack trace). This computation time can be better spent on producing test cases that are relevant for the bug.

8.1.2 Evaluation

To further improve the evaluation of the automated crash fault localization pipeline, we propose the following points that may be of interest for future work:

- Our evaluation included an end-to-end evaluation of the automated crash fault localization approach. So, we did not evaluate the effects caused by the randomness of BOTSING. Therefore, we should investigate the effects of the randomness by repeating the crash reproduction step and analyzing the fault localization process outcome.
- The extension of the current evaluation with new crashes, for example, with the `XWiki` or the `Elasticsearch` crashes from the JCRASHPACK dataset [72]. In addition to this, DEFECTS4J [47] has recently been upgraded with the addition of 11 new projects. This applies not only to the pipeline as a whole but also to BOTSING alone because it is not yet clear whether these crashes can be reproduced.
- With the aim of implementing automated crash fault localization in existing testing and debugging infrastructures in mind, it would be of great value to know the pipeline's runtime statistics when executed as a whole.
- EVOCRASH [71], the ancestor of BOTSING, has been evaluated using a controlled experiment to determine the usefulness for debugging [14]. In our evaluation, we only look at the automated crash fault localization approach with the diagnostic accuracy determined by the EXAM score. However, it would also be useful to assess the effectiveness of automated crash fault localization using a controlled experiment. Especially in case, a random input is generated, by BOTSING, that is useful for localizing faults, but where we are not sure if they are useful for the developer.

8.1.3 Other

Now that we have presented our views of interesting future research on the automated crash fault localization approach and the evaluation of the approach, we present the following points which can be conducted as future research:

- During the manual analyses, we observed that BOTSING could generate a lot of test cases (up to 300). These test cases are useful for the automated fault localization process, but we argue as well that these test cases are useful for the developer when trying to patch the program. However, determining which of the broad set of test cases is relevant could be a struggle in itself. A solution is to visualize the test cases that are the most useful for the developer when investigating a particular statement, such that the information within the test case can be utilized to patch the underlying bug.
- Our automated crash fault localization approach is based on frameworks designed explicitly for `Java`. However, given the benefits of the approach, it might be useful to extend the pipeline to support other programming languages.
- As mentioned in Chapter 6, there is an unusual difference between the number of test cases generated by $Bot_{fail}{}^{1}$ and $Bot_{fail}{}^{+}$. We could not pinpoint the cause of this difference during our research, so it might be interesting for the researcher behind BOTSING to investigate this difference.

Bibliography

[1] Current world population. URL https://www.worldometers.info/world-population/.

[2] Software fail watch: The politics of software defects, q2, 2018, Aug 2019. URL https://www.tricentis.com/blog/software-fail-watch-q2-2018/.

[3] Bbp, productie en bestedingen; kwartalen, waarden, nationale rekeningen, 2020. URL https://opendata.cbs.nl/statline/#/CBS/nl/dataset/84114NED/table?ts=1589293156803.

[4] David Abramson, Ian Foster, John Michalakes, and Rok Sosic. Relative debugging and its application to the development of large numerical models. In *Supercomputing'95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, pages 51–51. IEEE, 1995.

[5] David Abramson, Clement Chu, Donny Kurniawan, and Aaron Searle. Relative debugging in an integrated development environment. *Software: Practice and Experience*, 39(14):1157–1183, 2009.

[6] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 39–46. IEEE, 2006.

[7] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, 2007.

[8] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82 (11):1780–1792, 2009.

[9] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. Spectrum-based multiple fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99. IEEE, 2009.

[10] Hira Agrawal, James L Alberi, Joseph R Horgan, J Jenny Li, Saul London, W Eric Wong, Sudipto Ghosh, and Norman Wilde. Mining system tests to aid software maintenance. *Computer*, 31(7):64–73, 1998.

[11] Hiralal Agrawal, Joseph R Horgan, Saul London, and W Eric Wong. Fault localization using execution slices and dataflow tests. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*, pages 143–151. IEEE, 1995.

[12] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 263–272. IEEE, 2017.

[13] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. Deploying search based software engineering with sapienz at facebook. In *International Symposium on Search Based Software Engineering*, pages 3–45. Springer, 2018.

[14] Aaron Ang, Alexandre Perez, Arie van Deursen, and Rui Abreu. Revisiting the practical use of automated software fault localization techniques. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 175–182. IEEE, 2017.

[15] Shay Artzi, Sunghun Kim, and Michael D Ernst. Recrash: Making software failures reproducible by preserving object states. In *European conference on object-oriented programming*, pages 542–565. Springer, 2008.

[16] Luciano C Ascari, Lucilia Y Araki, Aurora RT Pozo, and Silvia R Vergilio. Exploring machine learning techniques for fault localization. In *2009 10th Latin American Test Workshop*, pages 1–6. IEEE, 2009.

[17] Thomas Ball and James R Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360, 1994.

[18] Jonathan Bell, Nikhil Sarda, and Gail Kaiser. Chronicler: Lightweight recording to reproduce field failures. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 362–371. IEEE, 2013.

[19] Francesco A Bianchi, Mauro Pezzè, and Valerio Terragni. Reproducing concurrency failures from crash stacks. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 705–716, 2017.

[20] David W Binkley and Mark Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62(105178):105–178, 2004.

[21] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. Gzoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 378–381, 2012.

[22] José Campos, Rui Abreu, Gordon Fraser, and Marcelo d'Amorim. Entropy-based test generation for improved fault localization. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 257–267. IEEE, 2013.

[23] Yu Cao, Hongyu Zhang, and Sun Ding. Symcrash: selective recording for reproducing crashes. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 791–802, 2014.

[24] Mariano Ceccato, Alessandro Marchetto, Leonardo Mariani, Cu D Nguyen, and Paolo Tonella. An empirical study about the effectiveness of debugging when random test cases are used. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 452–462. IEEE, 2012.

[25] Mariano Ceccato, Alessandro Marchetto, Leonardo Mariani, Cu D Nguyen, and Paolo Tonella. Do automatically generated test cases make debugging easier? an experimental assessment of debugging effectiveness and efficiency. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(1):1–38, 2015.

[26] Ning Chen and Sunghun Kim. Star: Stack trace based automatic crash reproduction via symbolic execution. *IEEE transactions on software engineering*, 41(2):198–220, 2014.

[27] James Clause and Alessandro Orso. A technique for enabling and supporting debugging of field failures. In *29th International Conference on Software Engineering (ICSE'07)*, pages 261–270. IEEE, 2007.

[28] James S Collofello and Larry Cousins. Towards automatic software fault location through decision-to-decision path analysis. In *afips*, page 539. IEEE, 1899.

[29] Deborah S Coutant, Sue Meloy, and Michelle Ruscetta. Doc: A practical approach to source-level debugging of globally optimized code. *ACM SIGPLAN Notices*, 23(7): 125–134, 1988.

[30] Pouria Derakhshanfar, Xavier Devroey, Gilles Perrouin, Andy Zaidman, and Arie van Deursen. Search-based crash reproduction using behavioural model seeding. *Software Testing, Verification and Reliability*, 30(3):e1733, 2020.

[31] Pouria Derakhshanfar, Xavier Devroey, Andy Zaidman, Arie van Deursen, and Annibale Panichella. Crash Reproduction Using Helper Objectives. In *Genetic and Evolutionary Computation Conference Companion (GECCO '20 Companion)*, Cancún, Mexico, 2020. ACM. doi: 10.1145/3377929.3390077.

[32] Jermaine Charles Edwards. Method, system, and program for logging statements to monitor execution of a program, March 25 2003. US Patent 6,539,501.

[33] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

[34] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2012.

[35] Gordon Fraser and Andrea Arcuri. Evosuite: On the challenges of test case generation in the real world. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 362–369. IEEE, 2013.

[36] Gordon Fraser and Andrea Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):1–42, 2014.

[37] Meng Gao, Pengyu Li, Congcong Chen, and Yunsong Jiang. Research on software multiple fault localization method based on machine learning. In *MATEC Web of Conferences*, volume 232, page 01060. EDP Sciences, 2018.

[38] María Gómez, Romain Rouvoy, Bram Adams, and Lionel Seinturier. Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 88–99. IEEE, 2016.

[39] Yongfeng Gu, Jifeng Xuan, Hongyu Zhang, Lanxin Zhang, Qingna Fan, Xiaoyuan Xie, and Tieyun Qian. Does the fault reside in a stack trace? assisting crash localization by predicting crashing fault residence. *Journal of Systems and Software*, 148: 88–104, 2019.

[40] Tibor Gyimóthy, Árpád Beszédes, and István Forgács. An efficient relevant slicing method for debugging. In *Software Engineering—ESEC/FSE'99*, pages 303–321. Springer, 1999.

[41] Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–12. IEEE, 2015.

[42] Matthias Hauswirth and Trishul M Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Acm SIGPLAN notices*, volume 39, pages 156–164. ACM, 2004.

[43] John L Hennessy. Symbolic debugging of optimized code. 1979.

[44] James A Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, 2005.

[45] James A Jones, Mary Jean Harrold, and John T Stasko. Visualization for fault localization. In *in Proceedings of ICSE 2001 Workshop on Software Visualization*. Citeseer, 2001.

[46] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 467–477. IEEE, 2002.

[47] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, 2014.

[48] Fitsum Kifetew, Xavier Devroey, and Urko Rueda. Java unit testing tool competition-seventh round. In *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*, pages 15–20. IEEE, 2019.

[49] Ákos Kiss, Judit Jász, and Tibor Gyimóthy. Using dynamic information in the interprocedural static slicing of binary executables. *Software Quality Journal*, 13(3): 227–245, 2005.

[50] Bogdan Korel and Janusz Laski. Stad-a system for testing and debugging: User perspective. In *[1988] Proceedings. Second Workshop on Software Testing, Verification, and Analysis*, pages 13–20. IEEE, 1988.

[51] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 872–881, 2006.

[52] Wolfgang Mayer, Markus Stumptner, Dominik Wieland, and Franz Wotawa. Can ai help to improve debugging substantially? debugging experiences with value-based models. In *Proceedings of the 15th European Conference on Artificial Intelligence*, pages 417–421. IOS Press, 2002.

[53] Phil McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163. IEEE, 2011.

[54] Urko Rueda Molina, Fitsum Kifetew, and Annibale Panichella. Java unit testing tool competition-sixth round. In *2018 IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST)*, pages 22–29. IEEE, 2018.

[55] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 153–162. IEEE, 2014.

[56] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 284–295. IEEE, 2005.

[57] Mathieu Nayrolles, Abdelwahab Hamou-Lhadj, Sofiène Tahar, and Alf Larsson. A bug reproduction approach based on directed model checking and crash traces. *Journal of Software: Evolution and Process*, 29(3):e1789, 2017.

[58] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013.

[59] Karl J Ottenstein and Linda M Ottenstein. The program dependence graph in a software development environment. *ACM SIGSOFT Software Engineering Notes*, 9(3): 177–184, 1984.

[60] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2017.

[61] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. A large scale empirical comparison of state-of-the-art search-based test case generators. *Information and Software Technology*, 104:236–256, 2018.

[62] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proceedings of the 38th International Conference on Software Engineering*, pages 547–558, 2016.

[63] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 609–620. IEEE, 2017.

[64] Alexandre Perez, Rui Abreu, and Arie van Deursen. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 654–664. IEEE, 2017.

[65] Alexandre Perez, Rui Abreu, and Arie Van Deursen. A theoretical and empirical analysis of program spectra diagnosability. *IEEE Transactions on Software Engineering*, 2019. doi: 10.1109/tse.2019.2895640.

[66] Ju Qian and Baowen Xu. Scenario oriented program slicing. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 748–752, 2008.

[67] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on software engineering*, 21(1):19–31, 1995.

[68] Jeremias Rößler, Andreas Zeller, Gordon Fraser, Cristian Zamfir, and George Candea. Reconstructing core dumps. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 114–123. IEEE, 2013.

[69] Colin Runciman and David Wakeling. Heap profiling of lazy functional programs. *Journal of functional programming*, 3(2):217–245, 1993.

[70] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 201–211. IEEE, 2015.

[71] Mozhan Soltani, Annibale Panichella, and Arie Van Deursen. Search-Based Crash Reproduction and Its Impact on Debugging. *IEEE Transactions on Software Engineering*, 2018. ISSN 0098-5589. doi: 10.1109/TSE.2018.2877664.

[72] Mozhan Soltani, Pouria Derakhshanfar, Xavier Devroey, and Arie Van Deursen. A benchmark-based evaluation of search-based crash reproduction. *Empirical Software Engineering*, 25(1):96–138, 2020.

[73] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. jrapture: A capture/replay tool for observation-based testing. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 158–167, 2000.

[74] Frank Tip and TB Dinesh. A slicing-based approach for locating type errors. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(1):5–55, 2001.

[75] András Vargha and Harold D Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[76] Iris Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5):459–494, 1985.

[77] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[78] Eric Wong, Tingting Wei, Yu Qi, and Lei Zhao. A crosstab-based statistical method for effective fault localization. In *2008 1st international conference on software testing, verification, and validation*, pages 42–51. IEEE, 2008.

[79] W Eric Wong and Jenny Li. An integrated solution for testing and analyzing java applications in an industrial setting. In *12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 8–pp. IEEE, 2005.

[80] W Eric Wong and Yu Qi. Effective program debugging based on execution slices and inter-block data dependency. *Journal of Systems and Software*, 79(7):891–903, 2006.

[81] W Eric Wong and Yu Qi. Bp neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering*, 19(04):573–597, 2009.

[82] W Eric Wong, Tatiana Sugeta, Yu Qi, and Jose C Maldonado. Smart debugging software architectural design in sdl. *Journal of Systems and Software*, 76(1):15–28, 2005.

[83] W Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani Thuraisingham. Effective software fault localization using an rbf neural network. *IEEE Transactions on Reliability*, 61(1):149–169, 2011.

[84] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, 2013.

[85] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[86] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. Crashlocator: locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 204–214, 2014.

[87] Jifeng Xuan and Martin Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 52–63, 2014.

[88] Jifeng Xuan, Xiaoyuan Xie, and Martin Monperrus. Crash reproduction via test case mutation: let existing test cases help. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 910–913, 2015.

[89] Shin Yoo, Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, and Mark Harman. No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist. *RN*, 14(14):14, 2014.

[90] Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[91] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[92] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faulty code by multiple points slicing. *Software: Practice and Experience*, 37(9):935–961, 2007.