Fit2Crash: Specialising Fitness Functions for Crash Reproduction

Master's Thesis

Shang Xiang

Fit2Crash: Specialising Fitness Functions for Crash Reproduction

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Shang Xiang
born in Hubei, China

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

Fit2Crash: Specialising Fitness Functions for Crash Reproduction

Author: Shang Xiang
Student id: 4702522
Email: S.Xiang@student.tudelft.nl

Abstract

Software applications inevitably crash, and it is time-consuming to recreate the crash conditions for debugging. Recently, researchers have developed frameworks relying on genetic algorithms, e.g. Botsing, for automated crash reproduction. However, the existing approaches process exceptions of different types as if they were the same. In this thesis, we study how the four most common types of Java exceptions are thrown and define specialised fitness functions for them. We have extended Botsing and carried out an evaluation against 52 real-world crashes from seven various open-source software applications. Our results show that our proposed fitness functions influence both the effectiveness and efficiency, negatively or positively depending on the type of the target exception. This thesis demonstrates how tailoring the fitness functions according to the exception type can improve search-based crash reproduction.

Thesis Committee:

Chair:	Prof.	Dr.	A.	Zaidman	Faculty EEMCS, TU Delft
University supervisor:	Prof.	Dr.	A.	Zaidman	Faculty EEMCS, TU Delft
Committee Member:		Dr.	A.	Panichella	Faculty EEMCS, TU Delft
		Dr.	J.	Cockx	Faculty EEMCS, TU Delft
		Dr.	X.	Devroey	Faculty EEMCS, TU Delft
			P.	Derakhshanfar	Faculty EEMCS, TU Delft

Preface

It feels like yesterday when I landed my feet on Mekelweg for the first time. Having never been outside of my home country, I had no idea what lay in store for me. Nearly three years have flown by, and I cannot believe that this journey is coming to an end now. It has always been a ride full of adventures, excitements as well as challenges and not-so-bright moments.

I am especially grateful to Professor Andy Zaidman for giving me the opportunity to work on this thesis project under his supervision and shedding light on me when I was at the darkest moment of my life. I would also like to thank my fellow countrywoman Qianqian for introducing me to Professor Zaidman. Without the tremendous help from my two brilliant mentors, Xavier, who may or may not have telepathic superpowers, and Pouria, who loves Rick and Morty as much as I do, it would not have been possible for me to finish the project.

During the one thousand days of being an expatriate, my homesickness can always be cured by Jasper and Xiaodong's emotional support and Xin's fantastic cooking skill of Chinese food. Gijs' volunteering to be my mental coach and academic counsellor Susanne's professional help pulled me back from an abyss. Thank you all. Last but not least, I want to thank my parents for the upright upbringing, unconditional love and support, and Silviu for simply existing in my life.

And finally, I dedicate this report to my two stolen bikes. I hope you enjoy reading it.

Shang Xiang
Delft, the Netherlands
March 8, 2020

Contents

Preface iii

Contents v

List of Figures vii

List of Tables viii

1 Introduction 1
- 1.1 Challenges in Highly-Reliable Software Development . . . 1
- 1.2 Crash Reproduction . . . 2
- 1.3 Research Questions . . . 3
- 1.4 Thesis Outline . . . 3

2 Background 5
- 2.1 Search-Based Optimisation Algorithms . . . 5
- 2.2 Search-Based Software Testing . . . 9
- 2.3 Automated Crash Reproduction for Debugging . . . 19
- 2.4 Botsing . . . 24
- 2.5 Limitation . . . 26

3 Common Runtime Exceptions 27
- 3.1 `ArrayIndexOutOfBoundsException` . . . 27
- 3.2 `StringIndexOutOfBoundsException` . . . 30
- 3.3 `IllegalStateException` and `IllegalArgumentException` . . . 32
- 3.4 `NullPointerException` . . . 34

4 Crash-Specific Fitness Functions 37
- 4.1 Adapting Integration Testing Fitness Function for Indexed Access . . . 37
- 4.2 Many-Objectivisation with Helper Objectives for Branching Variables . . . 41
- 4.3 Summary . . . 47

5 Evaluation and Results **49**
5.1 Experiment Set-up 49
5.2 Results 53
5.3 Discussion 58
5.4 Threats to Validity 66

6 Conclusions and Future Work **69**
6.1 Contributions 69
6.2 Conclusions 70
6.3 Future Work 70

Bibliography **73**

A Glossary **79**

List of Figures

2.1 Example Function f . . . 5
2.2 Random Search . . . 5
2.3 Hill Climbing . . . 6
2.4 Overview of a Genetic Algorithm . . . 7
2.5 Crossover . . . 8
2.6 Crossover Operation Example [1] . . . 12
2.7 Malformed Crossover Offsprings [1] . . . 13
2.8 Control Flow Graph of `checkTriangleType()` Method in the `Triangle` Class 15
2.9 Weighted Sum versus Many-Objective . . . 18
2.10 Overview of EvoCrash . . . 20

3.2 Operand Stack When Loading . . . 28
3.3 Operand Stack When Storing . . . 28
3.1 Accessing Array Elements in Java . . . 29

4.1 Related Components to Indexed Access in Botsing . . . 37
4.2 $d_{exception}$ (blue dashed line) versus $d^*_{exception}$ (red dotted line) . . . 38
4.3 Instrumentation Before Executing a Loading Instruction . . . 40
4.4 Related Components to Branching Variable Diversity in Botsing . . . 42
4.5 `BranchingVariableDiversityObjective` Class . . . 44

5.1 Reproduction Status on the Case Level . . . 54
5.2 Reproduction Status on the Application Level . . . 55

List of Tables

2.1 Test Cases 14
2.2 Approach Levels of the Test Cases 16
2.3 Distance Formulae [2] 16
2.4 Fitness Values of the Test Cases 17
2.5 Crash Cases Composition of JCrashPack [3] 23

3.1 Bytecode Instructions for Array Access 29
3.2 Causes of `StringIndexOutOfBoundsException` Crashes in JCrashPack . . . 30
3.3 Different Types of Methods That Throws `StringIndexOutOfBoundsException` 31

4.1 Diversity Objectives for Different Types of Variables 43

5.1 Detailed Composition of the Picked Crash Cases 50
5.2 Complexity Statistics of Crashes Used 51
5.3 Odds Ratio Between [**IA**] and [**IA-control**] 56
5.4 Odds Ratio Between [**BV**] and [**BV-control**] 57
5.5 Vargha and Delaney's $\hat{A}_{12}$ Measure Between [**IA**] and [**IA-control**] 57
5.6 Vargha and Delaney's $\hat{A}_{12}$ Measure Between [**BV**] and [**BV-control**] 58
5.7 Effect Size of LANG-19b 62

Chapter 1

Introduction

As ICT applications infiltrate our daily activities more and more, the reliability of software that we use has become of crucial importance to the quality of our lives and the functioning of the society.

Unreliable software implies high costs on various aspects of the life-cycle of a software product. For the owner of the product, it means high maintenance cost. In 2002, the estimated labour cost of unreliable software was forty billion US dollar per year [4]. The number has only gone up after that. A 2015 survey showed that maintenance takes 60% of the resources of the total cost of the software [5]. Furthermore, that number can surge up to 90% when the software is extremely unreliable.

For consumers of the software, unreliability costs user satisfaction. In 2012, Apple faced massive backlash for replacing the built-in Maps application of iOS with its buggy software.

For the general public, it means threats to life or access to food and shelter. Malfunctioning software in the Boeing 737 Max resulted in the death of hundreds in two consecutive aeroplane accidents in 2019 [6]. Previously, Ko et al. collected news reports about software failures of three decades, from 1980 to 2012, and showed that every month there has been on average one news report about casualties due to unreliable software [7].

The academics and the industry have put an enormous amount of efforts into coming up with theories and best practices to develop reliable software systems to avoid the high costs.

1.1 Challenges in Highly-Reliable Software Development

In essence, we can break down the procedure to develop more reliable software systems into three aspects [8]. The first one is to adopt a thoughtful design methodology that leads to a highly-reliable product. The second one is to test thoroughly before shipping. The last one is to make hard efforts to debug the software when issues do emerge. Developing reliable software poses many challenges.

We are yet to find a design methodology that could, in theory, eliminate all the threats to reliability. Let alone implementing such a design in the real world. Meanwhile, well-designed software can at most be issue-resistant, but not issue-proof.

When a software product reaches the hand of an end-user, environment and behaviours of the user are challenging to predict. Therefore, the developer can test only a small fraction of all possible execution scenarios, making the unreliability of software not able to be solved sufficiently by testing.

As for debugging, typically a crash report, containing a certain amount of run-time information of the software when the issue happened, is presented to the developer. It is often a stack trace fragment. The information within is considerably limited and far away from being enough to disclose the nature of the issue behind the crash. It takes empirical knowledge and trial-and-error attempts of the developer to locate the bug and then to solve it, which can usually be time-consuming.

1.2 Crash Reproduction

Glenford and Myers [9] defined debugging as:

> *The process of diagnosing the precise nature of a known error and then correcting it.*

Crash reproduction, i.e. to create a test case that causes the program to crash in the way specified by the report, is helpful for debugging. As pointed out by Zeller [10], the significance of reproducing the crash is twofold: (i) to bring the crash under control, such that the deficiency of the software can be observed; and (ii) to verify the success of the fix afterwards.

To ease the process of crash reproduction, the software engineering research community has developed automation techniques. One category of methods to automate crash reproduction is record-replay, which relies on program run-time data [11, 12, 13, 14, 15, 16]. It records the program execution data in order to use it for identifying the program states and execution path that have led to the crash. Record-replay systems either incur high run-time overhead due to monitoring, or require specialised hardware that is not widely available [17, 18]. Another concern is the potential infringement to privacy protection regulations [19].

Another direction is post-failure approaches, which collect information from the crash itself and take action after the crash [20, 21, 22, 23]. One of the innovative techniques in the post-failure area that researchers have been exploring recently is to use search-based optimisation algorithms like genetic algorithms, for crash reproduction [24]. EvoCrash/Botsing [25, 26] is such a framework. It uses EvoSuite [27] to instrument code and implements a guided genetic algorithm to search for the inputs and the call sequence that caused the crash.

A crucial component of genetic algorithms is the fitness function, which quantifies the distance between the generated test cases and the optimal test case that is able to reproduce the given crash. Current researches evaluate generated test cases for different kinds of crashes with the same fitness function, ignoring the type of the actual exception causing the crash. And a binary exception distance penalty is imposed on the fitness value depending only on whether the target exception is thrown.

1.3 Research Questions

`RuntimeException`s in Java are extended into various sub-classes according to the nature of the exceptions. For example, an `ArrayIndexOutOfBoundsException` is thrown when accessing an array with an illegal index. The information beneath is clearly characterised differently from that of a `NullPointerException`, which is thrown when dereferencing a `null` reference. That information is ignored as a result of the binary definition of the exception distance used in the fitness functions of Botsing.

We hypothesise that by taking the type of the target exception into account, fitness functions can be specialised. By collection additional information according to the type, we can make the fitness values of the generated test cases more granular, and whence improve the search process. Therefore we formulate the following research questions:

RQ1 How to define a distance between the execution flow of a test case and the execution flow throwing a specific type of exception?

RQ2 How to include the additional information provided by this distance in a fitness function to improve the guidance of the search process?

RQ3 What is the impact of the new fitness function in terms of the effectiveness and efficiency of search-based crash reproduction?

To answer those research questions, we first studied how the most common types of Java exceptions are typically thrown, and defined fitness functions with specialised exception distances. Then we integrated the fitness functions into Botsing, a search-based crash reproduction framework, and extended the underlying bytecode instrumentation to provide adequate information to those fitness functions. Eventually, we evaluated our new fitness functions with 52 crashes coming from various real-world open-source software applications. Our results show that including the additional information in the search process influenced both the effectiveness and efficiency, negatively or positively depending on the type of the target exception.

1.4 Thesis Outline

In Chapter 2, we discuss background knowledge needed and previous work about crash reproduction. In Chapter 3, we study four of the most common types of Java exceptions, detailing how they are typically thrown. In Chapter 4, we present the implementation details of our solution to the situation. In Chapter 5, we describe how the evaluation experiments were carried out and analyse the results of the experiments. Finally, in Chapter 6, we conclude and reflect on future improvements to be done.

Chapter 2

Background

In this chapter, we discuss the background knowledge that we base our project on and the current state of the art of automated crash reproduction.

2.1 Search-Based Optimisation Algorithms

In a typical optimisation problem, one is faced with a function f that given a set of input arguments $x_0, x_1, x_2, \ldots$ generates a value y as output:

$$y = f(x_0, x_1, x_2, \ldots)$$

The goal of the optimisation is to find certain input arguments, $x_0\prime, x_1\prime, x_2\prime, \ldots$ that make f generate the maximal or minimal output. The domain of definition of f is referred to as the input domain or search space. See Figure 2.1 for an example function f, where there is only one input argument for the simplicity of writing, and the goal of the optimisation is to find the maximum.

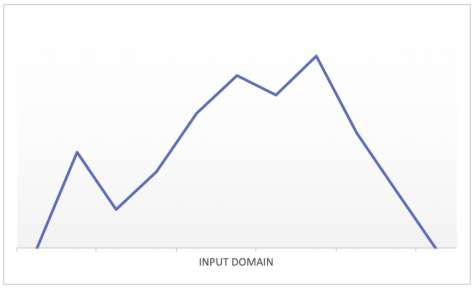


Figure 2.1: Example Function f

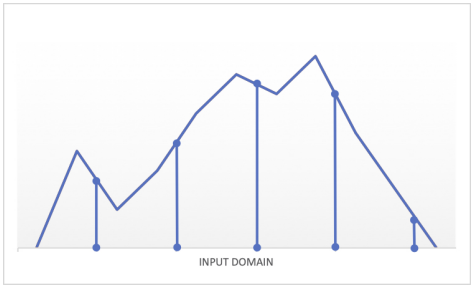


Figure 2.2: Random Search

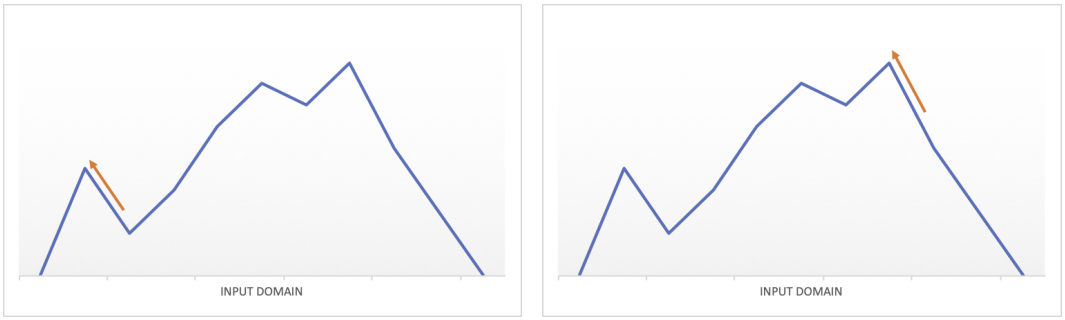
2.1.1 Random Search

The most easy-to-implement search algorithm is *Random Search*. As its name suggests, inputs are randomly generated until the goal of optimisation is achieved. As can bee seen

in Figure 2.2, random search fumbles through the input domain without any guidance and thus can be highly inefficient to reach the specific goal.

2.1.2 Hill Climbing

Hill Climbing still starts at a random point in the input domain, but it seeks guidance from the nearby area. When a neighbouring point has a more desirable output value, hill climbing switches to the new point and re-evaluates the neighbouring points. It repeats until no neighbouring points have a better output value. However, this means that hill climbing is vulnerable to local optima, as shown in Figure 2.3a. Typically, the algorithm is restarted certain times to counter that situation. When lucky enough, the algorithm will initialise at a random point that can lead it to the global optimum, as shown in Figure 2.3b.



(a) Stuck at a local optimum (b) Restart and get to the global optimum

Figure 2.3: Hill Climbing

2.1.3 Simulated Annealing

A further improved version of hill climbing is called *Simulated Annealing* [28]. It simulates the metallurgic process when a melted solid is slowly cooled down to a particular shape by controlling the temperature.

The significant difference between hill climbing and simulated annealing is that the former is strictly restricted to only moving towards neighbouring points that have better outputs. As for simulated annealing, the direction of moving is probabilistic. The probability is related to the parameter *temperature*. At the beginning, the temperature is hot, and there is a higher chance for the search process to move to a neighbouring point with a worse output. As the search goes on, temperature drops and the chance gets lower. Eventually, the temperature reaches a threshold called the *freezing point*, and from there on simulated annealing behaves the same as hill climbing. The existence of *temperature* makes simulated annealing relatively more robust against local optima.

The aforementioned search algorithms are considered as *local search approaches* as they consider merely one point at a time and evaluate only the local neighbours of that point.

2.1.4 Genetic Algorithms

Genetic Algorithms (GA), proposed by Holland et al. back in the 1970s [29], are a type of *global search approaches*. Inspired by Darwin's theory of evolution, the algorithm simulates the process of natural selection. The main loop of a genetic algorithm is shown in Figure 2.4.

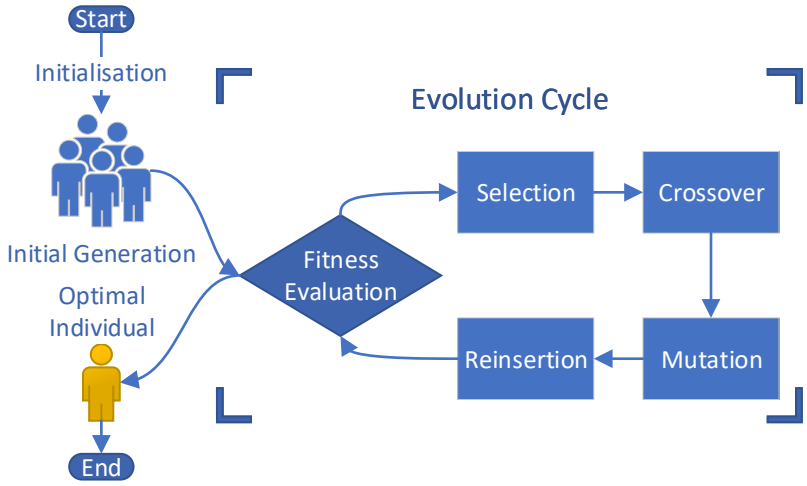


Figure 2.4: Overview of a Genetic Algorithm

At the *initialisation* phase, a certain number of points from the input domain are randomly selected to form the *Initial Generation*. A single input point is referred to as an *individual* or a *chromosome*. The number of individuals is called the *population size*.

Then the initial generation goes through the evolution cycle to produce offspring generations. The first step in the cycle is to evaluate the fitness value of each individual with the **fitness function**. Fitness value should be problem-specific and numerically describe how far an individual is away from being the optimal solution to the problem. In the previous example to find the maximum of function f, f itself is the fitness function.

The *selection* process selects a portion of individuals to breed the offspring generation. According to the "*Survival of the fittest*" principle, individuals with better fitness values are more likely to pass their chromosomes onto the next generation. Therefore, fitter individuals are more likely to be selected. However, to avoid local optima, the selection process cannot be merely ranking and selecting individuals with the best fitness values.

The next step is the *crossover*, depicted in Figure 2.5. A cut-point in the chromosome is randomly picked for each parent individual. Chromosome fragments after the point are swapped among the two parents, resulting in two new offspring individuals. The cut-points are not necessarily the same in two parent chromosomes, and there can be multiple cut-points in one crossover operation.

Followed by one or more *mutation*s, during which the gene at a randomly selected location in the chromosome mutates. Mutation operators are specific to the representation of the chromosome. One example mutation is that an integer input parameter to the component under test is replaced with its inverse code or even a random number.

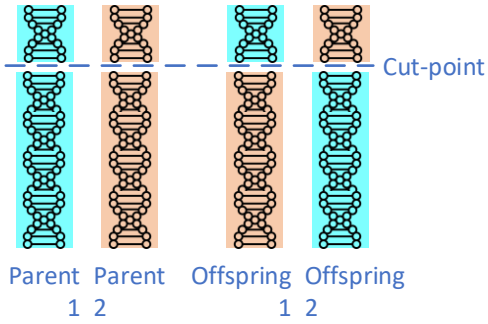


Figure 2.5: Crossover

The *reinsertion* process is there to ensure that the population size in the new generation stays the same as its parent generation. If not enough offspring have been created, the new generation consists of all the offspring individuals and some selected parent individuals.

The algorithm terminates under two conditions: (1) the optimisation goal has been achieved; or (2) the searching budget, such as the number of generations and execution time, has been used up. The optimal individual is the outcome of the genetic algorithm.

Compared with local search approaches, genetic algorithms consider multiple inputs at the same time, and they are sparsely located within the input domain. While the selection progress ensures that the overall specimen is becoming fitter and fitter with each generation, the two genetic operations, crossover and mutation, add extra randomness and variety to the pool to avoid local optima.

Exploration versus Exploitation

Exploration and exploitation are two fundamental concepts in search-based optimisation algorithms. Črepinšek et al. [30] defined them as follows:

> ***Exploration*** *is the process of visiting entirely new regions of a search space, while* ***exploitation*** *is the process of visiting those regions of a search space within the neighbourhood of previously visited points.*

Striking a balance between these two during the search process is of essential importance to the success of any search-based algorithm.

Each component of genetic algorithms is vital in achieving them. Random initialisation is the starting point of exploration as it should be a good sampling of the overall search space. Crossover, with the hypothesis that two fit parents will produce better offspring, exploits existing individuals and creates vastly different new individuals at the same time. Therefore, it contributes to both exploitation and exploration. Mutation mainly contributes to exploitation as the introduced changes are within the nearby area of existing individuals in the search space. The selection process, concerning fitness values, drives the search towards the regions of the best individuals, hence is a means of exploitation.

2.2 Search-Based Software Testing

Search-based optimisation algorithms have been applied to many fields, including software testing. The first work in search-based software testing (SBST) was presented by Miller and Spooner in 1976 [31]. Automated test data generation techniques back then focused on constraint solving and symbolic execution. Their work utilised optimisation algorithms instead. Each group of generated data is evaluated by a cost function (i.e. fitness function). And those that lead the execution *closer* to the desired path result in lower cost values while high-cost data are dropped. Eventually, the search algorithm outputs test data that execute the exact path desired.

Their work later caught widespread attention in the 1990s and inspired researchers to apply algorithms like GA to more aspects and levels of software testing. To adapt it, the most critical two tasks are:

- to find the proper individual chromosome representation that is suitable for the two genetic operations, crossover and mutation;
- to formulate a fitness function that accurately describes how good a solution to the problem is.

Both are problem-specific and following are some examples of the adaptation.

2.2.1 Input Vector Representation

Some components can be easily isolated from the surrounding system and environments. A *pure function*, for example, is a method of which the output only depends on the inputs, and that has no observable side effects. When testing against such components, an individual chromosome can be encoded as an n-dimensional input vector where n is the number of input arguments the component takes. The search domain is the n-dimensional space where the vector is valid for the component.

Crossover Operation

The crossover operation is straightforward swapping corresponding elements of the two parent vectors.

Mutation Operation

The mutation operation is a set of mutation operators, which are different according to the type of the element. For example, it can be $\{++, --, \ll 1, \gg 1\}$ for an integer, or $\{!\}$ for a boolean.

2.2.2 Call Sequence Representation

The input vector representation does not suffice for more complex software testing purposes. For example, when the component under test is a class, it must be tested against its constructors and methods. Each of the constructors and methods is parametrised differently.

Moreover, the sequence of calling them contributes to the variance of a test case as well. Tonella [1] proposed the syntax to represent the test case of a call sequence, see Listing 2.1.

```
1  <chromosome>     ::=     <actions> @ <values>
2  <actions>        ::=     <action> {: <actions>}?
3  <action>         ::=     $id = constructor({<parameters>}?)
4                    |       $id = class # null
5                    |       $id . method ({<parameters>}?)
6  <parameters>     ::=     <parameter> {, <parameters>}?
7  <parameter>      ::=     builtin-type {<generator>}?
8                    |       $id
9  <generator>      ::=     [low; up]
10                   |       [genClass]
11 <values>         ::=     <value> {, <values>}?
12 <value>          ::=     integer
13                   |       real
14                   |       boolean
15                   |       string
```

Listing 2.1: Syntax of chromosomes [1]

A chromosome consists of two parts, the ⟨*actions*⟩ and the ⟨*values*⟩ joined by the "@" sign. The first part ⟨*actions*⟩ is made of a sequence of invocations of constructors or methods, connected by ":". And the second part ⟨*values*⟩ contains the actual values of the arguments to each invocation, separated by ",".

For the first part of the chromosome: **$id** denotes a variable. Therefore an ⟨*action*⟩ can be: (i) to instantiate an object and assign it to a variable (line 3); (ii) to assign *null* to a variable (line 4); or (iii) to call a method on a variable (line 5). Invocations of constructors or methods are parameterised with ⟨*parameters*⟩, and one ⟨*parameter*⟩ can either be a built-in type value (line 7) or a variable (line 8). A built-in type value can be optionally generated from a ⟨*generator*⟩ by either randomly choosing (line 9) within the range [*low*;*up*] or an external class (line 10).

For the second part, each ⟨*value*⟩ belongs to a built-in type, therefore, having the form of an integer, a real number, a boolean value or a constant string. With the defined syntax, the Java code in Listing 2.2 can be represented as a chromosome shown in Listing 2.3.

```
A a = new A();
B b = new B();
b.f(2);
a.m(5, b);
```

Listing 2.2: Example Test Case [1]

```
$a=A():$b=B():$b.f(int):$a.m(int, $b) @ 2,5
```

Listing 2.3: Example Chromosome [1]

For the two genetic operations, we first introduce the mutation operation as some of the mutation operators are re-used in the crossover operation.

Mutation Operation

The following four kinds of mutation operators have been introduced by Tonella [1].

Mutation of Input Value The first mutation operator is to replace a ⟨*value*⟩ with a randomly generated one of the same type. See Listing 2.4 for example, where the *int* value passed to the constructor of class B is changed from 9 to 5.

```
$a=A():$b=B(int):$b.f():$a.m(int,$b) @ 9,7
                                      ⇓
$a=A():$b=B(int):$b.f():$a.m(int,$b) @ 5,7
```

Listing 2.4: Mutation of Input Value [1]

Constructor Change The second mutation operator is to randomly change one of the constructors in an ⟨*action*⟩. The previously used constructor is replaced by one with a different signature. If new values or variables are required, they are inserted accordingly. See Listing 2.5 for example, where the constructor of class B is changed from one that takes an *int* to one that takes a class C object. The instantiation action of variable *c* is inserted, and the value 1 is dropped.

```
$a=A():$b=B(int)       :$b.f():$a.m(int, $b) @ 1,5
       ⇓                                      ⇓
$a=A():$c=C():$b=B($c):$b.f():$a.m(int ,$b) @   5
```

Listing 2.5: Constructor Change [1]

Insertion of Method Invocations The third mutation operator is to insert new ⟨*action*⟩s of method invocations of existing variables. See Listing 2.6 for example. Variable *b* is randomly chosen, and method invocation $b.g(int)$ is inserted. As $b.g(int)$ takes an *int* parameter, integer 4 is inserted to the values at the corresponding location.

```
$a=A():$b=B(int):          $b.f():$a.m(int, $b) @ 1  ,5
                 ⇓                                 ⇓
$a=A():$b=B(int):$b.g(int):$b.f():$a.m(int, $b) @ 1,4,5
```

Listing 2.6: Insertion of Method Invocations [1]

Removal of Method Invocations The last mutation operator is to randomly select and remove ⟨*action*⟩s. In the example of Listing 2.7, the action $\$b.h(\$c)$ is randomly chosen to be removed. As this is the only usage of variable c, it is removed as well as its initial value 4.

```
$a=A():$b=B(int):$c=C(int):$b.h($c):$b.f():$a.m(int, $b) @ 1,4,5
                  ⇓                                      ⇓
$a=A():$b=B(int):                    $b.f():$a.m(int, $b) @ 1  ,5
```

Listing 2.7: Removal of Method Invocations [1]

Crossover Operation

The general idea of a crossover is to randomly pick a cut-point (see dashed lines between line 2 and line 3 in Figure 2.6) and then to swap the fragments after the point. Notice that in the chromosome representation, the crossover should be done in both the ⟨*actions*⟩ and the ⟨*values*⟩ parts at corresponding locations.

```
1 A a = new A();
2 B b = new B(1);
  -----------------
3 b.f();
4 a.m(5, b);
```

(a) Parent 1

```
A a = new A(0, 3);
B b = new B();
------------------
b.g();
a.m(4, b);
```

(b) Parent 2

```
1 A a = new A();
2 B b = new B(1);
3 b.g();
4 a.m(4, b);
```

(c) Offspring 1

```
A a = new A(0, 3);
B b = new B();
b.f();
a.m(5, b);
```

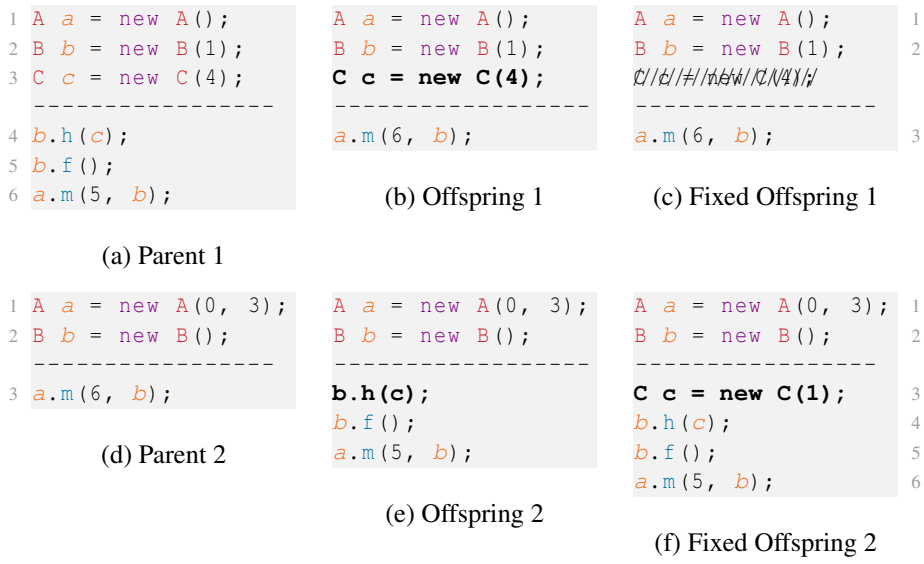
(d) Offspring 2

Figure 2.6: Crossover Operation Example [1]

The crossover shown in Figure 2.6 is straightforward as after the operation, all variables involved have been defined before their first usage, no unused variable exists, and there is no conflict among variables. This is, however, not often the case because cut-points are randomly picked, and the call sequences of two parent chromosomes are of variable-length.

Take the two offspring, (b) and (e), in Figure 2.7 for example. After the crossover operation, the variable c (line 3) is never used in offspring 1. Meanwhile, in offspring 2 at line 3, an undeclared variable c is referred to as the argument in the method invocation $b.h()$.

To resolve this, a specific clean-up needs to be done after the crossover operation. With the already defined removal mutation operator, the redundant actions related to the variable c in offspring 1 can be removed. And with the insertion mutation operator, a necessary



(a) Parent 1 (b) Offspring 1 (c) Fixed Offspring 1

(d) Parent 2 (e) Offspring 2 (f) Fixed Offspring 2

Figure 2.7: Malformed Crossover Offsprings [1]

constructor invocation of the variable c can be inserted to offspring 2. See the last column of Figure 2.7 for examples of the fixed offspring. Notice that it is not simply moving the instantiation action from offspring 1 to offspring 2 as the initial value of the variable c in offspring 2 is newly randomly generated.

2.2.3 Fitness Functions

Fitness functions should be of high specificity to the characteristics of the problem to be solved. In this sub-section, we introduce a few different types of software testing and their corresponding fitness functions when to be solved with genetic algorithms.

Temporal Testing

Temporal testing against a component aims at finding the worst-case and best-case execution times (i.e. WCET and BCET respectively). The optimisation goal is to achieve the maximal execution time for WCET and minimal execution time for BCET. Intuitively, the fitness value is the execution time of the component under test.

Functional Testing

Functional testing is a type of *black-box testing*, i.e. the internal implementation of the component under test is not given. Focusing on testing the functionality, the optimisation goal is often to generate input vectors that make the component malfunction, and the fitness function describes how far it is away from malfunctioning.

Buehler and Wegener [32] present an application of SBST to test a standalone controller in an automated car parking system. A parking scenario is an individual chromosome in this

case, which is represented as a series of geometric data describing what area of the parking lot the car can enter. The search-based algorithm generates parking scenarios with the optimisation goal of leading the system to collide into the parking scene. The fitness value is the shortest distance to collision points during the parking process.

Structural Testing

Unlike the previous type, structural testing is a kind of *white-box testing*, which means the details of the internal implementation are available. Moreover, the internal structures and control flows within a component are exactly what structural testing is interested in, with the aim to cover a particular part of the component or achieve maximal overall coverage.

It is also the area that has attracted the most attention when it comes to search-based software testing. Previous researchers have developed fitness functions for branch coverage, path coverage and data flow coverage [33]. We introduce the two metrics, *approach level* and *branch distance*, and the popular fitness function for search-based structural testing that uses these two.

Take the `Triangle` class in Listing 2.8 for example [34]. It is a simple class containing four fields, with `a`, `b` and `c` denoting the lengths of the three sides and `type` storing the type of the triangle. The only method of the class is `checkTriangleType()`, which checks whether the triangle is *equilateral*, *isosceles*, or *scalene*. The control flow graph (CFG) of that method is depicted in Figure 2.8. Each `if`-statement corresponds to a diamond node in the CFG, and such nodes are called *control dependent node*s as the flow of the execution departs there into separate paths.

If the coverage target is to reach line 10, by combining the line number of `if`-statements it visits to reach there, the target path t can be denoted as $p_t = \langle 6,7,9,10 \rangle$. Consider the four random generated test cases shown in Table 2.1.

Table 2.1: Test Cases

Test Case	Type	Path Notation
$x_1 = (5,5,5)$	Equilateral	$p_{x_1} = \langle 6,13,14 \rangle$
$x_2 = (5,6,5)$	Isosceles	$p_{x_2} = \langle 6,7,8 \rangle$
$x_3 = (5,6,4)$	Scalene	$p_{x_3} = \langle 6,7,9,12 \rangle$
$x_4 = (5,6,3)$	Scalene	$p_{x_4} = \langle 6,7,9,12 \rangle$
t	Isosceles	$p_t = \langle 6,7,9,10 \rangle$

Intuitively, $path_{x_3}$ and $path_{x_4}$ are closer to the target path t as they cover the deepest `if`-statement at line 9, while $path_{x_1}$ is further away from t as it covers only the very first `if`-statement at line 6. The approach level is defined to numerically describe this distance:

Definition 1 (Approach Level)**.** *Given the execution path p_x obtained by running the program with the generated test case x, the approach level $AL(p_x)$ is the minimum number of control dependent nodes between an executed statement and the coverage target t [33].*

```
1  class Triangle {
2    int a, b, c; // sides
3    String type = null;
4
5    void checkTriangleType() {
6      if (a != b)
7        if (a == c)
8          type = "Isosceles";
9        else if (b == c)
10         type = "Isosceles";
11       else
12         type = "Scalene";
13     else if (b == c)
14       type = "Equilateral";
15     else
16       type = "Isosceles";
17   }
18 }
```

Listing 2.8: The `Triangle` class [34]

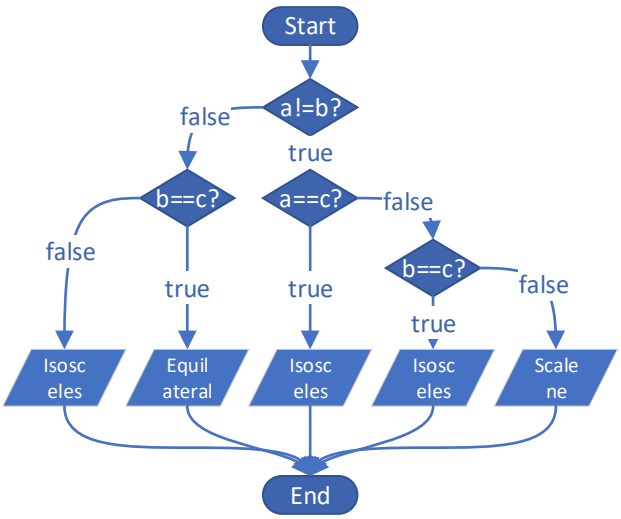


Figure 2.8: Control Flow Graph of `checkTriangleType()` Method in the `Triangle` Class

Table 2.2: Approach Levels of the Test Cases

Test Case	x_1	x_2	x_3	x_4
Approach Level	2	1	0	0

With the above definition, we have the approach levels for each test case listed in Table 2.2. However, we still need to decide whether test case x_3 or x_4 is closer to the target path t. Hence the branch distance is defined:

Definition 2 (Branch Distance)**.** *Given the first control dependent node where the execution p_x diverges from the target t, the predicate at such node is converted to a distance $d()$ (from taking the desired branch), normalised between 0 and 1, therefore, the branch distance $BD(p_x)$ [33]:*

$$BD(p_x) = norm(d(p_x))$$

For the computation of $d()$, different distance formulae are applied, depending on the relational operator of the control dependent node. Tracey et al. [2] presented a full list of formulae for different relational operators, see Table 2.3.

Table 2.3: Distance Formulae [2]

Relational Operator	**Distance** $d()$
$boolean$	0 if $true$; K if $false$
$a = b$	0 if $a = b$; otherwise $abs(a-b)+K$
$a \neq b$	0 if $a \neq b$; otherwise K
$a < b$	0 if $a < b$; otherwise $a-b+K$
$a \leqslant b$	0 if $a \leqslant b$; otherwise $a-b+K$
$a > b$	0 if $a > b$; otherwise $b-a+K$
$a \geqslant b$	0 if $a \geqslant b$; otherwise $b-a+K$
$a \vee b$	$\min(d(a), d(b))$
$a \wedge b$	$d(a)+d(b)$
$\neg a$	negation is propagated inside a

Definition 3 (Fitness Function for Structural Testing)**.** *With the two metrics, approach level and branch distance, defined, the widely used fitness function for statement coverage in search-based structural testing is defined as follows [33]:*

$$f(x) = AL(p_x) + BD(p_x)$$

With the fitness function, Table 2.2 can be updated with additional values. For normalisation, Arcuri [35] evaluated different normalisation methods and suggested to use $norm(a) = \frac{a}{a+1}$, which is what has been used to calculate the branch distance in Table 2.4.

As can be seen, test case x_3 is closer to reach the target t and therefore, it is favoured among the other cases in the selection process of GA to produce offspring.

Table 2.4: Fitness Values of the Test Cases

Test Case	x_1	x_2	x_3	x_4
Approach Level	2	1	0	0
Branch Distance	0.50	0.50	0.67	0.75
Fitness Value	2.50	1.50	0.67	0.75

2.2.4 EvoSuite

EvoSuite[1] is a popular SBST tool that automatically generates test suites for Java programs, proposed by Fraser and Arcuri [27]. It is also one of the bedrocks of this thesis project. Several characteristics of EvoSuite make it stand out among other tools, and they are discussed here.

Whole Test Suite Optimisation

A typical SBST generation tool for white-box testing generates test cases to satisfy one single coverage goal at a time (e.g., to cover a particular program branch or to follow a specific control flow path). This strategy is, however, flawed as not all goals are equally difficult or important to reach, and some are dependent on each other.

Instead, EvoSuite works on the level of a whole test suite, where the individual chromosome is a collection of test cases. The crossover operation is hence to exchange randomly chosen test cases among the two parents, and the mutation operation is to add, remove or alter single test cases. The search-based optimisation process integrates state-of-the-art techniques like hybrid search and dynamic symbolic execution to generate appropriate data and improve the generations through evolution. The individual chromosome fitness is evaluated against a set of coverage criteria instead of single coverage goals.

Mutation-Based Assertion Generation

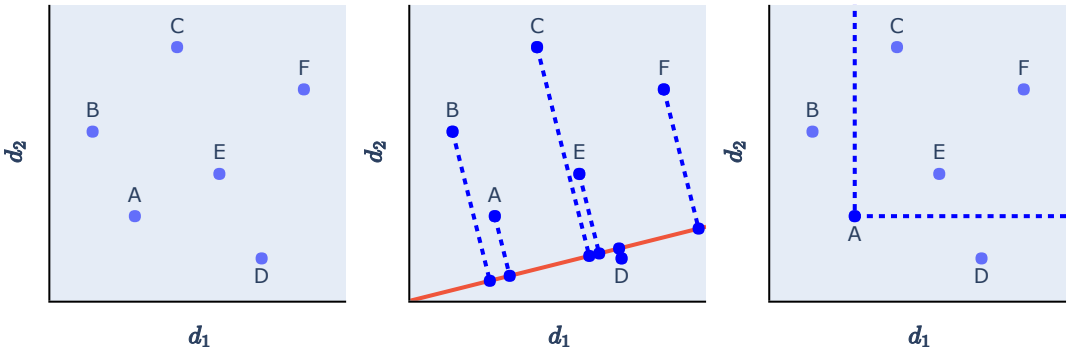
An automated oracle can easily detect faults in the program when executing the test cases if they lead to crashes, deadlocks, or violate some formal specifications. It, however, fails to capture the functional incorrectness when the program runs smoothly. It is feasible to generate assertions to the program as manual oracles, but it is hard to control the proper amount of assertions so that they do not over-specify the test case.

EvoSuite makes use of mutation testing to determine the importance and effectiveness of an assertion. In mutation testing, mutants, which are artificial defects, are seeded into the program, and a test case is evaluated whether it can detect the mutants from the original program. On the one hand, if a mutant stays undetected, it means that the test suite is not thorough enough and that new test cases should be added, or there should be better assertions for existing ones. On the other hand, if an assertion captures no mutants, it is probably irrelevant to the test case and therefore, can be removed.

[1] `http://www.evosuite.org`

Many-Objective Sorting Algorithms

With the whole test suite strategy, the different coverage criteria are still integrated into a single fitness function to evaluate the fitness of individuals. Previous works in the field of numerical optimisation [36] have shown that reformulating a single-objective optimisation problem into a many-objective one reduces the probability for the search process to be stuck at a local optimum and boosts the efficiency.



(a) Individuals (b) Weighted Sum (c) Many-Objective

Figure 2.9: Weighted Sum versus Many-Objective

Imagine an optimisation task with two distance metrics d_1 and d_2. Distance values of six example individuals are shown in Figure 2.9a. To evaluate these individuals with a weighted sum of d_1 and d_2 is equivalent to project the points onto a line according to the weights, shown in Figure 2.9b. Points, the projections of which are closer to the origin, result in better fitness values. In the example, individual B is the fittest. Individual D is considered poorly fit, even though it has the best d_2 value. The way different distance metrics are integrated plays a huge role and sometimes may cause diversity loss.

In many-objective algorithms, an individual t_1 is said to *dominate* another individual t_2 if and only if: (i) for all objectives, t_1 results in no worse distance values than t_2; and (ii) for at least one objective, t_1 has a better distance value than t_2. On the other hand, t_2 is called a *dominated* individual. If an individual gets 0 for all distance metrics, it is called the *optimal* individual. While the end goal is still to find the optimal individual, many-objective algorithms treat all *non-dominated* individuals as equally fit. In the example shown in Figure 2.9c, A, B and D are therefore the non-dominated individuals, and they have an equal chance to pass their chromosomes onto the next generation.

EvoSuite provides a many-objective GA named Many-Objective Sorting Algorithm (MOSA), proposed by Panichella et al. [37]. Viewing all the branch distances in the class under test (CUT) as the many objectives to optimise, it is highly scalable and can handle hundreds of objectives at the same time. Panichella et al. further proposed DynaMOSA, the improved version of MOSA that dynamically selects optimisation objectives based on the control dependency hierarchy [38]. With MOSA, all objectives are considered since the initial generation. However, some latter objectives can only be satisfied if certain precedent

objectives have already been satisfied. DynaMOSA tackles this and has seen improvements in coverage and especially efficiency, when given a limited search budget.

Java Bytecode Instrumentation Infrastructure

Built upon the ASM [2] library, EvoSuite provides a handy instrumentation framework to extend for our study. With customisable `ClassVisitor`s and `MethodVisitor`s, the CUT can be instrumented on the bytecode level easily. And EvoSuite provides a centralised singleton `ExecutionTracer` to store all the run-time information retrieved by the instrumentation when executing one generated test case.

2.3 Automated Crash Reproduction for Debugging

Proper testing and debugging are essential to developing reliable software [39]. The evolution of integrated development environments (IDEs) has made it easy for developers to spot predetermined errors, like missing `';'` at the end of a Java statement or undefined variables, that are language-specific before compilation. There have also been empirical studies on how to use debugging tools like debuggers, assertions [40, 41], and breakpoints [42] better to locate the bugs more efficiently.

However, in order to use these tools, one first needs to be able to reproduce the crash so that it can be observed, and the fix can be verified afterwards. Researchers have studied ways to automate crash reproduction, and one novel direction is to utilise SBST technologies. They are discussed in this section.

2.3.1 Different Approaches

Approaches for automated crash reproduction can be grouped into two main categories: (1) *record-replay* approaches, and (2) *post-failure* approaches.

Record-replay approaches record run-time execution data of the software program. When a crash happens, they use the recorded data to identify the states and the execution path of the program that lead to the crash. Earliest work in crash production, like jRapture [11], Bugnet [12] and Saff et al. [13], and later research [14, 15, 16] all fall into this category. In order to record the run-time data, the program is instrumented, and therefore, it implies performance overhead and may raise privacy concerns of the user.

Post-failure approaches, on the other hand, try to reproduce the crash by using data derived from the crash itself or provided by the operating system. As those data are all available after the crash, there is no need to instrument the program that is delivered to the user [20, 21]. STAR [22] takes only the stack trace and applies backwards symbolic execution for the computation of crash conditions. Some researchers also make use of SBST techniques to generate test cases for crash reproduction. For example, Rößler et al. proposed RECORE [23], which generates method sequences with information from core dumps with the help of search-based algorithms.

[2] `https://asm.ow2.io/`

2.3.2 EvoCrash

EvoCrash [25] is a post-failure crash reproduction framework, extending EvoSuite. It addresses some challenges that previous post-failure approaches have been facing, like path explosion and inability to handle environmental dependencies. See Figure 2.10 for the overview of EvoCrash.

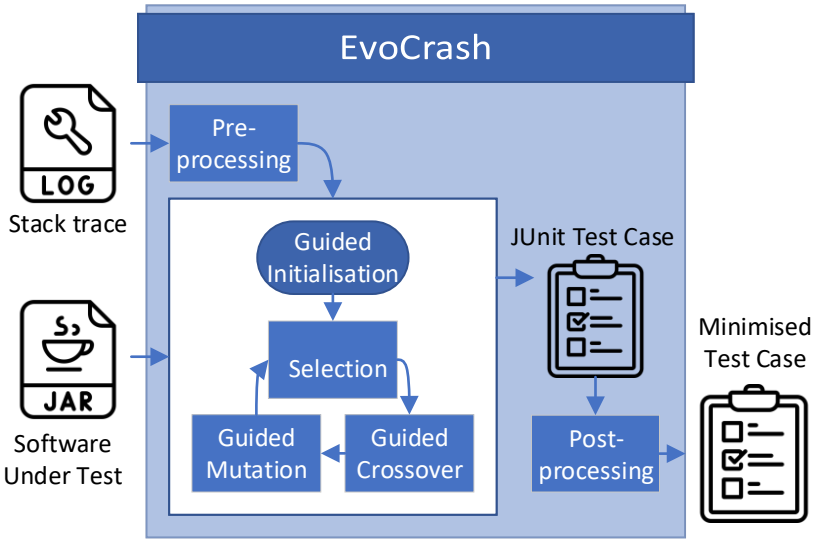


Figure 2.10: Overview of EvoCrash

The input to EvoCrash contains a stack trace generated by the crash itself and all the related jar files of the original software under test (SUT). After pre-processing of the stack trace, it utilises a novel genetic algorithm called Guided Genetic Algorithm (GGA). When the crash is reproduced, the GGA outputs a JUnit test case. Afterwards, the post-processing optimises the test case. The final output of the whole process is a concise executable JUnit test case that triggers the crash for the engineer, or an `info` message if it fails to converge with the given search budget.

Empirical evaluations showed that the GGA in EvoCrash addresses the path explosion problem and the mocking strategies in EvoSuite help to partially tackle the environment dependency challenge [26]. Each component in the framework is tailored for crash reproduction, and they are explained in this sub-section.

Pre-Processing

As shown in Listing 2.9, a typical Java crash stack trace contains the type of thrown exception (line 1), and a list of frames in the stack at the time of the crash (line 2 to 11). While each frame tells the class name, method name and the line number where the exception is propagated, the inner-most frame (line 2) is where the exception is thrown.

The pre-precessing of EvoCrash notes down the classes and methods in the stack trace as its “interests”. The inner-most frame is the devoted target frame hereafter, and the class

```
java.lang.ArrayIndexOutOfBoundsException: 4
  at org.apache.commons.lang3.time.FastDateParser.toArray(FastDateParser.
    java:413)
  at org.apache.commons.lang3.time.FastDateParser.getDisplayNames(
    FastDateParser.java:381)
  at org.apache.commons.lang3.time.FastDateParser$TextStrategy.addRegex(
    FastDateParser.java:664)
  at org.apache.commons.lang3.time.FastDateParser.init(FastDateParser.
    java:138)
  at org.apache.commons.lang3.time.FastDateParser.<init>(FastDateParser.
    java:108)
  at org.apache.commons.lang3.time.FastDateFormat.<init>(FastDateFormat.
    java:370)
  at org.apache.commons.lang3.time.FastDateFormat$1.createInstance(
    FastDateFormat.java:91)
  at org.apache.commons.lang3.time.FastDateFormat$1.createInstance(
    FastDateFormat.java:88)
  at org.apache.commons.lang3.time.FormatCache.getInstance(FormatCache.
    java:82)
  at org.apache.commons.lang3.time.FastDateFormat.getInstance(
    FastDateFormat.java:165)
```

Listing 2.9: Crash Stack Trace Example (LANG-9b from JCrashPack [3])

from it is the main CUT to be instrumented later.

Fitness Function

To numerically evaluate how good a generated test case is, EvoCrash defines its fitness function based on three conditions: (1) the line number in the target frame should be covered; (2) an exception of the same type with the target exception should be thrown; and (3) the generated stack trace should be as similar to the given one as possible.

The two well-established metrics, approach level and branch distance introduced in Definition 1 and 2, are used to represent the **line distance** $d_{line}(x)$ of a generated test case x for the first condition. For the second condition, the **exception distance** $d_{exception}(x)$ is defined in Definition 5.

Definition 4 (Line Distance)**.**

$$d_{line}(x) = AL(x) + BD(x)$$

Definition 5 (Exception Distance)**.** *$d_{exception}(x)$ is a binary value as it is zero if and only if the given exception is thrown, otherwise it is one:*

$$d_{exception}(x) = \begin{cases} 0 & \textit{if the target exception is thrown;} \\ 1 & \textit{otherwise.} \end{cases}$$

For the third condition, to calculate the trace distance is a bit more complicated. A stack trace frame is denoted as $e_i = (C_i, m_i, l_i)$ where i is the frame level, C_i is the class name of

that level, m_i the method name and l_i the line number. First, we have the frame distance defined in Definition 6. Thereafter, the **trace distance** $d_{trace}(x)$ is defined in Definition 7.

Definition 6 (Frame Distance)**.** *The distance between an actual frame e_j and a frame from the target stack trace e_i^* is:*

$$diff(e_i^*, e_j) = \begin{cases} 3 & \text{if } C_i^* \neq C_j; \\ 2 & \text{if } C_i^* = C_j \text{ and } m_i^* \neq m_j; \\ \varphi\left(\left|l_i^* - l_j\right|\right) \in [0,1] & \text{otherwise.} \end{cases}$$

where $\varphi()$ is the often used normalisation function $\varphi(x) = \frac{x}{x+1}$.

Definition 7 (Trace Distance)**.** *The distance between the generated stack trace S and the target stack trace S^* is:*

$$d_{trace}(x) = \sum_{i=1}^{n} min\left\{diff(e_i^*, e_j) : e_j \in S\right\}$$

where n is the number of frames in S^* and e_j is the closest frame in S to e_i^* from S^*.

Definition 8 (Weighted Sum Fitness Function)**.** *The final fitness function of a test case x is a weighted sum of the norms of the aforementioned three distances:*

$$f(x) = \omega_1 \varphi(d_{line}(x)) + \omega_2 d_{exception}(x) + \omega_3 \varphi(d_{trace}(x))$$

where $\langle \omega_1, \omega_2, \omega_3 \rangle$ are the weights. In the implementation, the weight vector is set to $\langle 3,2,1 \rangle$. Therefore, the fitness function can be broken down into:

$$f(x) = \begin{cases} 3 \times \varphi(d_{line}(x)) + 2 \times 1 + 1 & \text{if the target line is not reached;} \\ 3 \times 0 + 2 \times d_{exception}(x) + 1 & \text{if the target line is reached;} \\ 3 \times 0 + 2 \times 0 + \varphi(d_{trace}(x)) & \text{if the target exception is thrown.} \end{cases}$$

Distance $d_{exception}(x)$ is not normalised as we already have $d_{exception}(x) \in \{0,1\}$.

Guided Genetic Algorithm

Unlike traditional genetic algorithms used in SBST that target all methods in the CUT, the GGA used by EvoCrash is modified in such a way that it prioritises methods involved in the target crash. The idea is to create test cases that always exercise the target method in order to increase the chance of leading to the target crash.

Initial Population The main difference in initialisation between EvoSuite and EvoCrash is that the former tries to maximise the number of methods from the CUT being covered, while the latter tries to include at least one invocation of the target method in all test cases. If the target method is `public` or `protected`, it is ensured to be called at least once in each test case in the initial population. However, if it is `private`, EvoCrash makes sure to indirectly call it at least once in each test case in the initial population. Non-target methods are invoked based on a probability or if they help to decrease the fitness value.

Guided Crossover With the traditional crossover operation, there is a possibility that one of the offspring may lose the invocation to the target method due to the randomness in choosing the cut-point. If this indeed happens after the crossover operation, the GGA drops the offspring with no target method invocation and substitute it with a copy of one of the parent test cases. Necessary constructor invocations are inserted so that the offspring test case is well-formed.

Guided Mutation Similar to the crossover operation, an offspring test case may lose the invocation to the target method while mutating as well, specifically with the *removal-of-method-invocations* operator. When that indeed happens, the GGA keeps mutating the said offspring until at least one target method invocation is re-inserted back into the test case.

2.3.3 JCrashPack

Soltani et al. [3] proposed a benchmark named JCrashPack[3] for evaluating search-based crash reproduction techniques. The benchmark consists of selected size-varying software applications with openly accessible binaries, source code and actively maintained issue tracers. The authors have collected 200 stack traces of crashes from 7 real-world Java applications, including *Apache Commons Lang*[4], *Apache Commons Math*[5], *mockito*[6], *Joda Time*[7], *JFreeChart*[8], *XWiki*[9], and *Elasticsearch*[10]. See Table 2.5a for the number of crashes from each software application and Table 2.5b for the number of crashes of each Java exception type.

Table 2.5: Crash Cases Composition of JCrashPack [3]

(a) versus Software Projects

Application	Crashes
commons-lang	22
commons-math	27
mockito	14
Joda Time	8
JFreeChart	2
XWiki	51
Elasticsearch	76
Total	200

(b) versus Exception Types

Exception Type	Crashes
`NullPointer`	49
`IllegalArgument`	24
`ArrayIndexOutOfBounds`	14
`ClassCast`	10
`StringIndexOutOfBounds`	9
`IllegalState`	7
Other	87
Total	200

[3] `https://github.com/STAMP-project/JCrashPack`
[4] `https://commons.apache.org/lang/`
[5] `https://commons.apache.org/math/`
[6] `https://site.mockito.org/`
[7] `https://www.joda.org/joda-time/`
[8] `http://www.jfree.org/jfreechart/`
[9] `https://www.xwiki.org/`
[10] `https://www.elastic.co/`

"Other" in Table 2.5b includes exceptions that only occur very few times or customised exceptions defined by the software applications.

2.4 Botsing

When EvoCrash was first implemented, it contained a full clone of EvoSuite. EvoSuite itself as a popular open-source project has many active contributors and has been updated frequently. Due to the fact that EvoCrash was deeply dependent on EvoSuite, it became difficult to maintain and update EvoCrash, and it was then deprecated [43].

The follow-up development of EvoCrash moved towards Botsing[11]. *Botsing* means "crash" in Dutch. While it fully re-implements the functionality of EvoCrash, it only takes EvoSuite as a maven dependency. Additionally, Botsing comes with an extensive test suite to further ease future developments and expansion. Furthermore, Botsing has also brought new techniques to the original EvoCrash, two of which are the *integration testing* fitness function and *many-objective optimisation*. They are discussed in this section.

Limitations of the Weighted Sum Fitness Function

The weighted sum fitness function (see Definition 8) from EvoCrash views the stack trace from a top-down perspective. Giving d_{line} the highest weight 3, it prioritises on making sure that the line number of the target frame is covered. Then $d_{exception}$ is given the weight 2 to focus on throwing the target exception. And eventually, it checks the similarity between the generated stack trace and the target stack trace. A huge penalty is given to the candidate when the first two conditions are not covered. Moreover, EvoCrash only treats the class in the target frame as the CUT, and only instruments that class.

```
SomeTypeOfException:
  at ClassA.method1(ClassA.java:435)
  at ClassB.method2(ClassB.java:156)
  at ClassC.method3(ClassC.java:851)
```

Listing 2.10: Dummy Stack Trace

Imagine a task to reproduce the third frame from the dummy stack trace shown in Listing 2.10, and in the source code, `method3` can invoke `method1` both directly or indirectly via `method2`. Both `method1` and `method4` have the potential to throw `SomeTypeOfException`.

```
SomeTypeOfException:
  at ClassA.method4(ClassA.java:652)
  at ClassC.method3(ClassC.java:851)
```

Listing 2.11: Resulting Stack Trace of x_1

```
method1:422, ClassA
method2:156, ClassB
method3:851, ClassC
```

Listing 2.12: Program Stack of x_2

[11] https://stamp-project.github.io/botsing/

Dummy test case x_1 throws the exception, the resulting stack trace is shown in Listing 2.11. While dummy test case x_2 does not throw any exception, shown in Listing 2.12 is the program stack.

With the weighted sum fitness function, x_1 is favoured as $d_{line}(x_1) = d_{exception}(x_1) = 0$, while x_2 might be dropped because no exception is thrown. However, x_2 is able to invoke `method1` via the desired path and should be considered more likely to reproduce the crash. Therefore, the search process is possibly misled to a local optimum. In essence, the nested nature of stack traces should have been respected throughout the search process.

2.4.1 Integration Testing

Inspired by RECORE [23], Botsing provides an integration testing scheme, which views the stack trace bottom-up. Classes from all frames inner to the target frame are treated as CUTs and instrumented. The integration testing fitness function gives the highest priority to reconstruct the stack trace frame by frame with its different definition of the **trace distance**:

Definition 9 (Integration Testing Trace Distance)**.**

$$d^*_{trace}(x) = |S^*| - |lcp| + \varphi(d_{line}(x, r_{lcp}))$$

where lcp is the longest common prefix between S and S^*, $|S^*| - |lcp|$ means the number of stack trace frames yet to reach. r_{lcp} denotes the first diverging stack trace frame, and $d_{line}(x, r_{lcp})$ is the line distance for test case x to reach the line number of that frame.

Definition 10 (Integration Testing Fitness Function)**.**

$$f(x) = d^*_{trace}(x) + d_{exception}(x)$$

The search process thus strives to maximise stack trace similarity, starting from the outermost frame. The definition can also be broken down into:

$$f(x) = \begin{cases} n + d_{line_n}(x) & \text{if the line number of the n-th frame is not covered;} \\ d_{exception}(x) & \text{if the line number of all frames are covered.} \end{cases}$$

where n is the level of the current frame and $n \in [1, total\_number\_of\_frames]$.

```
method1:435, ClassA
method2:156, ClassB
method3:851, ClassC
```

Listing 2.13: Program Stack of x_3

```
SomeTypeOfException:
  at ClassA.method1(ClassA.java:435)
  at ClassB.method2(ClassB.java:156)
  at ClassC.method3(ClassC.java:851)
```

Listing 2.14: Stack Trace of x_4

Consider two additional dummy cases x_3 and x_4 shown in Listing 2.13 and 2.14. Notice that x_3 does not throw any exception. Now with the integration testing fitness function, we have $f(x_1) \in (2,3)$ as the second frame is not covered, and $f(x_2) \in (1,2]$ as the second

frame is covered but not the first one. We have $f(x_3) = 1$ because it covers all three frames but throws no exception. And $f(x_4) = 0$ as all three frames are covered and the same type of exception is thrown.

In this way, the search process respects the nested nature of the stack trace, and fitness values decrease more smoothly.

2.4.2 Many-Objective Optimisation

Soltani et al. [44] applied a multi-optimisation technique for evolutionary crash reproduction. Instead of summing up d_{line}, $d_{exception}$, and d_{trace} with weights, each of them is set to be an objective of NSGA-II [45]. Their experiments show that by modifying the fitness function and loosing the constraints, it improves the efficiency, especially when reproducing non-trivial crashes which take several generations of evolution [44]. The limitation of NSGA-II is that it is not effective when solving problems with more than three objectives. Therefore, Botsing implemented the guided version of the MOSA in EvoSuite in addition to the original GGA from EvoCrash.

2.5 Limitation

We have stated how important the fitness function is to the applications of genetic algorithms. However, in existing search-based crash reproduction techniques, the exploitation power of the genetic algorithm is shackled by the underlying fitness function. Used in both the weighted sum fitness function and the integration testing fitness function, the **exception distance** $d_{exception}$ (Definition 5) is a binary value. When the fitness value reaches 3.0 in the case of weighted sum, or 1.0 in the case of integration testing, the search process has to wander around a plateau without further guidance to decrease the fitness value.

To address the limitation and to adjust and improve the fitness function, our study is carried out with the following steps:

1. Studying how common run-time exceptions in Java are triggered;
2. Defining distances for different kinds of exceptions to break the plateau; and
3. Implementing additional instrumentation to retrieve information necessary to the computation of the distances and thereafter the new fitness functions.

Chapter 3

Common Runtime Exceptions

There are numerous types of Java exceptions defined in the JDK. The package java.lang alone has seventeen types of RuntimeExceptions already, making it infeasible to specialise fitness functions against each one of them within this thesis project.

However, some types are more ubiquitous in Java programs. Soltani et al. [26] summarised that the most commonly examined exception types in automatic crash reproduction studies are: (1) NullPointer, 74%; (2) ArrayIndexOutOfBounds, 9%; (3) IllegalState and IllegalArgument, 3%. A survey about debugging Java applications on the Android platform also confirms that these types, along with ClassCast exception, have been reported more often than others [46].

Considering the stack traces provided in JCrashPack (see Table 2.5b), we have decided to focus on ArrayIndexOutOfBoundsExceptions, StringIndexOutOfBoundsExceptions, IllegalArgumentExceptions and IllegalStateExceptions. How each one of them is typically thrown is discussed in this chapter. And the reason why NullPointerExceptions are not included is explained at the end.

3.1 ArrayIndexOutOfBoundsException

Arrays are stored as a series of consecutive bytes in memory. Each element can be accessed with the address of the start of the array, an index indicating the offset, and the type of stored elements. In languages like C, malicious parties can exploit the index to access any address in the memory, raising safety concerns.

The Java virtual machine (JVM) performs boundary checks for array accesses at runtime to intercept any illegal memory access to ensure the security and functionality of the program. The lower bound of an array is always zero, and the upper bound is the length of the array minus one. If the index falls out of this range, an ArrayIndexOutOfBoundsException is thrown. In the JDK documentation, it is defined as follows:

Documentation (ArrayIndexOutOfBoundsException)**.** *Thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array [47].*

In Java, arrays are a special type of built-in data structure, for which there are bytecode instructions in the JVM to directly operate on them. Accessing an element of an array happens on the bytecode level. Therefore, the exception is mainly thrown by the JVM itself, and seldomly thrown explicitly from the source code.

Two sets of bytecode instructions are involved. One set of instructions read an element from an array and push its value onto the top of the operand stack. And the other set of instructions store a value that is on the top of the operand stack to an array. Within each set, there is a unique instruction for each primitive type (see Table 3.1, each instruction is followed by its hexadecimal code).

3.1.1 Loading

Consider the example of loading an array element in Figure 3.1. Java source code in Figure 3.1a has been transcompiled into bytecode in Figure 3.1c. At the time when `IALOAD` is called (line 11 of the bytecode), there are two relevant values in the operand stack. The lower one is the result of `ALOAD 1`, which is the reference to the local variable `array`. The upper one is the integer constant 5, which is the index of the element we are trying to access as indicated in line 3 of the source code.

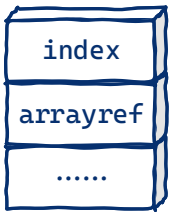


Figure 3.2: Operand Stack When Loading

3.1.2 Storing

The bytecode for storing an element in an array is very similar to loading, as shown in Figure 3.1b and 3.1d. The only difference is that now there are three relevant values in the operand stack when calling `IASTORE` (line 12 of the bytecode). The lower one `ALOAD 1` is the reference to the local variable `array`. The middle one `ICONST_M1` is the index `-1` that is indicated in line 3 of the source code. The upper one `BIPUSH 7` is the value of the element to be stored in the array.

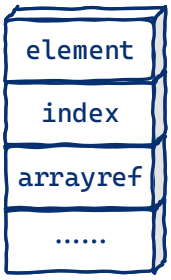


Figure 3.3: Operand Stack When Storing

If $0 \leqslant$ `index` $<$ `length` stands, the `index` is legal. Otherwise, the exception is thrown. The *exception distance* can, therefore, be defined to reflect how far the `index` is away from `0` or the `length`.

Table 3.1: Bytecode Instructions for Array Access

Type	Loading		Storing	
int	IALOAD	2E	IASTORE	4F
long	LALOAD	2F	LASTORE	50
float	FALOAD	30	FASTORE	51
double	DALOAD	31	DASTORE	52
reference	AALOAD	32	AASTORE	53
boolean	BALOAD	33	BASTORE	54
char	CALOAD	34	CASTORE	55
short	SALOAD	35	SASTORE	56

```java
void loadingExample() {
  int[] array = new int[5];
  int i = array[5];
}
```

(a) Loading Example (Source Code)

```java
void storingExample() {
  int[] array = new int[5];
  array[-1] = 7;
}
```

(b) Storing Example (Source Code)

```
loadingExample()V
  L0
    LINENUMBER 2 L0
    ICONST_5
    NEWARRAY T_INT
    ASTORE 1
  L1
    LINENUMBER 3 L1
    ALOAD 1
    ICONST_5
    IALOAD
    ISTORE 2
  L2
    LINENUMBER 4 L2
    RETURN
  L3
    LOCALVARIABLE array [I L1 L3 1
    LOCALVARIABLE i     I  L2 L3 2
    MAXSTACK = 2
    MAXLOCALS = 2
```

(c) Loading Example (Bytecode)

```
storingExample()V
  L0
    LINENUMBER 2 L0
    ICONST_5
    NEWARRAY T_INT
    ASTORE 1
  L1
    LINENUMBER 3 L1
    ALOAD 1
    ICONST_M1
    BIPUSH 7
    IASTORE
  L2
    LINENUMBER 4 L2
    RETURN
  L3
    LOCALVARIABLE array [I L1 L3 1
    MAXSTACK = 3
    MAXLOCALS = 1
```

(d) Storing Example (Bytecode)

Figure 3.1: Accessing Array Elements in Java

3.2 StringIndexOutOfBoundsException

In Java, strings are represented as instances of the `String` class. The class wraps several fields and utility methods around a `char` array. For the same reason mentioned at the beginning of the previous section, any illegal access to strings should not be allowed either. Therefore a `StringIndexOutOfBoundsException` is thrown when the access to a string is illegal.

Documentation (`StringIndexOutOfBoundsException`)**.** *Thrown by `String` methods to indicate that an index is either negative or greater than the size of the string. For some methods such as the `charAt` method, this exception also is thrown when the index is equal to the size of the string [48].*

However, unlike arrays, `String` is a typical Java class. To operate on a `String` object, one needs to invoke corresponding methods. Furthermore, as said in the above documentation, `StringIndexOutOfBoundsException`s are explicitly thrown from the source code and mostly from methods related to the `String` class. JCrashPack collected 9 crashes where a `StringIndexOutOfBoundsException` is thrown. The `String` method causing the exception for each crash is shown in Table 3.2.

Table 3.2: Causes of `StringIndexOutOfBoundsException` Crashes in JCrashPack

Crash	Causing String Method
Lang-6b	String.charAt(int)
Lang-19b	
Lang-44b	
Lang-51b	
Lang-27b	String.substring(int, int)
Lang-45b	
Math-101b	
XWIKI-14152	
ES-22997	

In `JDK1.8`, there are 46 places where a `StringIndexOutOfBoundsException` is thrown. It involves 24 `public` and `protected` methods from 3 classes, including `AbstractStringBuilder`, `String` and `StringBuffer`. The 24 methods can be categorised into several types, see Table 3.3. Notice that '`*`' in the table means that there are multiple matching types of the argument. And **Conditions** indicate the situation where **no** `StringIndexOutOfBoundsException` is thrown, instead of the opposite.

The methods are categorised mainly based on the arguments they take, and the conditions they have to meet to not throw the exception. For methods that fall into the same category, at the time of them being called, the operand stacks have the same combination of values. Those values have to go through the same checks. And similar to `ArrayIndexOutOfBoundsException`s, the *exception distance* can be defined to reflect how far the relevant values like the `index` or the `offset` are away from `0` or the `length`.

Table 3.3: Different Types of Methods That Throws `StringIndexOutOfBoundsException`

Signatures	Conditions	Methods

`int index`	$0 \leqslant \text{index} < \text{length}$	`String.charAt(int)`
		`String.codePointAt(int)`
		`StringBuffer.charAt(int)`
		`AbstractStringBuilder.charAt(int)`
		`AbstractStringBuilder.codePointAt(int)`
		`AbstractStringBuilder.deleteCharAt(int)`

	$1 \leqslant \text{index} \leqslant \text{length} + 1$	`String.codePointBefore(int)` `AbstractStringBuilder.codePointBefore(int)`
	$0 \leqslant \text{index} \leqslant \text{length}$	`String.substring(int)`
`int index, char c`	$0 \leqslant \text{index} < \text{length}$	`StringBuffer.setCharAt(int, char)` `AbstractStringBuilder.setCharAt(int, char)`
`int index, * str`	$0 \leqslant \text{index} \leqslant \text{length}$	`AbstractStringBuilder.insert(int, String)` `AbstractStringBuilder.insert(int, char[])`
`int newLength`	$0 \leqslant \text{newLength}$	`AbstractStringBuilder.setLength(int)`
`int start, int end`	$0 \leqslant \text{start} \leqslant \text{end} \leqslant \text{length}$	`String.substring(int, int)` `AbstractStringBuilder.substring(int, int)`
	$0 \leqslant \text{start} \leqslant \text{end}$	`AbstractStringBuilder.delete(int, int)`

`int start, int end,`
`*[] dst, int begin`
	$0 \leqslant \text{start} \leqslant \text{end} \leqslant \text{length}$	`String.getBytes(int, int, byte[], int)`
		`String.getChars(int, int, char[], int)`
		`AbstractStringBuilder.getChars(int, int, char[], int)`

`int start, int end,` `String str`	$0 \leqslant \text{start} \leqslant \text{length},$ $\text{start} \leqslant \text{end}$	`AbstractStringBuilder.replace(int, int, String)`
`*[] values,` `int offset, int count`	$0 \leqslant \text{offset}, 0 \leqslant \text{count},$ $\text{offset} + \text{count} \leqslant \text{values.length}$	`String(char[], int, int)` `String(int[], int, int)`
`int index, char[] values,` `int offset, int count`	$0 \leqslant \text{index} \leqslant \text{length},$ $0 \leqslant \text{offset}, 0 \leqslant \text{count},$ $\text{offset} + \text{count} \leqslant \text{values.length}$	`AbstractStringBuilder.insert(int, char[], int, int)`

3.3 `IllegalStateException` and `IllegalArgumentException`

Documentation (`IllegalStateException`)**.** *Signals that a method has been invoked at an illegal or inappropriate time. In other words, the Java environment or Java application is not in an appropriate state for the requested operation [49].*

Documentation (`IllegalArgumentException`)**.** *Thrown to indicate that a method has been passed an illegal or inappropriate argument [50].*

`IllegalStateException`s and `IllegalArgumentException`s are only explicitly thrown from the source code, similar to `StringIndexOutOfBoundsException`s. However, these two are more omnipresent compared with the previous two types of exceptions we have discussed. In `JDK1.8` itself, 583 instances of `IllegalStateException`s are thrown and 3,367 for `IllegalArgumentException`s. Numerous classes and methods are involved. Their usage is more general as well.

According to the documentation quoted above, an `IllegalStateException` is thrown when the environment state is inappropriate, and an `IllegalArgumentException` is thrown when the passed argument is inappropriate. The code snippet shown in Listing 3.1 is an example. The variable `hooks` at line 6 is a static field of the class. When it is `null`, an `IllegalStateException` is thrown. And `hook` at line 8 is a passed argument. When it violates the logic of the program, an `IllegalArgumentException` is thrown.

```
1  /**
2   * Add a new shutdown hook. Checks the shutdown state and the hook
3   * itself, but does not do any security checks.
4   */
5  static synchronized void add(Thread hook) {
6    if (hooks == null)
7      throw new IllegalStateException("Shutdown in progress");
8    if (hook.isAlive())
9      throw new IllegalArgumentException("Hook already running");
10   // ... rest of the program
11 }
```

Listing 3.1: `add` method from class `ApplicationShutdownHooks` from `JDK1.8`

However, this convention is not always true. Sometimes an `IllegalStateException` is thrown when only the argument violates the logic and sometimes it is hard to draw a line between the illegal arguments and illegal states. Take the code snippet in Listing 3.2 for example, the variables `comp1` and `comp2` are only related to the arguments, not any state outside this method, but `IllegalStateException`s are thrown at both line 14 and 16.

Unlike the first two types of exception, the usage of which is tied up with indexed access to a sequential data structure, the usage of `IllegalStateException`s and `IllegalArgumentException`s is difficult to characterise. The only pattern is that those are typically thrown within an `if`-statement checking the status of a variable.

```java
/**
 * If arr1 and arr2 are both arrays of the same component type,
 * return an array of that component type that consists of the
 * elements of arr1 followed by the elements of arr2.
 * Throws IllegalArgumentException otherwise.
 */
public static Object concatenateArrays(Object arr1, Object arr2) {
  Class comp1 = arr1.getClass().getComponentType();
  Class comp2 = arr2.getClass().getComponentType();
  int len1 = Array.getLength(arr1);
  int len2 = Array.getLength(arr2);

  if ((comp1 == null) || (comp2 == null))
    throw new IllegalStateException("Arguments must be arrays");
  if (!comp1.equals(comp2))
    throw new IllegalStateException(
      "Arguments must be arrays with the same component type");
  // ... rest of the program
}
```

Listing 3.2: `concatenateArrays` method from class `ObjectUtility` from `JDK1.8`

3.3.1 Jump Instructions

Two types of bytecode instructions are involved in `if`-statements. The first type is *jump instructions* including: (1) `IF_ICMPEQ`, `IF_ICMPNE`, `IF_ICMPLT`, `IF_ICMPGE`, `IF_ICMPGT`, and `IF_ICMPLE`; (2) `IFEQ`, `IFNE`, `IFLT`, `IFGE`, `IFGT`, and `IFLE`; (3) `IF_ACMPEQ`, and `IF_ACMPNE`; (4) `IFNULL` and `IFNONNULL`.

In the JVM, values of type `byte`, `char`, `short`, `int`, `boolean` or their wrapper classes are all represented as integers. While instructions in group (1) compare two integer values against each other, instructions in group (2) compare one integer value against zero. Instructions in group (3) check whether two references are the same and the ones in group (4) check whether one reference is `null` or not.

After the comparison, the flow of execution directly jumps into one of the branches of the `if`-statement.

3.3.2 Compare Instructions

The other type of bytecode instructions is normal instructions that perform the comparison task, including `LCMP`, `FCMPL`, `FCMPG`, `DCMPL` and `DCMPG`.

The first instruction compares two `long` values against each other, while the next two compare `float` values. The last two compare `double` values to each other.

After the comparison, the instruction returns an integer indicating the result of the comparison. That integer is then compared to zero by calling one instruction from group (2). The flow of execution jumps into a certain branch in the end.

A previous study showed that reformulating a single-objective fitness function into a many-objective one and loosing the constraints have a positive effect on search-based crash reproduction [44]. Along with the fact that these two types of exceptions are mainly thrown within `if`-statements, the new fitness function can look at the data diversity of branching variables at `if`-statements. Instead of focusing on only guiding the execution path into a certain branch, the many-objective fitness function can broaden the diversity of the context (i.e. values of the different variables) in addition.

3.4 `NullPointerException`

Documentation (`NullPointerException`)**.** *Thrown when an application attempts to use `null` in a case where an object is required. These include:*

- *Calling the instance method of a `null` object;*
- *Accessing or modifying the filed of a `null` object;*
- *Taking the length of `null` as if it was an array;*
- *Accessing or modifying the slots of `null` as if it was an array;*
- *Throwing `null` as if it was a `Throwable` value.*

Applications should throw instances of this class to indicate other illegal uses of the `null` object. `NullPointerException` objects may be constructed by the virtual machine as if suppression were disabled and/or the stack trace was not writable [51].

As `IllegalArgumentException`s and `IllegalStateException`s have covered a large portion of `null` checks in Java programs regarding use cases mentioned in the above documentation, `NullPointerException`s are mainly thrown by the JVM at run-time. The dereferenced object can either be `null` or not `null`. Therefore to throw the exception is a dichotomous event, for which it is hard to describe the exception distance with continuous values to specialise the fitness function.

```
a.b.c.d = 99;
n[i][j][k] = 99;
e.f = g.h;
x().y().m = 99;
```

Listing 3.3: Complex Situations for `NullPointerException`s

Additionally, Oracle Corporation and the OpenJDK Community [1] are aware that the information contained in a `NullPointerException` stack trace is not informative nor helpful [52]. For statements shown in Listing 3.3, when a `NullPointerException` is thrown, the JVM provides the filename and the line number where it is thrown from. However, it is not possible to pinpoint down which object is `null`. The two parties, at the time of writing this thesis report, have started working on computing the `null`-detail message to be incorporated into `NullPointerException`s for the future releases of the `JDK` and the JVM.

[1] `http://openjdk.java.net/`

Romano et al. [53] combined a static backward path-sensitive analysis with a search-based algorithm to recreate NullPointerException crashes. However, their approach requires manual intervention to initiate the search process, and the path analysing component is computationally expensive, which makes it only feasible to target one specific object at a time.

Therefore, NullPointerExceptions are out of the score of this thesis project.

Chapter 4

Crash-Specific Fitness Functions

In this chapter, we explain the design and implementation details of the crash-specific fitness functions for the aforementioned four types of Java exceptions, based on the analyses presented in Chapter 3.

4.1 Adapting Integration Testing Fitness Function for Indexed Access

A `StringIndexOutOfBoundsException` is thrown in essence because of the illegal access to the `char[]` that is wrapped around in the `String` class. Based on this similarity, `ArrayIndexOutOfBoundsException`s and `StringIndexOutOfBoundsException`s are grouped together. For these two types of exception, we have implemented an instrumentation component in EvoSuite and a fitness function in Botsing. See Figure 4.1 for the new components. Unchanged components from Figure 2.10 are omitted.

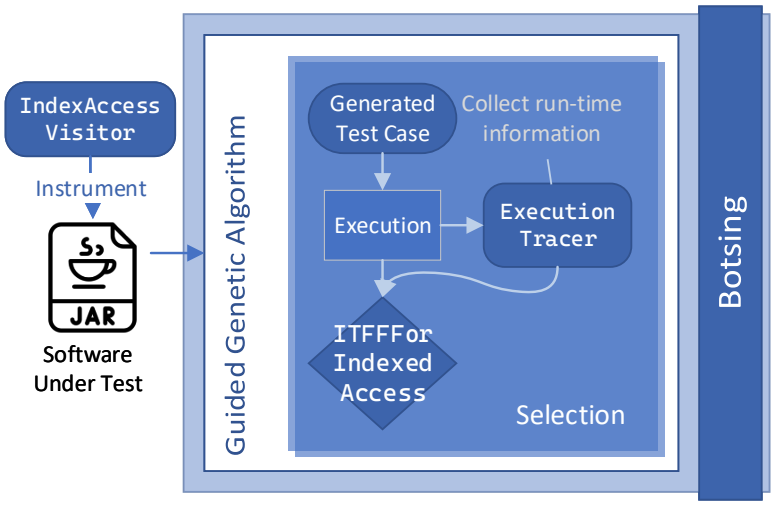


Figure 4.1: Related Components to Indexed Access in Botsing

The overall process is the same as described in Sub-section 2.3.2. The input to Botsing includes the crash stack trace and the relevant jar files. The pre-processing process logs classes in the stack trace as CUTs and instruments the jar files with our `IndexedAccessVisitor` in addition to other necessary visitors from EvoSuite. The GGA then generates the initial population of test cases made of direct or indirect invocations of methods and constructors appearing in the stack trace. Each test case is executed, and the data acquired by the instrumentation is stored into the `ExecutionTracer`, the singleton provided by EvoSuite for storing all run-time information. The information is used in the selection process to calculate the fitness value with our proposed *integration testing fitness function for indexed access*. With the guided crossover and guided mutation operators, the GGA evolves the test cases with the goal of reducing the fitness value to zero. Eventually, the optimal test case, i.e. the test case with a fitness value of zero, is found and minimised, and it is the final output of Botsing. All new components are explained in detail in this section.

4.1.1 Exception Distance

We have formulated a new exception distance $d^*_{exception}(x)$ intending to describe how far the accessed `index` is away from being negative or greater or equal to the length of the array. It is defined as follows:

Definition 11 (Exception Distance for Indexed Access)**.**

$$d^*_{exception}(x) = \begin{cases} 0 & \text{if } index < 0; \\ 1 - \frac{|index - mid|}{length - mid} & \text{if } 0 \leqslant index < length; \\ 0 & \text{if } length \leqslant index \end{cases}$$

where x is the generated test case, $length$ is the length of the array or `String` instance at the target line, $index$ is the queried index, and $mid = \frac{length-1}{2}$ is the median of all legal indices.

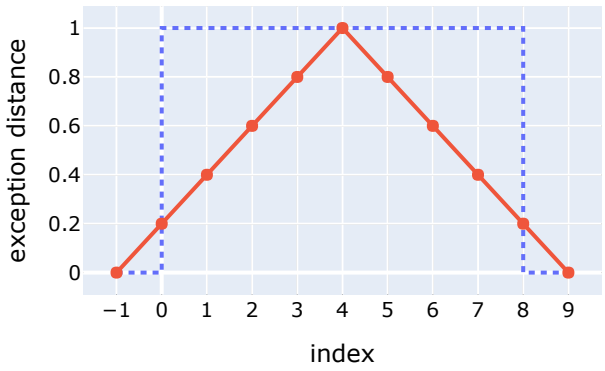


Figure 4.2: $d_{exception}$ (blue dashed line) versus $d^*_{exception}$ (red dotted line)

See Figure 4.2 for example. For an array of length 9, all integers from 0 to 8 are legal indices. With the newly defined $d^*_{exception}$, the median index, 4, results in a distance value

of 1 because it is the furthest to being either negative or greater or equal to 9. As the index moves either left or right, the distance value decreases and turns 0 as soon as the index becomes illegal. With the previously used $d_{exception}$, distance values stay 1 for all legal indices and immediately drop to 0 when illegal.

4.1.2 Instrumentation

To calculate $d^*_{exception}$, two values are essential. One is the accessed `index`, and the other one is the `length` of the array or `String` instance at the target line. To acquire the information at run-time, we have added a customised `MethodVisitor` to EvoSuite's instrumentation infrastructure, which is called `IndexedAccessVisitor`. It adds bytecode instrumentation to the CUT in pre-processing. When a generated test case is executed, the added instrumentation will log desired information into the `ExecutionTracer`.

`ArrayIndexOutOfBoundsException`

As explained in section 3.1, indexed array accesses are performed directly by bytecode instructions. Therefore at the target lines in the SUT, these related bytecode instructions are instrumented.

```
1  DUP2
2  SWAP
3  ARRAYLENGTH
4  LDC layer++
5  LDC className
6  LDC methodName
7  INVOKESTATIC org/evosuite/testcase/execution/ExecutionTracer.
     passedIndexedAccess (IIILjava/lang/String;Ljava/lang/String;)V
```

Listing 4.1: Bytecode Snippet Inserted Before Each Loading Instruction

For all loading related instructions, including `IALOAD`, `LALOAD`, `FALOAD`, `DALOAD`, `AALOAD`, `BALOAD`, `CALOAD` and `SALOAD`, the bytecode snippet shown in Listing 4.1 is inserted before the said instruction.

Figure 4.3 depicts the changes in the operand stack when executing the inserted bytecode snippet. The instruction `DUP2` duplicates the top two values of the operand stack. `SWAP` switches the position of the top two values so later on `arrayref` can be consumed by `ARRAYLENGTH`, and it gets the `length` of the array in return. The variables `layer`, `className` and `methodName` are all fields of the `IndexedAccessVisitor`. The field `layer` denotes how many indexed accesses on the target line have been visited so that situations like a multi-dimensional array or multiple arrays on one line can be handled. The field `methodName` is the full method name with signature. The instruction `LDC` loads these three fields onto the operand stack. `INVOKESTATIC` invokes a static method `passedIndexedAccess` from the `ExecutionTracer`, which logs the top five values of the operand stack (namely `index`, `length`, `layer`, `className` and `methodName`) into a nested map in the `ExecutionTracer`.

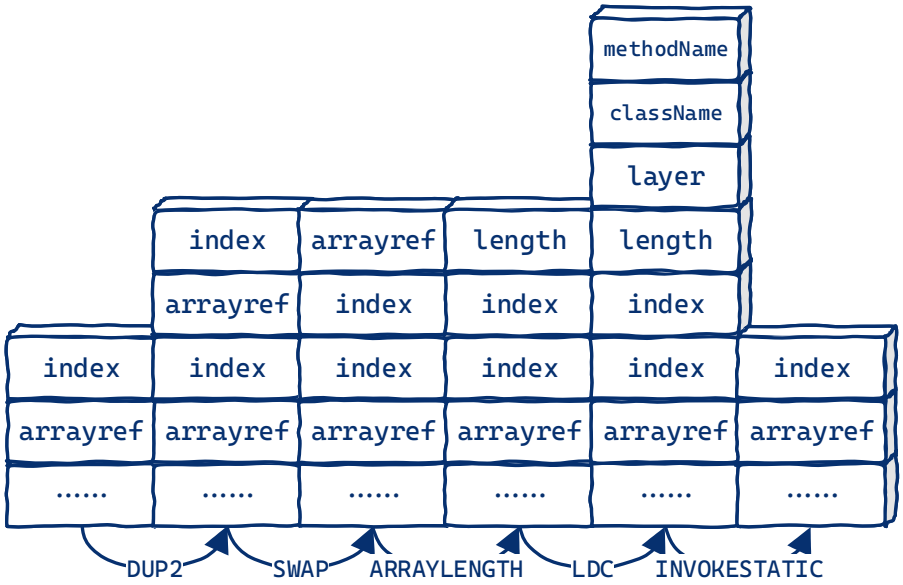


Figure 4.3: Instrumentation Before Executing a Loading Instruction

For all storing related instructions, including `IASTORE`, `LASTORE`, `FASTORE`, `DASTORE`, `AASTORE`, `BASTORE`, `CASTORE` and `SASTORE`, the instrumentation is similar to that for the loading instructions. The only difference is that the `element` value at the top of the operand stack (see Figure 3.3) should be temporarily saved to log the `index` and the `length` of the array. Afterwards, it should be restored back to the top of the operand stack.

`StringIndexOutOfBoundsException`

Instead of by bytecode instructions, indexed `String` accesses are performed, and the exception is thrown, by a set of methods in the string related classes (see Table 3.3). Therefore at the target lines in the SUT, invocations to these methods should be instrumented.

However considering that in JCrashPack there are only crashes caused by two methods from the `String` class (see Table 3.2), our effects have yet only included those two methods together with the other seven methods from the `String` class. One addition is `CharSequence.charAt(int)` as `String.charAt(int)` is an implementation of it. In total, there are ten methods.

Three bytecode instructions are involved in the invocation to those methods, including `INVOKEVIRTUAL`, `INVOKESPECIAL` and `INVOKEINTERFACE`. The bytecode snippet is inserted if the to-be-invoked method is one of the ten methods mentioned before. According to different signatures and conditions listed in Table 3.3, different values need to be logged into the `ExecutionTracer`. Therefore the bytecode snippet varies. It is largely similar to that of indexed array accesses, with one major difference. Take `String.charAt(int)` for example, the bytecode snippet for which is shown in Listing 4.2. On line 3, instead of the bytecode instruction `ARRAYLENGTH`, an invocation to the method `String.length()` is inserted to consume the `stringref` and get the `length` of the `String` instance back.

```
1 DUP2
2 SWAP
3 INVOKEVIRTUAL java/lang/String.length ()I
4 LDC layer++
5 LDC className
6 LDC methodName
7 INVOKESTATIC org/evosuite/testcase/execution/ExecutionTracer.
    passedIndexedAccess (IIILjava/lang/String;Ljava/lang/String;)V
```

Listing 4.2: Bytecode Snippet Inserted Before Invoking `String.charAt(int)`

One critical aspect of the instrumentation is that, after the execution of the insert code snippet, the operand stack should return to its previous form (see Figure 4.3) so that the original instructions can be executed without any error.

4.1.3 Integration Testing Fitness Function for Indexed Access

With the newly defined $d^*_{exception}$ from Definition 11 and bytecode instrumentation to collect necessary information, we have extended the integration testing fitness function used in Botsing (see Definition 10) for indexed access, defined as follows:

Definition 12 (Integration Testing Fitness Function for Indexed Access)**.**

$$f_{indexed\_access}(x) = \begin{cases} n + d_{line_n}(x) & \textit{if the line number of the n-th frame is not covered;} \\ d^*_{exception}(x) & \textit{if the line number of all frames are covered.} \end{cases}$$

4.2 Many-Objectivisation with Helper Objectives for Branching Variables

In previously used fitness functions, branch distance guides the search process to go strictly into one specific branch of the `if`-statements. However in the context of crash reproduction, we know that the flow has ended up at one point (i.e. the line where the exception is thrown), but we do not know in which context (i.e. values of the different variables) it reaches there. Soltani et al. [44] showed that relaxing the constraints and adapting a multi-objective genetic algorithm can improve the results. Hence our improvement for `IllegalArgumentExceptions` and `IllegalStateExceptions` consists in loosing the branching constraints and creating multiple diversity objectives for branching variables. For that purpose, we have implemented an instrumentation component in EvoSuite, a static analysis component and a set of fitness functions for different types of variables in Botsing. A simplified overview of our work is shown in Figure 4.4, the detail of which is explained in this section.

4.2.1 Diversity Objectives

Clauses contained in an `if`-statement typically include checking whether a reference is `null`, checking whether a `Collection`-like instance is empty, and comparison between

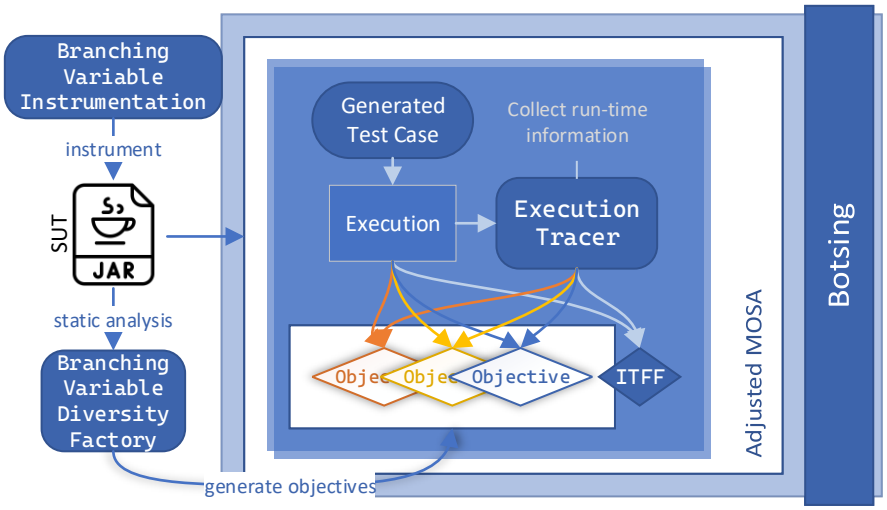


Figure 4.4: Related Components to Branching Variable Diversity in Botsing

values. Based on `IOCoverageConstants` [1] from EvoSuite, we have defined 20 objectives for different types of variables, listed in Table 4.1.

4.2.2 Objective Factory

At run-time, the execution of one generated test case can only cover a particular path of the program, because of which the instrumentation only logs information about variables that it has run into within that specific path, leaving out variables on other paths. Therefore it is necessary to get all the objectives set beforehand, and we have implemented the `BranchingVariableDiversityFactory` to do that.

At the pre-processing phase, the `BranchingVariableDiversityFactory` collects all the methods and constructors appearing in the input stack trace. When going through the CFG of one method or constructor, the factory creates a set of `BranchingVariable-DiversityObjectives` as helper objectives when it runs into an `if`-statement, according to the types of the branching variables.

Take method `toLocale` in Listing 4.3 for example. At line 99, the local variable `len` is compared against three constants, `2`, `5`, and `7`. It is an `int` value; therefore, objectives `NUM_POSITIVE`, `NUM_ZERO` and `NUM_NEGATIVE` are created. At line 104, local variables `ch0` and `ch1` are compared against literals `'a'` and `'z'`. They are `char` values; therefore, objectives `CHAR_ALPHA`, `CHAR_DIGIT` and `CHAR_OTHER` are created for each one of them. This is done for each method or constructor that has appeared in the stack trace.

[1] `https://github.com/EvoSuite/evosuite/blob/master/client/src/main/java/org/evosuite/coverage/io/IOCoverageConstants.java`

Table 4.1: Diversity Objectives for Different Types of Variables

(Based on `IOCoverageConstants` from EvoSuite)

Type				**Objectives**
Classes	`Object`			`REF_NULL`
				`REF_NONNULL`
		`Collection`	`List`	`LIST_EMPTY`
				`LIST_NONEMPTY`
			`Set`	`SET_EMPTY`
				`SET_NONEMPTY`
		`Array`		`ARRAY_EMPTY`
				`ARRAY_NONEMPTY`
		`Map`		`MAP_EMPTY`
				`MAP_NONEMPTY`
		`String`		`STRING_EMPTY`
				`STRING_NONEMPTY`
		`Wrapper`	`Character`	`CHAR_ALPHA`
				`CHAR_DIGIT`
				`CHAR_OTHER`
			`Boolean`	`BOOL_TRUE`
				`BOOL_FALSE`
			`Number`	`NUM_POSITIVE`
				`NUM_ZERO`
				`NUM_NEGATIVE`
Primitive Types	`char`			`CHAR_ALPHA`
				`CHAR_DIGIT`
				`CHAR_OTHER`
	`boolean`			`BOOL_TRUE`
				`BOOL_FALSE`
	`byte`, `short`, `int`, `long`, `float`, `double`			`NUM_POSITIVE`
				`NUM_ZERO`
				`NUM_NEGATIVE`

```
94 public static Locale toLocale(String str) {
95   if (str == null) {
96     return null;
97   }
98   int len = str.length();
99   if (len != 2 && len != 5 && len < 7) {
100    throw new IllegalArgumentException("Invalid locale format: " + str);
101  }
102  char ch0 = str.charAt(0);
103  char ch1 = str.charAt(1);
104  if (ch0 < 'a'  ch0 > 'z'  ch1 < 'a'  ch1 > 'z') {
105    throw new IllegalArgumentException("Invalid locale format: " + str);
106  }
   // ... rest of the program
}
```

Listing 4.3: `LocaleUtil.toLocale` from Commons-lang

4.2.3 Diversity Distances

Each objective is, in essence, a fitness function (see Figure 4.5), hence we have also defined a set of distance functions, inspired by `InputCoverageTestFitness` [2] from EvoSuite, to calculate the fitness value to each objective.

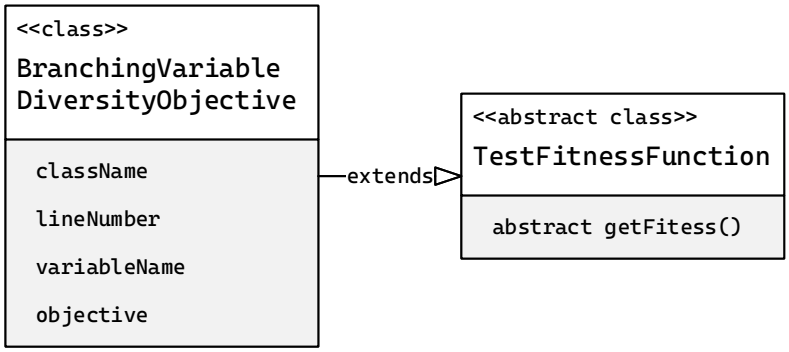


Figure 4.5: `BranchingVariableDiversityObjective` Class

For objectives with a dichotomous nature, including `REF_NULL`, `REF_NONNULL`, `LIST_NONEMPTY`, `SET_NONEMPTY`, `ARRAY_NONEMPTY`, `MAP_NONEMPTY`, `STRING_NONEMPTY`, `BOOL_TRUE` and `BOOL_FALSE`, the distance function remains binary:

Definition 13 (Diversity Distance for Binary Objectives)**.**

$$d_{binary}(b) = \begin{cases} 0 & \textit{the } b \textit{ satisfies the objective;} \\ 1 & \textit{otherwise.} \end{cases}$$

[2] https://github.com/EvoSuite/evosuite/blob/master/client/src/main/java/org/evosuite/coverage/io/input/InputCoverageTestFitness.java

For objectives for a `Collection`-like variable to be empty, the more elements it contains, the further it is away from being empty. Therefore the distance function is defined with its size:

Definition 14 (Diversity Distance for Collection Being Empty)**.**

$$d_{empty}(x) = \begin{cases} \texttt{x.length} & \textit{if } x \textit{ is an array or } \texttt{String} \textit{ instance;} \\ \texttt{x.size()} & \textit{if } x \textit{ is a } \texttt{List}, \texttt{Set} \textit{ or } \texttt{Map} \textit{ instance.} \end{cases}$$

To calculate the distances for the rest of the objectives, the following distance functions have been defined:

Definition 15 (Diversity Distance for Char Being Alphabet Letters)**.**

$$d_{char\_alpha}(c) = \begin{cases} \texttt{'A' - c} & \textit{if } \texttt{c < 'A';} \\ \texttt{min(c - 'Z', 'a' - c)} & \textit{if } \texttt{'Z' < c < 'a';} \\ \texttt{c - 'z'} & \textit{if } \texttt{'z' < c;} \\ \texttt{0} & \textit{otherwise.} \end{cases}$$

Definition 16 (Diversity Distance for Char Being Digits)**.**

$$d_{char\_digit}(c) = \begin{cases} \texttt{'0' - c} & \textit{if } \texttt{c < '0';} \\ \texttt{0} & \textit{if } \texttt{'0'} \leqslant \texttt{c} \leqslant \texttt{'9';} \\ \texttt{c - '9'} & \textit{if } \texttt{'9' < c.} \end{cases}$$

Definition 17 (Diversity Distance for Char Being Neither Letter Nor Digits)**.**

$$d_{char\_other}(c) = \begin{cases} \texttt{min(c - '0', '9' - c) + 1} & \textit{if } \texttt{'0'} \leqslant c \leqslant \texttt{'9';} \\ \texttt{min(c - 'A', 'Z' - c) + 1} & \textit{if } \texttt{'A'} \leqslant c \leqslant \texttt{'Z';} \\ \texttt{min(c - 'a', 'z' - c) + 1} & \textit{if } \texttt{'a'} \leqslant c \leqslant \texttt{'z';} \\ \texttt{0} & \textit{otherwise.} \end{cases}$$

Definition 18 (Diversity Distance for Number Being Positive)**.**

$$d_{num\_positive}(n) = \begin{cases} \texttt{1 - n} & \textit{if } n \leqslant 0; \\ \texttt{0} & \textit{otherwise.} \end{cases}$$

Definition 19 (Diversity Distance for Number Being Zero)**.**

$$d_{num\_zero}(n) = \texttt{abs(n)}$$

Definition 20 (Diversity Distance for Number Being Negative)**.**

$$d_{num\_negative}(n) = \begin{cases} \texttt{0} & \textit{if } n < 0; \\ \texttt{1 + n} & \textit{otherwise.} \end{cases}$$

4.2.4 Instrumentation

To compute the aforementioned diversity distances, the actual values of variables at runtime is needed. For that purpose, an instrumentor component, `BranchingVariableInstrumentation`, has been added to EvoSuite.

Information about local variables is vital to the task because the instrumentation should log the names of the variables to differentiate different objectives. The instrumentation should also be able to tell the source of the operands in the comparison, for example, whether one operand is a constant or a return value of a comparison instruction as explained in Sub-section 3.3.2. However, related information is not available when checking bytecode instruction by instruction. Therefore unlike `IndexedAccessVisitor`, which checks the transcompiled bytecode on the instruction level, `BranchingVariableInstrumentation` works on the level of method nodes. A `MethodNode` contains a list of all the bytecode instructions and a table of local variable nodes in that method.

If one of the jump instructions or compare instructions mentioned in Sub-section 3.3.1 or 3.3.2 is spotted when going through the list of bytecode instructions, the instrumentor goes backwards in the list to check the source of each operand. Based on the previous instructions, six kinds of sources can be identified:

1. a local variable;
2. the return value of a method call;
3. a `new` object;
4. a field of the object or static field of the class;
5. a constant or literal;
6. the return value of a comparison instruction.

For operands of source 1, the bytecode snippet shown in Listing 4.4 is inserted after loading the local variable. A `new` object is essentially the output of a constructor invocation; therefore, those of source 2 and 3 can be addressed by the *output coverage diversity* option already existing in EvoSuite. Operands of source 4 are left for future work. Operands of source 5 should be ignored as they are not variables. Operands of source 6 should be ignored as well since the comparison instructions are already instrumented.

```
1 DUP
2 INVOKESTATIC java/lang/Integer.valueOf (I)Ljava/lang/Integer;
3 LDC className
4 LDC currentLine
5 LDC variableName
6 INVOKESTATIC org/evosuite/testcase/execution/ExecutionTracer.
    passedBranchingVariable (Ljava/lang/Object;Ljava/lang/String;ILjava/
    lang/String;)V
```

Listing 4.4: Bytecode Snippet Inserted After Loading Local Variable

The instruction `DUP` duplicates the top value of the operand stack, which is the local variable just loaded. If the local variable is of a primitive type, the `valueOf` method of its corresponding wrapper class is called to turn it into an object. `Integer.valueOf(int)` in the case of the example shown. If it is already an object, the idle instruction `NOP` is used here. The variables `className` and `currentLine` are fields of the instrumentor, the latter of which denotes the line number in the source code of the current bytecode instruction. And the `variableName` is acquired by checking the local variable table contained in the `MethodNode`. Eventually, the static method `passedBranchingVariable` from the `ExecutionTracer` is invoked to log the information into a nested map.

```
1  LINENUMBER 99 L3
2  ILOAD 1
3  ICONST_2
4  IF_ICMPEQ L4
5  ILOAD 1
6  ICONST_5
7  IF_ICMPEQ L4
8  ILOAD 1
9  BIPUSH 7
10 IF_ICMPGE L4
```

Listing 4.5: Bytecode of Line 99 of method `toLocale`

Take the example of `LocaleUtil.toLocale` again. The transcompiled bytecode of line 99 of it is shown in Listing 4.5. At line 4, a jump instruction `IF_ICMPEQ` is spotted. The instrumentor goes backwards in search of operands of the comparison. First `ICONST_2` is found, which loads the integer constant `2` onto the operand stack. As it is a constant, it is ignored. Then `ILOAD 1` is found, which loads the variable with index `1` from the local variable table onto the operand stack. The instrumentor retrieve the name `len` from the local variable table and insert the bytecode snippet shown in Listing 4.4 to the original list of bytecode.

4.2.5 Many-Objectivisation With Helper Objectives

Together with a typical integration testing fitness function, all the diversity objectives created by the factory are used by MOSA to evaluate the generated test cases. The inserted instrumentation collects the required information to calculate fitness values for each of the objectives.

As described in Sub-section 2.2.4, while treating all *non-dominated* test cases equally, original Guided MOSA still aims to find the *optimal* test case that fulfils all objectives. For a specific variable, it is only possible to meet one diversity objective. Therefore, no *optimal* test case exists.

Hence Guided MOSA is adjusted in our implementation. The integration testing fitness function is set to be the *primary objective*, and all the diversity objectives are *helper objectives*. Once the primary objective is achieved, the optimisation goal is set to be reached. The search process terminates, and post-processing starts.

4.3 Summary

4.3.1 Exception Distance (RQ1)

For `ArrayIndexOutOfBoundsException`s and `StringIndexOutOfBoundsException`s, the exception distance is defined to numerically reflect how far the accessed index is away

from being out of bounds (i.e. being negative or larger than the length of the array or `String`), see Definition 11.

And for `IllegalArgumentException`s and `IllegalStateException`s, based on `InputCoverageTestFitness` from EvoSuite, a set of distance functions are defined to broaden the diversity of values of branching variables, see Definition 13 to 20.

4.3.2 Fitness Function (RQ2)

For `ArrayIndexOutOfBoundsException`s and `StringIndexOutOfBoundsException`s, a customised `MethodVisitor` named `IndexedAccessVisitor` is implemented in EvoSuite to instrument CUTs. Index access information at run-time is logged into the `ExecutionTracer` so that they can be used later to calculate the fitness value with the newly defined integration testing fitness function for indexed access (see Definition 12).

And for `IllegalArgumentException`s and `IllegalStateException`s, the task is many-objectivised. The `BranchingVariableDiversityFactory` checks the CFGs of CUTs to create different diversity objectives for different types of branching variables as helper objectives. A customised instrumentor `BranchingVariableInstrumentation` instruments CUTs so necessary information of the branching variables can be collected by the `ExecutionTracer`. The helper objectives, together with an integration testing fitness function as the primary objective, are then used by the adjusted Guided MOSA to evaluate the generated test cases.

Chapter 5

Evaluation and Results

To evaluate the effectiveness and efficiency of our crash-specific fitness functions and to answer **RQ3**, we performed an evaluation against real-world crashes from various open-source applications with our customised implementation of EvoSuite and Botsing. We also run the original Botsing to compare the results. We describe the experiment set-up and analyse the outcome in this chapter.

5.1 Experiment Set-up

5.1.1 Experiment Protocol

Case Selection

We base our experiments on JCrashPack [3], the benchmark introduced in Sub-section 2.3.3. As we focus on the four aforementioned types of exceptions, in total, 52 crashes have been picked, consisting of crashes from all seven real-world Java software applications included in JCrashPack. See Table 5.1 for the detailed composition of the 52 crashes.

For every selected crash, we have targeted Botsing at each frame which points to a class of the application. Other frames have been discarded to avoid generating test cases for external dependencies. For `ArrayIndexOutOfBoundsException`s and `StringIndexOutOfBoundsException`s, listed in the left half of Table 5.1, there are 128 valid frames from the 23 crashes. And for `IllegalArgumentException`s and `IllegalStateException`s, listed in the right column of Table 5.1, there are 246 frames from the 29 crashes. To address the random nature of the genetic algorithm, we repeated each execution 30 times, as suggested by Arcuri and Briand [54].

Configuration Parameters

We run Botsing with four different configurations, namely:

[IA] indexed access;
[IA-control] the control group for indexed access;
[BV] branching variable; and
[BV-control] the control group for branching variable.

Table 5.1: Detailed Composition of the Picked Crash Cases

Exception	Application	Crash	Exception	Application	Crash
ArrayIndex OutOfBounds Exception	ElasticSearch	19891	Illegal Argument Exception	JFreeChart	13b
		21911		Elasticsearch	14457
		22786			20045
		23324			20333
		24674			20479
		25933			21457
	Commons-lang	9b			21974
		12b			23381
	Commons-math	3b			26184
		81b			26651
		98b		Commons-lang	2b
		100b			5b
	Mockito	3b			54b
		34b		Commons-math	90b
StringIndex OutOfBounds Exception	Elasticsearch	22997			95b
	Commons-lang	6b			97b
		19b		Joda-Time	2b
		27b			8b
		44b			20b
		45b		XWiki	12667
		51b			13196
	Commons-math	101b			13546
	XWiki	14152			13942
			Illegal State Exception	Elasticsearch	19026
					22119
					23218
					23675
					25849
					26513

For all four configurations, the *population size* of one generation is set to the default value of EvoSuite, 50. And the *search budget* is set to 62,328 fitness evaluations. In addition to that, an execution will be killed if it has been running for 15 minutes, which is roughly the time needed for 62,328 evaluations. Botsing parameter `-integration_testing` is set to `true` so that all classes, having appeared in the stack trace, can be instrumented, and the integration testing scheme introduced in Sub-section 2.4.1 can be used.

Configurations [**IA**] and [**IA-control**] are used against `ArrayIndexOutOfBounds-Exception` and `StringIndexOutOfBoundsException` crashes. The *fitness function* of [**IA**] is set to our customised `ITFFForIndexedAccess`, while for [**IA-control**] it is the original `IntegrationTestingFF` from Botsing. All other configuration parameters are set

to the default values of Botsing and EvoSuite.

Meanwhile, configurations [**BV**] and [**BV-control**] are used on `IllegalArgumentException` and `IllegalStateException` crashes. The *search algorithm* of both configurations is set to `Guided_MOSA` and both *fitness functions* to the integration testing fitness function. For [**BV**] it has the property `branching_variable_diversity` set to `true`, because of which the integration testing fitness function will be used as the primary objective, and our customised diversity factory will generate additional helper objectives. That property is left out for [**BV-control**] so that the integration testing fitness function is the only objective to be satisfied. Similarly, all other configuration parameters are set to the defaults.

Table 5.2 shows the complexity statistics of crashes used for the two groups of configurations. *Cr* denotes the number of crashes. $\overline{frm}$ means the average number of frames per stack trace. $\overline{CYC}$ is the average cyclomatic complexity [55], which is a metric that measures the number of linearly independent paths of a software application, therefore describing its complexity. And $\overline{NCSS}$ represents the average number of statements.

Table 5.2: Complexity Statistics of Crashes Used

(a) Crashes Used for [**IA**] and [**IA-control**]

Application	*Cr*	$\overline{frm}$	$\overline{CYC}$	$\overline{NCSS}$
Elasticsearch	7	11.00	1.82	135.15k
Commons-lang	8	2.75	3.28	13.65k
Commons-math	5	2.00	2.37	22.83k
Mockito	2	6.00	1.79	6.04k
XWiki	1	7.00	1.87	164.67k
Total	23	5.57	2.45	58.52k

(b) Crashes Used for [**BV**] and [**BV-control**]

Application	*Cr*	$\overline{frm}$	$\overline{CYC}$	$\overline{NCSS}$
JFreeChart	1	5.00	2.77	59.09k
Elasticsearch	15	12.73	1.80	118.81k
Commons-lang	3	1.00	3.32	14.20k
Commons-math	3	2.00	2.32	15.63k
Joda-Time	3	1.33	2.12	19.44k
XWiki	4	9.25	1.89	204.02k
Total	29	8.48	2.09	96.73k

5.1.2 Data Analysis Procedure

To check whether our customised implementation of Botsing and EvoSuite can improve the crash reproduction effectiveness, we analyse the status of the search process after each execution. We define the following five states:

#1 **not started** - the search process does not start due to failing to generate the initial generation;

#2 **crash location not reached** - the target line of the inner-most frame is not covered;
#3 **crash location reached** - the target line of the inner-most frame is covered, but the target exception is not thrown;
#4 **reproduced** - the target exception is thrown, and the given stack trace is reproduced up to the target frame level; and
#5 **unsatisfactory** - the target exception is thrown, but the crash should not be considered reproduced (for example, the crash is reproduced by an obvious mock object).

Notice that Botsing reports a successful reproduction for both state #4 and #5. An example of why state #5 is defined and why those executions should not be considered reproduced is explained in Sub-section 5.3.1.

Reproduction Rate Effect Size

The crash reproduction task is, in essence, a dichotomous problem that the target stack trace is either reproduced or not in the end. *Reproduction ratio* is fundamental to our evaluation. However, as pointed out by Arcuri and Briand [54], only comparing the reproduction ratio is not statistically sufficient to address the random nature of genetic algorithms. Therefore, we have used the *odds ratio* to measure the effect size of our proposed methods against the original ones. It is computed with the following equation:

$$\psi = \frac{a+\rho}{n-a+\rho} / \frac{b+\rho}{n-b+\rho}$$

where a is the number of executions where our proposed methods successfully reproduce the target frame. And b is the number of executions where the control groups successfully reproduce the target frame. For configuration [**BV**] and [**BV-control**], n is the number of total executions, which is always 30. However for configuration [**IA**] and [**IA-control**], as our adapted fitness function only kicks in after reaching the crash location, n, therefore, is the number of executions of states #3, #4 and #5, i.e. having a fitness value smaller or equal to 1.0. And ρ is a constant used to avoid the appearance of zeros in the denominator. In our evaluation, it is set to $\rho = 0.5$.

$\psi = 1$ means that the two compared methods have identical performance. While $\psi > 1$ means that the former has a higher chance of success, $\psi < 1$ means that the latter outperforms the former.

In addition, we have applied Fisher's exact test with $\alpha = 0.05$ for Type I error to evaluate the statistical significance of the observed data. As a statistical convention, a resulting p-value of less than 0.05 is considered statistically significant enough to draw a conclusion that one of the compared methods performs differently compared to the other.

Reproduction Efficiency Effect Size

For crashes that have been reproduced by both our proposed methods and the control groups, we have collected *the number of fitness evaluations* having been performed before the search process reaching the optimal test case. Those numbers are used to investigate the efficiency of the compared methods. Again it is not statistically convincing to simply compare the

numbers themselves. An effect size measurement is needed. As the number of fitness evaluations is no longer dichotomous, we performed the $\hat{A}_{12}$ analysis proposed by Vargha and Delaney [56] (VD.A) instead. It is computed with the following equation:

$$\hat{A}_{12} = \left(\frac{R_1}{s_1} - \frac{s_1 + 1}{2} \right) / s_2$$

where s_1 is the number of executions where our proposed methods successfully reproduce the target frame, and s_2 is the number of executions where the control groups successfully reproduce the target frame. For configuration [**BV**], R_1 is the rank sum of the number of fitness evaluations that our proposed method uses to reproduce the target frame. Again for configuration [**IA**], our adapted fitness function only starts to affect the search process once fitness value 1.0 has been reached. R_1, therefore, is the rank sum of the number of fitness evaluations that our proposed method uses to reduce the fitness value from 1.0 to 0.0.

$\hat{A}_{12} = 0.5$ means that the two compared methods have an identical efficiency. While $\hat{A}_{12} < 0.5$ means that the former takes fewer fitness evaluations to reach the optimal test case, $\hat{A}_{12} > 0.5$ means the latter outperforms the former. The VD.A analysis also comes with a magnitude measure, ranging from *negligible*, *small*, *medium* to *large*, categorising the impact of the difference. Similarly, to evaluate the statistical significance, we have applied the Wilcoxon Rank Sum test with $\alpha = 0.05$ for Type I error.

For example, if our proposed method reproduces the target frame successfully three times with $\{20, 30, 40\}$ fitness evaluations, while the control group reproduces the target frame five times with $\{50, 27, 35, 52, 28\}$ fitness evaluations, the rank of our proposed method is therefore $\{1, 4, 6\}$ and $R_1 = 11$. The resulting effect size $\hat{A}_{12} = 0.3333$.

5.1.3 Evaluation Infrastructure

The evaluations have been performed on the BSR clusters [57] with 20 CPU-cores, 384 GB memory and 482 GB hard drive. With all 374 frames for our customised Botsing and EvoSuite and the control group, we have executed a total of 22,440 independent runs. It has taken about 11 days in total.

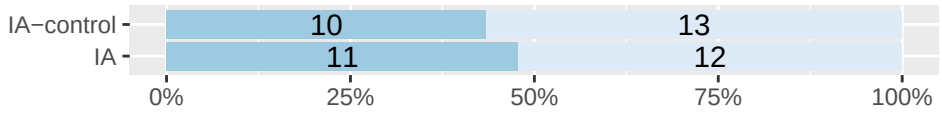
5.2 Results

In this section, we present the results of the evaluation for our proposed methods versus the control groups.

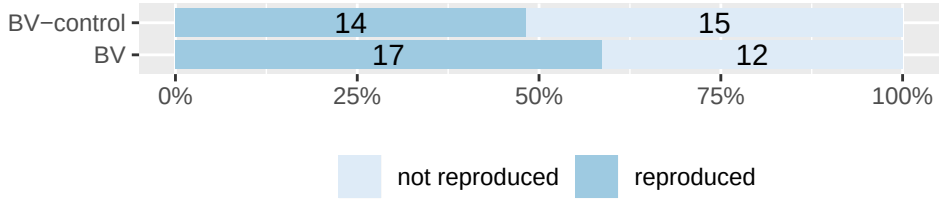
5.2.1 Reproduction Rate

Figure 5.1 shows the reproduction status overview for all 52 crashes grouped by the configurations. One crash is considered reproduced if one frame of its stack trace has been successfully reproduced at least once by one of the configurations among the 30 repeated executions.

Configurations [**IA**] and [**IA-control**] have been executed against JCrashPack crashes of `ArrayIndexOutOfBoundsException`s and `StringIndexOutOfBoundsException`s. As



(a) [**IA**] versus [**IA-control**]



(b) [**BV**] versus [**BV-control**]

Figure 5.1: Reproduction Status on the Case Level

indicated in Figure 5.1a, [**IA-control**] has reproduced 10 crashes. [**IA**] has reproduced one additional crash, which is **ES-23324**.

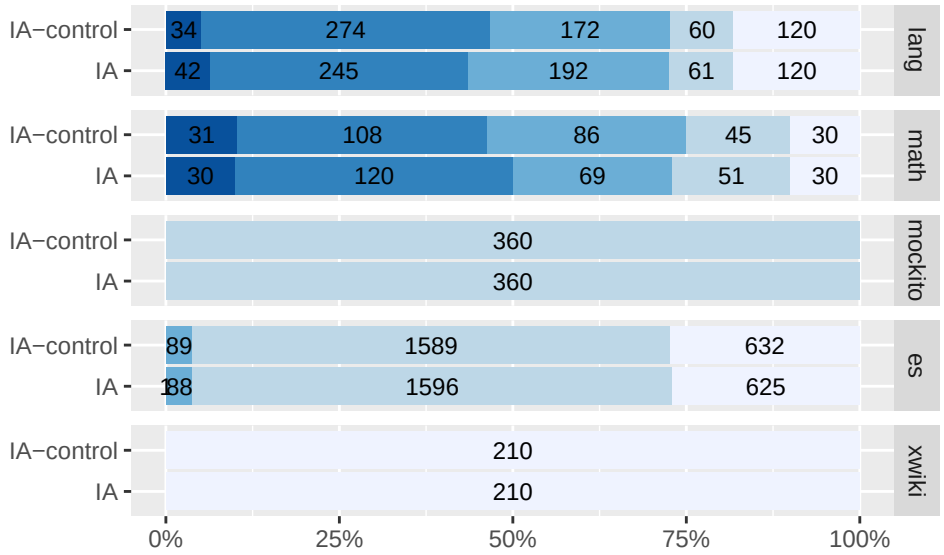
Figure 5.1b shows that 14 `IllegalArgumentException` and `IllegalStateException` crashes have been reproduced by configuration [**BV-control**]. [**BV**] reproduces three more crashes in addition to those, which are **MATH-97b**, **TIME-20b** and **ES-14457**.

Figure 5.2 shows the more detailed reproduction rate for all 22,440 runs grouped by the software application and the configuration. For the first two configurations, with Commons-lang crashes, there is a delta of 29 executions. [**IA-control**] has 31 more successes with crash **LANG-19b** and [**IA**] has two more successes with the second frame of crash **LANG-12b**. With Commons-math crashes, there is a delta of 12 executions. [**IA**] has 11 more successes with crash **MATH-81b** and 1 more success with crash **MATH-101b**. For the other software applications, the performance of our proposed method and that of the control group are very similar.

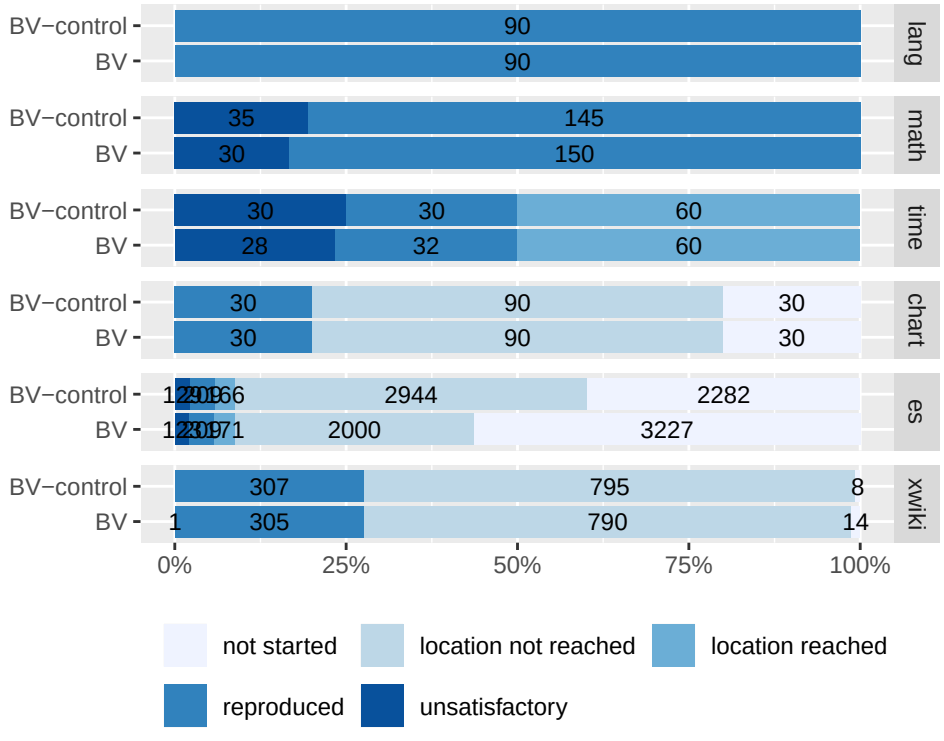
The reproduction status shown in Figure 5.2b indicates that the two configurations perform very similar to each other. A significant difference is that a large number of executions of our proposed method on Elasticsearch crashes fail to start the search process compared to the control group, which is explained in Sub-section 5.3.1. For a few executions with Commons-math and Joda-Time crashes, [**BV**] has resulted in less *unsatisfactory* ones and more *reproduced* ones.

5.2.2 Odds Ratio

For all crashes that have been successfully reproduced at least once by at least one of the configuration pairs, we have calculated the odds ratios and their corresponding p-values, see Tables 5.3 and 5.4. Rows coloured blue are where our proposed methods outperform the control groups, while rows coloured orange are where the control groups perform better. Notice that for crashes with multiple frames reproduced, the outer-most reproduced frame is selected to represent the crash. Even though the second frame of crash **ES-22119** is



(a) [**IA**] versus [**IA-control**]



(b) [**BV**] versus [**BV-control**]

Figure 5.2: Reproduction Status on the Application Level

only reproduced by [**BV**], the first frame is reproduced by both our proposed method and the control group. Therefore, it is not illustrated as a delta crash in Figure 5.1b. And for configuration [**IA**] and [**IA-control**], the number of executions, that have reached the crash location, is included in the table as well.

Table 5.3: Odds Ratio Between [**IA**] and [**IA-control**]

Crash	**Frame**	**IA**		**IA-control**		ψ	p-**value**[*]
		success	**line reached**	**success**	**line reached**		
ES-23324	1	1	1	0	2	15.0000	0.33333
MATH-101b	1	30	30	29	30	3.1017	1.00000
LANG-12b	2	29	30	27	30	2.5030	0.61195
MATH-81b	all 6	60	129	49	135	1.5212	0.10460
LANG-27b	1	30	30	30	30	1.0000	1.00000
LANG-44b	1	30	30	30	30	1.0000	1.00000
LANG-45b	1	30	30	30	30	1.0000	1.00000
LANG-51b	1	30	30	30	30	1.0000	1.00000
LANG-9b	6	1	30	1	30	1.0000	1.00000
MATH-98b	1	30	30	30	30	1.0000	1.00000
LANG-19b	3	12	30	24	30	0.1793	0.00333

[*] Notice that it must have $p \leqslant 0.05$ for the observed results of one crash to be statistically significant.

From Table 5.3, we can see that the control group outperforms our proposed method with the third frame of crash **LANG-19b** with a p-value of 0.00333, which means that it is statistically significant. For the rest of the crashes, if Botsing is able to reach the crash location, our proposed method performs no worse than the control group. With crashes **ES-23324**, **MATH-81b**, **MATH-101b** and **LANG-12b**, the results even suggest that our proposed method can reproduce the target frame of the crash more often, especially with the first two where Botsing has difficulty in reaching the crash location. [**IA**] has a reproduction rate of 100.00% and 46.51% respectively for **ES-23324** and **MATH-81b** against 0.00% and 36.30% with [**IA-control**]. However, the p-values are too high to draw statistically significant conclusions. It is explained in Sub-section 5.3.1 why all six frames of **MATH-81b** are grouped together.

Table 5.4 shows that configuration [**BV**] outperforms [**BV-control**] with the top five crashes and falls behind with the bottom two crashes. However, none of the p-values is small enough to draw any conclusion. For the remaining 10 crashes, there is no improvement nor deterioration in terms of reproduction rate.

5.2.3 Vargha and Delaney's $\hat{A}_{12}$ Measures

Tables 5.5 and 5.6 contain the results of the VD.A analyses performed between configurations. Rows coloured blue indicate that our proposed methods decrease the number of

Table 5.4: Odds Ratio Between [**BV**] and [**BV-control**]

Crash	Frame	BV	control	ψ	p-value*
MATH-97b	1	3	0	7.7636	0.23729
TIME-20b	1	2	0	5.3509	0.49153
ES-21457	1	1	0	3.1017	1.00000
ES-22119	2	1	0	3.1017	1.00000
ES-14457	3	11	6	2.2229	0.25157
CHART-13b	2	30	30	1.0000	1.00000
LANG-2b	1	30	30	1.0000	1.00000
LANG-54b	1	30	30	1.0000	1.00000
LANG-5b	1	30	30	1.0000	1.00000
MATH-90b	1	30	30	1.0000	1.00000
MATH-95b	4	30	30	1.0000	1.00000
TIME-8b	1	30	30	1.0000	1.00000
XWIKI-12667	2	30	30	1.0000	1.00000
XWIKI-13196	2	30	30	1.0000	1.00000
XWIKI-13546	2	30	30	1.0000	1.00000
XWIKI-13942	5	5	7	0.6758	0.74805
ES-21974	5	18	21	0.6540	0.58888

Table 5.5: Vargha and Delaney's $\hat{A}_{12}$ Measure Between [**IA**] and [**IA-control**]

Crash	Frame	$\hat{A}_{12}$	p-value*	Magnitude
LANG-19b	3	0.1771	0.00191	large
LANG-51b	1	0.3356	0.02847	small
LANG-27b	1	0.4372	0.40749	negligible
LANG-12b	2	0.4630	0.14604	negligible
LANG-45b	1	0.5000	1.00000	negligible
MATH-81b	all 6	0.6194	0.03271	small
LANG-44b	1	0.6272	0.09189	small
MATH-101b	1	0.6351	0.07576	small
MATH-98b	1	0.6433	0.05687	small
LANG-9b	6	1.0000	1.00000	large

* Notice that it must have $p \leqslant 0.05$ for the observed results of one crash to be statistically significant.

fitness evaluations needed to reach the optimal test case, while rows coloured orange are where the control groups were more efficient in reproducing the target frame of the crash.

10 crashes have been reproduced by both configuration [**IA**] and [**IA-control**]. With p-values of 0.00191 and 0.02847, our proposed method is significantly more efficient in reproducing frames **LANG-19b-3** and **LANG-51b-1**. (For the convenience of writing, the n-th frame of crash CR-XXX is denoted as frame CR-XXX-n in the following sections.)

However, the control group has done slightly more efficient against crash **MATH-81b**, with a p-value of 0.03271. The general trend is that for Commons-lang crashes, our proposed method improves the efficiency, while for Commons-math crashes, it slows the search process down. Notice that even though frame **LANG-9b-6** shows a *large* deterioration, the frame has only been reproduced once by each of the two configurations.

Table 5.6: Vargha and Delaney's $\hat{A}_{12}$ Measure Between [**BV**] and [**BV-control**]

Crash	Frame	$\hat{A}_{12}$	p-value*	Magnitude
ES-21974	5	0.3095	0.04305	medium
XWIKI-13942	5	0.4286	0.74271	negligible
MATH-95b	4	0.4594	0.57056	negligible
CHART-13b	2	0.4711	0.70610	negligible
TIME-8b	1	0.4983	0.97734	negligible
ES-22119	1	0.5000	1.00000	negligible
LANG-2b	1	0.5000	1.00000	negligible
LANG-5b	1	0.5000	1.00000	negligible
MATH-90b	1	0.5000	1.00000	negligible
XWIKI-12667	2	0.5000	1.00000	negligible
XWIKI-13196	2	0.5000	1.00000	negligible
XWIKI-13546	2	0.5000	1.00000	negligible
ES-14457	3	0.5227	0.91965	negligible
LANG-54b	1	0.6800	0.01695	medium

* Notice that it must have $p \leqslant 0.05$ for the observed results of one crash to be statistically significant.

With [**BV**], five crashes have shown better efficiencies, judging by the $\hat{A}_{12}$ measures from Table 5.6. Only one of the five is statistically significant, which is crash **ES-21974** with $p = 0.04305$. While two crashes have seen small increase in the number of fitness evaluations needed. One of the two is statistically significant, which is crash **LANG-54b** with $p = 0.01695$. All the other seven crashes have witnessed no difference in terms of the efficiency between our proposed method and the control group.

5.3 Discussion

5.3.1 Manual Analyses

We try to understand and explain the results to answer **RQ3** by looking into cases where there is a significant difference.

Unsatisfactory Reproductions

For several executions, Botsing has terminated the search process and declares a successful reproduction. However, after manual analyses, we decide to consider those generated test cases as *unsatisfactory*. Frame **LANG-6b-1** is one example. It has been *reproduced* exactly

once by configuration [**IA-control**], but not by [**IA**]. The input stack trace of it is shown in Listing 5.1, while the test case generated by the control group is shown in Listing 5.2. Listing 5.3 contains the source code of the `translate` method from the first frame of the stack trace.

The `translate(CharSequence, int, Writer)` method is stubbed at line 4 of Listing 5.2 to return `3663` for any parameter. Originally, it should return the count of codepoints consumed with the invocation. Notice that this is neither the same `translate` method from the first frame, nor the one from the second frame of the stack trace. However, it is invoked at line 85 of the `translate` method from the first frame, see Listing 5.3. Later, at line 94, the returned value is used as the boundary when accessing the `CharSequence`. The forged value `3663`, therefore, inevitably results in a `StringIndexOutOfBoundsException` being thrown at line 95, reproducing the target frame.

Botsing treated it as a valid solution and terminated the search process. However, the method would not return such a value if it was not stubbed. The generated test case is far-fetched and does not reveal any information about the bug either. Therefore the crash should not be considered reproduced.

We have identified 72 unsatisfactory reproductions for configuration [**IA**], 65 for [**IA-control**] respectively out of 3,840 runs. With Fisher's exact test, the resulting p-value is 0.5478. For configuration [**BV**], 182 unsatisfactory reproductions have been identified out of 7,380 runs, and 194 for [**BV-control**]. With Fisher's exact test, we have $p = 0.5656$. Therefore, Botsing's behaviour of generating unsatisfactory reproductions is not related to our proposed methods, and it is out of the scope of this thesis project to investigate the reason behind this behaviour. Relevant crashes and frames of those unsatisfactory reproductions are logged in our replication package, see Sub-section 5.4.4.

MATH-81b

Crash **MATH-81b** has seen improvement in the reproduction rate, but a decrease in the efficiency. Shown in Listing 5.4 is the input stack trace of it.

The interface `EigenDecomposition` defines methods to calculate the eigen decomposition of a real matrix. The provided implementation of it, `EigenDecompositionImpl`, translates an algorithm from a Fortran library LAPACK[1] to fulfil the interface. During the translation, several mistakes were made, resulting in the bug.

All inner five frames point to `private` methods, and the constructor `<init>` at frame 6 is the only `public` method in the stack trace. As described in Sub-section 2.3.2, the GGA can only target frames of `public` or `protected` methods. When the target method is `private`, it tries to invoke that method indirectly with `public` or `protected` methods. In the source code, the methods from the inner five frames have only one `public` indirect caller, which is the constructor from frame 6. Setting the *target frame* to any one of the inner five frames is in effect the same as setting it to the sixth frame. Therefore, all the 180 executions are grouped together for our evaluation.

Because of the fact that the exception is thrown after several complicated mathematical computations and it is buried six frames deep, to reproduce the crash, one needs to pro-

[1] `http://performance.netlib.org/lapack/`

```
java.lang.StringIndexOutOfBoundsException: String index out of range: 2
  at org.apache.commons.lang3.text.translate.CharSequenceTranslator.
    translate(CharSequenceTranslator.java:95)
  at org.apache.commons.lang3.text.translate.CharSequenceTranslator.
    translate(CharSequenceTranslator.java:59)
  at org.apache.commons.lang3.StringEscapeUtils.escapeCsv(
    StringEscapeUtils.java:556)
```

Listing 5.1: Stack Trace for LANG-6b

```
1 @Test(timeout = 4000)
2 public void test0() throws Throwable {
3   CharSequenceTranslator charSequenceTranslator0 = mock(
      CharSequenceTranslator.class, CALLS_REAL_METHODS);
4   doReturn(3663).when(charSequenceTranslator0).translate(any(java.lang.
      CharSequence.class), anyInt(), any(java.io.Writer.class));
5   StringWriter stringWriter0 = new StringWriter(609);
6   // Undeclared exception!
7   charSequenceTranslator0.translate((CharSequence) "261", (Writer)
      stringWriter0);
8 }
```

Listing 5.2: Generated Test Case for LANG-6b-1

```
75 public final void translate(CharSequence input, Writer out) throws
       IOException {
76   if (out == null) {
77     throw new IllegalArgumentException("The Writer must not be null");
78   }
79   if (input == null) {
80     return;
81   }
82   int pos = 0;
83   int len = input.length();
84   while (pos < len) {
85     int consumed = translate(input, pos, out);
86     if (consumed == 0) {
87       char[] c = Character.toChars(Character.codePointAt(input, pos));
88       out.write(c);
89       pos+= c.length;
90       continue;
91     }
92   // contract with translators is that they have to understand codepoints
93   // and they just took care of a surrogate pair
94     for (int pt = 0; pt < consumed; pt++) {
95       pos += Character.charCount(Character.codePointAt(input, pos));
96     }
97   }
98 }
```

Listing 5.3: Source code of the `translate` method from the first frame

```
java.lang.ArrayIndexOutOfBoundsException: -1
  at org.apache.commons.math.linear.EigenDecompositionImpl.
    computeShiftIncrement(EigenDecompositionImpl.java:1544)
  at org.apache.commons.math.linear.EigenDecompositionImpl.goodStep(
    EigenDecompositionImpl.java:1071)
  at org.apache.commons.math.linear.EigenDecompositionImpl.
    processGeneralBlock(EigenDecompositionImpl.java:893)
  at org.apache.commons.math.linear.EigenDecompositionImpl.
    findEigenvalues(EigenDecompositionImpl.java:657)
  at org.apache.commons.math.linear.EigenDecompositionImpl.decompose(
    EigenDecompositionImpl.java:246)
  at org.apache.commons.math.linear.EigenDecompositionImpl.<init>(
    EigenDecompositionImpl.java:205)
```

Listing 5.4: Stack Trace of MATH-81b

vide two sophisticatedly composed `double` arrays to the constructor. With `IntegrationTestingFF`, no guidance regarding how the two arrays should be mutated is provided. With `ITFFForIndexedAccess`, the distance of how far the accessed index is away from being out of bounds, indeed leads the search process to the optimal test case more often, as supported by the results. However, the increment in the reproduction rate, with an odds ratio of 1.5212, comes with a decrement in the efficiency, with $\hat{A}_{12} = 0.6194$.

LANG-19b

Crash **LANG-19b** has seen differences in both the reproduction rate and efficiency. The input stack trace of it contains 3 frames, as shown in Listing 5.5. And the VD.A measures of all 3 frames are shown in Table 5.7.

```
java.lang.StringIndexOutOfBoundsException: String index out of range: 19
  at org.apache.commons.lang3.text.translate.NumericEntityUnescaper.
    translate(NumericEntityUnescaper.java:54)
  at org.apache.commons.lang3.text.translate.CharSequenceTranslator.
    translate(CharSequenceTranslator.java:86)
  at org.apache.commons.lang3.text.translate.CharSequenceTranslator.
    translate(CharSequenceTranslator.java:59)
```

Listing 5.5: Stack Trace of LANG-19b

The `NumericEntityUnescaper` extends the `CharSequenceTranslator` and overwrites the `translate` method to translate an XML formatted numeric entity into a codepoint in a `String`. For example, "`Coke &#174;`" is translated into "Coke ®". Each numeric entity ends with a '`;`', and `NumericEntityUnescaper` locates the semi-column with the loop shown in Listing 5.6.

However, when a numeric entity is mal-formatted in the way that the closing semicolumn is missing, the loop continues increasing the local variable `end` till it is equal to the `length` of the `input` when a `StringIndexOutOfBoundsException` is thrown at line

```
52 int end = start;
53 // Comment
54 while(input.charAt(end) != ';')
55 {
56   end++;
57 }
```

Listing 5.6: Bug of LANG-19b

Table 5.7: Effect Size of LANG-19b

Frame	$\hat{A}_{12}$	p-value	Magnitude
3	0.1771	0.00191	large
2	0.2969	0.23737	medium
1	0.3229	0.04552	medium

54. Therefore, to reproduce the crash, one needs to call the `translate` method with a mal-formatted XML numerical entity.

Taking the example of "`&#174;`" again, one straightforward way that Botsing uses to mutate it is to append another valid numeric entity to it, which results in "`&#174;`" for example. During Botsing's search process, with the original `IntegrationTestingFF`, the two strings result in a fitness value of `1.0` as there is no exception thrown. Botsing keeps mutating the `input`. Chances are that the ending '`;`' will be removed and the crash will be reproduced.

With `ITFFForIndexedAccess`, the `length` of "`&#174;`" is `6` and the last visited index is `5`, which results in a fitness value of `0.2857`. As for "`&#174;`", it gets a `length` of `12` and the last visited index `11`. The fitness value is then reduced to `0.1538`. The decrement in the fitness value is heavily favoured by the selection process, and Botsing tends to append the well-formatted numeric entities longer and longer as the fitness value keeps decreasing. In the end, the search process is lead to a local optimum, with fitness values as small as $3.202e^{-4}$, and may never try the correct mutation of removing the ending '`;`'.

When the search process is not trapped, VD.A measures in Table 5.7 show a significant improvement for both the first frame and the third frame in terms of efficiency to reduce fitness value from 1.0 to 0.0.

LANG-51b

Both [**IA**] and [**IA-control**] have a reproduction rate of 100% with crash **LANG-51b**, however our proposed method performs significantly more efficient, with $\hat{A}_{12} = 0.3356$ while $p = 0.02847$. The input stack trace of it consists of one frame, as shown in Listing 5.7.

The method `toBoolean` converts a `String` object to a `boolean` value. The bug of the method lies in line 682 of Listing 5.8 as there is no `break` to the switch clause. For example, `"tru"` is of length 3. After failing the check for `case 3`, `false` should have already been returned. However, it continues into the check for `case 4` and there a `StringIndexOutOfBoundsException` is thrown at line 686 when accessing `charAt(3)`.

```
java.lang.StringIndexOutOfBoundsException: String index out of range: 3
  at org.apache.commons.lang.BooleanUtils.toBoolean(BooleanUtils.java
    :686)
```

Listing 5.7: Stack Trace of LANG-51b

```
649 public static boolean toBoolean(String str) {
    // ...
662   switch (str.length()) {
      // ...
670     case 3: {
671       char ch = str.charAt(0);
672       if (ch == 'y') {
673         return
674           (str.charAt(1) == 'e'  str.charAt(1) == 'E') &&
675           (str.charAt(2) == 's'  str.charAt(2) == 'S');
676       }
677       if (ch == 'Y') {
678         return
679           (str.charAt(1) == 'E'  str.charAt(1) == 'e') &&
680           (str.charAt(2) == 'S'  str.charAt(2) == 's');
681       }
682     }
683     case 4: {
684       char ch = str.charAt(0);
685       if (ch == 't') {
686         return
687           (str.charAt(1) == 'r'  str.charAt(1) == 'R') &&
688           (str.charAt(2) == 'u'  str.charAt(2) == 'U') &&
689           (str.charAt(3) == 'e'  str.charAt(3) == 'E');
690       }
691       if (ch == 'T') {
692         return
693           (str.charAt(1) == 'R'  str.charAt(1) == 'r') &&
694           (str.charAt(2) == 'U'  str.charAt(2) == 'u') &&
695           (str.charAt(3) == 'E'  str.charAt(3) == 'e');
696       }
697     }
698   }
699   return false;
700 }
```

Listing 5.8: Bug of LANG-51b

Any `String` of size 3 not starting with `'y'` or `'Y'` and any `String` of size 4 starting with `'t'` can reach line 686 easily. The original integration testing function cannot provide any further guidance after reaching the line. However, with our proposed fitness function outputting a more granular fitness value based on later accessed indices, it guides the search process to compose a `String` starting with `'t'`, then to `"tR"` or `"tr"`, and eventually to `"tRU"`, `"tRu"`, `"trU"`, or `"tru"` and triggering the exception being thrown.

LANG-54b and ES-21974

Frame **LANG-54b-1** and **ES-21974-5** are the only two frames with p-values less than 0.05 in the VD.A analyses between [**BV**] and [**BV-control**].

Frame **LANG-54b-1** is straightforward to reproduce as both configurations have a re-

production rate of 100% for frame **LANG-54b-1**. However, it has seen a significant efficiency drop with our proposed method, as shown in Table 5.6. Part of the source code of the method, where the exception is thrown, is already shown in Listing 4.3. On average, it takes 344.33 fitness evaluations for [**BV**] to cover the target frame. However, fairly early in the search process (7.3 fitness evaluations on average), 11 out of the 24 diversity objectives are already covered. Afterwards, new diversity objectives are rarely covered. Frame **ES-21974-5** has seen a significant improvement in the efficiency. One observation is that new diversity objectives have been covered throughout the search process.

Failure to Start the Search Process

As shown in Figure 5.2b, a large number of executions have failed to start the search process with configuration [**BV**]. Notice that for those frames, the control group has not reached the target line of the target frame either. These pre-processing failures can be grouped into two categories.

Fail to Instrument `JDK` Classes The first category consists of crashes where there are `JDK` classes in the stack trace. Crash **ES-23218** is an example, of which the stack trace is shown in Listing 5.9.

```
java.lang.IllegalStateException: No match found
  at java.util.regex.Matcher.group(Matcher.java:536)
  at org.elasticsearch.monitor.os.OsProbe.getControlGroups(OsProbe.java
    :216)
  at org.elasticsearch.monitor.os.OsProbe.getCgroup(OsProbe.java:414)
  at org.elasticsearch.monitor.os.OsProbe.osStats(OsProbe.java:466)
  at org.elasticsearch.monitor.os.OsService.<init>(OsService.java:45)
  at org.elasticsearch.monitor.MonitorService.<init>(MonitorService.java
    :45)
  at org.elasticsearch.node.Node.<init>(Node.java:345)
  at org.elasticsearch.node.Node.<init>(Node.java:232)
  at org.elasticsearch.bootstrap.Bootstrap$6.<init>(Bootstrap.java:241)
  at org.elasticsearch.bootstrap.Bootstrap.setup(Bootstrap.java:241)
  at org.elasticsearch.bootstrap.Bootstrap.init(Bootstrap.java:333)
  at org.elasticsearch.bootstrap.Elasticsearch.init(Elasticsearch.java
    :121)
```

Listing 5.9: Stack Trace of ES-23218

Due to the frequently released updates of `JDK`, Botsing developers decide not to instrument any `JDK` class to avoid discrepancies between line numbers reported in the stack trace and line numbers of the installed `JDK`. Therefore, when our `BranchingVariableDiversityFactory` queries the method node, a `NullPointerException` is thrown, interrupting the pre-processing. [**BV-control**] ignores the exception; however, it cannot move any further either without the necessary instrumentation.

Fail to Instrument Synthetic Methods The second category consists of crashes where the stack trace contains synthetic method frames. Crash **ES-26184** is an example, of which the stack trace is shown in Listing 5.10.

```
java.lang.IllegalArgumentException: invalid IP address [*] for [_ip]
  at org.elasticsearch.cluster.node.DiscoveryNodeFilters.lambda$static$0(
    DiscoveryNodeFilters.java:58)
  at org.elasticsearch.common.settings.Setting$3.get(Setting.java:908)
  at org.elasticsearch.common.settings.Setting$3.get(Setting.java:885)
  at org.elasticsearch.cluster.metadata.IndexMetaData$Builder.build(
    IndexMetaData.java:1026)
  at org.elasticsearch.cluster.metadata.IndexMetaData$Builder.
    fromXContent(IndexMetaData.java:1240)
  at org.elasticsearch.cluster.metadata.IndexMetaData$1.fromXContent(
    IndexMetaData.java:1302)
  at org.elasticsearch.cluster.metadata.IndexMetaData$1.fromXContent(
    IndexMetaData.java:1293)
  at org.elasticsearch.gateway.MetaDataStateFormat.read(
    MetaDataStateFormat.java:202)
  at org.elasticsearch.gateway.MetaDataStateFormat.loadLatestState(
    MetaDataStateFormat.java:322)
```

Listing 5.10: Stack Trace of ES-26184

The first frame points to a lambda expression, which is transcompiled into a synthetic method in bytecode. EvoSuite does not provide CFG for synthetic methods or classes. Hence, when our `BranchingVariableDiversityFactory` queries the method node, a `NullPointerException` is thrown, interrupting the pre-processing. The exception is ignored by [**BV-control**]. But it cannot proceed without the necessary instrumentation.

5.3.2 Summary

In this sub-section, we answer **RQ3** with the analysed observations.

Adapting Integration Testing Fitness Function for Indexed Access

By providing a continuous exception distance in the calculation of fitness values, our proposed method enhances the exploitation power of the search process when an individual shows a likelihood of accessed indices being out of bounds.

With 22 out of the 23 selected crashes, our proposed method does not show any deterioration in terms of crash reproduction effectiveness. The results suggest improvement with crashes where Botsing has difficulties in reaching the crash location, i.e. with crash **ES-23324** and **MATH-81b**. However, as the added exploitation power only kicks in after reaching the crash location, the number of executions taken into consideration is massively reduced and therefore cannot lead us to any solid conclusion. With the other crash, **LANG-19b**, the additional power of exploitation within the search space actually leads it to local optima and results in a worse reproduction rate.

In terms of crash reproduction efficiency, almost all the reproduced `ArrayIndexOutOfBoundsException` and `StringIndexOutOfBoundsException` crashes take less than ten fitness evaluations to reproduce, making it hard to observe notable changes. As shown in our manual analyses, with bugs of sophisticated logical errors, which require better-thought composition of inputs to trigger, the efficiency is generally dragged down by our proposed fitness function. On the other hand, bugs related to implementation mistakes that require less complicated input to trigger have seen a tendency of improvements in efficiency with the additional exploitation power.

Many-Objectivisation with Helper Objectives for Branching Variables

By reformulating the reproduction task into a many-objective optimisation with helper objectives problem, our approach tries to loose the constraints imposed on the search process and to empower its exploration ability.

The 246 frames of 29 crashes of `IllegalArgumentException` and `IllegalStateException` crashes have revealed no significant sign of improvement or deterioration in terms of reproduction effectiveness. Several executions have failed to start the search process due to technical difficulties in bytecode instrumentation with the provided framework from EvoSuite.

Branches that the original Botsing is already able to cover are now entered with a more diverse context (i.e. values of the branching variables). It results in a considerable improvement in terms of reproduction efficiency for several complex Elasticsearch crashes. If the coverage of diversity objectives goes in agreement with the search process, the number of fitness evaluations needed is significantly reduced. However, if all the possible diversity objectives are covered at the beginning of the search process, they, as a matter of fact, impose additional constraints and slow down the search process.

5.4 Threats to Validity

5.4.1 Internal Validity

We cannot guarantee that our extensions to Botsing and EvoSuite are defect-free, especially with the instrumentation components, which is an established challenge [58]. We mitigate this threat by unit testing the extensions and manually analysing the results.

Furthermore, to counteract the random nature of genetic algorithms, we have repeated each execution 30 times. However, the p-values of the two effect size analyses still suggest that it is not enough, especially with configuration [**IA**] and its control group, where our proposed fitness function only kicks in after reaching the crash location. For instance, crash **ES-23324** has the potential to show improvements. However, only 1 execution of [**IA**] and 2 executions of [**IA-control**] have managed to reach the crash location, and therefore only 3 data samples have been taken into account, way less than the suggested minimum by Arcuri and Briand [54].

5.4.2 External Validity

We cannot guarantee our results generalise to all crashes of the four studied types of exceptions. However, we have evaluated against all seven software projects included in JCrash-Pack. The seven projects differ much in terms of their sizes, complexities and functions, providing a wide variety, which mitigates this threat.

5.4.3 Conclusion Validity

The crashes used in our evaluation mostly result in two scenarios. One where the original Botsing can quickly reproduce with less than one hundred fitness evaluations with a reproduction rate of 100%. The other one where the original Botsing struggles to make any progress after initialisation. Because our proposed methods are extensions to the original Botsing, this phenomenon makes it difficult to observe significant differences in performance, threatening the conclusion validity. And some observations came out as counter-intuitive because they are considerably affected by marginal samples.

A future evaluation should consider not only the variety of software applications but also the variety of how easy it is for Botsing to solve. That is to say, not only crashes, that can be reproduced easily by Botsing or cannot be reproduced at all, should be included, but also crashes that Botsing can only reproduce with a possibility.

5.4.4 Replication Package

A replication package of our evaluation is available at `https://github.com/CoolTomatos/fit2crash-replication-package`. Along with scripts to run the evaluation and to analyse the outcome, the complete results and the full manual analyses are provided in this package. Our extension to Botsing is available at `https://github.com/stamp-project/Botsing`, and our extension to EvoSuite is available at `https://github.com/STAMP-project/evosuite-ramp`.

Chapter 6

Conclusions and Future Work

To specialise fitness functions used in search-based crash reproduction, we studied the four most common types of Java exceptions and grouped them into two categories. The first category is made of `ArrayIndexOutOfBoundsException`s and `StringIndexOutOfBoundsException`s, while the second category consists of `IllegalArgumentException`s and `IllegalStateException`s. In this chapter, by recalling the research questions, we summarise the contributions of this thesis project and draw conclusions from the results of our evaluation. After that, we discuss ideas for future work.

6.1 Contributions

The first research question is about specialising the exception distance according to the type of the target exception. It is stated as follows:

> ***RQ1*** *How to define a distance between the execution flow of a test case and the execution flow throwing a specific type of exception?*

For the first category of `ArrayIndexOutOfBoundsException`s and `StringIndexOutOfBoundsException`s, $d^*_{exception}(x)$ is defined to numerically represent how far an accessed index is away from being out of the bounds, see Definition 11.

And for the second category of `IllegalArgumentException`s and `IllegalStateException`s, we have defined a set of *branching variable diversity objective*s to broaden the diversity of values of branching variables, see Table 4.1. A set of diversity distances are defined to calculate the distance of one variable to meet the objectives, see Definition 13 to 20.

The second research question asks how to incorporate the newly defined distances into a fitness function. It is stated as follows:

> ***RQ2*** *How to include the additional information provided by this distance in a fitness function to improve the guidance of the search process?*

For the first category of `ArrayIndexOutOfBoundsException`s and `StringIndexOutOfBoundsException`s, we have implemented a customised method visitor, `IndexedAccessVisitor` in EvoSuite to collect the information. We define the *integration testing fitness function for indexed access* and replace the original binary exception distance with our $d^*_{exception}(x)$, see Definition 12. We have implemented it in Botsing. After reaching the crash location, the newly defined exception distance is used to make the fitness value continuous.

For the second category of `IllegalArgumentException`s and `IllegalStateException`s, we have reformulated the fitness function into a many-objective one. We have implemented `BranchingVariableDiversityFactory` in Botsing to create *branching variable diversity objective*s as helper objectives. The original integration testing fitness function is used as the primary objective. We have added a customised instrumentor, `BranchingVariableInstrumentation`, to EvoSuite to collect the information. And finally, we have adjusted the guided MOSA so that the search process will terminate once the primary objective is met.

6.2 Conclusions

Based on JCrashPack, we have performed an evaluation of 11 days on 52 real-world crashes from 7 various open-source Java applications to answer the third research question, which is stated as follow:

> ***RQ3*** *What is the impact of the new fitness function in terms of effectiveness and efficiency of search-based crash reproduction?*

For the first category of `ArrayIndexOutOfBoundsException`s and `StringIndexOutOfBoundsException`s, the result of our evaluation suggests that the additional exploitation power brought by the *integration testing fitness function for indexed access* tends to improve the effectiveness of crash reproduction, at the risk of getting trapped in local optima. We have shown that the efficiency decreases when the crash requires sophisticated input to trigger, while the efficiency increases when sophisticated inputs are not required.

For the second category of `IllegalArgumentException`s and `IllegalStateException`s, the result reveals no significant improvement nor deterioration in crash reproduction effectiveness by adding helper objectives to enforce the branching variable diversity. However, it suggests a moderate enhancement in terms of efficiency.

6.3 Future Work

The project itself has much room for improvement, and the research in the field of automated crash reproduction with search-based optimisation algorithms still has much to be discovered. In this section, we list some recommendations of possible directions for future work.

- As described in Sub-section 4.1.2, only methods of a subset of `JDK` classes that throw `StringIndexOutOfBoundsException`s are instrumented. And as described in Sub-section 4.2.4, currently comparison operands of fields of an object or static fields of a class are ignored in the instrumentation. Future work of the project can extend the added instrumentation components of EvoSuite to include the missing methods and fields. To work the latter one out, one might need to incorporate mocking mechanisms for the states of one object or class.

- Currently, with the *integration testing fitness function for indexed access*, the added exploitation power only kicks in when the search process manages to reach the crash location (i.e. the target line of the inner-most frame of the stack trace). However, for many crashes, Botsing does not have the exploration power to reach the location in the first place. Future work should, therefore, improve the exploration power. It can be done by (i) relaxing the constraints [59]; (ii) combining different evolutionary algorithms [30]; and (iii) defining new genetic operators [30].

- Currently, with the *branching variable diversity factory*, every single branching variable in the CFG it runs into is instrumented. However, some variables are not relevant to the throwing of the target exception, and some later diversity objectives can only be achieved if a previous objective is met. Future work can incorporate backward path-sensitive analysing [53] to identify more relevant variables and utilise DynaMOSA [38] to add follow-up objectives dynamically based on the different paths the generated test cases are taking. And the objective factory can take the constant one branching variable is compared with into consideration for defining the diversity objectives.

- The central idea of the project is to study the patterns how the four most common types of Java run-time exceptions are thrown. To specialise fitness functions for other exceptions, new patterns must be identified. However, human perception of programming patterns can be limited. Future work in the field can utilise machine learning to cluster different types of exceptions based on their throwing patterns and extract feature to be used in the exception distance $d_{exception}(x)$ for specialised fitness functions.

Bibliography

[1] Paolo Tonella. Evolutionary testing of classes. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 119–128. ACM, 2004.

[2] Nigel Tracey, John Clark, Keith Mander, and John McDermid. An automated framework for structural test-data generation. In *Proceedings 13th IEEE International Conference on Automated Software Engineering (Cat. No. 98EX239)*, pages 285–288. IEEE, 1998.

[3] Mozhan Soltani, Pouria Derakhshanfar, Xavier Devroey, and Arie van Deursen. A benchmark-based evaluation of search-based crash reproduction. *Empirical Software Engineering*, pages 1–43, 2019.

[4] Strategic Planning. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, 2002.

[5] Uttamjit Kaur and Gagandeep Singh. A review on software maintenance issues and how to reduce maintenance efforts. *International Journal of Computer Applications*, 118(1), 2015.

[6] Andy Pasztor, Andrew Tangel, Robert Wall, and A Slider. How boeing's 737 max failed. *The Wall Street Journal (Mar. 2019)(cit. on p. 11)*, 2019.

[7] Andrew J Ko, Bryan Dosono, and Neeraja Duriseti. Thirty years of software problems in the news. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 32–39, 2014.

[8] Hermann Kopetz et al. *Software reliability*. Macmillan International Higher Education, 2016.

[9] J Myers Glenford and S Myers. Software reliability principles and practices, 1976.

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

[11] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. *jRapture: A capture/replay tool for observation-based testing*, volume 25. ACM, 2000.

[12] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 284–295. IEEE Computer Society, 2005.

[13] David Saff, Shay Artzi, Jeff H Perkins, and Michael D Ernst. Automatic test factoring for java. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 114–123. ACM, 2005.

[14] 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.

[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] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K Chang. Ocat: object capture-based automated testing. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 159–170. ACM, 2010.

[17] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, pages 53–64, 2010.

[18] Pablo Montesinos, Matthew Hicks, Samuel T King, and Josep Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, pages 73–84, 2009.

[19] Mathieu Nayrolles, Abdelwahab Hamou-Lhadj, Sofiène Tahar, and Alf Larsson. Jcharming: A bug reproduction approach using crash traces and directed model checking. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 101–110. IEEE, 2015.

[20] Andreas Leitner, Alexander Pretschner, Stefan Mori, Bertrand Meyer, and Manuel Oriol. On the effectiveness of test extraction without overhead. In *2009 International Conference on Software Testing Verification and Validation*, pages 416–425. IEEE, 2009.

[21] Cristian Zamfir and George Candea. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems*, pages 321–334. ACM, 2010.

[22] 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.

[23] 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.

[24] Fitsum Meshesha Kifetew, Wei Jin, Roberto Tiella, Alessandro Orso, and Paolo Tonella. Sbfr: A search based approach for reproducing failures of programs with grammar based input. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 604–609. IEEE, 2013.

[25] Mozhan Soltani, Annibale Panichella, and Arie Van Deursen. A guided genetic algorithm for automated crash reproduction. In *Proceedings of the 39th International Conference on Software Engineering*, pages 209–220. IEEE Press, 2017.

[26] Mozhan Soltani, Annibale Panichella, and Arie Van Deursen. Search-based crash reproduction and its impact on debugging. *IEEE Transactions on Software Engineering*, 2018.

[27] 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. ACM, 2011.

[28] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[29] John Henry Holland et al. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[30] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. Exploration and exploitation in evolutionary algorithms: A survey. *ACM Computing Surveys (CSUR)*, 45(3):35, 2013.

[31] Webb Miller and David L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, (3):223–226, 1976.

[32] Oliver Buehler and Joachim Wegener. Evolutionary functional testing of an automated parking system. In *Proceedings of the International Conference on Computer, Communication and Control Technologies (CCCT'03) and the 9th. International Conference on Information Systems Analysis and Synthesis (ISAS'03), Florida, USA*, 2003.

[33] Joachim Wegener, André Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and software technology*, 43(14): 841–854, 2001.

[34] Phil McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.

[35] Andrea Arcuri. It does matter how you normalise the branch distance in search based software testing. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 205–214. IEEE, 2010.

[36] Joshua D Knowles, Richard A Watson, and David W Corne. Reducing local optima in single-objective problems by multi-objectivization. In *International conference on evolutionary multi-criterion optimization*, pages 269–283. Springer, 2001.

[37] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*, pages 1–10. IEEE, 2015.

[38] 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.

[39] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[40] David S Rosenblum et al. Towards a method of programming with assertions. In *14th International Conference on Software Engineering: Proceedings*, page 92. Association for Computing Machinery, 1992.

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

[42] 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.

[43] STAMP-project. Background of botsing — crash reproduction made easy! `https://stamp-project.github.io/botsing/pages/background.html`, 2019.

[44] Mozhan Soltani, Pouria Derakhshanfar, Annibale Panichella, Xavier Devroey, Andy Zaidman, and Arie van Deursen. Single-objective versus multi-objectivized optimization for evolutionary crash reproduction. In *International Symposium on Search Based Software Engineering*, pages 325–340. Springer, 2018.

[45] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[46] Roberta Coelho, Lucas Almeida, Georgios Gousios, and Arie van Deursen. Unveiling exception handling bug hazards in android based on github and google code issues. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 134–145. IEEE, 2015.

[47] Arrayindexoutofboundsexception (java plateform se 8). https://docs.oracle.com/javase/8/docs/api/java/lang/ArrayIndexOutOfBoundsException.html.

[48] Stringindexoutofboundsexception (java plateform se 8). https://docs.oracle.com/javase/8/docs/api/java/lang/StringIndexOutOfBoundsException.html.

[49] Illegalstateexception (java plateform se 8). https://docs.oracle.com/javase/8/docs/api/java/lang/IllegalStateException.html.

[50] Illegalargumentexception (java plateform se 8). https://docs.oracle.com/javase/8/docs/api/java/lang/IllegalArgumentException.html.

[51] Nullpointerexception (java plateform se 8). https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html.

[52] Goetz Lindenmaier and Ralf Schmelter. Jep 358: Helpful nullpointerexceptions. https://openjdk.java.net/jeps/358, 2019.

[53] Daniele Romano, Massimiliano Di Penta, and Giuliano Antoniol. An approach for search based testing of null pointer exceptions. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 160–169. IEEE, 2011.

[54] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[55] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[56] 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.

[57] 3tu: Big software on the run. http://www.3tu-bsr.nl/.

[58] 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.

[59] Carlos A Coello Coello. Constraint-handling techniques used with evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 675–701, 2017.

Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

AL: Approach Level

BCET: Best-Case Execution Time

BD: Branch Distance

CFG: Control Flow Graph

CUT: Class Under Test

FF: Fitness Function

GGA: Guided Genetic Algorithm

IDE: Integrated Development Environment

ITFF: Integration Testing Fitness Function

ITFFForIndexedAccess: Integration Testing Fitness Function for Indexed Access

JDK: Java Development Kit

JVM: Java Virtual Machine

MOSA: Many-Objective Sorting Algorithm

NSGA-II: Elitist Non-dominated Sorting Genetic Algorithm

SBST: Search-Based Software Test

SUT: Software Under Test

VD.A: Vargha and Delaney's $\hat{A}_{12}$ Measure

WCET: Worst-Case Execution Time