Carving Information Sources to Drive Search-Based Crash Reproduction and Test Case Generation

Carving Information Sources to Drive Search-Based Crash Reproduction and Test Case Generation

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof. dr. ir. T.H.J.J. van der Hagen,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen
op donderdag 22 april 2021 om 10.00 uur

door

Pouria DERAKHSHANFAR

Master of Science in Computer Engineering,
Sharif University of Technology Tehran, Iran,
geboren te Tehran, Iran.

Dr. X. Devroey has, as daily-supervisor, contributed significantly to the preparation of this dissertation.
The work in the thesis has been partially funded by the EU Project STAMP ICT-16-10 No.731529.
The work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Keywords: Search-based Software Testing, Crash Reproduction, Class Integration Testing, Carving Information Sources

Printed by: IPSKAMP printing (`www.proefschriften.net`)

Cover: 'Genetic gradient' by Atefeh Alaeddin and Arefeh Alaeddin

Style: TU Delft House Style, with modifications by Moritz Beller
`https://github.com/Inventitech/phd-thesis-template`

The author set this thesis in LaTeX using the Libertinus and Inconsolata fonts.

ISBN 978-94-6421-312-6

An electronic version of this dissertation is available at
`http://repository.tudelft.nl/`.

As you start to walk on the way, the way appears.

Rumi

Contents

[2] This section is partly based on B. Cherry, X. Devroey, P. Derakhshanfar, and B. Vanderose. Crash reproduction difficulty, an initial assessment, BENEVOL'20 [1]

Summary

Software testing plays a crucial role in software development to improve the software's consistency and performance. Since software testing activities demand considerable effort in the development process, many automated techniques have been introduced to aid developers and testers in various testing phases, thereby reducing the costs related to these tasks. One category of these automated approaches seeks to generate software tests automatically using different strategies. One of the successful strategies, which is applied to industrial cases, uses metaheuristic search-based approaches for test generation automation. These approaches use various search techniques to produce tests for different levels, such as unit testing and system-level tests. The assessments of these techniques confirm their usefulness in fault detection and debugging practices. However, most of these techniques use structural coverage criteria (*e.g.,* line and branch coverage) for test generation. Despite the usefulness of these general criteria, it has been shown that they are not always enough for revealing faults. Previous studies show that these criteria have a fault detection likelihood of about 50%.

This thesis investigates the application of novel search objectives and search-based test generation methods, based on information carved from multiple sources (*e.g.,* source code, hand-written tests, *etc.*), on search-based test generation. In particular, in the first part of the thesis, we introduce new search objectives and methods to cover an instance of specific software behavior called crash reproduction. Then, we present a new search-based approach for testing integration points between two coupled classes to find class integration-level faults. Finally, we propose new search objectives to generate unit-level tests exercising the software common and uncommon execution patterns observed during the software operation.

Our results regarding the assessment of new search-based crash reproduction strategies show that these introduced techniques improve the search process's effectiveness and efficiency. In other words, these techniques drove the state-of-the-art in search-based crash reproduction to reproduce more crashes and more quickly. Moreover, evaluating the novel search-based class-integration test generation approach indicates that this approach complements the state-of-the-art search-based unit test generation in fault detection. Finally, this thesis reports mixed results for the search objectives introduced for exercising common and uncommon execution patterns. We observed that these objectives improve the mutation score achieved by the generated tests in some cases, while we see the opposite in some other cases.

In summary, this thesis introduced new techniques for search-based test generation by looking at the existing knowledge carved from different resources. The results reported in this thesis confirm these approaches' positive impact on generating tests covering undetected bugs and faults with higher efficiency. This thesis is a step towards the development of fully-automated tools helping developers in software testing.

Samenvatting

Het testen van software speelt een cruciale rol bij softwareontwikkeling om de consistentie en prestaties van de software te verbeteren. Aangezien het testen van software een aanzienlijke inspanning vergt in het ontwikkelingsproces, zijn er meerdere geautomatiseerde technieken voorgesteld om ontwikkelaars en testers te helpen in verschillende testfasen, waardoor de kosten die hieraan gekoppeld zijn, verlaagd worden. Een categorie van deze geautomatiseerde technieken tracht om automatisch softwaretests te genereren met behulp van verschillende strategieën. Een strategie die hier succesvol in is, welke ook toegepast wordt voor industriële doeleinden, maakt gebruik van meta heuristische zoek gebaseerde benaderingen voor het automatisch genereren van deze testen. Deze benaderingen gebruiken verschillende zoektechnieken om tests te produceren voor verschillende niveaus, zoals unit-tests en tests op systeemniveau. De resultaten die verkregen worden met deze technieken bevestigen hun nut bij het opsporen van fouten en het debuggen van de code. De meeste van deze technieken gebruiken echter structurele dekkingscriteria (bijv. Dekking van lijnen en aftakkingen) voor het genereren van de tests. Ondanks het nut van deze algemene criteria is aangetoond dat ze niet altijd voldoende zijn voor het ontdekken van fouten. Eerdere studies hebben aangetoond dat deze criteria fouten kunnen opsporen met een kans van 50%.

Dit proefschrift onderzoekt de toepassing van nieuwe zoekdoelstellingen en zoek gebaseerde testgeneratiemethoden, gebaseerd op informatie uit meerdere bronnen (bijv. Broncode, handgeschreven tests, enz.), op zoek gebaseerde testgeneratie. In het bijzonder introduceren we in het eerste deel van het proefschrift nieuwe zoekdoelen en methoden om een specifiek soort softwaregedrag, genaamd crashreproductie, te behandelen. Vervolgens presenteren we een nieuwe zoek gebaseerde benadering voor het testen van de integratiepunten tussen twee gekoppelde klassen om fouten op klasse-integratieniveau te vinden. Ten slotte stellen we nieuwe zoekdoelen voor om tests op unit-niveau te genereren door gebruik te maken van de veelvoorkomende en ongebruikelijke uitvoeringspatronen van de software die worden waargenomen tijdens de werking van de software.

Onze resultaten betreffende de beoordeling van nieuwe zoek gerelateerde crashreproductiestrategieën laten zien dat deze geïntroduceerde technieken de effectiviteit en efficiëntie van het zoekproces verbeteren. Met andere woorden, deze technieken hebben ervoor gezorgd dat de state-of-the-art in zoek gebaseerde crashreproductie meer crashes kan vinden en dit ook sneller doet. Bovendien geeft de evaluatie van deze nieuwe methode voor het genereren van klasse-integratietests aan dat deze benadering een aanvulling vormt op de state-of-the-art.

Ten slotte rapporteert dit proefschrift gemengde resultaten voor de zoekdoelen die zijn geïntroduceerd voor het uitvoeren van veelvoorkomende en ongebruikelijke uitvoeringspatronen. We hebben vastgesteld dat deze doelstellingen in sommige gevallen de mutatiescore verbeteren, terwijl we in sommige andere gevallen het tegenovergestelde zien.

Samenvattend introduceerde dit proefschrift nieuwe technieken voor zoek gebaseerde testgeneratie door te kijken naar de bestaande kennis die verworven wordt uit verschillend bronnen. De resultaten die in dit proefschrift worden gerapporteerd, bevestigen de positieve impact van deze technieken op het genereren van tests voor niet-gedetecteerde bugs en fouten met een hogere efficiëntie. Dit proefschrift is een stap in de richting van de ontwikkeling van volledig geautomatiseerde tools die ontwikkelaars helpen bij het testen van software.

Acknowledgments

When I moved to the Netherlands to start my Ph.D., I knew that I am here to learn many new materials about doing research on Software Engineering (especially Software Testing). However, I did not realize that this journey will also teach me lots of valuable lessons that will change my life. When I look back, I see that this path helped me to know myself and my life much better. I owe all of these to all the people who helped during my Ph.D. First of all, I should thank my supervisors/promoters Andy, Xavier, Annibale, and Arie: thank you for everything. Without your supervision and supports, I would not achieve to this point. Any meeting with you was a class for me, and this dissertation wouldn't be like this without your feedback and revisions. I also want to thank the committee members for reading and helping to improve this thesis.

In the following paragraphs, I would like to express my appreciation to some of the people who had significant roles in this beautiful yet challenging adventure. Please accept my apologies if I did miss out on some of the names. I am thankful for any person who helped me in the last three and a half years.

Andy: as I always advise any person who is searching for a Ph.D. position, the first important factor in improving during your Ph.D. is a supervisor who believes in you, supports you, and gives you the freedom to find your research path while making sure that you are not taking the wrong way. This is the lesson that I learned when I worked with you. You always helped me with even the most minor issues I had, even though you had a busy schedule. Even the first day that I talked with you, I was so nervous that I couldn't even speak proper English. However, after 5 minutes, you gave me enough confidence to pull myself together. I am always thankful for all of your help and supports.

Xavier: you are both one of the best colleagues and one of the best friends that I have had in my life. During my Ph.D., You were there for me every single day that I wanted to talk with you. The talk could be either about a new crazy idea for the next paper or even when I just wanted to speak with someone about my personal issues. In both cases, you helped me as a supportive friend. I am that much lucky that I cannot say if this thesis is my most significant achievement during my Ph.D. or our friendship. Thank you for everything, including being my beer sommelier ☺🍺.

Annibale: you are highly knowledgeable as a senior researcher. But at the same time, you can be as cool as a junior bachelor student :D I managed to study the concepts of search-based software engineering by reading the relevant papers. Still, I started to gain more profound knowledge about this topic after having regular meetings with you. Thank you for all of the chats that we had about either research or general things. By the way, with all of the respect, I still think that a broken pasta which is cooked two times is more delicious :)).

Arie: being part of the Software Engineering Research Group was one of the best experiences that I had in my life. I always appreciate the chance that this group gave me to have a Ph.D. here. Moreover, I am also thankful for your guidance, feedback, and revisions for my research. Also, Thanks to you for teaching me to have impactful communication.

Enrique: you are one of the kindest persons that I have ever seen in my life. I hope that we can hang out again in the near future.

Mehdi: having a cool friend with the same nationality is always a blessing in an international group. Thank you for being that person for me ☺.

Luís: it is always fascinating for me to find a person who likes the same type of music as me. I really appreciate that I found my officemate as that person. I will never forget all of the freaky pieces of music that we have listened, while we were working at the office ♫.

Mitchell thank you for helping me whenever I needed help (including translating the summary of this thesis). Also, hanging out with you was always fun.

Carolin your positiveness and the good mood was always encouraging for me ☺. Thank you for that.

Gilles: visiting Namur university was quite pleasant and exciting for me. Thank you for inviting me there. Also, thank you for collaborating with me. Hopefully, we can meet again soon and brainstorm about future collaborations while we are having a beer.

Minaksie: during my Ph.D., I was always sure that anything which leads to you will be smoothly handled. Thank you for making every paperwork much easier ☺.

Davide, *Jean*, *Luca*, *Marco*, *Anand*, *Joseph*, and *Leonhard*: thank you for all of the coffee breaks. Hanging out with you guys was always refreshing and lovely.

I would like to also thank the Master students who worked with Xavier and me for their thesis: *Boris*, *Shang*, *Björn*, and *Sven*. I wish you success in your future careers.

Atefeh: I don't know how to thank you for all of the things you have done for me. When I have started my Ph.D., you were utterly supportive about that. Your patience, kindness, and each of your sacrifices were an inspiration for me. We made the impossible, possible. We have passed many hard days, but finally, we tackled all of them. I loved you for any second that I missed you while we were away from each other. And, I will love you for any second that we will spend with each other in our future.

Last but not least, my parents, *Maman Baba*, thank you for supporting me while I was studying overseas and away from you. Without you, I would not be here. I hope that you are proud of me.

This adventure had its ups and downs for me. I am happy that I have experienced it, and now I am finishing it with writing this thesis. Now, I am ready to start a new chapter of my life. Again, Thank you, everybody.

Pouria
Delft, March 2021

1 Introduction

Software testing is one of the essential and expensive tasks in software development. Hence, many approaches were introduced to automate different software testing tasks. Among these techniques, search-based test generation techniques have been vastly applied in real-world cases and have shown promising results. These strategies apply search-based methods for generating tests according to various test criteria such as line and branch coverage.

In this thesis, we introduce new search objectives and techniques using various knowledge carved from resources like source code, hand-written test cases, and execution logs. These novel search objectives and approaches (i) improve the state-of-the-art in search-based crash reproduction, (ii) present a new search-based approach to generate class-integration tests covering interactions between two given classes., and (iii) introduce two new search objectives for covering common/uncommon execution patterns observed during the software production.

This chapter is partly based on 🗎 P. Derakhshanfar. Well-informed Test Case Generation and Crash Reproduction, ICST'20 (Doctoral Symposium) [2].

Software testing is an indispensable part of software engineering, widely studied from various aspects by researchers in this field. As mentioned by Bertolino [3], one of the biggest dreams in software testing research is 100% automatic testing, and one of the research paths towards reaching this dream is the automation of test generation.

A survey by McMinn [4] shows that search-based software testing techniques are applicable to a vast range of automated software testing problems, including automated test generation. The application of metaheuristic approaches for automating the process of software test generation has been an interesting research path in recent years. The approaches model the software testing goals, which should be achieved by hand-written test cases, into optimization problems and solve them using search algorithms [5].

The approaches aim to produce tests for different levels of testing. For instance, many approaches are proposed for unit testing [6–9] and system-level testing [10–15]. Moreover, these approaches can be categorized into white-box [6–10], grey-box [16–19], and black-box [11–15] testing techniques.

The evaluations that were performed indicate the usefulness of the generated tests. More specifically, the generated tests can not only achieve high structural and mutation coverage [20, 21], but are also helpful for catching faults [22] and debugging [23]. They have also been successfully deployed in industry [24, 25].

Most of these approaches aim at a general coverage criteria (*e.g.,* line and branch coverage). However, generated tests with high structural coverage are not always successful at detecting faults. Gay *et al.* [26] have shown that these types of coverage criteria are poor indicators for failure detection and mutation score in some cases. As an example, a test case can cover a statement without passing failure revealing data. In this case, we have the coverage, but the fault will remain undetected without the adequate test oracle. Moreover, Shamshiri *et al.* [22] reported that the tests generated by EvoSuite, which is one of the better automatic unit test generation tools, are only successful in exposing about 50% of industrial faults despite the high structural coverage scores.

Furthermore, search-based test generation for specific problems has a lot of open challenges. Among them, fitness functions defined for search-based test generation suffer from a lack of guidance and underuse contextual information. In particular, Salahirad *et al.* [27] indicated that the strongest fitness function (branch coverage) has about 25% likelihood of fault detection. As an outcome, they suggest using classical branch and line coverage as primary objectives and using other objectives that aid to trigger the faults as secondary objectives.

In this thesis, we go beyond classical search objectives that aim at maximizing structural coverage. In particular, we investigate how information collected from different sources (*i.e.,* source code, hand-written tests, *etc.*) can help to design and reinforce search objectives. In doing so, we hypothesize that we can exercise specific behaviors, and thus trigger specific kinds of faults.

First, we focus on one of the instances of test generation for specific software behaviors: **search-based crash reproduction**. These crash reproduction approaches [28–33] accept crash-related data as input and generate a test that reproduces this given crash. Our studies on crash reproduction focus on leveraging the collected contextual information to improve the effectiveness (*i.e.,* the number of reproduced crashes and how often they can be reproduced) and efficiency (*i.e.,* the time required to reproduce a crash) of the search process, trying to reproduce the given crash.

Second, we concentrate on generating tests for **exercising integration points between two classes**. We consider the execution of different scenarios in the interaction between two coupled classes as test objectives for the test generation process. We introduce a new test criterion for class integration testing, which is suitable for defining search objectives. Then, we design a search-based test generation algorithm according to this newly defined criterion. We investigate if this algorithm can reveal integration level faults which are not detectable with search-based unit testing.

Finally, we investigate a novel search objective for search-based unit testing that covers the **common and uncommon execution patterns**. To detect the common and uncommon patterns, we monitor the execution patterns during the operation of the software.

1.1 Background & Context

This section presents an overview of search-based software test generation and automated crash reproduction and how they are connected to this thesis.

1.1.1 Search-based Software Test Generation

McMinn [4] defined search-based software testing (SBST) as *“using a meta-heuristic optimizing search technique, such as a genetic algorithm, to automate or partially automate a testing task”*. Within this realm, test data generation at different testing levels (such as *unit testing*, *integration testing*, *etc.*) has been actively investigated [4]. This section provides an overview of earlier work in this area.

One of the most successfully used optimization techniques in search-based test generation is the genetic algorithm [34]. Figure 1.1 depicts how search-based test generation techniques uses genetic algorithm for test generation.

In the first step, the algorithm generates a population of individuals. Each individual is either a single test case or test suite (a set of test cases). These individuals can be entirely randomly generated, or the algorithm can use seeding strategies in which it uses some information, such as hand-written tests, and generate tests using these existing data. Then, each generated individual is evaluated for fitness using one or multiple fitness functions (box 2 in Figure 1.1). Next, it uses the fittest individuals (according to fitness function(s)) to generate the next population of individuals (box 3).

For producing the next generation of individuals, first, the algorithm gets the individuals with the best fitness values (box 3.1 in Figure 1.1). Then, it uses two genetic operators for generating new tests: *Crossover* (box 3.2 in Figure 1.1), which combines two selected individuals (parents) to create two new individuals (offsprings), and *Mutation* (box 3.3 in Figure 1.1), which add/remove/modify one or more statements in a selected test to generate a new one. Finally, the newly generated individuals are re-inserted into a new population (box 3.4 in Figure 1.1).

The iteration between fitness evaluation (box 2 in Figure 1.1) and producing the next generation (box 3 in Figure 1.1) will continue until either the allocated budget is exhausted

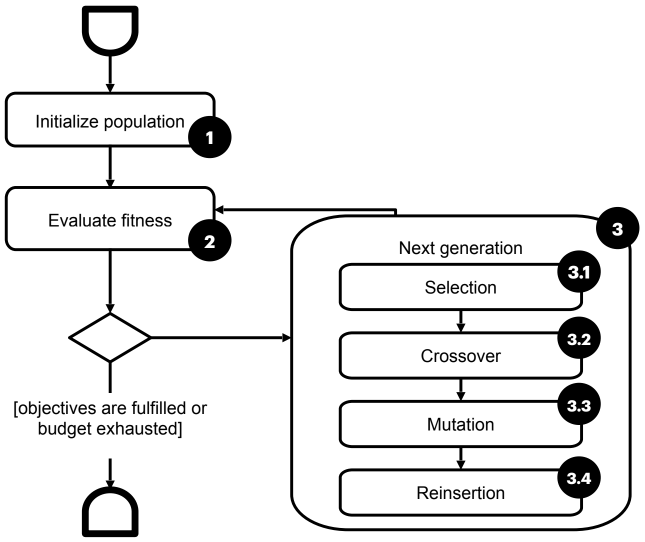


Figure 1.1: General overview of search-based test generation techniques using genetic algorithm.

or all of the search objectives (represented as fitness functions) are fulfilled.

Search-based approaches for unit testing

Search-based software test generation algorithms have been extensively used for unit test generation. Previous studies confirmed that thus generated tests achieve a high code coverage [20, 35], real-bug detection [25], and debugging cost reduction [36, 37], complementing hand-written tests. Also, a recent study by Panichella *et al.* [38] empirically showed that the unit tests generated by search-based test generation techniques have a low rate of test smell occurrence.

From McMinn's [4] survey about search-based test data generation, we observe that most of the current approaches rely on the control flow graph (CFG) to abstract the source code and represent possible execution flows. The $CFG_m = (N_m, E_m)$ represents a method[1] m as a directed graph of **basic blocks** (*i.e.,* sequences of statements execute one after each other) of code (the nodes N_m), while E_m is the set of the control flow edges. An edge connects a basic block n_1 to another one n_2 if the control may flow from the last statement of n_1 to the first statement of n_2.

Listing 1.1 presents the source code of `Person`, a class representing a person and they transportation habits. A `Person` can drive home (lines 4-10), or add energy to her car (lines 12-18). Figure 1.2 presents the CFG of two of Person's methods, with the labels of the nodes representing the line numbers in the code. Since method `driveToHome` calls method `addEnergy`, `node 6` is transformed to two nodes, which are connected to the entry

[1]Or function in procedural programming languages.

Example 1.1: Class Person

```java
class Person{
    private Car car = new Car();
    protected boolean lazy = false;
    public void driveToHome(){
        if (car.fuelAmount < 100) {
            addEnergy();
        } else {
            car.drive();
        }
    }

    protected void addEnergy(){
        if (this.lazy) {
            takeBus();
        } else {
            car.refuel();
        }
    }
  }
```

and exit point of the called method. This transformation is explained in the last paragraph of this section.

Many approaches based on CFGs combine two common heuristics to reach a high branch and statement coverage in unit-level testing. These two heuristics are the *approach level* and the *branch distance*. The *branch distance* measures (based on a set of rules) the distance to *satisfying* (true branch) and the distance to *not satisfying* (false branch) a particular branching node in the program.

The *approach level* measures the distance between the execution path and a target node in a CFG. For that, it relies on the concepts of **post-dominance** and **control dependency** [39]. A *node A* is control dependent on *node B* in a control flow graph if *node B* contains a branch, which can change the execution path away from reaching *node A*.

As an example, in Figure 1.2, *node 8* is control dependent on *node 5* and *node 8* post-dominates edge $\langle 5, 8 \rangle$. The *approach level* is the minimum number of control dependencies between a target node and an executed path by a test case.

One of the best tools for performing search-based test generation is EvoSuite [40]. This tool gets a Java application and one of its classes as a class under test. Then, it starts a genetic algorithm to generate a test suite fulfilling different testing criteria for the given class. This tool contains multiple genetic algorithms [41, 42]. Also, it can generate tests according to various criteria such as line coverage, branch coverage, weak mutation, *etc.*

This thesis leverages the information carved from different sources (*e.g.,* hand-written tests, source code, *etc.*) to introduce search objectives complementing approach level and branch distance for different test generation scenarios.

Evolutionary-based approaches for integration testing

Search-based approaches are widely used for test ordering [43–46, 46–54], typically with the aim of executing those tests with the highest likelihood of failing earlier on. However, to the best of our knowledge, search-based approaches have rarely been used for generating integration tests. Ali Khan *et al.* [55] have proposed a high-level evolutionary approach that detects coupling paths in data-flow graphs of classes and generates tests for the detected coupling paths. Moreover, they proposed another approach for the same

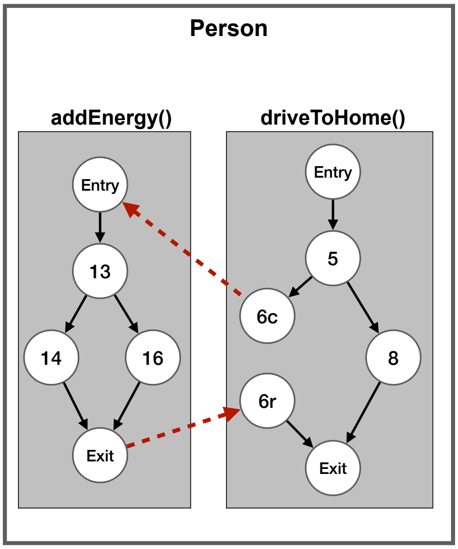


Figure 1.2: Class-level CFG for class `Person`

goal, which uses Particle Swarm Optimization [56]. However, the paper does not describe the fitness function and genetic algorithm used in their approach, nor any evaluation for examining the quality of the tests generated by this approach. The paper also does not check whether the tests can complement tests generated by existing search-based unit testing approaches. Besides, since objectives are defined according to the *def-use* paths between classes, the number of search objectives can grow exponentially, thus severely limiting the scalability of the approach.

In Chapter 6 we propose a novel approach for class integration test generation. Instead of using the data flow graph, which is relatively expensive to construct as it needs to find the coupling paths, we use the information available about the the integration between classes to calculate the fitness of the generated tests.

Search-based approaches for other testing levels

Arcuri [10] proposed EvoMaster, an evolutionary-based white-box approach for system-level test generation for RESTful APIs. A test for a RESTful web service is a sequence of HTTP requests. EvoMaster tries to cover three types of targets: (i) the statements in the System Under Test (SUT); (ii) the branches in the SUT; and (iii) different returned HTTP status codes. Although EvoMaster tests different classes in the SUT, it does not systematically target different integration scenarios between classes.

In contrast to EvoMaster, other approaches perform fuzzing [11], *"an automated technique providing random data as input to a software system in the hope to expose a vulnerability."* Fuzzing uses information like grammar specifications [11, 13–15] or feedback from the program during the execution of tests [12] to steer the test generation process. These

Example 1.2: XWIKI-13377 crash stack trace [59]

```
java.lang.ClassCastException: [...]
    at [...].BaseStringProperty.setValue([...]:45) (@@)
    at [...].PropertyClass.fromValue([...]:615) (@@)
    at [...].BaseClass.fromMap([...]:413)
    [...]
```

fuzzing approaches generate only input data but do not generate a full test case containing method call sequences.

The approaches introduced in this thesis perform white-box testing.

1.1.2 Search-based Crash Reproduction [2]

Another application of search-based test generation techniques is in **automated crash reproduction**. Information about a software crash, like a *stack trace* for Java applications, are usually reported to the developers through an issue tracker. Based on the report's information, the developers debug the software by identifying the crash's root cause and applying a fix to the code. To ease their investigation, developers can start their debugging process by *reproducing* and *exposing* the crash, and (latter) write a test case to ensure that the fix does not induce regression errors [57]. Recent developments lead to the (partial) automation of the crash reproduction and exposure process. When a new issue is created, an automated process fetches the stack trace and try to generate a crash reproducing test case able to reproduce and expose the crash [58]. Various approaches have been developed to automate the generation of a crash reproducing test case [28, 30, 31, 33]. Among those, search-based crash reproduction yields the best results by reproducing more crashes and generating helpful test cases [28]. It has also been confirmed that crash reproducing test cases generated by this approach aid developers in fixing bugs [28].

Search-based crash reproduction takes as input the application in which the crash happened, and a stack trace (reported in a crash report) with one of its frames indicated as the *target frame*. Then, it initiates a search process to generate a test case, which reproduced the given stack trace from the deepest frame up to the target frame. For instance, by passing the stack trace in Listing 1.2 as the given tack trace and frame 2 as the target frame, search-based crash reproduction generates a test case which reproduces the first two frames of the given stack trace with the same type of exception (`ClassCastException`).

Fitness function

To reproduce a given crash, search-based crash reproduction relies on a fitness function called *Crash Distance* (described in Equation 1.1) to evaluate the generated test cases, thereby guiding an evolutionary algorithm towards generating a crash reproducing test case for a given stack trace.

$$f(t) = \begin{cases} 3 \times d_s(t) + 2 \times max(d_e) + max(d_t) & \textit{line is not reached} \\ 2 \times d_e(t) + max(d_t) & \textit{line is reached} \\ d_t(t) & \textit{exception is thrown} \end{cases} \quad (1.1)$$

[2] This section is partly based on B. Cherry, X. Devroey, P. Derakhshanfar, and B. Vanderose. Crash reproduction difficulty, an initial assessment, BENEVOL'20 [1]

Where $d_s(t) \in [0,1]$ measures the distance between the execution of a generated test t from reaching the line of the target frame (*target line*) using the approach level and branch distance [4]; $d_e(t) \in \{0,1\}$ is a binary value indicating if t throws the same type of exception as the given stack trace ($d_e(t) = 0$) or not ($d_e(t) = 1$); $d_t(t) \in [0,1]$ compares the similarity of the frames in the given thrown stack trace by test t against the frames in the given stack trace; and $max(.)$ indicates the maximum possible value for each heuristic.

Since considering $d_e(t)$ and $d_t(t)$ is only relevant if test t covers the target line ($d_s(t) = 0$), *Crash Distance* (first line of Equation 1.1) sets the maximum value for these two heuristics before achieving the target line coverage. Therefore, $f(t) \in]3,6]$ before reaching the target line. Likewise, as shown by the second line of Equation 1.1, measuring the stack trace similarity ($d_t(t)$) is not relevant before fulfilling the exception coverage ($d_e(t)$), and thereby *Crash Distance* sets the maximum possible value for $d_t(t)$. Hence, $f(t) \in]1,3]$ before t throws the same type of exception as the given stack trace. Finally, when $d_s(t)$ and $d_e(t)$ are zero, $f(t) \in [0,1]$ according to the value of $d_t(t)$. Since the process is a minimization process, the three heuristics are equal to zero for a crash reproducing test case.

In this thesis, we have implemented a new open-source search-based crash reproduction framework called Botsing. This framework contains the previously introduced search-based crash reproduction approach (*e.g.,* EvoCrash). Also, the novel techniques introduced in this thesis are all implemented in this framework.

1.2 Challenges In Search-based Crash Reproduction And Test Generation

Since the search-based crash reproduction approaches are evaluated by a limited number of crashes, the limits and challenges of these techniques remained largely unrecognized. In this thesis, we first identify search-based crash reproduction challenges. Hence, we empirically evaluate search-based crash reproduction by a new Java crash benchmark called JCrashPack and identify the challenges by performing an extensive manual analysis. Some of the identified challenges are dedicated only to the crash reproduction problem, but some other challenges are general search-based test generation issues.

After identifying the challenges, we intend to address them by introducing novel solutions to improve the effectiveness and efficiency of the crash reproduction search process. For this goal, we investigate the application of contextual information collected from different sources such as source code, hand-written test cases. While some identified challenges in search-based crash reproduction are due to the existing general search-based test generation limitations, we go beyond crash reproduction and introduce new search-based techniques to cover other specific software behaviors.

1.3 Research Goals & Questions

This thesis seeks to understand the challenges in search-based crash reproduction and test case generation and utilizes the existing information in different sources such as source code and hand-written test cases to address the identified challenges. Hence, to present indications towards this thesis, we seek to answer the following research questions:

Since our initial goal is improving the effectiveness and efficiency of the search-based crash reproduction, first, we need to understand its challenges. Hence, the first research question tries to address this goal.

RQ$_1$: *What are the challenges in search-based crash reproduction?*

After identifying the challenges, we study novel ways to tackle them by enhancing the search process from different aspects. This enhancement is done by utilizing the contextual information collected from source code and existing tests. The second research question investigates the new techniques addressing the detected challenges using the observed contextual information.

RQ$_2$: *Based on the identified challenges, how can we leverage the existing knowledge, carved from information sources, to steer the crash reproduction search process?*

Since some of the detected challenges in search-based crash reproduction are observable in other search-based test generation techniques, the observed contextual information can guide the search process to generate tests for other criteria, as well. Hence, the last question concentrates on novel search-based techniques for testing in two levels of unit testing and class integration testing.

RQ$_3$: *How can we leverage the existing knowledge, carved from information sources, to design search-based test generation approaches for unit and class integration testing?*

This thesis answers RQ_3 by (i) introducing a whole new search-algorithm for class integration testing using the collected information about the method calls from one class (*caller class*) to the other one (*callee class*), and (ii) introducing a new search objective for search-based unit testing considering the common and uncommon execution paths in the class under test.

After answering these research questions, we will be able to understand the challenges in search-based software test generation better. Besides, we can confirm that (i) automated test generation for specific software behaviors can cover, and reveal, faults that are not detectable by other search-based test generation techniques, using only the classical structural coverage search objectives; and (ii) using other contextual information collected from various sources (such as source code, existing test cases, and execution logs) guides the search process to achieve higher fault detection.

1.4 Research Outline

This section briefly presents the various chapters in this thesis. Table 1.1 outlines the connections between each defined research question and chapters in this thesis.

Chapter 2: Crash reproduction approaches help developers during debugging by generating a test case that reproduces a given crash. Several solutions have been proposed

Table 1.1: Connection of chapters with research questions

Research Question	Chapters
RQ_1: What are the challenges in search-based crash reproduction?	2
RQ_2: Based on the identified challenges, how can we leverage the existing knowledge, carved from information sources, to steer the crash reproduction search process?	3 to 5
RQ_3: How can we leverage the existing knowledge, carved from information sources, to design search-based test generation approaches for unit and class integration testing?	6 & 7

to automate this task. However, the proposed solutions have been evaluated on a limited number of projects, making comparison difficult. In this chapter, we enhance this line of research by proposing JCrashPack, an extensible benchmark for Java crash reproduction, together with ExRunner, a tool to simply and systematically run evaluations. JCrashPack contains 200 stack traces from various Java projects, including industrial open source ones, on which we run an extensive evaluation of EvoCrash, the state-of-the-art approach for search-based crash reproduction. Our results include a detailed manual analysis of EvoCrash outputs, from which we derive 14 current challenges for crash reproduction. Finally, based on those challenges, we discuss future research directions for search-based crash reproduction for Java.

Chapter 3: According to the results of Chapter 2, one of the fundamental challenges of search-based crash reproduction is creating objects needed to trigger the crash. One way to overcome this limitation is seeding: using information about the application during the search process. With seeding, the existing usages of classes can be used in the search process to produce realistic sequences of method calls which create the required objects. In this chapter, we introduce behavioral model seeding: a new seeding method which learns class usages from both the system under test and existing test cases. Learned usages are then synthesized in a behavioral model (state machine). Then, this model serves to guide the evolutionary process. To assess behavioral model-seeding, we evaluate it against test-seeding (the state-of-the-art technique for seeding realistic objects) and no-seeding (without seeding any class usage). Our results indicate that behavioral model-seeding outperforms both test seeding and no-seeding by a minimum of 6% without any notable negative impact on efficiency.

Chapter 4: The state-of-the-art search-based crash reproduction approaches use a single fitness function called *Crash Distance* to guide the search process toward reproducing a target crash. Despite the reported achievements, these approaches do not always successfully reproduce some crashes due to a lack of test diversity (premature convergence). In this study, we introduce a new approach, called *MO-HO*, that addresses this issue via multi-objectivization. In particular, we introduce two new Helper-Objectives for crash reproduction, namely *test length* (to minimize) and *method sequence diversity* (to maximize), in addition to *Crash Distance*. We assessed *MO-HO* using five multi-objective evolutionary algorithms (NSGA-II, SPEA2, PESA-II, MOEA/D, FEMO) on crashes selected from JCrash-Pack. Our results indicate that SPEA2 is the best-performing multi-objective algorithm for *MO-HO*. We evaluated this best-performing algorithm for *MO-HO* against the state-of-the-art: single-objective approach (Single-Objective Search) and decomposition-based multi-objectivization approach (*De-MO*). Our results show that *MO-HO* reproduces five crashes that cannot be reproduced by the current state-of-the-art. Besides, *MO-HO* improves the effectiveness (+10% and +8% in reproduction ratio) and the efficiency in 34.6% and 36% of crashes (i.e., significantly lower running time) compared to Single-Objective Search and *De-MO*, respectively. For some crashes, the improvements are very large, being up to +93.3% for reproduction ratio and -92% for the required running time.

Chapter 5: Search-based crash reproduction approaches rely on the approach level and branch distance heuristics to guide the search process and generate test cases covering the lines, which appeared in the given stack trace. Despite the positive results achieved by these two heuristics, they only use the information related to the coverage of explicit branches (*e.g.,* indicated by conditional and loop statements), but ignore potential implicit branchings within basic blocks of code. If such implicit branching happens at runtime (*e.g.,* if an exception is thrown in a branchless-method), the existing fitness functions cannot guide the search process. To address this issue, we introduce a new secondary objective, called Basic Block Coverage (*BBC*), which takes into account the coverage level of relevant basic blocks in the control flow graph. We evaluated the impact of *BBC* on search-based crash reproduction because the implicit branches commonly occur when trying to reproduce a crash, and the search process needs to cover only a few basic blocks (*i.e.,* blocks that are executed before crash happening). We combined *BBC* with existing fitness functions (namely *STDistance* and *Crash Distance*) and ran our evaluation on JCrash-Pack crashes. Our results show that *BBC*, in combination with *STDistance* and *Crash Distance*, reproduces 6 and 1 new crashes, respectively. *BBC* significantly decreases the time required to reproduce 26.6% and 13.7% of the crashes using *STDistance* and *Crash Distance*, respectively. For these crashes, *BBC* reduces the consumed time by 44.3% (for *STDistance*) and 40.6% (for *Crash Distance*) on average.

Chapter 6: Search-based approaches have been used in the literature to automate the process of creating unit test cases. However, related work has shown that generated unit-tests with high code coverage could be ineffective, i.e., they may not detect all faults or kill all injected mutants. In this chapter, we propose Cling, an integration-level test case generation approach that exploits how a pair of classes, the caller and the callee, interact with each other through method calls. In particular, Cling generates integration-level test cases that maximize the Coupled Branches Criterion (CBC). CBC is a novel integration-level coverage criterion, measuring the degree to which a test suite exercises the interactions between a caller and its callee classes. We implemented Cling and evaluated the approach on 140 pairs of classes from five different open-source Java projects. Our results show that (1) Cling generates test suites with high CBC coverage; (2) such generated suites can kill on average 7.7% (with a maximum of 50%) of mutants that are not detected by tests generated at the unit level; (3) Cling can detect integration faults (32 for our subject systems) that remain undetected when using automatically generated unit-level test suites.

Chapter 7: Various search-based test generation techniques have been proposed to automate the generation of unit tests fulfilling different criteria (*e.g.,* line coverage, branch coverage, mutation score, *etc.*). Despite several advances made over the years, search-based unit test generation still suffers from a lack of guidance due to the limited amount of information available in the source code that, for instance, hampers the generation of complex objects. Previous studies introduced many strategies to address this issue, *e.g.,* dynamic symbolic execution or seeding, but do not take the internal execution of the methods into account. This chapter introduces a novel secondary objective called commonality score, measuring how close the execution path of a test case is from reproducing a common or uncommon execution pattern observed during the operation of the software. To assess the commonality score, we implemented it in EvoSuite and evaluated its application on 150 classes from JabRef, open-source software for managing bibliographic references. Our results are mixed. Our approach leads to test cases that indeed follow common or uncommon execution patterns. However, if the commonality score can have a positive impact on the structural coverage and mutation score of the generated test suites, it can also be detrimental in some cases.

Chapter 8: Finally, we summarize our findings and conclusions in this thesis. This chapter also elaborates on the potential future work that can, first, improve the search-based crash reproduction, and second, investigate novel search-based algorithms for covering other software specific behaviors that can be interesting for developers.

1.5 Research Methodology

This thesis answers the aforementioned research questions by following an approach based on design science [60]. The design science paradigm contains two iterative phases: **Build** and **Evaluation**. The former phase concentrates on developing a purposeful artifact to solve an unsolved problem (here, search-based test generation techniques for specific behaviors such as crash reproduction and class integrations). The latter phase evaluates the designed artifact. The *Evaluation* phase reveals the limitations and challenges in the built artifact, and thereby, the weaknesses can be identified and resolved in the next phase of the *Build* process. This iteration usually continues multiple times, and in each iteration, one (or more) novel techniques are introduced (to improve the existing artifact) and be assessed by the *Evaluation* process. Also, both *Build* and *Evaluation* evolve in this process according to the new findings.

In this thesis, we define a framework for search-based test case generation for crash reproduction. This framework includes a benchmark for crash reproduction approaches containing real-world and non-trivial crashes, an extensible platform for search-based crash reproducing test case generation, and accompanying guidelines for efficient usage of the platform in an industrial setting. This framework applies the existing search-based crash reproduction approach to real-world crashes and identifies the challenges (to answer RQ_1). We then improve the crash reproduction approach by addressing the identified challenges (to answer RQ_2). Finally, we go beyond search-based crash reproduction and introduce a novel approach for generating tests to cover other specific behaviors (*e.g.,* class integration testing). Accordingly, we extend the evaluation process to assess the new artifacts, as well (to address RQ_3).

1.6 Origins Of The Chapters

All chapters of this thesis (except Chapter 7, which is currently under review) have been published in peer-reviewed journals and conferences. Hence, all chapters contain a dedicated background, related work, and conclusion section. This section lists the origin of each chapter.

For all chapters, except Chapter 7, the author of this thesis was the lead author of the paper, responsible for the design of the algorithms and experiments, tool implementation, carrying out experiments, analysis of the results, and the writing of the paper. For chapter 2, this role was shared with Mozhan Soltani.

- *Chapter 2* was published in the paper ”A benchmark-based evaluation of search-based crash reproduction” in Empirical Software Engineering (EMSE) 2020.

- *Chapter 3* was published in the paper ”Search-based crash reproduction using behavioural model seeding” in theJournal of Software: Testing, Validation, and Reliability (STVR) 2020.

- *Chapter 4* was published in the paper ” Good Things Come In Threes: Improving Search-based Crash Reproduction With Helper Objectives” at the International Conference on Automated Software Engineering (ASE) 2020.

- *Chapter 5* was published in the paper ”It is not Only About Control Dependent Nodes: Basic Block Coverage for Search-Based Crash Reproduction” at the Symposium on Search-Based Software Engineering (SSBSE) 2020.

- *Chapter 6* was published as a paper titled ”Generating Class-Level Integration Tests Using Call Site Information”, which is currently under revision in Transactions on Software Engineering journal (TSE).

- *Chapter 7* was published in the paper ”Commonality-Driven Unit Test Generation” at Symposium on Search-Based Software Engineering (SSBSE) 2020.

1.7 Open Science

Open science is the “movement to make scientific research, data and dissemination accessible to all levels of an inquiring society” [61]. All of the implementations used in our studies are available via GitHub. Also, replication packages of all of our studies, presented in this thesis, are openly available in Zenodo. These replication packages contain the list of subject systems used in each study, a Docker-based infrastructure to rerun all of the experiments, and all test cases generated by each of the search-based approaches.

Table 1.2 shows the replication packages of each chapter in this thesis.

Table 1.2: Connection of chapters with replication packages

Chapter	Replication package	Zenodo DOI
2	[62]	10.5281/zenodo.3766689
3	[63]	10.5281/zenodo.3673916
4	[64]	10.5281/zenodo.3979097
5	[65]	10.5281/zenodo.3953519
6	[66]	10.5281/zenodo.4300634
7	[67]	10.5281/zenodo.3894711

1.7.1 Open-source Search-based Test Case Generation Implementations

Search-based crash reproduction (Botsing)

In this thesis, we present Botsing[3]: an open-source, extendable search-based crash reproduction framework. Botsing implements search-based crash reproduction approaches introduced in previous studies [28, 32, 68]. The tool takes as input a stack trace and software under test. Then, it starts a single-objective or multi-objective search process to generate a test reproducing the crash.

Botsing has been designed as an extendable framework for implementing new features and search algorithms for crash reproduction. For example, in Chapter 3 we perform a study on the impact of various seeding strategies on crash reproduction, for which we have implemented multiple seeding strategies in Botsing.

From an industrial perspective, Botsing is used by our partners in the STAMP project.[4] They confirmed the relevance of Botsing for debugging and fixing application crashes [69]. The feedback —as well as the crash reproducing test cases— from our partners using Botsing is openly available in the STAMP GitHub repository.[5]

Search-based test generation for class integration (Cling)

Chapter 6 addresses RQ_3 by introducing a novel search-based technique to test the integration between two classes looking at their call-sites information. We have implemented this approach as an open-source tool called Cling[6]. This tool gets application's bytecode and two classes in the application (the caller class and callee class) and produces a test suite that covers the various interactions between these two classes.

Common/uncommon execution patterns test generation in unit testing

In Chapter 7, we implement novel secondary objectives considering the common/uncommon execution patterns in EvoSuite.

1.7.2 Open-source Evaluation Infrastructures

Assessing crash reproduction

To evaluate the different search-based crash reproduction techniques, we created JCrashPack, an open-source crash benchmark, which contains 200 non trivial Java crashes col-

[3]`https://github.com/STAMP-project/botsing`

[4]Available at `http://stamp-project.eu/`

[5]Available at `https://github.com/STAMP-project/botsing-usecases-output`.

[6]`https://github.com/STAMP-project/botsing/tree/master/cling`

lected from seven open-source projects: *Closure compiler*, *Apache commons-lang*, *Apache commons-math*, *Mockito*, *Joda-Time*, *XWiki*, and *ElasticSearch*. Moreover, to ease benchmarking using JCrashPack, we developed a bash-based execution runner, openly available on GitHub.[7] This experiment runner (called ExRunner) runs different instances of a crash reproduction tool (here, Botsing) in parallel processes and collects relevant information about the execution in a CSV file. These collected data helps to identify the search-based crash reproduction benchmark.

Assessing class integration

To assess Cling against the state-of-the-art, we used subjects from five Java projects, namely *Closure compiler*, *Apache commons-lang*, *Apache commons-math*, *Mockito*, and *Joda-Time*. These projects have been used in prior studies to assess the coverage and the effectiveness of unit-level test case generation [22, 42, 70, 71], program repair [72, 73], fault localization [74, 75], and regression testing [76, 77].

Moreover, we have implemented another open-source runner [8] (similar to ExRunner). This runner collects more information about the test suites generated by different approaches: *branch coverage* and mutation score measured by PIT[9], which is a state-of-the-art mutation testing tool for Java code, to mutate the callee classes.

Assessing common/uncommon execution patterns test generation

To assess our common/uncommon execution patterns search objective, we choose JabRef (46 KLOC), an open-source Java bibliography reference manager with a graphical user interface working with BibTex files. We instrumented JabRef using Spoon [78] to monitor the execution paths while users are using it. We sampled 150 classes from this project for our evaluation.

Since we want to measure the strong mutation score of test suites generated by EvoSuite + our novel secondary objectives against regular EvoSuite, we use the same open-source infrastructure as the one we used to assess Cling.

[7]`https://github.com/STAMP-project/ExRunner-bash`

[8]`https://github.com/STAMP-project/Cling-application`

[9]http://pitest.org

2
A Benchmark-Based Evaluation of Search-BasedCrash Reproduction

EvoCrash is the state-of-the-art in automated crash reproduction. It has been previously evaluated on 54 crashes[36], and its relevance for debugging has been confirmed [28]. Also. it has been shown that EvoCrash outperforms other approaches based on backward symbolic execution [30], test case mutation [33], and model-checking [31], evaluated on smaller benchmarks [28].

However, all those crashes benchmarks were not selected to reflect challenges that are likely to occur in real-life stack traces, raising threats to external validity. Thus the questions of whether the selected applications and crashes were sufficiently representative, if EvoCrash will work in other contexts, and what limitations are still there to address remained unanswered.

The goal of this chapter is to identify challenges in search-based crash reproduction by performing an empirical evaluation. To that end, we devise a new benchmark of real-world crashes, called JCrashPack. It contains 200 crashes from seven actively maintained open-source and industrial projects. These projects vary in their domain application and include an enterprise wiki application, a distributed RESTful search engine, several popular APIs, and a mocking framework for unit testing Java programs. JCrashPack is extensible, and can be used for large-scale evaluation and comparison of automated crash reproduction techniques for Java programs.

To illustrate the use of JCrashPack, we adopt it to extend the reported evaluation on EvoCrash [36] and identify the areas where the approach can be improved. In this chapter, we provide an account of the cases that were successfully reproduced by EvoCrash (87 crashes out of 200). We also analyze all failed reproductions and distill 14 categories of research and engineering limitations that negatively affected reproducing crashes in our study. Some of those limitations are in line with challenges commonly reported for search-based structural software testing in the community [21, 34, 79] and others are specific to search-based crash reproduction.

This chapter is published as a paper with two first authors. The author of this thesis and the other first author contributed equally to this work.

Our categorization of challenges indicates that environmental dependencies, code complexity, and limitations of automated input data generation often hinder successful crash reproduction. In addition, stack frames (i.e., lines in a stack trace), pointing to varying types of program elements, such as interfaces, abstract classes, and anonymous objects, influence the extent to which a stack trace-based approach to crash reproduction is effective.

Finally, we observe that the percentage of successfully reproduced crashes drops from 85% (46 crashes out of 54 reported by Soltani *et al.* [28]) to 43% (87 out of 200) when evaluating crashes that are from industrial projects. In our observations, generating input data for microservices, and unit testing for classes with environmental dependencies, which may frequently exist in enterprise applications, are among the major reasons for the observed drop in the reproduction rate. These results are consistent with the paradigm shift to context-based software engineering research that has been proposed by Briand et al. [80].

The key contributions of this chapter are:

- JCrashPack,[1] a carefully composed benchmark of 200 crashes, as well as their correct system version and its libraries, from seven real-world Java projects, together with an account of our manual analysis on the characteristics of the selected crashes and their constituting frames, including size of the stack traces, complexity measures, and identification of buggy and fixed versions.

- ExRunner,[2] a bash script for automatically running experiments with crash reproduction tools in Java.

- Empirical evidence [62], demonstrating the effectiveness of search-based crash reproduction on real world crashes taken from JCrashPack.

- The identification of 14 categories of research and engineering challenges for search-based crash reproduction that need to be addressed in order to facilitate uptake in practice of crash reproduction research.

The remainder of the chapter is structured as follows: Sections 2.2 to 2.4 describe the design protocol for the benchmark, the resulting benchmark JCrashPack, as well as the ExRunner tool to run experiments on JCrashPack. Sections 2.5 to 2.7 cover the experimental setup for the EvoCrash evaluation, the results from our evaluation, and the results challenges that we identified through our evaluation. Sections 2.8 to 2.11 provide a discussion of our results and future research directions, an analysis of the threats to validity, and a summary of our overall conclusions.

2.1 Background and related work

2.1.1 Automated Crash Reproduction

Software crashes commonly occur in operating environments and are reported to developers for inspection. When debugging, reproducing a reported crash is among the tasks a

[1]Available at `https://github.com/STAMP-project/JCrashPack`.

[2]Available at `https://github.com/STAMP-project/ExRunner-bash`

developer needs to do in order to identify the conditions under which the reported crash is triggered [57, 81]. In particular, for Java programs, when a crash occurs, an exception is thrown. A developer strives to reproduce it to understand its cause, then fix the bug, and finally add a (non-)regression test to avoid reintroducing the bug in future versions.

Manual crash reproduction can be a challenging and labor-intensive task for developers: it is often an iterative process that requires setting the debugging environment in a similar enough state as the environment in which the crash occurred [57].

To help developers in this process, various automated techniques have been suggested. These techniques can be divided into three categories, based on the kind of data used for crash reproduction: record-replay approaches [82–88] record data from the running program; post-failure approaches [32, 89–94] collect data from the crash, like a memory dump; and stack-trace based post-failure [29–31, 33, 36] use only the stack trace produced by the crash. We briefly describe each category hereafter.

Record-replay approaches. These approaches record the program runtime data and use them during crash reproduction. The main limitation is the availability of the required data. Monitoring software execution may violate privacy by collecting sensitive data, the monitoring process can be an expensive task for the large scale software, and may induce a significant overhead [30–32]. Tools like ReCrash [82], ADDA [83], Bugnet [84], jRapture [85], MoTiF [86], Chronicler [87], and SymCrash [88] fall in this category.

Post-failure approaches. Techniques from this category use the software data collected directly after the occurrence of a failure. For instance, RECORE [32] applies a search-based approach to reproduce a crash by using the stack trace and a core dump, produced by the system when the crash happened, to guide the search.

Although these tools limit the quantity of monitored and recorded data, the availability of such data still represents a challenge. Yu *et al.* [89] addressed this issue for system-level concurrency failure reproduction by introducing DESCRY. This approach only uses the default execution logs and applies both static and dynamic analysis combined with symbolic execution to generate the input data and interleaving schedule. However, even this approach suffers from two limitations: (i) since this tool relies on symbolic execution, applying it on the large and complex projects leads to path explosion; (ii) the performance of this tool is strongly linked to the quality of the software log. Other *post-failure approaches* include: Weeratunge *et al.* [90], Leitner *et al.* [91, 92], and Kifetew *et al.* [93, 94].

Stack-trace based post-failure. Recent studies in crash reproduction [29–31, 33, 36] focuses on utilizing data only from a given crash stack trace to enhance the practical application. Table 2.1 illustrates an example of a crash stack trace from Apache Ant[3] [95] which is comprised of a crash type (`java.lang.NullPointerException`) and a stack of frames pointing to all method calls that were involved in the execution when the crash happened. From a crash stack frame, we can retrieve information about: the crashing method, the line number in the method where the crash happened, and the fully qualifying name of the class where the crashing method is declared.

[3]ANT-49755: `https://bz.apache.org/bugzilla/show_bug.cgi?id=49755`

Table 2.1: The crash stack trace for Apache Ant-49755.

java.lang.**NullPointerException**:

Level	Frame
1	**at** org.apache.tools.ant.util.FileUtils.**createTempFile**(FileUtils.java:**888**)
2	**at** org.apache.tools.ant.taskdefs.TempFile.**execute**(TempFile.java:**158**)
3	**at** org.apache.tools.ant.UnknownElement.**execute**(UnknownElement.java:**291**)

The state of the research in crash reproduction [29–31, 33, 36, 96, 97] aims at generating test code that, once executed, produces a stack trace that is as similar to the original one as possible. They, however, differ in their means to achieve this task.

ESD [96], a debugger based on execution synthesis, uses forward symbolic execution and static analysis to reach reproduction. This tool focuses more on concurrency and memory safety bugs. Similarly, BugRedux [97] uses forward symbolic execution. BugRedux is a crash reproduction tool for C programs.

Since these two tools rely on forward symbolic execution, they can be applied only on medium-size applications. Also, as illustrated by Braione *et al.* [7], symbolic execution test generation approaches face limitations when generating complex input data structures. To address these limitations, STAR [30] applies optimized backward symbolic execution and uses a novel technique for method sequence composition to generate a unit test that satisfies the computed preconditions, and eventually reproduces the target crash. However, as reported by Chen *et al.* [30], STAR still suffers from the path explosion problem stemming from utilizing symbolic execution. It only supports 3 types of exceptions: explicitly thrown exceptions, `NullPointerException`, and `ArrayIndexOutOfBoundsException`.

JCHARMING [31] applies model checking to reproduce the reported bugs. To prevent state explosion in the model, it utilizes program slicing. Since JCHARMING can be applied to any frame from a given crash stack trace, the approach can reproduce any fraction of the target crash stack trace.

MuCrash [33] is based on exploiting existing test cases written by developers and mutating them until they trigger the target crash. Test case mutation in MuCrash is directed by selecting tests for the classes included in the target crash stack trace.

Finally, Concrash [29] focuses on reproducing *concurrency* failures that violate thread-safety of a class. Concrash iteratively generates test code and looks for a thread interleaving that triggers a concurrency crash. In order to steer the test generation process and avoid expensive computations, Concrash applies the pruning strategies to avoid redundant and irrelevant test code. In contrast to other crash reproduction techniques, Concrash only reproduces the minority of the crashes in the issue tracking systems. As reported by Yuan *et al.* [98], inter-leaving crashes cause only 10% of failures in the distributed data-intensive systems. Besides, Coelho *et al.* [99] state that this number is even lower in Android applications (2.9%).

2.1.2 Search-based Crash Reproduction With EvoCrash

Search-based algorithms have been increasingly used for software engineering problems since they are shown to suite complex, non-linear problems, with multiple optimization objectives that may conflict or competing [100]. EvoCrash [28, 36] is a search-based approach to crash reproduction, which applies a *guided genetic algorithm* to search for a unit test that reproduces the target crash. They have shown that this search-based technique outperforms other automated crash reproduction approaches.

EvoCrash takes as input a stack trace with one of its frames set as the target frame. The target frame is composed of a (i) target class, the class to which the exception has been propagated; a (ii) `target method`, the method in that class; and a (iii) target line, the line in that method where the exception has been propagated. It then seeks to generate a unit test that replicates the given stack trace from the target frame (at level n) to the deepest frame (at level 1). For instance, if we pass the stack trace in Table 2.1 as the given trace and indicate the second frame as the target frame (level 2), the output of EvoCrash will be a unit test for the class `TempFile` which replicates the first two frames of the given stack trace with the same type of the exception (`NullPointerException`).

Guided genetic algorithm

The search process in EvoCrash begins by randomly generating unit tests for the target frame. In this phase, called guided initialization, the target method corresponding to the selected frame (i.e., the failing method to which the exception is propagated) is injected in every randomly generated unit test. During subsequent phases of the search, guided crossover and guided mutation, standard evolutionary operations are applied to the unit tests. However, applying these operations involves the risk of losing the injected failing method. Therefore, the algorithm ensures that only unit tests with the injected failing method call remain in the evolution loop. If the generated test by crossover does not contain the failing method, the algorithm replaces it with one of its parents. Also, if the resulting test does not contain the failing method after a mutation, the algorithm redoes the mutation until the failing method is added to the test again. The search process continues until either the search budget is over or a crash reproducing test case is found.

To evaluate the generated tests, EvoCrash applies the following weighted sum fitness function [28] (called *Crash Distance*, hereafter) to a generated test t:

$$f(t) = \begin{cases} 3 \times d_s(t) + 2 \times max(d_{except}) + max(d_{trace}) & \textit{if the line is not reached} \\ 3 \times min(d_s) + 2 \times d_{except}(t) + max(d_{trace}) & \textit{if the line is reached} \\ 3 \times min(d_s) + 2 \times min(d_{except}) + d_{trace}(t) & \textit{if the exception is thrown} \end{cases} \tag{2.1}$$

Where:

- $d_s \in [0, 1]$ indicates the distance between the execution of t and the target statement s located at the target line. This distance is computed using the *approach level*, measuring the minimum number of control dependencies between the path of the code executed by t and s, and normalized *branch distance*, scoring how close t is to satisfying the branch condition for the branch on which$s$ is directly control dependent [4]. If the target line is reached by the test case, $d_l(t)$ equals to 0.0;
- $d_{except}(t) \in \{0, 1\}$ indicates if the target exception is thrown ($d_e = 0$) or not ($d_e = 1$);
- $d_{trace}(t) \in [0, 1]$ indicates the similarity of the input stack trace and the one generated by t by looking at class names, methods names and line numbers;
- $max(\cdot)$ denotes the maximum possible value for the function.

Since the stack trace similarity is relevant only if the expected exception is thrown by t, and the check whether the expected exception is thrown or not is relevant only if the target line where the exception propagates is reached, d_{except} and d_{trace}) are computed only upon the satisfaction of two constraints: the target exception has to be thrown in the target line s and the stack trace similarity should be computed only if the target exception is actually thrown.

Unlike other stack trace similarity measures (e.g., [32]), Soltani *et al.* [28] do not require two stack traces to share the same common prefix to avoid rejecting stack traces where the difference is only in one intermediate frame. Instead, for each frame, $d_{trace}(t)$ looks at the closest frame and computes a distance value. Formally, for an original stack trace S^* and a test case t producing a stack trace S, $d_{trace}(t)$ is defined as follows:

$$d_{trace}(t) = \varphi\left(\sum_{f^* \in S^*} min\{diff(f^*, f) : f \in S\}\right) \tag{2.2}$$

Where $\varphi(x) = x/(x+1)$ is a normalization function [4] and *diff*(f^*, f) measures the difference between two frames as follows:

$$diff(f^{**}, f) = \begin{cases} 3 & \textit{if the classes are different} \\ 2 & \textit{if the classes are equal but the methods are different} \\ \varphi(|l^* - l|) & \textit{otherwise} \end{cases} \tag{2.3}$$

Where l (resp. l^*) is the line number of the frame f (resp. f^*).

Each of the three components if the fitness function defined in Equation 2.1 ranges from 0.0 to 1.0, the overall fitness value for a given test case ranges from 0.0 (crash is fully reproduced) to 6.0 (no test was generated), depending on the conditions it satisfies.

Comparison with the state-of-the-art

Crash reproduction tools. Table 2.2 presents the number of crashes used in the benchmarks used to evaluated stack-trace based post-failure crash reproduction tools as well as their crash reproduction rates. EvoCrash has been evaluated on various crashes reported in other studies and has the highest reproduction rate.

EvoSuite. Table 2.2 also reports the comparison of EvoCrash with EvoSuite, using exception coverage as the primary objective, applied by Soltani *et al.* [28]. All the crashes reproduced by EvoSuite could also be reproduced by EvoCrash on average 170% faster and with a higher reproduction rate.

Usefulness for debugging

When reproducing a stack trace with EvoCrash, there is no guarantee that the generated test completely reproduces the conditions in which the crash happened in the first place. Besides the random nature of search-based approaches, test cases are generated at the unit level, while crashes usually happen at the system level. However, rather than reproducing the exact same conditions of the crash, the goal of crash reproduction is to help developers fix the underlying bug.

Table 2.2: The number of crashes used in each crash reproduction tool experiment, the gained reproduction by them, and the involved projects.

Tool	Reproduced/Total	Rate	Projects
EvoCrash [28, 36]	46/54	85%	Apache Commons Collections Apache Ant Apache Log4j ActiveMQ DnsJava JFreeChart
EvoSuite [28]	18/54	33%	Apache Commons Collections Apache Ant Apache Log4j ActiveMQ DnsJava JFreeChart
STAR [30]	30/51	59%	Apache Commons Collections Apache Ant Apache Log4j
MuCrash [33]	8/12	66%	Apache Commons Collections
JCharming[31]	8/12	66%	Apache Ant Apache Log4j ActiveMQ DnsJava JFreeChart

Chen *et al.* [30] introduced a usefulness criterion for the crash reproduction approaches. According to this criterion, a crash reproducing test is useful to the developers if it covers the buggy frame, *i.e.,* if the target frame for which the reproduction is successful is higher than the frame that points to the buggy method. Soltani *et al.* [28] refined that criterion through a controlled experiment with 35 master students in computer science and two crashes to assess the degree to which the tests generated by EvoCrash helps to debug code. Their results indicate that the reproducing tests generated by EvoCrash help the participants to fix the bugs more often, although not significantly, and significantly faster. They confirmed the usefulness criterion defined by the Chen *et al.* [30], but also found evidence that test cases categorized as not useful, according to this criterion, can still help developers fix the bug.

2.2 Benchmark Design

Benchmarking is a common practice to assess a new technique and compare it to the state of the art [101]. For instance, SF110 [21] is a sample of 100 Java projects from SourceForge, and 10 popular Java projects from GitHub, that may be used to assess (search based) test case selection techniques. In the same way, Defects4J [71] is a collection of bugs coming from popular open-source projects: for each bug, a buggy and a fixed version of the projects, as well as bug revealing test case, are provided. Defects4J is aimed to assess various testing techniques like test case selection or fault localization.

In their previous work, Soltani et al. [36], Xuan et al. [33], and Chen and Kim [30] used Apache Commons Collections [102], Apache Ant [95], and Apache Log4j [103] libraries. In addition to Apache Ant and Apache Log4j, Nayrolles et al. [31] used bug reports from 8 other open-source software.

In this chapter we enhance previous efforts to build a benchmark dedicated to crash reproduction by collecting cases coming from both state of the art literature and actively maintained industrial open-source projects with well documented bug trackers.

2.2.1 Projects Selection Protocol

As Table 2.2 clearly shows, current crash reproduction tools are not evaluated using a common benchmark. This hampers progress in the field as it makes it hard to compare approaches. To be able to perform analysis of the results of a crash reproduction attempt, we define the following benchmark requirements for our benchmark:

BR1, to be part of the benchmark, the projects should have openly accessible binaries, source code, and crash stack traces (in an issue tracker for instance);

BR2, they should be under active maintenance to be representative of current software engineering practices and ease communication with developers;

BR3, each stack trace should indicate the version of the project that generated the stack trace; and

BR4, the benchmark should include projects of varying size.

To best of our knowledge, there is no benchmark fulfilling those requirements. The closest benchmark is Defects4j. However, only 25% of the defects manifest through a crash stack trace (**BR1**) and the projects are relatively small (**BR4**). To address those limitations, we built a new benchmark dedicated to the crash reproduction tools.

To build our benchmark, we took the following approach. First, we investigated projects collected in SF110 [21] and Defects4J [71] as state of the art benchmarks. However, as most projects in SF110 have not been updated since 2010 or earlier, we discarded them from our analysis (**BR2**). From Defects4J, we collected 73 cases where bugs correspond to actual crashes: i.e., the execution of the test case highlighting the bug in a given buggy version of a project generates a stack trace that is not a test case assertion failure.

As also discussed by Fraser and Arcuri [21], to increase the representativeness of a benchmark, it is important to include projects that are popular and attractive to end-users. Additionally to Defects4J, we selected two industrial open-source projects: XWiki [104] and Elasticsearch [105]. XWiki is a popular enterprise wiki management system. Elasticsearch, a distributed RESTful search and analytic engine, is one of the ten most popular projects on GitHub[4]. To identify the top ten popular projects from Github, we took the following approach: (i) we queried the top ten projects that had the highest number of forks; (ii) we queried the top ten projects that had the highest number of stars; (iii) we queried the top ten trending projects; and (iv) took the intersection of the three.

Four projects were shared among the above top-ten projects, namely: Java-design-patterns [106], Dubbo[107], RxJava [108], and Elasticsearch. To narrow down the scope

[4]This selection was performed on 26/10/2017.

of the study, we selected Elasticsearch, which ranked the highest among the four shared projects.

2.2.2 Stack Trace Collection And Preprocessing

For each project, we collected stack traces to be reproduced as well as the project binaries, with specific versions on which the exceptions happened.

Defects4J. From the 395 buggy versions of the Defects4J projects, we kept only the bugs relevant to our crash reproduction context (73 cases), i.e., the bugs that manifest as crashes. We manually inspected the stack traces generated by the failing tests and collected those which are not JUnit assertion failures (i.e., those which are due to an exception thrown by the code under test and not by the JUnit framework). For instance, for one stack trace from the Joda-Time project:

```
0 java.lang.IllegalArgumentException:
1   at org.joda.time.Partial.<init>(Partial.java:224)
2   at org.joda.time.Partial.with(Partial.java:466)
3   at org.joda.time.TestPartial_Basics.testWith_baseAndArgHaveNoRange(...)
```

We only consider the first and second frames (lines 1 and 2). The third and following lines concern testing classes of the project, which are irrelevant for crash reproduction. They are removed from the benchmark, resulting in the following stack trace with two frames:

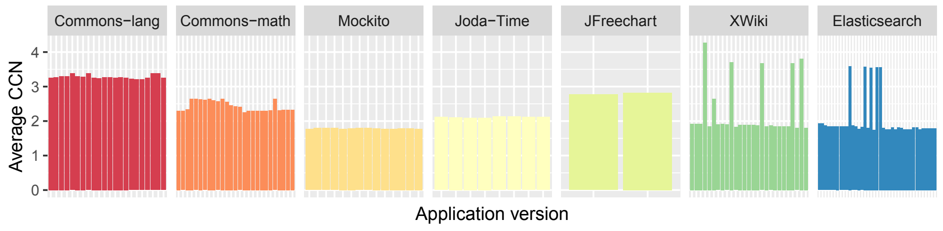
```
0 java.lang.IllegalArgumentException:
1   at org.joda.time.Partial.<init>(Partial.java:224)
2   at org.joda.time.Partial.with(Partial.java:466)
```

We proceeded in the same way for each Defects4J project and collected a total of 73 stack traces coming from five (out of the six) projects: JFreeChart, Commons-lang, Commons-math, Mockito, and Joda-Time. All the stack traces generated by the Closure compiler test cases are JUnit assertion failures.

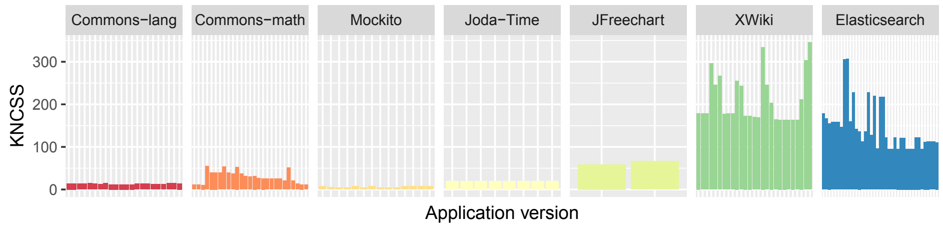
Elasticsearch. Crashes for Elasticsearch are publicly reported to the issue tracker of the project on GitHub[5]. Therefore, we queried the reported crashes, which were labelled as bugs, using the following string ”`exception is:issue label:bug`”. From the resulting issues (600 approx.), we manually collected the most recent ones (reported since 2016), which addressed the following: (i) the version which crashed was reported, (ii) the issue was discussed by the developers and approved as a valid crash to be fixed. The above manual process resulted in 76 crash stack traces.

XWiki. XWiki is an open source project which has a public issue tracker[6]. We investigated first 1000 issues which are reported for XWIK-7.2 (released in September 2015) to XWIK-9.6 (released in July 2017). We selected the issues where: (i) the stack trace of the crash was included in the reported issue, and (ii) the reported issue was approved by developers as a valid crash to be fixed. Eventually, we selected a total of 51 crashes for XWIKI.

[5]`https://github.com/elastic/elasticsearch/issues`
[6]`https://jira.xwiki.org/browse/XWIKI/`



(a) Average methods Cyclomatic Complexity Number (CCN)



(b) Thousands of Non-Commenting Sources Statements (KNCSS)

Figure 2.1: Complexity and size of the different projects

2.3 The JCrashPack Benchmark

The result of our selection protocol is a benchmark with 200 stack traces called JCrashPack . For each stack trace, based on the information from the issue tracker and the Defects4J data, we collected: the Java project in which the crash happened, the version of the project where the crash happened and (when available) the fixed version or the fixing commit reference of the project; the buggy frame (i.e., the frame in the stack trace targeting the method where the bug lays); and the Cyclomatic Complexity Number (CCN) and the Non-Commenting Sources Statements (NCSS) of the project, presented in Figure 2.1. Due to the manual effort involved in filtering, verifying and cleaning up stack traces, issues, the collection of stack traces and binaries (including the project's dependencies binaries) took about 4.5 person-months in total.

Figure 2.1 presents the average Cyclomatic Complexity Number (CCN) per method for each project and the Non-Commenting Sources Statements (NCSS) per project, ordered by version number, to give an idea of the complexity of a project. Also, Table 2.3 gives the number of versions and the average number of non-commenting source statement for each project in JCrashPack. As illustrated in the table and figure, JCrashPack contains projects of diverse complexities (the CCN for the least complex project is 1.77, and for the most complex is 3.38) and sizes (the largest project has 177,840 statements, and the smallest one holds 6,060 statements on average), distributed among different versions.

Table 2.4 shows the distribution of stack traces per exception type for the six most common ones, the *Other* category denoting remaining exception types. According to this table, the included stack traces in JCrashPack covers different types of the exceptions. Also, they are varied in the size (number of frames): the smallest stack traces have one frame and the largest, a user-defined exception in *Other*, has 175 frames.

Table 2.3: The number of versions and average number of statements ($\overline{NCSS}$) for each project.

Applications	Number of versions	$\overline{NCSS}$
Commons-lang	22	13.38k
Commons-math	27	29.98k
Mockito	14	6.06k
Joda-Time	8	19.41k
JFreechart	2	63.01k
XWiki	32	177.84k
Elasticsearch	46	124.36k
Total	151	62.01k

JCrashPack is extensible and publicly available on GitHub.[7] We provide guidelines to add new crashes to the benchmark and make a pull request to include them in JCrashPack master branch. The detailed numbers for each stack trace and its project are available on the JCrashPack website.

[7]At `https://github.com/STAMP-project/JCrashPack`

Table 2.4: Number of stack traces (*st*), total number of frames (*fr*), and average number of frames ($\overline{fr}$) and standard deviation (σ) per stack trace for the different exceptions: NullPointerException (NPE), IllegalArgumentException (IAE), ArrayIndexOutOfBoundsException (AIOOBE), ClassCastException (CCE), StringIndexOutOfBoundsException (SIOOBE), IllegalStateException (ISE), and other exceptions types (Other).

Applications		**NPE**	**IAE**	**AIOOBE**	**CCE**	**SIOOBE**	**ISE**	**Other**	**Total**
Commons-lang	*st*	5.0	3.0	2.0	0.0	6.0	0.0	6.0	22.0
	fr	8.0	3.0	12.0	0.0	10.0	0.0	12.0	45.0
	$\overline{fr}$	1.6	1.0	6.0		1.7		2.0	2.0
	σ	0.9	0.0	5.7		1.0		1.5	2.1
Commons-math	*st*	3.0	3.0	4.0	2.0	1.0	0.0	14.0	27.0
	fr	8.0	7.0	9.0	11.0	1.0	0.0	70.0	106.0
	$\overline{fr}$	2.7	2.3	2.2	5.5	1.0		5.0	3.9
	σ	0.6	1.5	2.5	6.4	NA		3.0	3.0
Mockito	*st*	2.0	0.0	2.0	2.0	0.0	0.0	8.0	14.0
	fr	3.0	0.0	12.0	2.0	0.0	0.0	48.0	65.0
	$\overline{fr}$	1.5		6.0	1.0			6.0	4.6
	σ	0.7		7.1	0.0			3.8	4.1
Joda-Time	*st*	0.0	3.0	0.0	0.0	0.0	0.0	5.0	8.0
	fr	0.0	5.0	0.0	0.0	0.0	0.0	26.0	31.0
	$\overline{fr}$		1.7					5.2	3.9
	σ		0.6					1.5	2.2
JFreechart	*st*	1.0	1.0	0.0	0.0	0.0	0.0	0.0	2.0
	fr	6.0	6.0	0.0	0.0	0.0	0.0	0.0	12.0
	$\overline{fr}$	6.0	6.0						6.0
	σ	NA	NA						0.0
XWiki	*st*	20.0	4.0	0.0	6.0	1.0	0.0	20.0	51.0
	fr	535.0	39.0	0.0	131.0	8.0	0.0	687.0	1400.0
	$\overline{fr}$	26.8	9.8		21.8	8.0		34.4	27.5
	σ	33.3	3.7		22.2	NA		47.0	37.0
Elasticsearch	*st*	18.0	10.0	6.0	0.0	1.0	7.0	34.0	76.0
	fr	222.0	152.0	102.0	0.0	15.0	135.0	717.0	1343.0
	$\overline{fr}$	12.3	15.2	17.0		15.0	19.3	21.1	17.7
	σ	9.8	9.2	18.0		NA	11.9	13.4	12.5
Total	*st*	49.0	24.0	14.0	10.0	9.0	7.0	87.0	200.0
	fr	782.0	212.0	135.0	144.0	34.0	135.0	1560.0	3002.0
	$\overline{fr}$	16.0	8.8	9.6	14.4	3.8	19.3	17.9	15.0
	σ	23.9	8.5	13.3	19.3	4.8	11.9	26.3	22.3

2.4 Running Experiments With ExRunner

We combine JCrashPack with ExRunner, a tool that can be used for running experiments with a given stack trace-based crash reproduction tool. This tool (i) facilitates the automatic parallel execution of the crash reproduction instances, (ii) ensures robustness in the presence of failures during the crash reproduction failure, and (iii) allows to plug different crash reproduction tools to allow a comparison of their capabilities.

Figure 2.2 gives an overview of ExRunner architecture. The job generator takes as input the stack traces to reproduce, the path to the Jar files associated to each stack trace, and the configurations to use for the stack trace reproduction tool under study. For each stack trace, the job generator analyzes the stack frames and discards those with a target method that does not belong to the target system, based on the package name. For instance, frames with a target method belonging to the Java SDK or other external dependencies are discarded from the evaluation. For each configuration and stack trace, the job generator creates a new job description (i.e., a JSON object with all the information needed to run

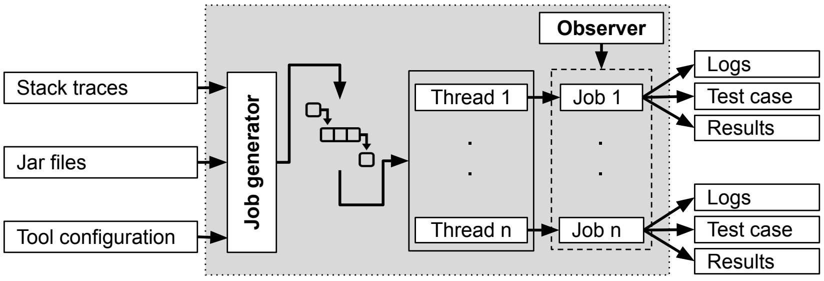


Figure 2.2: ExRunner overview

the tool under study) and adds it to a queue.

To speed-up the evaluation, ExRunner multithreads the execution of the jobs. The number of threads is provided by the user in the configuration of ExRunner and depends on the resources available on the machine and required by one job execution. Each thread picks a job from the waiting queue and executes it. ExRunner users may activate an observer that monitors the jobs and takes care of killing (and reporting) those that do not show any sign of activity (by monitoring the job outputs) for a user-defined amount of time. The outputs of every job are written to separate files, with the generated test case (if any) and the results of the job execution (output results from the tool under study).

For instance, when used with EvoCrash, the log files contain data about the target method, progress of the fitness function value during the execution, and branches covered by the execution of the current test case (in order to see if the line where the exception is thrown is reached). In addition, the results contain information about the progress of search (best fitness function, best line coverage, and if the target exception is thrown), and number of fitness evaluations performed by EvoCrash in an output CSV file. If EvoCrash succeeds to replicate the crash, the generated test is stored separately.

As mentioned by Fraser et al. [109], any research tool developed to generate test cases may face specific challenges. One of these is long (or infinite) execution time of the test during the generation process. To manage this problem, EvoSuite uses a timeout for each test execution, but sometimes it fails to kill sub-processes spawned during the search [109]. We also experienced EvoCrash freezing during our evaluation. In order to handle this problem, ExRunner creates an observer to check the status of each thread executing an EvoCrash instance. If one EvoCrash execution does not respond for 10 minutes (66% of the expected execution time), the Python script kills the EvoCrash process and all of its spawned threads.

Another challenge relates to garbage collection: we noticed that, at some point of the execution, one job (i.e., one JVM instance) allocated all the CPU cores for the execution of the garbage collector, preventing other jobs to run normally. Moreover, since EvoCrash allocates a large amount of heap space to each sub-process responsible to generate a new test case (since the execution of the target application may require a large amount of memory) [109], the garbage collection process could not retrieve enough memory and got stuck, stopping all jobs on the machine. To prevent this behaviour, we set -XX:ParallelGCThreads JVM parameter to 1, enabling only one thread for garbage collection, and limited the number of parallel threads per machine, depending on the maximal amount of allocated memory space. We set the number of active threads to 5 for running on virtual machines, and 25 for running on two powerful machines. Using the logging mechanism in EvoCrash, we are able to see when individual executions ran out of memory.

ExRunner is available together with JCrashPack.[8] It has been used to perform benchmarking for search-based crash reproduction approaches, both EvoCrash and Botsing (an open-source search-based crash reproduction framework for assessing the new techniques introduced in this thesis), yet it has been designed to be extensible to other available stack trace reproduction tools using a plugin mechanism. Integrating another crash reproduction tool requires the definition of two handlers, called by ExRunner: one to run the tool with the inputs provided by ExRunner (i.e. the stack trace, the target frame, and the classpath of the software under test); and one to parse the output produced by the tool to pick up relevant data (e.g., the final status of the crash reproduction, progress of the tool during the execution, *etc.*). Relevant data are stored in a CSV file, readily available for analysis.[9]

2.5 Application To EvoCrash: Setup

Having JCrashPack available allowed us to perform an extensive evaluation of EvoCrash, a state-of-the-art tool in search-based crash replication [28]. Naturally, our first research question deals with the capability of EvoCrash to reproduce crashes from JCrashPack:

RQ$_{1.1}$ *To what extent can EvoCrash reproduce crashes from JCrashPack?*

Since the primary goal of our evaluation is to identify current limitations, we refine the previous research question to examine which frames of the different crashes EvoCrash is able to reproduce:

RQ$_{1.2}$ *To what extent can EvoCrash reproduce the different frames of the crashes from JCrashPack?*

The diversity of crashes in JCrashPack also allows us to investigate how certain types of crashes affect reproducibility. Thus, we investigate whether the exception type and the project nature have an influence on the reproduction rate:

RQ$_{2.1}$ *How does project type influence performance of EvoCrash for crash reproduction?*

In addition, different types of projects might have impact on how costly it is to reproduce the reported crashes for them. The second research question studies the influence of the exception and project type on the performance of EvoCrash:

RQ$_{2.2}$ *How does exception type influence performance of EvoCrash for crash reproduction?*

Finally, we seek to understand why crashes could not be reproduced:

RQ$_3$ *What are the main challenges that impede successful search-based crash reproduction?*

[8]See `https://github.com/STAMP-project/ExRunner-bash`.

[9] The ExRunner documentation includes a detailed tutorial describing how to proceed, available at `https://github.com/STAMP-project/EvoCrash-JCrashPack-application#run-other-crash-replication-tools-with-exrunner`.

2.5.1 Evaluation Setup

Number of executions. Due to the randomness of Guided Genetic Algorithm in EvoCrash, we executed the tool multiple times on each frame. The number of executions has to strike a balance between the threats to external validity (i.e., the number of stack traces considered) and the statistical power (i.e., number of runs) [21, 110]. In our case, we do not compare EvoCrash to other tools (see for instance Soltani *et al.* [28, 36]), but rather seek to identify challenges for crash reproduction. Hence we favor external validity by considering a larger amount of crashes compared to previous studies [28] and ran EvoCrash 10 times on each frame. In total, we executed 18,590 EvoCrash runs.

Search parameters. We used the default parameter values [21, 111] with the following additional configuration options: we chose to keep the reflection mechanisms, used to call private methods, deactivated. The rationale behind this decision is that using reflection can lead to generating invalid objects that break the class invariant [112] during the search, which results in test cases helplessly trying to reproduce a given crash [30].

After a few trials, we also decided to activate the implementation of functional mocking available from EvoSuite [113] in order to minimize possible risks of environmental interactions on crash reproduction. Functional mocking works as follows: when, in a test case, a statement that requires new specific objects to be created (as parameters of a method call for instance) is inserted, either a plain object is instantiated by invoking its constructor, or (with a defined probability, left to its default value in our case) a mock object is created. This mock object is then refined using `when-thenReturn` statements, based on the methods called during the execution of the generated test case. Functional mocking is particularly useful in the cases where the required object cannot be successfully initialized (for instance, if it relies on environmental interactions or if the constructor is accessible only through a factory).

Investigating the impact of those parameters and other parameters (e.g., crossover rate, mutation rate, etc. to overcome the challenges as identified in **RQ**$_3$) is part of our future work.

Search budget. Since our evaluation is executed in parallel on different machines, we choose to express the budget time in terms of number of fitness evaluations: i.e., the number of times the fitness function is called to evaluate a generated test case during the execution of the guided generic algorithm. We set this number to 62,328, which corresponds to the average number of fitness evaluations performed by EvoCrash when running it during 15 minutes on each frame of a subset of 4 randomly selected stack traces on one out of our two machines. Both of the machines have the same configuration: A cluster running Linux Ubuntu 14.04.4 LTS with 20 CPU-cores, 384 GB memory, and a 482 GB hard drive.

We partitioned the evaluation into two, one per available machine: all the stack traces with the same kind of exception have been run on one machine for 10 rounds. For each run, we measure the number of fitness evaluations needed to achieve reproduction (or the exhaustion of the budget if EvoCrash fails to reproduce the crash) and the best fitness value achieved by EvoCrash (0 if the crash is reproduced and higher otherwise). The whole process is managed using ExRunner. The evaluation itself was executed during 10 days on our 2 machines.

2.6 Application To EvoCrash: Results

In this section, we answer the first two research questions on the extent to which the selected crashes and their frames were reproduced and the impact of the project and the exception type on the performance of EvoCrash. We detail the results by analyzing the outcome of EvoCrash in a majority of 10 executions for each frame of each stack trace. We classify the outcome of each execution in one of the five following categories:

reproduced: when EvoCrash generated a test that successfully reproduced the stack trace at the given frame level;

ex. thrown: when EvoCrash generated a test that cannot fully reproduce the stack trace, but covers the target line and throws the desired exception. The frames of the exception thrown, however, do not contain all the original frames;

line reached: when EvoCrash generated a test that covers the target line, but does not throw the desired exception;

line not reached: when none of the tests produced by EvoCrash could cover the target line within the available time budget; and

aborted: when EvoCrash could not generate an initial population to start the search process.

Each outcome denotes a particular state of the search process. For the reproduced frames, EvoCrash could generate a crash-reproducing test within the given time budget (here, 62,328 fitness evaluations). For the frames that could not be reproduced, either EvoCrash exhausted the time budget (for ex. thrown, line reached, and line not reached outcomes) or could not perform the guided initialization (i.e., generate at least one test case with the target method) and did not start the search process (aborted outcomes). For instance, if the class in the target frame is abstract, EvoCrash may fail to find an adequate implementation of the abstract class to instantiate an object of this class during the guided initialization.

2.6.1 Crash Reproduction Outcomes (RQ1)

For $\mathbf{RQ}_1$, we first look at the reproduced and non-reproduced crashes to answer $\mathbf{RQ}_{1.1}$. If EvoCrash was successful in reproducing any frame of a stack trace in a majority of 10 executions, we count the crash as a **reproduced crash**. Otherwise, we count the crash as **not reproduced**. To answer $\mathbf{RQ}_{1.2}$, we detail the results by analyzing the outcome of EvoCrash in a majority of 10 executions for each frame of each stack trace.

Figure 2.3 shows the number of reproduced and not reproduced crashes for each project (and all the projects) and type of exception. EvoCrash is successful in reproducing the majority of crashes (more than 75%) from *Commons-lang*, *Commons-math*, and *Joda-Time*. For the other projects, EvoCrash reproduced 50% or less of the crashes, with only 2 out

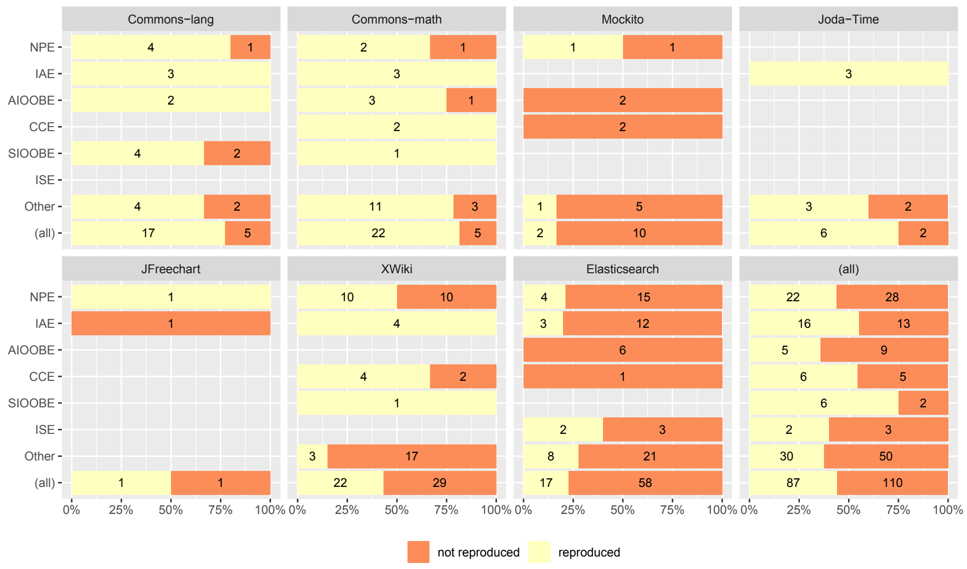


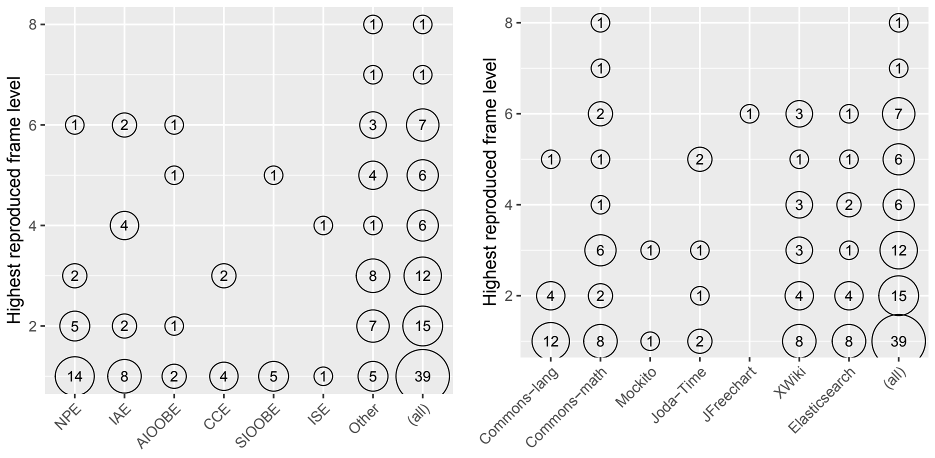
Figure 2.3: Reproduction outcome for the different crashes

of 12 crashes reproduced for *Mockito*. Crashes with an *IllegalArgumentException* are the most frequently reproduced crashed: 16 out of 29 (55%).

Before detailing the results of each frame of each crash, we first look at the frame levels that could be reproduced. Figure 2.4 presents for the 87 stack traces that could be reproduced, the distribution of the highest frame level that could be reproduced for the different crashes for each type of exception (in Figure 2.4a) and each application (in Figure 2.4b). As we can see, EvoCrash replicates lower frame levels more often than higher levels. For instance, for 39 out of the 87 reproduced stack traces, EvoCrash could not reproduce frames beyond level 1 and could reproduce frames up to level 5 for only 9 crashes.

Figure 2.4a indicates that EvoCrash can replicate only the first frame in 14 out of 22 NPE crashes, while there is only one NPE crash for which EvoCrash could reproduce a frame above level 3. In contrast, it is more frequent for EvoCrash to reproduce higher frame levels of IAE stack traces: the highest reproduced frames in 6 out of 16 IAE crashes are higher than 3. Those results suggest that, when trying to reproduce a crash, propagating an illegal argument value through a chain of method calls (i.e., the frames of the stack trace) is easier than propagating a `null` value. According to Figure 2.4b, EvoCrash can reproduce frames higher than 6 only for *Commons-math* crashes. The highest reproduced frames in most of the reproduced crashes in this project are higher than level 2 (12 out of 22). In contrast, for *Elasticsearch* the highest reproduced frame is 1 in most of the crashes.

Both the number of crashes reproduced and the highest level at which crashes could be reproduced confirm the relevance of our choice to consider crashes from XWiki and Elasticsearch, for which the average number of frames (resp. 27.5 and 17.7) is higher than for Defects4J projects (at most 6.0 for JFreeChart), as they represent an opportunity to evaluate and understand current limitations.



(a) In each type of exception (b) In each type of application

Figure 2.4: Highest reproduced frame levels

Frames reproduction outcomes

To answer **RQ**$_{1.2}$, we analyze the results for each frame individually. Figure 2.5 presents a summary of the results with the number of frames for the different outcomes. Figure 2.6 details the same results by application and exception.

Overall, we see in Figure 2.5 that EvoCrash reproduced 171 frames (out of 1,859), from 87 different crashes (out of 200) in the majority of the ten rounds. If we consider the frames for which EvoCrash generated a crash-reproducing test at least once in the ten rounds, the number of reproduced frames increases to 201 (from 96 different crashes). In total, EvoCrash exhausted the time budget for 950 frames: 219 with a test case able to throw the target exception, 245 with a test case able to reach the target line, and 486 without a test case able to reach the line. EvoCrash aborted the search for 738 frames, 455 of which were from Elasticsearch, the application for which EvoCrash had the most difficulties to reproduce a stack trace.

Figure 2.6 details the results by applications (columns) and exceptions (lines). The last line (resp. column), denoted *(all)*, provides the global results for the applications (resp. exceptions). In the remainder of this section, we discuss the results for the different applications and exceptions.

Defects4J applications

For the Defects4J applications, presented in the first five columns in Figure 2.6, in total, 90 (out of 244) of the frames from 48 (out of 71) different crashes were reproduced. For 94 frames, EvoCrash exhausted the time budget (46 ex. thrown, 25 line reached, and 23 line not reached) and aborted for 60 frames from the Defects4J projects.

In particular, only 4 frames out of 61 frames for Mockito were successfully reproduced. For instance, EvoCrash could not reproduce `MOCKITO-4b`, which has only one frame. From

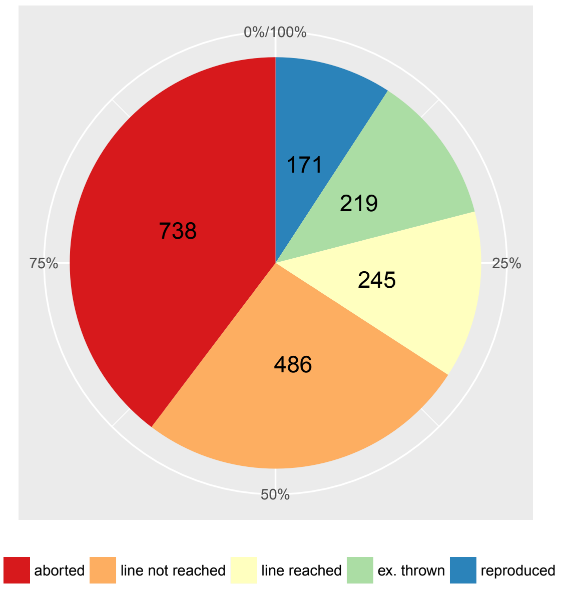


Figure 2.5: An overview of the reproduction outcome

our evaluation, we observe that one very common problem when trying to reproduce a *ClassCastException* is to find which class should be used to trigger the exception.

```
public void noMoreInteractionsWantedInOrder(Invocation undesired){
  throw new VerificationInOrderFailure(join( ...,
                 "..." + undesired.getMock() + "':", ...));
}
```

The exception happens when the `undesired.getMock()` call returns an object that cannot be cast to `String`. During the search, EvoCrash mocks the `undesired` object and assigns some random value to return when the `getMock` method is called. EvoCrash generates a test able to cover the target line, but failing to trigger an exception. Since the signature of this method is `Object getMock()`, EvoCrash assigns only random `Object` values to return, where, from the original stack trace, a `Boolean` value is required to trigger the exception.

XWiki and Elasticsearch

XWiki is one of the industrial open source cases in the evaluation, for which 53 (out of 706) frames were successfully reproduced, 430 could not be reproduced with the given time budget (125 ex. thrown, 127 line reached, and 178 line not reached), and 223 aborted during the generation of the initial population. EvoCrash reproduced only 28 (out of 909) frames from Elasticsearch, for which, the majority of frames (455) aborted during the generation of the initial population. However, EvoCrash was able to start the search for 426 frames (48 ex. thrown, 93 line reached, and 285 line not reached).

Variability of the reproductions. We also observed that XWiki and Elasticsearch have the highest variability in their outcomes. For XWiki (resp. Elasticsearch), 4 (resp. 3)

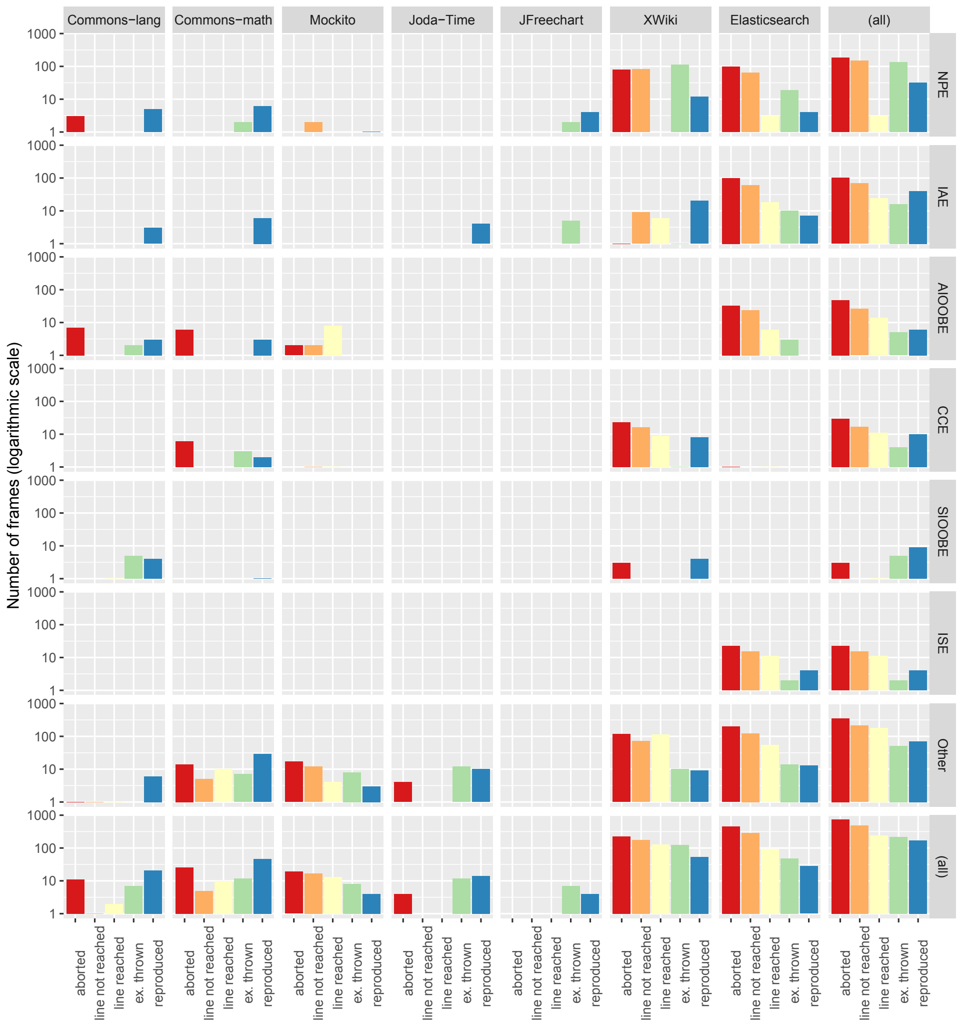


Figure 2.6: Detailed reproduction outcome for the different frames.

frames that could be reproduced in a majority of time could however not be reproduced 10 out of 10 times, compared to 2 frames for Commons-lang and Commons-math. This could indicate a lack of guidance in the current fitness function of EvoCrash. For instance, for the Elasticsearch crash ES-26833, EvoCrash could only reproduce the third frame 4 times out of 10 and was therefore not considered as reproduced. After a manual inspection, we observed that EvoCrash gets stuck after reaching the target line and throwing the expected exception. From the intermediate test cases generated during the search, we see that the exception is not thrown by the target line, but a few lines after. Since the fitness value improved, EvoCrash got stuck into a local optima, hence the lower frequency of reproduction for that frame.[10] Out future work includes improvement of the guidance in the fitness function and a full investigation of the fitness landscape to decrease the variability of EvoCrash outcomes.

Importance of large industrial applications. Compared to Defects4J and XWiki applications, the crash reproduction rate drops from 36.9% for Defects4J, to 7.5% for XWiki, and only 3% for Elasticsearch. Those results emphasize the importance of large industrial applications for the assessment of search-based crash reproduction and enforce the need of context-driven software engineering research to identify relevant challenges [80].

Additionally to the larger variability of reproduction rate, we observe that frequent use of *Java generics* and *static initialization*, and most commonly, automatically generating suitable input data that resembles `http` requests are among the major reasons for the encountered challenges for reproducing Elasticsearch crashes. In Section 2.7 we will describe **14** categories of challenges that we identified as the underlying causes for the presented execution outcomes.

Exceptions

The lines in Figure 2.6 presents the outcomes for the different exceptions. In particular, NPE, IAE, AIOOBE, and CCE are the most represented exceptions in JCrashPack. For those exceptions, EvoCrash could reproduce, respectively, 32 (out of 499), 40 (out of 250), 6 (out of 99), and 10 (out of 72) frames. Looking at the reproduction frequency, IAE is the most frequently reproduced exception (16%), followed by CCE (13.8%), NPE (6.4%), and AIOOBE (6%).

This contrast with the number of frames for which EvoCrash aborted the search, where NPE has the lowest frequency (181 frames, 36.2%), followed by IAE (101 frames, 40.4%), CCE (30 frames, 41.6%), and AIOOBE (48 frames, 48.4%). Interestingly, those numbers show that EvoCrash is able to complete the guided initialization for NPEs more often than for other exceptions.

Figure 2.6 also shows that the number of test cases that reach the line is low for NPEs, meaning that whenever EvoCrash generates at test able to cover the line (line reached), the evolution process will be able to progress and generate another test that throws an exception (ex. thrown).

Summary (RQ$_1$) To what extent can EvoCrash reproduce crashes from JCrashPack, and how far it can proceed in the stack traces? Overall, EvoCrash reproduced 171 frames (out of 1,859 - 9%), from 87 different crashes (out of 200 - 43.5%) in a majority out of 10 executions. Those numbers climb to 201 frames (10.8%) from 96 crashes (48%) if we consider at least one reproduction in one of the 10 executions. In most of the reproduced crashes, EvoCrash can only reproduce the first two frames. It indicates that since EvoCrash needs higher accuracy in setting the state of the software under test for reproducing higher frames, increasing the length of the stack trace reduces the chance of this tool for crash reproduction. When looking at larger industrial applications, the crash

[10]A detailed analysis is available at https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/manual-analysis/Elasticsearch/ES-26833.md

Table 2.5: Statistics for the average number of fitness evaluations for the *reproduced* frames (**fr**) belonging to different stack traces (**st**), grouped by **applications**, out of 10 rounds of execution. The confidence Interval (**CI**) is calculated for the median bootstrapping with *100,000* runs, at a 95% confidence level.

Applications	st	fr	Min	Lower Quart.	Median CI	Med.	Upper Quart.	Max
Com.-lang	19	213	1	2.0	[5.0 ,22.0]	15.0	237.0	52,240
Com.-math	24	471	1	13.0	[124.0 ,211.0]	178.0	1,046.5	58,731
Mockito	2	40	1	1.0	[1.0 ,1.0]	1.0	5.2	138
Joda-Time	6	138	1	15.5	[79.1 ,369.0]	253.5	1,290.2	40,189
JFreechart	1	41	1	10.0	[-292.0 ,350.0]	221.0	1,188.0	20,970
XWiki	25	531	1	2.5	[14.0 ,30.0]	23.0	209.0	34,089
Elasticsearch	19	287	1	4.0	[5.0 ,32.0]	23.0	125.0	17,461
Total	96	1721	1	4.0	[34.0 ,59.0]	48.0	534.0	58,731

Table 2.6: Statistics for the average number of fitness evaluations for the *reproduced* frames (**fr**) belonging to different stack traces (**st**), grouped by **exceptions**, out of 10 rounds of execution. Confidence Interval (**CI**) is calculated for median with bootstrapping with *100,000* runs, at 95% confidence level.

Applications	st	fr	Min	Lower Quart.	Median CI	Med.	Upper Quart.	Max
NPE	26	330	1	6.0	[9.0 ,63.0]	44.5	220.0	34,089
IAE	16	399	1	2.0	[7.0 ,12.0]	10.0	49.0	38,907
AIOOBE	5	58	1	15.5	[252.0 ,1,104.5]	675.0	1,671.2	53,644
CCE	6	103	1	6.5	[74.0 ,210.0]	120.0	560.0	10,197
SIOOBE	8	95	1	12.5	[122.0 ,945.0]	505.0	2,326.0	52,240
ISE	2	42	1	1.0	[1.0 ,3.0]	2.0	105.8	1,138
Other	33	694	1	7.0	[99.0 ,139.0]	125.5	825.0	58,731
Total	96	1721	1	4.0	[34.0 ,59.0]	48.0	534.0	58,731

reproduction rates drop from 36.9% for Defects4J to 7.5% for XWiki and 3% for Elasticsearch. The most frequently reproduced exceptions are IllegalArgumentExceptions. The exceptions for which EvoCrash is the most frequently able to complete the guided initialization are NullPointerExceptions.

2.6.2 Impact of Exception Type and Project on Performance (RQ2)

To identify the distribution of fitness evaluations per exception type and project, we filtered the reproduced frames out of the 10 rounds of execution. Tables 2.5 and 2.6 present the statistics for these executions, grouped by application and exception types, respectively.

We filtered out the frames that were not reproduced to analyze the impact of project and exception types on the average number of fitness evaluations and, following recommendations by Arcuri and Briand [110], we replaced the test of statistical difference by a confidence interval. For both groups, we calculated confidence intervals with a 95% confidence level for medians with bootstrapping with 100,000 runs.[11]

As Table 2.5 shows, for four projects (Commons-lang, Mockito, XWiki, and Elasticsearch) the median number of fitness evaluations is low. On the contrary, the cost of crash reproductions for Commons-math, Joda-Time, and JFreechart are higher in comparison to the rest of projects. By comparing those results with the projects sizes reported in Table 2.3, where the largest projects are XWiki (with $\overline{NCSS} = 177.84k$) and Elasticsearch (with $\overline{NCSS} = 124.36k$), we observe that the effort required to reproduce a crash cannot be

[11]We used the *boot* function from the *boot* library in R to compute the *basic* intervals with bootstrapping. See `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/tree/master/results` to reproduce the statistical analysis.

solely predicted by the project size. This is consistent with the intuition that the difficulty of reproducing a crash only depends on the methods involved in the stack trace.

Similarly, according to Figure 2.1a, the average CCN for Mockito, XWiki, and Elasticsearch is lower compared to other projects. Table 2.5 shows that reproducing crashes from these projects is less expensive, and that reproducing crashes from Commons-math, Joda-Time, and JFreechart, which all have higher average CCN, is more expensive. We also observe that the average CCN for Commons-lang is high, however, contradicting the intuition that crashes from projects higher CCN are more expensive to reproduce, the cost for reproducing crashes in Commons-lang is low compared to other projects. This can be explained by the levels of the frames reproduced by EvoCrash: according to Figure 2.4, the average level of the reproduced frames in the crashes from Commons-lang is low compared to the other projects and, as we discussed in the previous section, reproducing crashes with fewer frames is easier for EvoCrash.

In general, we observe that the performance of EvoCrash depends on the complexity of the project and the frame level in the stack trace. Future work includes further investigations to determine which other factors (e.g., code quality) can influence EvoCrash performance.

From Table 2.6, we observe that for *CCE*, *SIOOBE*, and *AIOOBE*, the cost of generating a crash-reproducing test case is high, while for *NPE*, *IAE*, and *ISE*, the cost is lower. One possible explanation could be that generating input data which is in a suitable state for causing cast conflicts, or an array which is in the right state to be accessed by an illegal index is often non-trivial.

In contrast, to trigger an NPE, it is often enough to return a `null` value not checked by the crashing method. For example, Listing 2.1 shows the stack trace of CHART-4b, a crash from the JFreeChart application. The crash happens at line 1490 of the `createScatterPlot` method presented in Listing 2.2. Listing 2.3 shows the test case generated by EvoCrash that reproduces the 6th frame (line 6 in Listing 2.1) of the stack trace. First, the test initializes the mocks used as mandatory parameters values (from line 2 to 4), before calling the `createScatterPlot` method (at line 5). The `ds` `XYDataset` mock is used along the various calls (from line 6 to 1 in Listing 2.1), up to the method `getDataRange` presented in Listing 2.4 that triggers the NPE at line 4493. In our case, the `null` value is returned by the `getRendererForDataset` call with the propagated `ds` mock at line 4491.

Example 2.1: Stack trace for the crash CHART-4b

```
0 java.lang.NullPointerException
1         at org.jfree.chart.plot.XYPlot.getDataRange(XYPlot.java:4493)
2         at org.jfree.chart.axis.NumberAxis.autoAdjustRange(NumberAxis.java:434)
3         at org.jfree.chart.axis.NumberAxis.configure(NumberAxis.java:417)
4         at org.jfree.chart.axis.Axis.setPlot(Axis.java:1044)
5         at org.jfree.chart.plot.XYPlot.<init>(XYPlot.java:660)
6         at org.jfree.chart.ChartFactory.createScatterPlot(ChartFactory.java:1490)
```

Example 2.2: Code excerpt from JFreeChart `ChartFactory.java`

```
1478 public static JFreeChart createScatterPlot(String title, String xAxisLabel,
1479     String yAxisLabel, XYDataset dataset, PlotOrientation orientation,
1480     boolean legend, boolean tooltips, boolean urls) {
1481
1482         if (orientation == null) {
1483                 throw new IllegalArgumentException(”Null␣’orientation’␣argument.”);
```

```
        }
        NumberAxis xAxis = new NumberAxis(xAxisLabel);
        xAxis.setAutoRangeIncludesZero(false);
        NumberAxis yAxis = new NumberAxis(yAxisLabel);
        yAxis.setAutoRangeIncludesZero(false);

        XYPlot plot = new XYPlot(dataset, xAxis, yAxis, null);

        [...]
}
```

Example 2.3: The test case generated by EvoCrash for reproducing the 6th frame of CHART-4b

```
public void test()  throws Throwable  {
 XYDataset ds = mock(XYDataset.class, new ViolatedAssumptionAnswer());
 doReturn(0).when(ds).getSeriesCount();
 PlotOrientation pl = mock(PlotOrientation.class, new ViolatedAssumptionAnswer());
 ChartFactory.createScatterPlot((String) null, (String) null, (String) null, ds, pl, true,
   true, true);
}
```

Example 2.4: Code excerpt from JFreeChart `XYPlot.java`

```
public Range getDataRange(ValueAxis axis) {
 XYItemRenderer r = getRendererForDataset(d); // d == ds and getRendererForDataset(d)
   returns null
 [...]
 Collection c = r.getAnnotations(); // r is null and throws a NPE
 [...]
}
```

Considering the presented results in Figure 2.6 and Table 2.5, crash replication for various exceptions may be dependent on project type. Figure 2.7 presents the results of crash reproduction grouped both by applications and exception types. As the figure shows, the cost of reproducing NPE is lower for Elasticsearch, compared to XWiki and JFreechart, and the cost of reproducing IAE is lower for Commons-lang than for Elasticsearch. We also observe differences in terms of costs of reproducing AIOOBE and SIOOBE for different projects.

Summary (RQ$_{2.1}$) How does project type influence performance of EvoCrash for crash reproduction? We observed that the factors are (i) the complexity of the the project, and (ii) the level of the reproduced frames (reproducing higher frame requires more effort). Furthermore, we see no link between the size of the project and the effort required to reproduce one of its crashes.

Summary (RQ$_{2.2}$) How does exception type influence performance of EvoCrash for crash reproduction? For the exceptions, we observe that for ClassCastException, ArrayIndexOutOfBoundsException and StringIndexOutOfBoundsException, the cost of generating a crash-reproducing test case is high, while for NullPointerException, IllegalArgumentException, and IllegalStateException, the cost is lower. This result indicates that the cost of reproducing types of exceptions for a non-trivial scenario (*e.g.*, class conflicts or accessing an illegal state of an array) needs a more complex input generation. Furthermore, accessing the corresponding complex state is more time consuming for the search process.

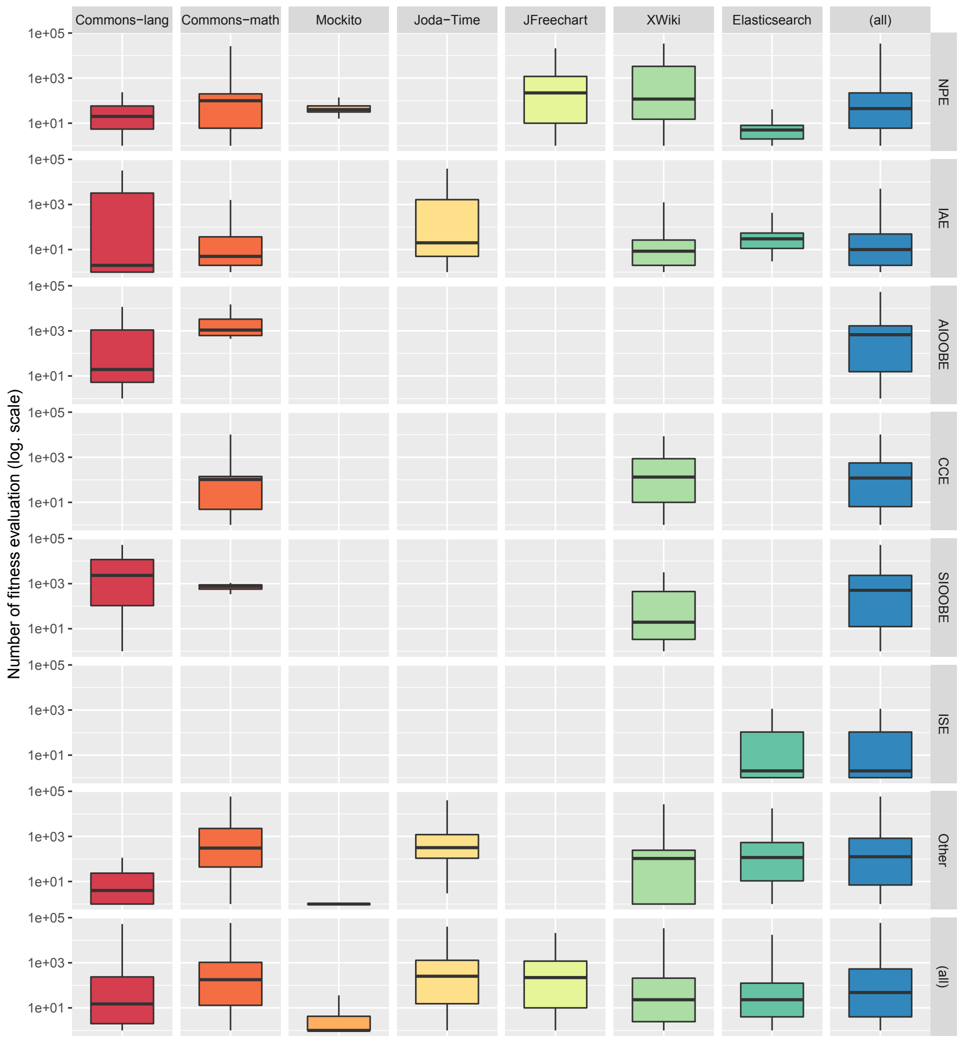


Figure 2.7: Average number of fitness evaluations for the *reproduced* frames for each applications and exception type.

2.7 Challenges For Crash Reproduction (RQ3)

To identify open problems and future research directions, we manually analyzed the execution logs of 1,653 frames that could not be reproduced in any of the 10 executions. This analysis includes a description of the reason why a frame could not be reproduced.[12] Based on those descriptions, we grouped the reason of the different failures into 13 categories and identified future research directions. Table 2.7 provides the number and frequency of frames classified in each category.[13] The complete categorization table is available in our replication package.[14]

For each challenge, we discuss to what extent it is crash-reproduction-specific and its relation to search-based software testing in general. In particular, for challenges previously identified by the related literature in search-based test case generation, we highlight the differences originating from the crash reproduction context.

2.7.1 Input Data Generation

Generating complex input objects is a challenge faced by many automated test generation approaches, including search-based software testing and symbolic execution [7]. Usually, the input space of each input is large and generating proper data enabling the search process to cover its goals is difficult.

As we can see from Table 2.7, this challenge is substantial in search-based crash reproduction. Trying to replicate a crash for a target frame requires to set the input arguments of the target method and all the other calls in the sequence properly such that when calling the target method, the crash happens. Since the input space of a method is usually large, this can be challenging. EvoCrash uses randomly generated input arguments and mock objects as inputs for the target method. As we described in Section 2.6, we observe that a widespread problem when reproducing a *ClassCastException* (CCE) is to identify which types to use as input parameters such that a CCE is thrown. In the case of a CCE, this information can be obtained from the error message of the exception. Our future work includes harvesting additional information, like error messages, to help the search process.

We also noticed that some stack traces involving Java generic types make EvoCrash abort the search after failing to inject the target method in every generated test during the guided initialization phase. Generating generic type parameters is also a recognized challenge for automated testing tools for Java [114]. To handle these parameters, EvoCrash, based on EvoSuite's implementation [114], collects candidate types from `castclass` and `instanceof` operators in Java bytecode, and randomly assign them to the type parameter. Since the candidate types may themselves have generic type parameters, a threshold is used to avoid large recursive calls to generic types. One possible explanation for the crashes in these cases could be that the threshold is not correctly tuned for the kind of classes involved in the recruited projects. Thus, the tool fails to set up the target method to inject to the tests. Based on the results of our evaluation, handling Java generics in Evo-

[12]Available at `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/tree/master/results/manual-analysis`.

[13]For each category, we provide illustrative examples from `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/tree/master/results/examples`.

[14]The full table is available at `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/manual-analysis/categorisation.csv`.

Table 2.7: Challenges with the number and percentage of frames identified for this challenge.

Category	**Frames**	**Frequency**
Input Data Generation	825	49.91%
Abstract Class	242	14.64%
Anonymous Class	142	8.59%
Static Initialization	141	8.53%
Complex Code	118	7.14%
Private Inner Class	56	3.39%
Environmental Dependencies	52	3.15%
Irrelevant Frame	37	2.24%
Unknown Sources	16	0.97%
Nested calls	10	0.60%
try/catch	7	0.42%
Interface	6	0.36%
Empty Enum Type	1	0.06%
Total	1653	100%

Example 2.5: Excerpt of the stack trace for the crash XWIKI-13708

```
java.lang.NullPointerException: null
  at com.xpn.xwiki.internal.template.TemplateListener.onEvent(TemplateListener.java:79)
  at org.xwiki.observation.internal.DefaultObservationManager.notify([...]:307)
  at org.xwiki.observation.internal.DefaultObservationManager.notify([...]:269)
  [...]
```

Crash needs further investigation to identify the root cause(s) of the crashes and devise effective strategies to address them.

For instance, EvoCrash cannot reproduce the first frame of crash XWIKI-13708[15], presented in Listing 2.5. The target method `onEvent` (detailed in Listing 2.6) has three parameters. EvoCrash could not reach the target line (line 78 in Listing 2.6) as it failed to generate a fitted value for the second parameter (`source`). This (`Object`) parameter should be castable to `XWikiDocument` and should return values for `getXObject()` or `getAttachment()` (using mocking for instance).

Chosen examples: XWIKI-13708, frame 1; ES-22922, frame 5; ES-20479, frame 10.[16]

2.7.2 Complex Code

Generating tests for complex methods is hard for any search-based software testing tool [115]. In this study, we indicate a method as complex if (i) it contains more than 100 lines of code and high cyclomatic complexity; (ii) it holds nested predicates [115, 116]; or (iii) it has the *flag problem* [34, 116], which include (at least one) branch predicate with a binary

[15]`https://jira.xwiki.org/browse/XWIKI-13708`

[16]See `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/examples/InputDataGeneration.md`.

Example 2.6: Code excerpt from method `onEvent` in `TemplateListener.java`

```java
public void onEvent(Event event, Object source, Object data) {
  XWikiDocument document = (XWikiDocument) source;

  if (document.getXObject(WikiSkinUtils.SKINCLASS_REFERENCE) != null) {
    if (event instanceof AbstractAttachmentEvent) {
      XWikiAttachment attachment = document.getAttachment(((AbstractAttachmentEvent)
        event).getName());
      String id = this.referenceSerializer.serialize(attachment.getReference()); // target
        line
      [...]
    }
  }
}
```

(boolean) value, making the landscape of the fitness function flat and turning the search into a random search [115].

As presented in Section 2.1.2, the first component of the fitness function that is used in EvoCrash encodes how close the algorithm is to reach the line where the exception is thrown. Therefore, frames of a given stack trace pointing to methods with a high code complexity[17] are more costly to reproduce, since reaching the target line is more difficult.

Handling complex methods in search-based crash reproduction is harder than in general search-based testing. The search process in crash reproduction should cover (in most cases) only one specific path in the software under test to achieve the reproduction. If there is a complex method on this path, the search process cannot achieve reproduction without covering it. Unlike the more general coverage driven search-based testing approach (with line coverage for instance), where the are usually multiple possible executions paths to cover a goal.

Chosen examples: XWIKI-13096, frame 3; ES-22373, frame 10.[18]

2.7.3 Environmental Dependencies

As discussed by Arcuri et al. [117], generating unit tests for classes which interact with the environment leads to (i) difficulty in covering certain branches which depend on the state of the environment, and (ii) generating flaky tests [118], which may sometimes pass, and sometimes fail, depending on the state of the environment. Despite the numerous advances made by the search-based testing community in handling environmental dependencies [21, 117], we noticed that having such dependencies in the target class hampers the search process. Since EvoCrash builds on top of EvoSuite [41], which is a search-based *unit* test generation tool, we face the same problem in the crash reproduction problem as well.

For instance, Listing 2.7 shows the stack trace of the crash XWIKI-12584.[19] During the

[17]In some cases for Elasticsearch, the failing methods have nearly 300 lines of source code.

[18]See `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/examples/ComplexCode.md`.

[19]Reported at `https://jira.xwiki.org/browse/XWIKI-12584` and analyzed at `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/manual-analysis/Xwiki/`

Example 2.7: Stack trace for the crash XWIKI-12584

```
java.lang.ClassCastException: [Ljava.lang.Object; cannot be cast to java.lang.String
  at [...].XWikiHibernateStore.searchDocumentReferencesInternal([...]:2457)
  at [...].XWikiHibernateStore.searchDocumentsNamesInternal([...]:2440)
  at [...].XWikiHibernateStore.searchDocumentsNames([...]:2246)
  at [...].XWikiHibernateStore.searchDocumentsNames([...]:2230)
  at [...].XWikiCacheStore.searchDocumentsNames([...]:373)
  at [...].XWiki.searchDocuments([...]:576)
```

evaluation, EvoCrash could not reproduce any of the frames of this stack trace. During our manual analysis, we discovered that, for the four first frames, EvoCrash was unable to instantiate an object of class `XWikiHibernateStore`,[20] resulting in an abortion of the search. Since the class `XWikiHibernateStore` relies on a connection to an environmental dependency (here, a database), generating unit test requires substantial mocking code[21] that is hard to generate for EvoCrash. As for input data generation, our future work includes harvesting and leveraging additional information from existing tests to identify and use relevant mocking strategies.

Chosen examples: ES-21061, frame 4; XWIKI-12584, frame 4.[22]

2.7.4 Static Initialization

In Java, static initializers are invoked only once when the class containing them is loaded. As explained by Fraser and Arcuri [21], these blocks may depend on static fields from other classes on the classpath that have not been initialized yet, and cause exceptions such as `NullPointerException` to be thrown. In addition, they may involve environmental dependencies that are restricted by the security manager, which may also lead to unchecked exceptions being generated.

In our crash reproduction benchmark, we see that about 9% (see Table 2.7) of the cases cannot be reproduced as they point to classes that have static initializers. When such frames are used for crash reproduction with EvoCrash, the tool currently aborts the search without generating any crash reproducing test. As Fraser and Arcuri [21] discuss, automatically determining and solving all possible kinds of dependencies in static initializers is a non-trivial task that warrants dedicated research.

Chosen examples: ES-20045, frames 1 and 2.[23]

XWIKI-12584.md.

[20]See https://github.com/xwiki/xwiki-platform/blob/xwiki-platform-7.2-milestone-2/xwiki-platform-core/xwiki-platform-oldcore/src/main/java/com/xpn/xwiki/store/XWikiHibernateStore.java

[21]See https://github.com/xwiki/xwiki-platform/blob/xwiki-platform-7.2-milestone-2/xwiki-platform-core/xwiki-platform-oldcore/src/test/java/com/xpn/xwiki/store/XWikiHibernateStoreTest.java

[22]See https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/examples/EnvironmentalDependencies.md.

[23]See https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/examples/StaticInitialisation.md.

2.7.5 Abstract Classes And Methods

In Java, abstract classes cannot be instantiated. Although generating coverage driven unit tests for abstract classes is possible (one would most likely generate unit tests for concrete classes extending the abstract one or use a parameterized test to check that all implementations respect the contract defined by the abstract class), when a class under test is abstract, EvoSuite (as the general test generation tool for java) looks for classes on the classpath that extend the abstract class to create object instances of that class. In order to cover (e.g., using line coverage) specific parts of the abstract class, EvoSuite needs to instantiate the right concrete class allowing to execute the different lines of the abstract class.

For crash reproduction, as we can see from Table 2.7, it is not uncommon to see abstract classes and methods in a stack trace. In several cases from Elasticsearch, the majority of the frames from a given stack trace point to an abstract class. Similarly to coverage-driven unit test generation, EvoCrash needs to instantiate the right concrete class: if EvoCrash picks the same class that has generated the stack trace in the first place, then it can generate a test for that class that reproduces the stack trace. However, if EvoCrash picks a different class, it could still generate a test case that satisfies the first two conditions of the fitness function (section 2.1.2). In this last case, the stack trace generated by the test would match the frames of the original stack trace, as the class names and line numbers would differ. The fitness function would yield a value between 0 and 1, but it may never be equal to 0.

Chosen examples: ES-22119, frames 3 and 4; XRENDERING-422, frame 6.[24]

2.7.6 Anonymous Classes

As discussed in the study by Fraser *et al.* [41], generating automated tests for covering anonymous classes is more laborious because they are not directly accessible. We observed the same challenge during the manual analysis of crash reproduction results generated by EvoCrash. When the target frame from a given crash stack trace points to an anonymous object or a lambda expression, guided initialization in EvoCrash fails, and EvoCrash aborts the search without generating any test.

Chosen examples: ES-21457, frame 8; XWIKI-12855, frames 30 and 31.[25]

2.7.7 Private Inner Classes

Since it is not possible to access a private inner class, and therefore, not possible to directly instantiate it, it is difficult for any test generation tool in Java to create an object of this class. As for anonymous classes, this challenge is also present for crash reproduction approaches. In some crashes, the target frame points to a failing method inside a private inner class. Therefore, it is not possible to directly inject the failing method from this class during the guided initialization phase, and EvoCrash aborts the search.

[24]See `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/examples/AbstractClass.md`.

[25]See `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/examples/AnonymousClass.md`.

Chosen example: MATH-58b, frame 3.[26]

2.7.8 Interfaces

In 6 cases, the target frame points to an interface. In Java, similar to abstract classes, interfaces may not be directly instantiated. In these cases also, EvoCrash randomly selects the classes on the classpath that implement the interface and, depending on the class picked by EvoCrash, the fitness function may not reach 0.0 during the search if the class is different from the one used when the input stack trace has been generated. This category is a special case of Abstract classes and methods (described in Section 2.7.5), however, since the definition of a default behavior for an interface is a feature introduced by Java 8 [119] that has, to the best of our knowledge, not been previously discussed for search-based testing, we choose to keep it as a separate category.

Chosen example: ES-21457, frame 9.[27]

2.7.9 Nested Private Calls

In multiple cases, the target frame points to a private method. As we mentioned in Section 2.5, those private methods are not directly accessible by EvoCrash. To reach them, EvoCrash detects other public or protected methods which invoke the target method directly or indirectly and randomly choose during the search. If the chain of method calls, from the public caller to the target method, is too long, the likelihood that EvoCrash may fail to pick the right method during the search increases.

In general, calling private methods is challenging for any automated test generation approach. For instance, Arcuri *et al.* [113] address this problem by using the Java reflection mechanism to access private methods and private attributes during the search. As mentioned in Section 2.5.1, this can generate invalid objects (with respect to their class invariants) and lead to generating test cases helplessly trying to reproduce a given crash [30].

Chosen examples: XRENDERING-422, frames 7 to 9.[28]

2.7.10 Empty `enum` Type

In the stack trace of the ES-25849 crash,[29] the 4th frame points to an empty enumeration Java type.[30] Since there are no values in the enumeration, EvoCrash was not able to instantiate a value and aborted during the initialization of the population. Frames pointing to code in an empty enumeration Java type should not be selected as target frames and could be filtered out using a preliminary static analysis.

[26]See `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/examples/PrivateInnerClass.md`.

[27]See `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/examples/Interface.md`.

[28]See `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/examples/NestedPrivateCalls.md`.

[29]The analysis is available at `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/manual-analysis/Elasticsearch/ES-25849.md`.

[30]See `https://github.com/jimczi/elasticsearch/blob/0a4b38b60c2752cdc6de819f5bf3414bd01f88c5/core/src/main/java/org/elasticsearch/index/fielddata/ordinals/GlobalOrdinalsBuilder.java`.

Chosen example: ES-25849, frame 4.

2.7.11 Frames With `try/catch`

Some frames have a line number that designates a call inside a `try/catch` block. When the exception is caught, it is no longer thrown at the specific line given in the trace, rather it is typically handled inside the associated `catch` blocks. From what we observed, often catch blocks either (i) re-throw a checked exception, which yield chained stack traces with information that is not exactly as the input stack trace but can still be used for crash reproduction; or (ii) log the caught exception. Since EvoCrash only considers uncaught exceptions that are generated as the result of running the generated test cases during the search, the logged stack traces is presently no use for crash reproduction. Also, even if a stack trace is recorded to an error log, this stack trace is not the manifestation of a crash *per se*. Indeed, once the exception logged, the execution of the program continues normally.

For instance, for the crash ES-20298,[31] EvoCrash cannot reproduce the fourth frame of the crash. This frame points to the following method call in a `try` and `catch`:

```
try {
    processResponse(response);
} catch (Throwable t) {
    onFailure(t);
}
```

Even if an exception is thrown by the `processResponse` method, this exception is caught and logged, and the execution of the program continues normally.

Generally, if an exception is caught in one frame, it cannot be reproduced (as it cannot be observed) from higher level frames. For instance, for ES-20298, all frames above level 4 cannot be reproduced since the exception is catch in frame 4 and not propagated to the higher frames. This property of a crash stack trace implies that, for now, depending on where in the trace such frames exist, only a fraction of the input stack traces can actually be used for automated crash reproduction. Future development of EvoCrash can alleviate this limitation by, additionally to the monitoring of uncaught exceptions, read the error log to affecting the propagation of exceptions during execution. However, unlike other branching instructions relying on boolean values, for which classical coverage driven unit test generation can use the branch distance (see Section 2.1.2) to guide the search [4], there is little guidance offered for `try/catch` instructions since the branching condition is implicit in one or more instructions in the `try`.

Chosen example: ES-14457, frame 4.[32]

2.7.12 Missing Line Number

31 frames in JCrashPack have frames with a missing line number, as shown in Listing 2.8. This happens if the Java files have been compiled without any debug information (by default, the Java compiler add information about the source files and line numbers, for

[31]Reported at https://github.com/elastic/elasticsearch/issues/20298 and analyzed at https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/manual-analysis/Elasticsearch/ES-20298.md

[32]See https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/examples/TryCatch.md.

Example 2.8: An excerpt of the stack trace from the crash XRENDERING-422 with missing line numbers

```
at org.apache.xerces.parsers.XMLParser.parse(Unknown Source)
at org.apache.xerces.parsers.AbstractSAXParser.parse(Unknown Source)
at org.xml.sax.helpers.XMLFilterImpl.parse(XMLFilterImpl.java:357)
```

instance, when printing a stack trace) or if the frame points to a class part of the standard Java library and the program has been run in the Java Runtime Environment (JRE) and not the JDK.

Since EvoCrash currently requires a line number to compute the fitness values during the search, those frames have been ignored during our evaluation and do not appear in the results. Yet, as frames with missing line number appear in JCrashPack (and in other stack traces), we decided to mention this trial here as a search-based crash reproduction challenge. A possible solution, as the future work, is to relax the fitness function so that it can still approximate fitness if line numbers are missing.

Chosen example: XRENDERING-422.[33]

2.7.13 Incorrect Line Numbers

In 37 cases, the target frame points to the line in the source code where the target class or method is defined. This happens when the previous frame points to an anonymous class or a lambda expression. Such frames practically cannot be used for crash reproduction as the location they point to does not reveal where exactly the target exception occurs. One possible solution would be to consider the frame as having a missing line number and use the relaxed fitness function to approximate the fitness.

Chosen examples: MATH-49b, frames 1 and 4.[34]

2.7.14 Unknown

We were unable to identify why EvoCrash failed to reproduce 16 frames (out of 1,653 frames manually analyzed). In these cases, neither the logs nor the source code could help us understand how the exception was propagated.

Summary (RQ$_3$) What are the open problems that need to be solved to enhance search-based crash reproduction? Based on the manual analysis of the frames that could not be reproduced at least once out of 10 rounds of executions, we identified 13 challenges for search-based crash reproduction. We confirmed challenges previously identified in other search-based software testing approaches and specified how they affect search-based crash reproduction. And discovered new challenges, more specific to search-

[33]The stack trace is available at `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/evaluation/JarFiles/resources/logs/XWIKI/XRENDERING-422/XRENDERING-422.log`

[34]See `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/examples/IrrelevantFrames.md`.

based crash reproduction and explained how the can affect other search-based software testing approaches.

These challenges are related to the difficulty to generate test cases due to complex input data, environmental dependencies, or complex code; abstraction (static initialization, interfaces, abstract, and anonymous classes); encapsulation mechanisms (private inner classes and nested private calls in the given stack trace) of object-oriented languages; or the selection of the target frame in crash reproduction (in `try`/`catch` blocks, in empty enumerations, when the location in the source code is unknown, or when the frame has an incorrect line number).

2.8 Discussion

2.8.1 Empirical Evaluation For Crash Reproduction

Conducting empirical evaluation for crash reproduction is challenging. It requires to collect various artifacts from different sources and to analyze the results to determine, in the case of a negative outcome, the cause that prevents the crash reproduction. Some are easy to fix, like missing dependencies that were added to the project linked to the stack trace, and for which we rerun the evaluation on the stack traces. The others are detailed in Section 2.7, and serve to identify future research directions.

One of the most surprising causes is due to a line mismatch in some stack traces. During the manual analysis of our results, we found out that three frames in two different stack traces, coming from Defects4J projects, target the wrong lines in the source code: the line numbers in the stack traces point to lines in the source code that cannot throw the targeted exception. Since the stack traces were collected directly from the Defects4J data (which reports failing tests and their outputs), we tried to regenerate them using the provided test suite and found a mismatch between the line numbers of the stack traces indeed. We reported those two projects to the Defects4J developers:[35] a bug in JDK7 [120] causes this mismatch. Since EvoCrash relies on line numbers to guide its search, it could not reproduce the crashes. We recompiled the source code, updated the stack trace accordingly in JCrashPack, and rerun the evaluation for those two stack traces.

Thanks to JCrashPack and ExRunner, we are now able to ease empirical evaluation for crash reproduction. ExRunner can be extended to other crash reproduction tools[36] for comparison, or assess the development of new ideas in existing tools. Our future work also includes the prioritization of crashes from JCrashPack to allow quick feedback on new ideas in a fast and automated way [24].

2.8.2 Usefulness For Debugging

In our evaluation, we focused on the crash-replication capabilities of EvoCrash and identified problems affecting those capabilities. We considered the generated tests only to classify the outcomes of the EvoCrash generation process but did not assess their actual usefulness for debugging.

Chen *et al.* [30] introduced a usefulness criterion for the crash reproduction approaches. According to this criterion, a crash reproducing test is useful to the developers if it covers

[35]See the issue at `https://github.com/rjust/defects4j/issues/142`.

[36]See how to extend ExRunner at `https://github.com/STAMP-project/ExRunner`.

the buggy frame: i.e., if the target frame for which the reproduction is successful is higher than the frame that points to the buggy method.

In our previous work [28], we conducted a controlled experiment to assess the usefulness of EvoCrash for debugging and bug fixing of two crashes (one from Apache Commons Collections and one from Apache Log4j) with 35 master students. Results show that using a crash-replicating test case generated by EvoCrash may help to locate and fix the defects faster. Also, this study confirmed the usefulness criterion defined by the Chen *et al.* [30] but also found evidence that test cases categorized as not useful can still help developers to fix the bug.

Since JCrashPack also includes two open source industrial and actively maintained applications, it represents an excellent opportunity to confirm the usefulness of EvoCrash in an industrial setting. The key idea is to centralize the information in the issue tracker by providing a test case able to replicate the crash reported in an issue in the same issue (as an attachment for instance). This can be automated using, for instance, a GitHub, GitLab or JIRA plugin that executes EvoCrash when a new issue contains a stack trace. To assess the usefulness of EvoCrash in an industrial setting, we plan to setup a case study [121] with our industrial partners. Hereafter, we outline the main steps of the evaluation protocol using XWiki as subject: (i) select four crashes to fix (two from open issues and two from closed issues) for which EvoCrash could generate a crash reproducing test for frame 3 or higher; (ii) clone the XWiki Git repository in GitHub and open four issues, corresponding to the four crash; (iii) remove the fix for the two fixed issues; (iv) for each issue, append the test case generated by EvoCrash; (v) ask (non-XWiki) developers to fix the issues; and finally, (vi) repeat the same steps without adding the test cases generated by EvoCrash (i.e., omit step iv). We would measure the time required to fix the issues (by asking participants to log that time). For the two previously fixed issues, we will compare the fixes provided by the participants with the fixes provided by XWiki developers. And for the two open issues, we will ask feedback from the XWiki developers through a pull request with the different solutions.

2.8.3 Benchmark Building

JCrashPack is the first benchmark dedicated to crash reproduction. We deliberately made a biased selection when choosing Elasticsearch as the most popular, trending, and frequently-forked project from GitHub. Elasticsearch was among several other highly ranked projects, which addressed other application domains, and thus were interesting to explore. In the future, further effort should extend JCrashPack, possibly by: (i) using a *random selection* methodology for choosing projects; (ii) involving industrial projects from other application domains; and (iii) automatically collecting additional information about the crashes, the stack traces, and the frames to further understand current strengths and limitations of crash reproduction.

Building JCrashPack required substantial manual effort, not just for finding the issues, but also for collecting the right versions of the system itself and its dependencies needed to reproduce the given crash. Since we want it to be representative of current crashes, we need to automate this effort as much as possible: for instance, by mining stack traces from issue tracking systems [122].

Despite the benefits that the evaluation infrastructure could get from the inclusion of

JCrashPack bugs in Defects4J, i.e., the isolation of the bugs to ease replicability of the evaluations [71], we designed JCrashPack as a standalone instead of extending Defects4J. The main reason is that not all bugs in Defects4J manifest as crashes (only 73 out of 395 where selected to be part of JCrashPack). We also believe that the integration of the two benchmarks is not a smooth and easy process. Defects4J requires isolation of the buggy and fixed versions of the source code, as wel as a test case able to expose the bug [71]. However, not all issues were fixed at the time we collected the crashes in JCrashPack. Also, XWiki and Elasticsearch are much larger applications (124,000 NCSS for Elasticsearch, 177,000 NCSS for XWiki distributed in a hierarchy of several thousands of Maven projects) compared to the API libraries considered in Defects4J (63,000 NCSS for JFreeChart). Only building them with their default test suites already raised several issues. For those reasons, isolating the bug, the patch, and the non-regression test cases for such kind of large projects is not a trivial task.

2.9 Future Research Directions For Search-Based Crash Reproduction

From the evaluation and the challenges derived from our manual analysis, we devise the following future research directions. While the same challenge can be addressed in different ways, some requiring technical improvements of EvoCrash and other raising new research directions, we focus the discussion of this section on the latter.

2.9.1 Context Matters

While search-based crash-reproduction with EvoCrash [28, 36] outperformed other approaches based on (i) backward symbolic execution [30], (ii) test case mutation [33], and (iii) model-checking [31], our evaluation shows that the extent to which crashes are reproduced varies. These results indicate the need for taking various types of contexts and properties of software applications into account when devising an approach to a problem. Thus, we show that indeed, rather than seeking a universal approach to search-based crash reproduction, it is important to find out and address challenges specific to various types of application domains (e.g., RESTful microservices vs. enterprise wiki applications) [123].

Furthermore, search-based crash replication boils down to seeking the execution path that will reproduce a given stack trace. As with other search-based testing approaches, it faces challenges about input data generation during the search when the input space is large. Previous research on *mocking* and *seeding* [113, 124] address this problem by using functional mocking and extracting objects and constants from the bytecode.

We believe that taking context into account should go one step further for crash replication. With the development of DevOps [125] and continuous integration and delivery pipelines, there is an increasing amount of available data on the execution of the software. Those data can be used to guide the search more accurately. For instance, by seeding the search using values observed in the execution logs and setting up values for environmental dependencies (databases, external services, etc.).

2.9.2 Stack Trace Preprocessing And Target Frame Selection

Various factors may influence the selection of a target frame in a stack trace. As observed in our evaluation, when not performed cautiously, this selection leads to unsuccessful executions of EvoCrash. For instance, frames targeting code in a private inner class, or irrelevant source code location (like, as we observed, class header or annotation) should be discarded before performing the selection.

Frames targeting code in abstract classes or interfaces (only if the target method is defined in the interface, which is possible from Java 8) may be of some use to find the cause of the crash: for instance, to identify an incorrect subclass implementation [112]. However, as abstract classes and interfaces cannot be directly instantiated, the stack trace generated by EvoCrash can never be exactly the same as the given stack trace. And, as for input arguments and generic type parameters, EvoCrash has no indication on which subclass to pick, making the search difficult. In this case, considering higher level frames (i.e., frames that are lower in the stack trace) may help to pick the right subclass.

Those reasons motivate the need to develop stack trace analysis techniques in order to help the selection of a target frame. This analysis will discard irrelevant and unknown source location frames and provide a visualization to the developer to have a clear view on what are his or her options, for instance by marking stack traces that point to interfaces and abstract classes and recommend him to pick higher level frames.

For a given stack trace, this analysis will also identify frames pointing to a try/catch block. Those stack traces are commonly reported by users to issue tracking systems but cannot (for now) be completely reproduced by EvoCrash. Further investigation on current error handling practices in Java code [126, 127] and how they are reported by users [128] will help us to devise efficient approaches to replicate such stack traces.

2.9.3 Guided Search

Besides usage of contextual information to enhance the generation of test cases during the search process, we also consider to enhance the guidance itself. Search based testing algorithms have several parameters (365 in EvoCrash), like population size, search budget, probability of applying crossover and mutation, etc. As demonstrated by Arcuri and Fraser [111], default parameters values work well on average, but may be fare from optimal for specific frames and stack traces. A better characterization of the stack traces in JCrashPack, trying different parameters, as well as improving the fitness function itself are part of our future work. For instance the fitness function could take other elements into account (e.g., compute a similarity for exception messages). We will also consider multi-objectives search, where, for a given target frame, reproducing each lower frame becomes an objective of the search. We plan to reuse our evaluation infrastructure to compare those different approaches and investigate their different fitness landscapes to gain deeper understanding of the search process for crash reproduction. And eventually devise guidelines on EvoCrash settings to maximize crash reproduction for a given stack trace and its characteristics.

2.9.4 Improving Testability

Finally, as we observed, code complexity was among the major challenges in crash reproduction with EvoCrash. To improve testability, several testability transformation techniques [34, 115, 129–131] have been proposed in the literature so far. Future research may investigate testability transformation techniques and their impact on search-based crash reproduction.

2.10 Threats To Validity

Evaluations of crash reproduction approaches, such as the one we conducted for EvoCrash, come with threats to internal validity, external validity, and reliability. The overarching goal of JCrashPack is to reduce such threats for all evaluations of any crash reproduction tool, by offering a curated set of crashes to conduct such evaluations.

Concerning external validity, we carefully designed JCrashPack so that it offers a mix of small and large systems, as well as of different types of exceptions. Furthermore, it includes open source systems directly developed by industry. Nevertheless, any set is incomplete, which is why we keep JCrashPack open for extension, as discussed in Section 2.8. For example, there still remain several other domains, such as gaming or financial applications, for which there is no representative project in the benchmark.

With respect to internal validity, implementation faults can be a source of confounding factors. These can occur in the tools themselves, such as EvoCrash or EvoSuite, but also in the infrastructure used to actually conduct the experiment. To address the latter, JCrashPack comes with ExRunner, which automates the process of scheduling, executing, monitoring, and reporting crash reproduction attempts.

Concerning reliability, JCrashPack and ExRunner make it easy to repeat experiments, thus making it possible for researchers to independently replicate each others crash reproduction findings.

Besides these threats partially mitigated by JCrashPack, our evaluation of EvoCrash comes with additional threats to (internal and external) validity. This particularly relates to the randomized nature of genetic algorithms, which we addressed by running the evaluations 10 times, and following the guidelines by Arcuri and Briand [110] for analyzing the results. Furthermore, such threats concern the risk of bias during the manual analysis, which we mitigated by using cross-checking: the result of each manual analysis has been validated by at least one other person. In case of disagreement, we asked for a third opinion. Finally, our evaluation includes only one tool: EvoCrash. Previous work showed that EvoCrash performs better than other state-of-the-art crash reproduction tools. Unfortunately, since to the best of our knowledge, no other tool was publicly available, we were not able to confirm that conclusion on the crashes in JCrashPack. We believe that JCrashPack enhances the current state-of-the-practice in crash reproduction research by offering a publicly available benchmark for which other tool providers can report their results.

2.11 Conclusion

Experimental evaluation of crash reproduction research is challenging, due to the computational resources needed by reproduction tools, the difficulty of finding suitable real life crashes, and the intricacies of executing a complex system so that the crash can be reproduced at all.

To remedy this problem, this chapter sets out to create a benchmark of Java crashes, that can be reused for experimental purposes. To that end we propose JCrashPack and ExRunner, a curated benchmark of 200 real life crashes, and a tool to conduct massive experiments on these crashes. This benchmark is publicly available and can be used to compare existing and new tools against each other, as well as to analyze how proposed improvements to existing reproduction techniques actually constitute an improvement.

We applied the state of the art search-based Java crash reproduction tool, EvoCrash, to JCrashPack. Our findings include that the state of the art can reproduce 87 crashes out of 200 in a majority of time, that crash reproduction for industry-strength systems is substantially harder, and that `NullPointerExceptions` are generally easiest to reproduce. Furthermore, we identified 13 challenges that crash reproduction research needs to address to strengthen uptake in practice, as well a future research directions to address those challenges.

JCrashPack can be extended in various ways: by including more crashes from other types of applications; by automating the collection of information about the crashes and stack traces to further understand current strengths and limitations of crash reproduction; as well as automating the collection of the crashes themselves. Furthermore, since executing crash reproduction tools on 200 crashes may be time taking, JCrashPack could be extended to offer prioritization for benchmarks, based on the known theoretical strengths and limitations if the tools. For instance, by ordering crashes based on the cyclomatic complexity of the involved frames to evaluate search-based or symbolic execution-based crash reproduction approaches.

Finally, our future work for EvoCrash itself include improving input data generation by taking information from the execution context and the application (e.g., existing source code and test cases) into account. We also want to deeper our understanding of stack traces in order to be able to recommend target frames to the developers. Finally, we will improve the search process itself by refining the fitness function to improve the guidance through the different frames of the stack trace.

3 Search-based Crash Reproduction using Behavioral Model Seeding

As confirmed by Chapter 2, one of the challenges of search-based crash reproduction is to bring enough information into the test generation process. For instance, complex elements (like strings with a particular format or objects with a complex structure) are hard to initialize without additional information. This can lead to two different issues: first, complex elements take more time to be generated, which can prevent finding a solution within the time budget allocated to the search; and second, elements that require complex initialization procedures (*e.g.*, specific sequences of method calls to set up an object) may prevent starting the search if the search-based approach is unable to create an initial population.

Rojas *et al.* [124] demonstrated that seeding is beneficial for search-based unit test generation. More specifically, by analyzing source code (collecting information that relates to numeric values, string values, and class types) and existing tests (collecting information about the behavior of the objects in the test) and making them available for the search process, the overall coverage of the generated test improves. However, current seeding strategies focus on collecting and reusing values and object states as-is.

In this chapter, we define, implement, and evaluate a new seeding strategy, called behavioral model seeding, which abstracts behavior observed in the source code and test cases using transition systems. The transition systems represent the (observed) usages of the classes and are used during the search to generate objects and sequences of method calls on those objects.

Behavioral model seeding takes advantage of the advances made by the model-based testing community [132] and uses them to enhance search-based software testing. This seeding strategy helps the search process: (i) it provides the possibility of covering the given crash by collecting information from various resources (*e.g.*, source code and existing test cases) to infer a unique transition system; and (ii) it finds the most beneficial seeding candidates, for guiding the crash reproduction search process, by defining a rational procedure for the selection of abstract object behaviors from the inferred models.

We also adapt test seeding, introduced by Rojas *et al.*, for search-based crash reproduction. Contrarily to model seeding, test seeding relies only on the states of the objects observed during the execution of the test to seed a search process. Unlike search-based unit test generation, search-based crash reproduction does not seek to maximize the coverage of the class, but rather generates a specific test case able to reproduce a crash. Since test seeding has only been applied to search-based unit test generation [124], we first evaluate the use of test seeding for crash reproduction. We then compare the results of test seeding with the application of model seeding, which combines information on the objects states coming from the test cases with information collected in the source code, to search-based crash reproduction.

We performed an evaluation on 122 crashes from 6 open-source applications to answer the following research questions:

RQ1 What is the influence of test seeding used during initialization on search-based crash reproduction?

RQ2 What is the influence of behavioral model seeding used during initialization on search-based crash reproduction?

We consider both research questions from the perspective of effectiveness (of initializing the population and reproducing crashes) and efficiency. We also investigate the factors (*e.g.,* the cost of analyzing existing tests) that influence the test and model seeding approaches and gain a better insight into how search-based crash reproduction works and how it can be improved. Generally, our results indicate that behavioral model seeding increases the number of crashes that we can reproduce. More specifically, because of the randomness in the test generation process, we execute the crash replication multiple times and we observe that in the majority of these executions 4 crashes (out of 122) can be replicated; also, this seeding strategy can reproduce 9 crashes, which are not reproducible at all with no seeding, in at least one execution. In addition, this seeding strategy slightly improves the efficiency of the crash reproduction process. Moreover, model seeding enables the search process to start for three additional crashes. In contrast, using test seeding in crash-reproduction leads to a lower crash-reproduction rate and search initialization.

The contributions of this chapter are:

1. An evaluation of test seeding techniques applied to search-based crash reproduction;
2. A novel behavioral model seeding approach for search-based software testing and its application to search-based crash reproduction;
3. An open source implementation of model seeding in the Botsing toolset[1]; and
4. The discussion of our results demonstrating improvements in search-based crash reproduction abilities and contributing to a better understanding of the search-based process. All our results are available in the replication package [63].

The remainder of the chapter is structured as follows: Section 3.1 provides background on search-based crash reproduction, and model-based testing. Section 3.2 describes our behavioral model seeding strategy. Section 3.3 details our implementation, while Section 3.4 explains the evaluation setup. Section 3.5 presents our results. We discuss them and

[1]Available at `https://github.com/STAMP-project/botsing`.

explain threats to our empirical analyses in Section 3.6. Section 3.8 discusses future work and Section 3.9 wraps up the chapter.

3.1 Background And Related Work

Application crashes that happen while the system is operating are usually reported to developer teams through an issue tracking system for debugging purposes [133]. Depending on the amount of information reported from the operation environment, this debugging process may take more or less time. Typically, the first step for the developer is to try to reproduce the crash in his development environment [57]. Various approaches [29–31, 33, 36] automate this process and generate a crash-reproducing test case without requiring human intervention during the generation process. Previous studies [28, 30] show that such test cases are helpful for the developers to debug the application.

For Java programs, the information reported from the operations environment ideally includes a stack trace. For instance, Listing 3.1 presents a stack trace coming from the crash XWIKI-13372.[2] The stack trace indicates the exception thrown (`NullPointerException` here) and the frames, *i.e.,* the stack of method calls at the time of the crash, indexed from 1 (at line 1) to 26 (not shown here).

Various approaches use a stack trace as input to automatically generate a test case reproducing the crash. CONCRASH [29] focuses on reproducing *concurrency* failures that violate thread-safety of a class by iteratively generating test code and looking for a thread interleaving that triggers a concurrency crash. JCHARMING [31, 134] applies model checking and program slicing to generate crash reproducing tests. MuCrash [33] exploits existing test cases written by developers. MuCrash selects test cases covering classes involved in the stack trace and mutates them to reproduce the crash. STAR [30] applies optimized backward symbolic execution to identify preconditions of a target crash and uses this information to generate a crash reproducing test that satisfies the computed preconditions. Finally, RECORE [32] applies a search-based approach to reproduce a crash using both a stack trace and a core dump produced by the system when the crash happened to guide the search.

3.1.1 Search-Based Crash Reproduction

Search-based approaches have been widely used to solve complex, non-linear software engineering problems, which have multiple and sometimes conflicting optimization objectives [135]. Recently, Soltani *et al.* [36] proposed a search-based approach for crash reproduction called EvoCrash. EvoCrash is based on the EvoSuite approach [21, 41] and applies a new *guided genetic algorithm* to generate a test case that reproduces a given crash using a distance metric, similar to the one described by Rossler *et al.* [32], to guide the search. For a given stack trace, the user specifies a target frame relevant to his debugging activities: *i.e.,* the line with a class belonging to his system, from which the stack trace will be reproduced. For instance, applying EvoCrash to the stack trace from Listing 3.1 with a target frame 2 will produce a crash-reproducing test case for the class `BaseStringProperty` that produces a stack trace with the same two first frames.

[2]Described in issue `https://jira.xwiki.org/browse/XWIKI-13372`.

Example 3.1: Stack trace of the XWIKI-13372 crash

```
java.lang.NullPointerException: null
  at com[...]BaseProperty.equals([...]:96)
  at com[...]BaseStringProperty.equals([...]:57)
  at com[...]BaseCollection.equals([...]:614)
  at com[...]BaseObject.equals([...]:235)
  at com[...]XWikiDocument.equalsData([...]:4195)
  [...]
```

Soltani *et al.* [36] demonstrated the usefulness of the tests generated by EvoCrash for debugging and code fixing. They also compared EvoCrash to EvoSuite and showed that EvoCrash reproduces more crashes (85%) than EvoSuite (33%), and, for the crashes reproduced by both approaches, EvoCrash took on average 145 seconds while EvoSuite took on average 391 seconds. These results illustrate the limitations of high-code-coverage-driven test case generation and the need for adequate guidance for crash reproduction.

An overview of the EvoCrash approach is shown at the right part of Figure 3.2 (box 5). The first step of this algorithm, called guided initialization, is used to generate a random population. This random population is a set of random unit tests where a target method call (*i.e.,* the method in the target frame) is injected in each test. During the search, classical guided crossover and guided mutation are applied to the tests in such a way that they ensure that only the tests with a call to the target method are kept in the evolutionary loop. The overall process is guided by a weighted sum fitness function [68], applied to each test t:

$$fitness(t) = 3 \times d_l(t) + 2 \times d_e(t) + d_s(t) \tag{3.1}$$

The terms correspond to the following conditions when executing the test: (i) whether the execution distance from the target line (d_l) is equal to 0.0, in which case, (ii) if the target exception type is thrown (d_e), in which case, (iii) if all frames, from the beginning up until the selected frame, are included in the generated trace (d_s). The overall fitness value for a given test case ranges from 0.0 (crash is fully reproduced) to 6.0 (no test was generated), depending on the conditions it satisfies.

3.1.2 Seeding Strategies For Search-Based Testing

In addition to guided search, a promising technique is *seeding*. Seeding strategies use related knowledge to help the generation process and optimize the fitness of the population [136–138]. We focus here on the usage of the source code and the available tests as primary sources of information for search-based testing. Other approaches, for instance, search for string inputs on the internet [139], or use the existing test corpus [140] to mine relevant formatted string values (*e.g.,* XML or SQL statements).

Seeding from the source code

Three main seeding strategies exploit the source code for search-based testing [124, 136, 141]: (i) constant seeding uses static analysis to collect and reuse constant values appearing in the source code (*e.g.,* constant values appearing in boundary conditions); (ii) dynamic seeding complements constant seeding by using dynamic analysis to collect numerical and string values, observed only during the execution of the software, and reuse them for seeding; and (iii) type seeding is used to determine the object type that should be used as an input argument, based on a static analysis of the source code (*e.g.,* by looking at `instanceof` conditions or generic types for instance).

Seeding from the existing tests

Rojas *et al.* [124] suggest two test seeding strategies, using dynamic analysis on existing test cases: cloning and carving. Dynamic analysis uses code instrumentation to trace the different methods called during an execution, which, compared to static analysis, makes it easier to identify inter-procedural sequences of method calls (for instance, in the context of a class hierarchy). Cloning and carving have been implemented in EvoSuite and can be used for unit test generation.

For cloning, the execution of an existing test case is copied and used as a member of the initial population of a search process. Specifically, after its instrumentation and execution, the test case is reconstructed internally (without the assertions), based on the execution trace of the instrumented test. This internal representation is then used as-is in the initial population. Internal representation of the cloned test cases are stored in a test pool.

For carving, an object is reused during the initialization of the population and mutation of the individuals. In this case, only a subset of an execution trace, containing the creation of a new object and a sequence of methods called on that object, is used to internally build an object on which the methods are called. This object and the subsequent method calls are then inserted as part of a newly created test case (initialization) or in an existing test when a new object is required (mutation). Internal representations of the carved objects[3] are stored in an object pool.

The integration of seeding strategies into crash reproduction is illustrated in Figure 3.2, box 5. As shown, the test cases (respectively objects) to be used by the algorithm are stored in a test case (respectively object) pool, from which they can be used according to user-defined probabilities. For instance, if a test case only contains the creation of a new `LinkedList` (using `new`) that is filled using two `add` method calls, the sequence, corresponding to the execution trace `<new, add, add>`, may be used as-is in the initial population (cloning) or inserted by a mutation into other test cases (carving).

Challenges in seeding strategies

The existing seeding techniques use only one resource to collect information for seeding. However, it is possible that the selected resource does not provide enough information about class usages. For instance, test seeding only uses the carved call sequences from the execution of the existing test cases. If the existing test cases do not cover the behavior of the crash in the interesting classes, this seeding strategy may even misguide the search process. Additionally, if the number of observed call sequences is large, the seeding strategy needs a procedure to prioritize the call sequences for seeding. Using random call sequences as seeds can sometimes misguide the search process. Existing seeding strategies do not currently address these issues.

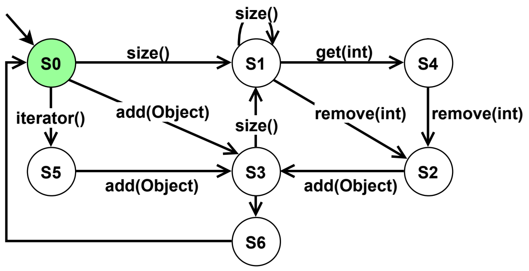


Figure 3.1: Transition system for method call sequences of the class `java.util.LinkedList` derived from Apache commons math source code and test cases.

3.1.3 Behavioral Model-Based Testing

Model-based testing [132] relies on abstract specifications (models) of the system under test to support the generation of relevant (abstract) test cases. *Transition systems* [142] have been used as a fundamental formalism to reason about test case generation and support the definition of formal test selection criteria [143]. Each abstract test case corresponds to a sequence of method calls on one object: *i.e.,* a path in the transition system starting from the initial state and ending in the initial state, a commonly used convention to deal with finite behaviours [144]. Once selected from the model, abstract test cases are concretized (by mapping the transition system's paths to concrete sequences of method calls) into executable test cases to be run on the system. In this chapter, we derive abstract test cases (called abstract object behavior hereafter) and concretize them, producing pieces of code creating objects and invoking methods on such objects. Those pieces of code serve as seeds for search-based crash reproduction.

Figure 3.1 shows an example of a transition system representing the possible sequences of method calls on `java.util.List` objects. Figure 3.1 illustrates usages of methods in `java.util.List` objects, learned from the code and tests, in terms of a transition system, from which *sequences* of methods calls can be derived.

The obtained transition system subsumes the behavior of the sequences used to learn it but also allows for new combinations of those sequences. These behaviors are relevant in the context of seeding as the diversity of the objects induced is useful for the search process. Also, generating invalid behaviors from the new combinations is not a problem here as they are detectable during the search process.

Abstract object behavior selection

The abstract object behaviors are selected from the transition system according to criteria defined by the tester. In the remainder of this paper, we use dissimilarity as selection criteria [145, 146]. Dissimilarity selection, which aims at maximizing the fault detection rate by increasing diversity among test cases, has been shown to be an interesting and scalable alternative to other classical selection criteria [146, 147]. This diversity is measured using a dissimilarity distance (here, 1 - the Jaccard index [148]) between the actions of two abstract object behaviors.

[3]In this paper, we use the term object to refer to a carved object, *i.e.,* an object plus the sequence of methods called on that object.

Model Inference
The model may be manually specified (and in this case will generally focus on specific aspects of the system) [132], or automatically learned from observations of the system [149–154]. In the latter case, the model will be incomplete and only contain the observed behavior of the system [155]. For instance, the sequence `<new, addAll >` is valid for a `java.util.List` object but cannot be derived from the transition system in Figure 3.1 as the `addAll` method call has never been observed. The observed behavior can be obtained via static analysis [156] or dynamically [157]. Model inference may be used for visualization [150, 154], system properties verification [158, 159], or generation [149, 151, 152, 156, 160, 161] and prioritization [144, 162] of test cases.

3.2 Behavioral Model and Test Seeding for Crash Reproduction

The goal of behavioral model seeding (denoted model seeding hereafter) is to abstract the behavior of the software under test using models and use that abstraction during the search. At the unit test level (which is the considered test generation level in this study), each model is a transition system, like in Figure 3.1, and represents possible usages of a class: *i.e.,* possible sequences of method calls observed for objects of that class.

The main steps of our model seeding approach, presented in Figure 3.2, are: the inference of the individual models ③ (described in Section 3.2.1) from the call sequences collected through static analysis ① performed on the application code (described in Section 3.2.1), and dynamic analysis ② of the test cases (described in Section 3.2.1); and for each model, the selection of abstract object behaviors ④, that are concretized into Java objects (described in Section 3.2.2), stored in an object pool from which the guided genetic algorithm ⑤ (described in Section 3.2.3) can randomly pick objects to build test cases during the search process.

3.2.1 Model inference

Call sequences are obtained by using static analysis on the bytecode of the application ① and by instrumenting and executing the existing test cases ②.

We use n-gram inference to build the transition systems used for model seeding. N-gram inference takes a set of sequences of actions as input to produce a transition system where the n^{th} action depends on the $n-1$ previously executed actions.

A large value of n for the n-gram inference would result in wider transition systems with more states and less incoming transitions, representing a more constrained behavior and producing less diverse test cases. In contrast, a small value of n enables better diversity in the behavior allowed by the model (ending up in more diverse abstract object behaviors), requires less observations to reach stability of the model, simplifies the inference, and results in a more compact model [151, 152]. For these reasons, we use 2-gram inference to build our models.

For each class, the model ③ is obtained using a 2-gram inference method using the call sequences of that class.

For instance, in the transition system of Figure 3.1, the action `size()`, executed from state s_3 at step k only depends on the fact that the action `add(Object)` has been executed

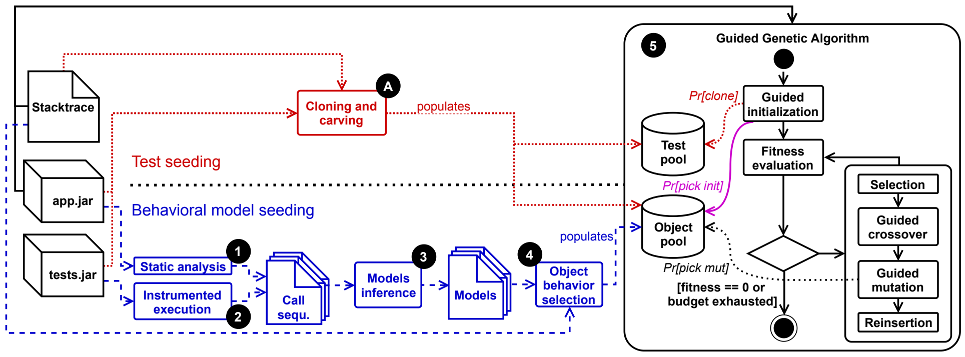


Figure 3.2: General overview of model seeding and test seeding for search-based crash reproduction

at step $k-1$, independently of the fact that there is a step $k-2$ during which the action `iterator()` has been executed.

Calls to constructors are considered as method calls during model inference. However, constructors may not appear in any transition of the model if no constructor call was observed during the collection of the call sequences. This is usually the case when the call sequences used to infer the model have been captured from objects that are parameters or attributes of a class. If an abstract object behavior does not start by a call to a constructor, a constructor is randomly chosen to initialize the object during the concretization.

For one version of the software under test, the model inference is a one time task. Models can then be directly reused for various crash reproductions.

Static analysis of the application

The static analysis is performed on the bytecode of the application. We apply this analysis to all of the available classes in the software under test. In each method of these classes, we build the control flow graph, and for each object of that method, we collect the sequences of method calls on that object. For each object, each path in the control flow graph will correspond to one sequence of method calls. For instance, if the code contains an `if-then-else` statement, the `true` and `false` branches will produce two call sequences. In the case of a loop statement, the `true` branch is considered only once. The static analysis is intraprocedural, meaning that only the calls in the current method are considered. If an object is passed as a parameter of a call to a method that (internally) calls other methods on that object, those internal calls will not appear in the call sequences. This analysis ensures collecting all of the existing relevant call sequences for any internal or external class, which is used in the project.

Dynamic analysis for the test cases

Since the existing manually developed test cases exemplify potential usage scenarios of the software under test, we apply dynamic analysis to collect all of the transpired sequences during the execution of these scenarios. Contrarily to static analysis, which would require an expensive effort and produce imprecise call sequences, dynamic analysis is interprocedural. Meaning that the sequences include calls appearing in the test cases, but also internal calls triggered by the execution of the test case (*e.g.,* if the object is passed as a parameter to a method and methods are internally called on that object). Hence, through dynamic analysis, we gain a more accurate insight into the class usages in these scenarios.

Dynamic analysis of the existing tests is done in a similar way to the carving approach of Rojas *et al.* [124]: instrumentation adds log messages to indicate when a method is called, and the sequences of method calls are collected after execution. In similar fashion to static analysis, we collect call sequences of any observed object (even objects which are not defined in the software under test). The representativeness of the collected sequences depends on the coverage of the existing tests.

3.2.2 Abstract Object Behaviors Selection

Abstract object behaviors are selected from the transition systems and concretized to populate the object pool used during the search. To limit the number of objects in the pool, we only select abstract object behaviors from two categories of models: models of internal classes (*i.e.,* classes belonging to packages of the software under test) and models of dependency classes (*i.e.,* classes belonging to packages of external dependencies) that are involved in the stack trace. Since we do not seek to validate the implementation of the application, the states are ignored during the selection process.

Selection

There exist various criteria to select abstract object behaviors from transition systems [132]. To successfully guide the search, we need to establish a good ratio between exploration (the ability to visit new regions of the search space) and exploitation (the ability to visit the neighborhood of previously visited regions) [163]. The guided genetic operators which are introduced in the EvoCrash approach [36] guarantee the exploitation by focusing the search based on the methods in the stack trace. However, depending on the stack trace, focusing on particular methods may reduce the exploration. Poor exploration decreases the diversity of the generated tests and may trap the search process in local optima.

To improve the exploration ability in the search process, we use dissimilarity as the criterion to select the abstract object behaviors. Compared to classical structural coverage criteria that seek to cover as many parts of the transition system as possible, dissimilarity tries to increase diversity among the test cases by maximizing a distance d (*i.e.,* the Jaccard index [148]):

$$d = 1 - \frac{\{call_{1i} \in b_1\} \cap \{call_{2j} \in b_2\}}{\{call_{1i} \in b_1\} \cup \{call_{2j} \in b_2\}}$$

Where $b_1 =< call_{11}, call_{12}, \ldots >$ and $b_2 =< call_{21}, call_{22}, \ldots >$ are two abstract object behaviors.

Concretization

Each abstract object behavior has to be concretized to an object and method calls before being added to the objects pool. In other words, for each abstract object behavior, if the constructor invocation is not the first action, one constructor is randomly called; and the methods are called on this object in the order specified by the abstract object

Example 3.2: Concretized abstract object behavior for `LinkedList` based on the transition system model of Figure 3.1

```
int[] t = new int[7];
t[3] = -2147483647;
EuclideanIntegerPoint ep = new EuclideanIntegerPoint(t);
LinkedList<[...]> lst = new LinkedList<>();
lst.add(ep);
lst.add(ep);
```

Example 3.3: Stack trace excerpt for MATH-79b

```
    java.lang.NullPointerException
     at ...KMeansPlusPlusClusterer.assignPointsToClusters()
     at ...KMeansPlusPlusClusterer.cluster()
```

behavior with randomly generated parameter values. Due to the randomness, each concretization may be different from the previous one. For each abstract object behavior, n concretizations (default value is $n = 1$ to balance scalability and diversity of the objects in the object pool) are done for each abstract object behavior and saved in the object pool. For instance, Listing 3.2 shows the concretized abstract object behavior `<add(Object), add(Object)>` derived from the transition system model of Figure 3.1. The type of the parameters (`EuclideanIntegerPoint`) is randomly selected during the concretization and created with required parameter values (an integer array here).

3.2.3 Guided Initialization and Guided Mutation

Classes are instantiated to create objects during two main steps of the guided genetic algorithm: guided initialization, where objects are needed to create the initial set of test cases; and guided mutation, where objects may be required as parameters when adding a method call. When no seeding is used, those objects are randomly created (as in the concretization step described in Section 3.2.2) by calling the constructor and random methods.

Finally, to preserve exploration in model seeding, objects are picked from the object pool during guided initialization (resp. guided mutation) according to a user-defined probability $Pr[pick\ init]$ (resp. $Pr[pick\ mut]$), and randomly generated otherwise. In our evaluation, we considered four different values for $Pr[pick\ init] \in \{0.2, 0.5, 0.8, 1.0\}$, to study the effect of model seeding on the initialization of the search process. Furthermore, we fixed the value of $Pr[pick\ mut] = 0.3$, corresponding to the default value of $Pr[pick\ mut]$ for test seeding for classical unit test generation in EvoSuite.

As an example of object picking in action, test case generation with model seeding generated the test case in Listing 3.4 for the second frame of the stack trace from the crash MATH-79b from the Apache commons math project, reported in Listing 3.3. The target method is the last method called in the test (line 10) and throws a `NullPointerException`, reproducing the input stack trace. The first parameter of the method has to be a `Collection<T>` object. In this case, the guided genetic algorithm picked the list object from the object pool (from Listing 3.2) and inserted it in the test case (lines 2 to 7). The algorithm also modified that object (during guided mutation) by invoking an additional method on the object (line 9).

Example 3.4: Test generated for frame 2 of MATH-79b (Listing 3.3)

```
public void testCluster() throws Exception{
 int[] t = new int[7];
 t[3] = (-2147483647);
 EuclideanIntegerPoint ep = new EuclideanIntegerPoint(t);
 LinkedList<[...]> lst = new LinkedList<>();
 lst.add(ep);
 lst.add(ep);
 KMeansPlusPlusClusterer<[...]> kmean = new KMeansPlusPlusClusterer<>(12);
 lst.offerFirst(ep);
 kmean.cluster(lst, 1, (-1357));}
```

3.2.4 Test Seeding

As described in Section 3.1.2, test seeding starts by executing the test cases (Figure 3.2 box Ⓐ) for carving and cloning, and subsequently populating the test and object pools. Like for model seeding, only internal classes and external classes appearing in the stack trace are considered.

For crash reproduction, the test pool is used only during guided initialization to clone test cases that contain the target class, according to a user-defined $Pr[clone]$ probability. If the target method is not called in the cloned test case, the guided initialization also mutates the test case to add a call to the target method. The object pool is used during the guided initialization and guided mutation to pick objects. As described by Rojas *et al.* [124], the properties of using the object pool during initialization ($Pr[pick\ init]$) and mutation ($Pr[pick\ mut]$) are indicated as a single property called `p_object_pool` in test seeding.

3.3 Implementation

Relying on the EvoCrash experience (Chapter 2), we developed Botsing, a framework for crash reproduction with extensibility in mind. Botsing also relies on EvoSuite [6] for the code instrumentation during test generation and execution by using *evosuite-client* as a dependency. Our open-source implementation is available at `https://github.com/STAMP-project/botsing`. The current version of Botsing includes both test seeding and model seeding as features.

3.3.1 Test Seeding

Test seeding relies on the implementation defined by Rojas *et al.* [124] and available in EvoSuite. This implementation requires the user to provide a list of test cases to consider for cloning and carving. In Botsing, we automated this process using the dynamic analysis of the test cases to automatically detect those accessing classes involved in a given stack trace. We also modified the standard guided initialization and guided mutation to preserve the call to the target method during cloning and carving.

3.3.2 Model Seeding

As mentioned in Section 3.2, Botsing uses a combination of static and dynamic analysis to infer models. The static analysis (① in Figure 3.2) uses the reflection mechanisms of EvoSuite to inspect the compiled code of the classes involved in the stack traces, and collect call sequences. The dynamic analysis (② in Figure 3.2) relies on the test seeding mechanism used for cloning that allows inspecting an internal representation of the test cases obtained after their execution and collect call sequences. The resulting call sequences are then used to infer the transition system models of the classes using a 2-gram inference tool called YAMI [144] (③ in Figure 3.2). From the inferred models, we extract a set of dissimilar (based on the Jaccard distance [148]) abstract object behaviors (④ in Figure 3.2). For abstract object behavior extraction, we use the VIBeS [164] model-based testing tool. Abstract object behaviors are then concretized into real objects. For this concretization, we rely on the EvoSuite API.

3.4 Empirical Evaluation

Our evaluation aims to assess the effectiveness of each of the mentioned seeding strategies (model and test seeding) on search-based crash reproduction. For this purpose, first, we evaluate the impact of each seeding strategy on the number of reproduced crashes. Second, we examine if using each of these strategies leads to a faster crash reproduction. Third, we see if each seeding strategy can help the search process to start more often. Finally, we characterize the impacting factors of test and model seeding.

Since the focus of this study is using seeding to enhance the guidance of the search initialization, we examine different probabilities of using the seeded information during the guided initialization in the evaluation of each strategy. Hence, we repeat each execution of test seeding with the following values for $Pr[clone]$: 0.2, 0.5, 0.8, and 1.0. Likewise, we execute each execution of model seeding with the same values for $Pr[pick\ init]$ (which is the only property that we can use for modifying the probability of the object seeding in the initialization of model seeding).

3.4.1 Research Questions

In order to assess the usage of test seeding applied to crash reproduction and our new model seeding approach during the guided initialization, we performed an empirical evaluation to answer the two research questions defined in introduction of this chapter.

RQ1 What is the influence of test seeding used during initialization on search-based crash reproduction? To answer this research question, we compare Botsing executions with test seeding enabled to executions where no additional seeding strategy is used (denoted no seeding hereafter), from their effectiveness to reproduce crashes and start the search process, the factors influencing this effectiveness, and the impact of test seeding on the efficiency. We divide **RQ1** into four sub-research questions:

RQ1.1 Does test seeding help to reproduce more crashes?
RQ1.2 Does test seeding impact the efficiency of the search process?
RQ1.3 Can test seeding help to initialize the search process?
RQ1.4 Which factors in test seeding impact the search process?

RQ2 What is the influence of behavioral model seeding used during initialization on search-based crash reproduction? To answer this question, we compare Botsing executions with model seeding to executions with test seeding and no seeding. We also divide **RQ2** into four sub-research questions:

Table 3.1: Projects used for the evaluation with the number of crashes (**Cr.**), the average number of frames per stack trace ($\overline{\textbf{frm}}$), the average cyclomatic complexity ($\overline{\textbf{CCN}}$), the average number of statements ($\overline{\textbf{NCSS}}$), the average line coverage of the existing test cases ($\overline{\textbf{LC}}$), and the average branch coverage of the existing test cases ($\overline{\textbf{BC}}$).

Application	**Cr.**	$\overline{\textbf{frm}}$	$\overline{\textbf{CCN}}$	$\overline{\textbf{NCSS}}$	$\overline{\textbf{LC}}$	$\overline{\textbf{BC}}$
JFreeChart	2	6.00	2.79	63.01k	67%	59%
Commons-lang	22	2.04	3.28	13.38k	91%	87%
Commons-math	27	3.92	2.43	29.98k	90%	84%
Mockito	12	5.08	1.79	6.06k	97%	93%
Joda-Time	8	3.87	2.11	19.41k	89%	82%
XWiki	51	27.45	1.92	181.68k	73%	71%

RQ2.1 Does behavioral model seeding help to reproduce more crashes compared to no seeding?
RQ2.2 Does behavioral model seeding impact the efficiency of the search process compared to no seeding?
RQ2.3 Can behavioral model seeding help to initialize the search process compared to no seeding?
RQ2.4 Which factors in behavioral model seeding impact the search process?

3.4.2 Setup

Crash selection

In Chapter 2, we introduced a new benchmark, called JCrashPack, containing 200 real-world crashes from seven projects: *JFreeChart*, a framework for creating interactive charts; *Commons-lang*, a library providing additional utilities to the `java.lang` API; *Commons-math*, a library of mathematics and statistics components; *Mockito*, a testing framework for object mocking; *Joda-time*, a library for date and time manipulation; *XWiki*, a popular enterprise wiki management system; and *ElasticSearch*, a distributed RESTful search and analytics engine. We use the same benchmark for the empirical evaluation of model-seeding and test-seeding on crash reproduction.

To use test and model seeding for reproducing the crashes of JCrashPack, first, we needed to apply static and dynamic analysis on different versions of projects in this benchmark. We successfully managed to run static analysis on all of the classes of JCrashPack. On the contrary, we observed that dynamic analysis was not successful in the execution of existing test suites of ElasticSearch. The reason for this failure stemmed from the technical difficulty of running ElasticSearch tests by the EvoSuite test executor. Since both of the seeding strategies need dynamic analysis, we excluded ElasticSearch cases from JCrashPack for this experiment. JCrashPack contains 122 crashes after excluding ElasticSearch cases. Table 3.1 provides more details about our dataset.

We used the selected crashes for the evaluation of no seeding and model seeding. Since test seeding needs existing test cases that are using the target class, we filtered out the crashes which contain only classes without any using tests. Hence, we used only 59 crashes for the evaluation of test seeding. More information about average number of used test classes for test seeding is available in Table 3.2.

Table 3.2: Information about test classes and models used, respectively, for test and model seeding in each project. $\overline{test}$ designate the average number of test classes used for test seeding. Also, $\overline{state}$, $\overline{trans}$, and $\overline{BFS}$ denote the average number of states, transitions, and BFS height of the used models, respectively. The standard deviations of each of these metrics (σ) are located beside them.

Project	$\overline{test}$	σ
chart	29.17	20.01
lang	1.45	2.03
math	1.24	1.37
mockito	0.73	2.15
time	9.24	9.55
xwiki	0.14	1.09

Project	$\overline{state}$	σ	$\overline{trans}$	σ	$\overline{BFS}$	σ
chart	56.67	50.40	157.50	167.86	21.00	17.50
lang	39.69	51.49	117.96	158.07	5.58	7.32
math	14.00	12.46	34.22	40.59	5.20	4.11
mockito	12.18	10.93	21.45	22.70	5.32	3.90
time	63.35	40.85	230.80	167.99	16.10	11.79
xwiki	47.94	90.94	139.15	323.75	11.08	17.04

Model inference

Since the selected crashes for this evaluation are identified before the model inference process, we have applied the dynamic analysis only on the test cases which use the classes involved in the crashes. During the static analysis, we spot all relevant test cases which call the methods of the classes that have appeared in the stack traces of the crashes. Next, we apply dynamic analysis only on the detected relevant test cases. This filtering process helps us to shorten the model inference execution time without losing accuracy in the generated models.

More information about the inferred models is available in Table 3.2.

Configuration parameters

We used a budget of 62,328 fitness evaluations (corresponding on average to 15 minutes of executing Botsing with no seeding on our infrastructure) to avoid side effects on execution time when executing Botsing on different frames in parallel. We also fixed the population size to 100 individuals as suggested by the latest study on search-based crash reproduction [68]. All other configuration parameters are set at their default value [124], and we used the default weighted sum scalarization fitness function (Equation 3.1) from Soltani *et al.* [68].

For test seeding executions, as we described at the beginning of this section, we execute each execution with four values for $Pr[clone]$: 0.2 (which is the default value), 0.5, 0.8, and 1.0. Also, we used the default value of 0.3 for `p_object_pool`.

We also use values 0.2, 0.5, 0.8, and 1.0 for $Pr[pick\ init]$ for model seeding executions. The value of $Pr[pick\ mut]$, which indicates the probability of using seeded information during the mutation, is fixed at 0.3. In addition to model seeding configurations, we fix the size of the selected abstract object behaviors to the size of the individual population in order to ensure that there are enough test cases to initiate the search.

For each frame (951 in total), we executed Botsing for no seeding (*i.e.,* no additional seeding compared to the default parameters of Botsing) and each configuration of model seeding. Since test seeding needs existing test cases which are using the target class, we filtered out the frames that do not have any test for execution of this seeding strategy. Therefore, we executed each configuration of test seeding on the subset of frames (171 in total).

Infrastructure
We used 2 clusters (with 20 CPU-cores, 384 GB memory, and 482 GB hard drive) for our evaluation. For each stack trace, we executed an instance of Botsing for each frame which points to a class of the application. We discarded other frames to avoid generating test cases for external dependencies. We ran Botsing on 951 frames from 122 stack traces for no-seeding and each model-seeding strategy configuration. Also, we ran Botsing with test-seeding on 171 frames from 59 crashes. To address the random nature of the evaluated search approaches, we repeated each execution 30 times. We executed a total of 186,560 independent executions for this study. These executions took about 18 days overall.

3.4.3 Data Analysis Procedure

To check if the search process can reach a better state using seeding strategies, we analyze the status of the search process after executing each of the cases (each run in one frame of a stack trace). We define 5 states:

(i) **not started**, the initial population could not be initialized, and the search did not start;
(ii) **line not reached**, the target line could not be reached;
(iii) **line reached**, the target line has been reached, but the target exception could not be thrown;
(iv) **ex. thrown**, the target line has been reached, and an exception has been thrown but produced a different stack trace; and
(v) **reproduced** the stack trace could be reproduced.

Since we repeat each execution 30 times, we use the majority of outcomes for a frame reproduction result. For instance, if Botsing reproduces a frame in the majority of the 30 runs, we count that frame as a *reproduced.*

To measure the impact of each strategy in the crash reproduction ratio (**RQ1.1** and **RQ2.1**), we use the Odds Ratio (OR) because of the binary distribution of the related data: a search process either reproduces a crash (the generated test replicates the stack trace from the highest frame which is reproduced by at least one of the other searches) or not. Also, we apply Fisher's exact test, with $\alpha = 0.05$ for the Type I error, to evaluate the significance of results.

Moreover, to answer **RQ1.2** and **RQ2.2**, which investigate the efficiency of the different strategies, we compare the number of fitness function evaluations needed by the search to reach crash reproduction. This metric indicates if seeding strategies lead to better initial populations that need fewer iterations to achieve the crash reproducing test. Since efficiency is only relevant for the reproduced cases, we only applied this comparison on the crashes which are reproduced at least once by no seeding or the seeding strategy (test seeding for **RQ1.2** and model seeding for **RQ2.2**). We use the Vargha-Delaney statistic [165] to appraise the effect size between strategies. In this statistic, a value lower than 0.5 for a pair of factors (A, B) gives that A reduces the number of needed fitness function evaluations, and a value higher than 0.5 shows the opposite. Also, we use the Vargha-Delaney magnitude measure to partition the results into three categories having large, medium, and small impact. In addition, to examine the significance of the calculated effect sizes, we use the non-parametric Wilcoxon Rank Sum test, with $\alpha = 0.05$ for Type I error. Moreover, we do note that since the reproduction ratio of each strategy is not 30/30 for each crash, executions that could not reproduce the frame simply reached the maximum allowed budget (62,328).

To measure the impact of each strategy in initializing the first population (**RQ1.3** and **RQ2.3**), we use the same procedure as **RQ1.1** and **RQ2.1** because the distribution of related data in this aspect is binary too (*i.e.,* whether the search process can start the search or not).

For all of the statistical tests in this study, we only use a level of significance $\alpha = 0.05$.

Since the model inference (in model seeding) and test carving (in test seeding) techniques can be applied as one time processes before running any search-based crash reproduction, we do not include them in the efficiency evaluation.

To answer **RQ1.4** and **RQ2.4**, we performed a manual analysis on the logs and crash reproducing test case (if any). We focused our manual analysis on the crash reproduction executions for which the search in one seeding configuration has a significant impact (according to the results of the previous sub-research questions) on (i) *initializing the initial population*, (ii) *crash reproduction*, (iii) or *search process efficiency* compared to no-seeding. Based on our manual analysis, we used a card sorting strategy by assigning keywords to each frame result and grouping those keywords to identify influencing factors.

3.5 Evaluation Results

We present the results of the evaluation and answer the two research questions by comparing each seeding strategy with no-seeding.

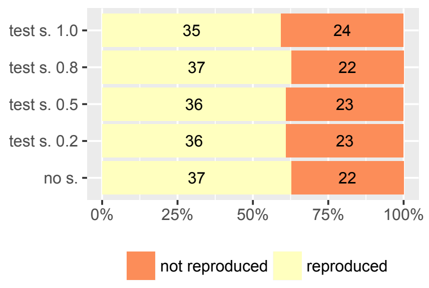
3.5.1 Test Seeding (RQ1)

Crash reproduction effectiveness (RQ1.1)

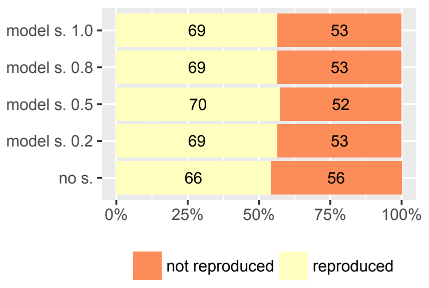
Figure 3.3 demonstrates the comparison of each seeding strategy (left-side of the figure is for test seeding and right-side is for model seeding) with the baseline (no seeding). Figures 3.3a and 3.3b show the overall comparison, while Figures 3.3c and 3.3d illustrate the per project comparison. In each of these figures, the yellow bar shows the number of reproduced crashes in the majority of the 30 executions, and the orange bar shows the non-reproduced crashes.

According to Figure 3.3a, *test s. 0.8* reproduced the same number of crashes. However, the other configurations of test-seeding reproduced fewer crashes in the majority of times. Moreover, according to Figure 3.3c, test seeding reproduces one more crash compared to no seeding. Also, some configurations of test seeding can reproduce one extra crash in XWiki and commons-lang projects. On the contrary, all of the configurations of test seeding missed one and two crashes in JFreeChart and commons-math, respectively. Finally, we cannot see any difference between test seeding and no seeding in the Joda-Time project.

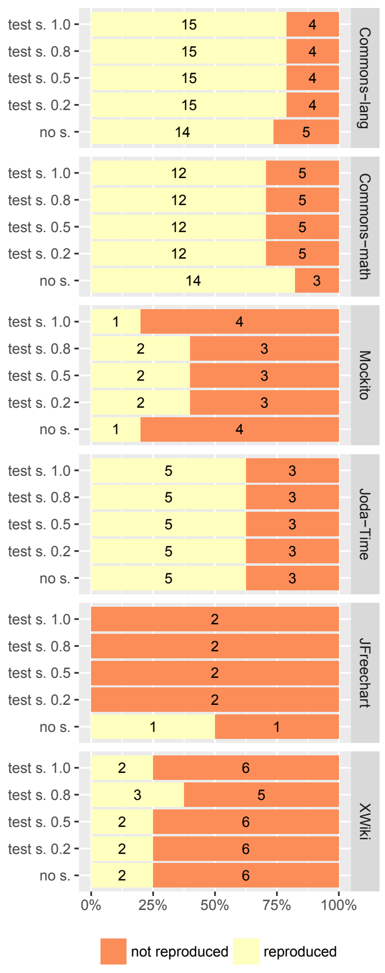
Table 3.4 demonstrates the impact of test-seeding on the crash reproduction ratio compared to no-seeding. It indicates that *test s. 0.2 & 0.5* have a better crash reproduction ratio for one of the crashes, while they perform significantly worse in 4 other crashes compared to no-seeding. The situation is almost the same for the other configurations of test seeding: *test s. 0.8 & 1.0* are significantly better in 2 crashes compared to no-seeding. However, they are significantly worse than no-seeding in 5 other crashes. The other interesting point in



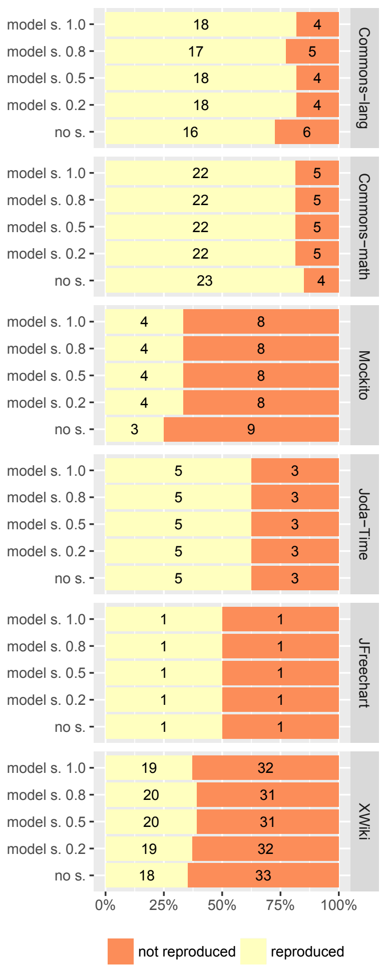
(a) test-seeding vs. no-seeding (for all projects together)



(b) model-seeding vs. no-seeding (for all projects together)



(c) test-seeding vs. no-seeding (per project)



(d) model-seeding vs. no-seeding (per project)

Figure 3.3: Outcomes observed in the majority of the executions for each crash in total and for each application.

this table is the standard deviation crash reproduction ratio. This value is slightly higher for all of the test seeding configurations compared to no seeding. The values of odds ratios and and p-values for crashes with significant difference is available in Table 3.3.

The underlying reasons for the observed results in this section are analyzed in **RQ1.4**.

Crash reproduction efficiency (RQ1.2)
Table 3.5 demonstrates the comparison of test-seeding and no-seeding in the number of needed fitness function evaluations for crash reproduction. The average number of fitness function evaluations increases when using test-seeding. It means that test-seeding is slower than no-seeding on average. *test s. 0.8* has the highest average fitness function evaluations.

Moreover, the standard deviations of both no seeding and test seeding are high values (more than 20k evaluations). This notable variation is explainable due to the nature of search-based approaches. In some executions, the initialized population is closer to the objectives, and the search process can achieve reproduction faster. Similar variations are reported in the JCrashPack empirical evaluation as well (Chapter 2). According to the reported standard deviations, we can see that this value increases for all of the configurations of test seeding compared to no seeding.

Also, the values of the effect sizes indicate that the number of crashes that receive (large or medium) positive impacts from *test s. 0.2 & 0.5* for their reproduction speed is higher than the number of crashes that exhibit a negative (large or medium) influence. However, this is not the case for the other two configurations. In the worst case, *test s. 1.0* is considerably slower than no-seeding (with large effect size) in 13 crashes.

Guided initialization effectiveness (RQ1.3)
Table 3.6 indicates the number of crashes where test-seeding had a significant (p-value < 0.05) impact on the search initialization compared to no-seeding. As we can see in this table, any configuration of test-seeding has a negative impact on the search starting process for 4 or 5 crashes. Additionally, this strategy does not have any significant beneficial impact on this aspect except on one crash in *test s. 0.8*. Also, the standard deviation of the average search initialization ratios, in all of the configurations of test seeding, is increased compared to no seeding. For instance, this value for *test s. 0.8* is about three times more than no seeding.

Influencing factors (RQ1.4)
To finding the influencing factors in test seeding, we manually analyzed the cases which cause significant differences, in various aspects, between no-seeding and test-seeding. From our manual analysis, we identified 3 factors of the test seeding process that influence the search: (i) **Crash-Test Proximity**, (ii) **Crash-Object Proximity**, and (iii) **Test Execution Cost**.

Crash-Test Proximity For the first factor, we observe that cloning existing test cases in the initial population leads to the reproduction of new crashes when the cloned tests include elements which are close to the crash reproducing test. For instance, all of the configurations of test seeding are capable of reproducing the crash LANG 6b, while no-seeding cannot reproduce it. For reproducing this crash, Botsing needs to generate a string of a specific format, and this format is available in the existing test cases, which are seeded to the search process.

However, manually developed tests are not always helpful for crash reproduction. According to the results of Table 3.5, *test s. 1.0*, which always clones test cases, is considerably and largely slower than no-seeding in 13 crashes. In these cases, cloning all of the test cases to form the initial population can prevent the search process from reaching the crash reproducing test. As an example, Botsing needs to generate a simple test case, which calls the target method with an empty string and null object, to reproduce crash LANG-12b. But, *test s. 1.0* clones tests which use the software under test in different ways. To summarize, the overall quality of results of our test seeding solution is highly dependent on the quality of the existing test cases in terms of factors like the distance of existing test cases to the scenario(s) in which the crash occurs and the variety of input data.

Crash-Object Proximity For the second factor, we observe that (despite the fixed value of $Pr[pick\ mut]$ for test seeding), the objects with call sequences carved from the existing tests and stored in the object pool can help during the search depending on their diversity and their distance from the call sequences that we need for reproducing the given crash. For instance, for crash MATH-4b, Botsing needs to initialize a `List` object with at least two elements before calling the target method in order to reproduce the crash. In test-seeding, such an object had been carved from the existing tests and allowed test seeding to reproduce the crash faster. Also, test-seeding can replicate this crash more frequently: the number of successfully replicated executions, in 30 runs, is higher with test-seeding.

In contrast, the carved objects can misguide the search process for some crashes which need another kind of call sequence. For instance, in crash MOCKITO-9b, Botsing cannot inject the target method into the generated test because the carved objects do not have the proper state to instantiate the input parameters of the target method.

In summary, if the involved classes in a given crash are well-tested (the existing tests contain all of the usage scenarios of these classes), we have more chances to reproduce by utilizing test-seeding.

Test Execution Cost The third factor points to the challenge of executing the existing test cases for seeding. The related tests for some crashes are either expensive (time/resource consuming) or challenging (due to the security issues) to execute. Hence, the EvoSuite test executor, which is used by Botsing, cannot carve all of them.

As an example of expensive execution, the EvoSuite test executor spends more than 1 hour during the execution of the related test cases for replicating frame 2 of crash Math-1b.

Also, as an example for security issues, the EvoSuite test executor is not successful in running some of the existing tests. It throws an exception during this task. For instance, this executor throws `java.lang.SecurityException` during the execution of the existing test cases for CHART-4b, and it cannot carve any object for seeding.

In some cases, test-seeding faces the mentioned problems during the execution of all of the existing test cases for a crash. If test seeding cannot carve any object from existing tests, there will be no useful call sequence in the object pool to seed during the search process. Hence, although the project contains some potentially valuable test scenarios for reproducing the given crash, there is no difference between no seeding and test seeding in these cases.

Summary (RQ1)
Test seeding (for any configuration) loses against no-seeding in the search initialization because some of the related test cases of crashes are expensive or even impossible to execute. Also, we observe in the manual analysis that the lack of generality in the existing test cases prevents the crash reproduction search process initialization. In these cases, the carved objects from the existing tests mismatch the search process in the target method injection. Moreover, this seeding strategy can outperform no seeding in the crash reproduction and search efficiency for some cases (*e.g.,* LANG 6b), thanks to the call sequences carved from the existing tests. However, these carved call sequences can be detrimental to the search process in some cases, if the carved call sequences do not contain beneficial knowledge about crash reproduction, overusing them can misguide the search process.

3.5.2 Behavioral Model Seeding (RQ2)

Crash reproduction effectiveness (RQ2.1)
Figure 3.3b draws a comparison between model-seeding and no-seeding in the crash reproduction ratio according to the results of the evaluation on all of the 122 crashes. As mentioned in Section 3.4.2, since model seeding collects call sequences both from source code and existing tests, it can be applied to all of the crashes (even the crashes that do not have any helpful test). As depicted in this Figure, all of the configurations of model-seeding reproduce more crashes compared to no-seeding in the majority of runs. We observe that *model s. 0.2 & 0.5 & 1.0* reproduce 3 more crashes than no-seeding. In addition, in the best performance of model-seeding, *model s. 0.8* reproduces 70 out of 122 crashes (6% more than no-seeding).

Figure 3.3d categorizes the results of Figure 3.3b per application. As we can see in this figure, model seeding replicates more crashes for XWiki, commons-lang, and Mockito. However, no-seeding reproduces one crash more than model-seeding for commons-math. For the other projects, the number of reproduced crashes does not change between no-seeding and different configurations of model-seeding.

We also check how many crashes can be reproduced at least once with model seeding, but not with no seeding. In total, model-seeding configurations reproduce nine new crashes that no-seeding cannot reproduce.

Table 3.4 indicates the impact of model-seeding on the crash reproduction ratio. As we can see in this table, *model s. 0.2* has a significantly better crash reproduction ratio in 3 crashes. Also, other configurations of model-seeding are significantly better than no seeding in 4 crashes. This improvement is achieved by model-seeding, while 2 out of 4 configurations of model-seeding have a significant unfavorable impact on only one crash. The values of odds ratios and and p-values for crashes with significant difference is available in Table 3.3.

Crash reproduction efficiency (RQ2.2)
Table 3.7 compares the number of the needed fitness function evaluations for crash reproduction in model-seeding and no-seeding. As we can see in this table, the average effort

is reduced by using model-seeding. On average *mode s. 1.0* achieves the fastest crash reproduction.

According to this table, and in contrast to test-seeding, model-seeding's efficiency is slightly positive. The number of crashes that model-seeding has a positive large or medium influence (as Vargha-Delaney measures are lower than 0.5) on varies between 3 to 5. Also, model-seeding has a large adverse effect size (as Vargha-Delaney measures are higher than 0.5) on one crash, while this number is higher for test-seeding (*e.g.,* 13 for *test s. 1.0*). Table 3.7 does not include the cost of model generation for seeding as mentioned in our experimental setup. In our case, model generation was not a burden and is performed only once per case study. We will cover this point in more detail in Section 3.6.

Guided initialization effectiveness (RQ2.3)
Table 3.6 provides a comparison between model-seeding and no-seeding in the search initialization ratio. As shown in this Table, *model s. 0.2 & 0.5* significantly outperform no seeding in starting the search process for two crashes. This number increases to 3 for *model s. 0.8 & 1.0.* In contrast to test-seeding, most of the configurations of model-seeding do not have any significant negative impact on the search initialization (only *model s. 0.2* is significantly worse than no-seeding in one crash). Notably, the average search initialization ratios for all of the model seeding configurations are slightly higher than no seeding. In the best case for model seeding, *model s. 0.8 & 1.0* is 30/30 runs, and the standard deviations for these two configurations are 0 or close to 0.

Influencing factors (RQ2.4)
We have manually analyzed the crashes which lead to significant differences between different configurations of model seeding and no seeding. In doing so, we have identified 4 influencing factors in model-seeding on search-based crash reproduction, namely: (i) using **Call sequence dissimilarity** for guided initialization, (ii) having **Information source diversity** to infer the behavioral models, (iii) **Sequence priority** for seeding by focusing on the classes involved in the stack trace, and (iv) having **Fixed size abstract object behavior selection** from usage models.

Call sequence dissimilarity Using dissimilar call sequences to populate the object pool in model seeding seems particularly useful for search efficiency compared to test seeding. In particular, if the number of test cases is large, model seeding enables (re)capturing the behavior of those tests in the model and regenerate a smaller set of call sequences which maximize diversity, augmenting the probability to have more diverse objects used during the initialization. For instance, Botsing with model-seeding is statistically more efficient than other strategies for replicating crash XWIKI-13141. Through our manual analysis we observed that model-seeding could replicate crash XWIKI-13141 in the initial population in 100% of cases, while the other seeding strategies replicate it after a couple of iterations. In this case, despite the large size of the target class behavioral model (35 transitions and 17 states), the diversity of the selected abstract object behaviors guarantees that Botsing seeds the reproducing test cases to the initial population.

Information source diversity Having multiple sources to infer the model from helps to select diversified call sequences compared to test seeding. For instance, the sixth frame of the crash XWIKI-14556 points to a class called `HqlQueryExecutor`. No seeding cannot replicate this crash because it does not have any guidance from existing solutions. Also, since the test carver could not detect any existing test which is using the related classes, this seeding strategy does not have any knowledge to achieve reproduction. In contrast, the knowledge required for reproducing this crash is available in the source code, and model-seeding learned it from static analysis of this resource. Hence, this seeding strategy is successful in accomplishing crash reproduction.

Sequence priority By prioritizing classes involved in the stack trace for the abstract object behaviors selection, the object pool contains more objects likely to help to reproduce the crash. For instance, for the 10th frame of the crash LANG-9b, model seeding could achieve reproduction in the majority of runs, compared to 0 for test and no seeding, by using the class `FastDateParser` appearing in the stack trace.

Fixed size abstract object behavior selection The last factor points to the fixed number of the generated abstract object behaviors from each model. In some cases, we observed that model-seeding was not successful in crash reproduction because the usage models of the related classes were large, and it was impossible to cover all of the paths with 100 abstract object behaviors. As such, this seeding strategy missed the useful dissimilar paths in the model. As an example, model-seeding was not successful in replicating crash XWIKI-8281 (which is replicated by no-seeding and test-seeding). In this crash, the unfavorable generated abstract object behaviors for the target class misguided the search process in model seeding.

Summary (RQ2)
Model seeding achieves a better search initialization ratio compared to no seeding. With respect to the best achievement of model seeding (*model s. 0.8 & 1.0*), they decrease the number of not started searches in 3 crashes. Moreover, compared to no seeding, model seeding increases the number of crashes that can be reproduced in the majority of times to 6%. It also reproduces 9 (out of 122) extra crashes that are unreproducible with no-seeding. In addition, model seeding improves the efficiency of search-based crash reproduction compared to no seeding. It takes, on average, less fitness function evaluations. Also, model seeding delivers more positive significant impact on the efficiency of the search process compared to no seeding.

In general, model seeding outperforms no seeding in all of the aspects of search-based crash reproduction. According to the manual analysis that we have performed in this study, model seeding achieves this performance thanks to multiple factors: Call sequence dissimilarity, Information source diversity, and Sequence priority. Nevertheless, we observe a negative impacting factor in model seeding, as well. This factor is the fixed size abstract object behavior selection.

Table 3.3: Odds ratios of model/test seeding configurations vs. no seeding in crash reproduction ratio. This table only shows the crashes, which reveal statistically significant differences (p-value < 0.05). An Odds ratio value higher than 1.0 gives that the seeding strategy is better than no seeding, and a value lower than 1.0 shows the opposite.

Conf.	**Crash**	**Odds Ratio (p-value)**
test s. 0.5	LANG-6b	Inf (2.37e-02)
	MATH-1b	0.00 (1.69e-17)
	MATH-61b	0.00 (1.69e-17)
	CHART-4b	0.00 (1.69e-17)
	TIME-20b	0.00 (1.94e-03)
	TIME-10b	207.79 (2.36e-12)
	TIME-5b	3.52 (3.52e-02)
test s. 0.8	LANG-6b	Inf (1.94e-03)
	MATH-1b	0.00 (1.69e-17)
	MATH-61b	0.00 (1.69e-17)
	CHART-4b	0.00 (1.69e-17)
	TIME-20b	0.00 (4.64e-05)
	TIME-10b	Inf (9.23e-14)
	TIME-7b	0.00 (6.19e-07)
test s. 1.0	LANG-51b	0.21 (8.21e-03)
	LANG-6b	Inf (4.64e-05)
	MATH-1b	0.00 (1.69e-17)
	MATH-61b	0.00 (1.69e-17)
	CHART-4b	0.00 (1.69e-17)
	TIME-20b	0.00 (5.83e-06)
	TIME-10b	69.79 (2.82e-10)
test s. 0.2	MATH-1b	0.00 (1.69e-17)
	MATH-61b	0.00 (1.69e-17)
	CHART-4b	0.00 (1.69e-17)
	TIME-20b	0.00 (3.19e-04)
	TIME-10b	Inf (9.23e-14)
	TIME-7b	0.00 (1.05e-02)

Conf.	**Crash**	**Odds Ratio (p-value)**
model s. 0.2	LANG-9b	Inf (1.94e-03)
	LANG-51b	0.17 (3.33e-03)
	MOCKITO-10b	Inf (1.43e-08)
	XWIKI-13141	13.95 (5.58e-03)
model s. 0.5	LANG-9b	Inf (2.37e-02)
	MOCKITO-10b	Inf (1.87e-07)
	XWIKI-13141	Inf (7.97e-04)
	XWIKI-14152	6.66 (7.41e-03)
model s. 0.8	LANG-9b	Inf (1.94e-03)
	LANG-51b	0.29 (3.70e-02)
	MOCKITO-10b	Inf (8.27e-10)
	XWIKI-13141	Inf (7.97e-04)
	XWIKI-14152	11.24 (2.51e-04)
model s. 1.0	LANG-9b	Inf (1.94e-03)
	MOCKITO-10b	Inf (5.34e-08)
	XWIKI-13141	13.95 (5.58e-03)
	XWIKI-14152	32.80 (5.62e-08)

Table 3.4: Evaluation results for comparing seeding strategies (test and model seeding) and no-seeding in crash reproduction. $\overline{\text{ratio}}$ and σ designate average crash reproduction ratio and standard deviation, respectively. The numbers in the comparison only count the statistically significant cases.

Conf.	Reproduction		Comparison to no s.		**Conf.**	Reproduction		Comparison to no s.	
	$\overline{\text{ratio}}$	σ	better	worse		$\overline{\text{ratio}}$	σ	better	worse
test s. 1.0	23.7	11.01	2	5	model s. 1.0	22.0	11.58	4	0
test s. 0.8	23.4	10.74	2	5	model s. 0.8	21.9	11.92	4	1
test s. 0.5	23.8	10.76	1	4	model s. 0.5	21.8	11.86	4	0
test s. 0.2	23.5	10.93	1	4	model s. 0.2	21.6	12.00	3	1
no s.	25.4	9.65	-	-	no s.	21.3	12.32	-	-

Table 3.5: Evaluation results for comparing test-seeding and no-seeding in the number of fitness evaluations $\overline{\text{evaluations}}$ and σ designate average fitness function evaluations needed for crash reproduction and standard deviation, respectively. The numbers in the comparison only count the statistically significant cases.

Conf.	Fitness		Comparison to no s.					
			large		medium		small	
	$\overline{\text{evaluations}}$	σ	< 0.5	> 0.5	< 0.5	> 0.5	< 0.5	> 0.5
no s.	10,467	22,368.13	-	-	-	-	-	-
test s. 0.2	14,089	25,464	4	3	1	1	2	-
test s. 0.5	13,366	25,043	5	3	1	-	2	1
test s. 0.8	14,254	25,496	3	4	1	5	1	3
test s. 1.0	13,856	25,097	3	13	4	3	1	3

Table 3.6: Evaluation results for comparing seeding strategies (test and model seeding) and no-seeding in search initialization. $\overline{\text{ratio}}$ and σ designate average successful search initialization ratio and standard deviation, respectively. The numbers in the comparison only count the statistically significant cases.

Conf.	Search started		Comparison to no s.		**Conf.**	Search started		Comparison to no s.	
	$\overline{\text{ratio}}$	σ	better	worse		$\overline{\text{ratio}}$	σ	better	worse
test s. 1.0	26.9	9.22	0	5	model s. 1.0	30.0	0.28	3	0
test s. 0.8	27.9	7.67	1	4	model s. 0.8	30.0	0.00	3	0
test s. 0.5	26.9	9.22	0	5	model s. 0.5	29.7	2.75	2	0
test s. 0.2	27.4	8.49	0	4	model s. 0.2	29.5	3.87	2	1
no s.	29.5	3.94	-	-	no s.	29.2	4.72	-	-

Table 3.7: Evaluation results for comparing model-seeding and no-seeding in the number of fitness evaluations $\overline{\text{evaluations}}$ and σ designate average fitness function evaluations needed for crash reproduction and standard deviation, respectively. The numbers in the comparison only count the statistically significant cases.

Conf.	Fitness		Comparison to no s.					
			large		medium		small	
	$\overline{\text{evaluations}}$	σ	< 0.5	> 0.5	< 0.5	> 0.5	< 0.5	> 0.5
no s.	18,713.1	28,023.93	-	-	-	-	-	-
model s. 0.2	18,016.1	27,699.61	2	1	1	1	2	1
model s. 0.5	17,646.9	27,463.02	2	1	2	-	2	1
model s. 0.8	17,564.5	27,400.27	3	1	2	-	1	3
model s. 1.0	17,268.8	27,190.73	3	1	2	-	1	2

3.6 Discussion

3.6.1 Practical Implications

Model derivation costs Generating seeds comes with a cost. For our worst case, XWIKI-13916, we collected 286K call sequences from static and dynamic analysis and generated 7,880 models from which we selected 6K abstract object behaviors. We repeated this process 10 times and found the average time for call sequence collection to be 14.2 seconds; model inference took 77.8 seconds; and abstract object behavior selection and concretization took 51.5 seconds. We do note however that the model inference is a one-time process that could be done offline (in a continuous integration environment). After the initial inference of models, any search process can utilize model seeding. To summarize, the total initial overhead is ~ 2.5 minutes, and the total nominal overhead is around ~ 1.25 minute. We argue that **the overhead of model seeding is affordable giving its increased effectiveness.** The initial model inference can also be incremental, to avoid complete regeneration for each update of the code, or limited to subparts of the application (like in our evaluation where we only applied static and dynamic analysis for classes involved in the stack trace). Similarly, abstract object behavior selection and concretization may be prioritized to use only a subset of the classes and their related model. In our current work, this prioritization is based on the content of the stack traces. Other prioritization heuristics, based for instance on the size of the model (reflecting the complexity of the behavior), is part of our future work.

Applicability and effectiveness Generally, test seeding alone does not make crash reproduction more effective. Actually, test seeding has a more negative impact on the search-based crash reproduction. Test seeding only uses dynamic analysis, which entails that it collects more accurate information from the potential usage scenarios of the software under test; it also means that this strategy collects more limited information for seeding. If these limited amounts of call sequences differ from the call sequences needed to reproduce the crash scenario, test seeding can misguide the crash reproduction search process.

In contrast to test-seeding, we observe that model seeding always performs better than no seeding with different configurations. As such, we observe that **model seeding can reproduce more crashes than other strategies**. Also, since model seeding also exploits test cases, thereby subsuming test seeding regarding the observed behavior of the application that is reused during the search, greater performance can be attributed to the analysis of the source code translated in the model.

In our experiments, various configurations of model seeding reproduced 8 new crashes that neither test seeding nor no seeding strategies could reproduce. Additionally, **only model seeding could reproduce stack traces with more than seven frames** (*e.g.*, LANG-9b). Still, model seeding missed the reproduction of one crash which is reproduced by no seeding. Despite the achieved improvements by model seeding, this seeding strategy could not outperform no-seeding dramatically (crash reproduction improved by 6%). To better understand the reasons for the results, we manually analyzed the logs of Botsing executions on the crashes for which model seeding could not show any improvements. Through this investigation, we noticed that the generated usage models in these cases are limited and they do not contain the beneficial call sequences for covering the particular path that we need for crash reproduction. The average size of the generated model in this study is 7 states and 14 transitions. We believe that by collecting more call sequences from different sources (*i.e.,* log files), model seeding can increase the number of crash reproductions.

We also observed two crashes that all of the test seeding configurations could reproduce them significantly more often compared to all of the configurations of model seeding: LANG-6b and TIME-5b. We manually analyzed the crash reproduction process in these two crashes to understand the reason for test seeding outperforming model seeding. In the former crash, test seeding is the only seeding strategy that can reproduce the crash because of the **Crash-Test Proximity** (explained in section 3.5.1 as an example of this factor). In the latter crash, we observed that the size of the inferred model for the target class is big (it has 99 states and more than 300 transitions).

We witnessed that the size of the generated abstract behaviors set is commensurate to the size of the inferred model. If we have a small model, and we choose too many abstract behaviors, we will get similar abstract behaviors that misguide the search process. The mutation operator may counter this negative impact during the search by potentially adding the missing method calls. In contrast, if we chose a small set of abstract behaviors from a behavioral model with a large size, we will miss the chance of using all of the potentials of the model for increasing the chance of crash reproduction by the search process.

Extendability The usage models can be inferred from any resource providing call sequences. In this study, we used the call sequences derived from the source code and existing test cases. However, we can extend the models with extra resources (*e.g.,* execution logs). Also, the abstract object behavior selection approach can be adapted according to the problem. In this study, we used the dissimilarity strategy to increase the diversity of the generated tests. Moreover, model seeding makes a distinction between using the object pool during guided initialization and guided mutation (as shown in Figure 3.2). This distinction enables us to study the influence of seeding during the different steps of the algorithm independently.

3.6.2 Model Seeding Configuration

Model seeding can be configured with different $Pr[pick\ init]$ and $Pr[pick\ mut]$ probabilities. Like many other parameters in search-based test case generation [166], the values of those parameters could influence our results. Although a full investigation of the effect of $Pr[pick\ init]$ and $Pr[pick\ mut]$ on the search process is beyond the scope of this paper, we set up a small experiment on a subset of crashes (10 crash in total) with 15 new configurations, each one run 10 times.

Tables 3.8 and 3.9 presents the configurations used for $Pr[pick\ init]$ and $Pr[pick\ mut]$ with, for each one, the crash reproduction effectiveness (Table 3.8), and the crash reproduction efficiency (Table 3.9). In general, we observe that changing the probability of picking an object during guided initialization ($Pr[pick\ init]$) has an impact on the search and leads to more reproduced crashes with a lower number of fitness evaluations. This confirms the results presented in Section 3.5. Changing the probability of picking an object during mutation ($Pr[pick\ mut]$) does not seem to have a large impact on the search.

Table 3.8: Evaluation results for comparing different configurations of model seeding in crash reproduction. $\overline{\text{rate}}$ and σ designate average crash reproduction rate and standard deviation, respectively. The numbers in the comparison only count the statistically significant cases.

Conf.	Reproduction		Comparison to other conf.	
	$\overline{\text{rate}}$	σ	better	worse
Pr[init]=0.0 Pr[mut]=0.3	18.8	13.81	0	11
Pr[init]=0.0 Pr[mut]=0.6	19.0	13.64	0	10
Pr[init]=0.0 Pr[mut]=0.9	19.0	13.55	0	13
Pr[init]=0.2 Pr[mut]=0.0	20.4	12.42	2	2
Pr[init]=0.2 Pr[mut]=0.3	19.6	12.87	0	7
Pr[init]=0.2 Pr[mut]=0.6	19.8	12.88	1	5
Pr[init]=0.2 Pr[mut]=0.9	19.4	13.15	0	7
Pr[init]=0.5 Pr[mut]=0.0	20.8	12.17	3	1
Pr[init]=0.5 Pr[mut]=0.3	20.6	12.29	3	2
Pr[init]=0.5 Pr[mut]=0.6	19.4	13.24	0	7
Pr[init]=0.5 Pr[mut]=0.9	20.0	12.58	1	5
Pr[init]=0.8 Pr[mut]=0.0	21.8	11.46	8	0
Pr[init]=0.8 Pr[mut]=0.3	21.6	11.53	6	0
Pr[init]=0.8 Pr[mut]=0.6	21.8	11.77	8	0
Pr[init]=0.8 Pr[mut]=0.9	20.8	11.96	3	2
Pr[init]=1.0 Pr[mut]=0.0	21.6	11.53	6	0
Pr[init]=1.0 Pr[mut]=0.3	23.0	11.31	12	0
Pr[init]=1.0 Pr[mut]=0.6	21.6	11.82	8	0
Pr[init]=1.0 Pr[mut]=0.9	22.6	11.30	11	0

A full investigation of the effects of $Pr[pick\ init]$ and $Pr[pick\ mut]$ on the search process is part of our future work.

Table 3.9: Evaluation results for comparing different configurations of model seeding in the number of fitness evaluations $\overline{rate}$ and σ designate average fitness function evaluations needed for crash reproduction and standard deviation, respectively. The numbers in the comparison only count the statistically significant cases.

Conf.	Fitness		Comparison to other configurations					
			large		medium		small	
	$\overline{\text{evaluations}}$	σ	< 0.5	> 0.5	< 0.5	> 0.5	< 0.5	> 0.5
Pr[init]=0.0 Pr[mut]=0.3	23,456.5	30,105.20	-	5	-	3	-	3
Pr[init]=0.0 Pr[mut]=0.6	23,066.3	29,976.23	-	2	-	5	-	3
Pr[init]=0.0 Pr[mut]=0.9	23,030.9	30,001.82	-	7	-	4	1	2
Pr[init]=0.2 Pr[mut]=0.0	20,179.0	29,012.80	-	-	1	2	1	-
Pr[init]=0.2 Pr[mut]=0.3	21,803.0	29,620.34	-	2	2	5	-	1
Pr[init]=0.2 Pr[mut]=0.6	21,448.9	29,441.74	-	2	-	3	1	2
Pr[init]=0.2 Pr[mut]=0.9	22,214.6	29,752.12	-	2	2	5	1	1
Pr[init]=0.5 Pr[mut]=0.0	19,371.3	28,668.58	-	-	2	1	3	-
Pr[init]=0.5 Pr[mut]=0.3	19,766.8	28,849.00	-	-	1	2	2	-
Pr[init]=0.5 Pr[mut]=0.6	22,245.2	29,729.80	-	-	-	4	-	3
Pr[init]=0.5 Pr[mut]=0.9	21,030.0	29,302.03	-	2	-	3	1	2
Pr[init]=0.8 Pr[mut]=0.0	17,329.0	27,693.98	2	-	6	-	-	1
Pr[init]=0.8 Pr[mut]=0.3	17,710.5	27,919.28	1	-	4	-	3	-
Pr[init]=0.8 Pr[mut]=0.6	17,327.0	27,694.60	2	-	6	-	-	-
Pr[init]=0.8 Pr[mut]=0.9	19,383.3	28,659.38	-	-	1	1	2	2
Pr[init]=1.0 Pr[mut]=0.0	17,730.5	27,906.92	1	-	4	-	3	1
Pr[init]=1.0 Pr[mut]=0.3	14,863.9	26,275.53	7	-	3	-	-	-
Pr[init]=1.0 Pr[mut]=0.6	17,692.5	27,930.17	2	-	5	-	1	-
Pr[init]=1.0 Pr[mut]=0.9	15,656.9	26,798.15	7	-	5	-	1	-

3.7 Threats To Validity

Internal Validity

We selected 122 crashes from 5 open source projects: 33 crashes have previously been studied [68] and we added additional crashes from XWiki and Defects4J (see Section 3.4). Since we focused on the effect of seeding during guided initialization, we fixed the $Pr[pick\ mut]$ value (which, due to the current implementation of Botsing, is also used as $Pr[pick\ init]$ value in test seeding) to 0.3, the default value used in EvoSuite for unit test generation. The effect of this value for crash reproduction, as well as the usage of test and model seeding in guided initialization, is part of our future work. We cannot guarantee that our extension of Botsing is free of defects. We mitigated this threat by testing the extension and manually analyzing a sample of the results. Finally, each frame has been run 30 times for each seeding configuration to take randomness into account and we derive our conclusions based on standard statistical tests [110, 167].

External validity

We cannot guarantee that our results are generalizable to all crashes. However, we used JCrashPack, which is the most recent benchmark for Java crash reproduction. This benchmark is assembled carefully from seven Java projects and contains 200 real-life crashes. Since the EvoSuite test executor is unsuccessful in running the existing test cases of one of the seven projects in JCrashPack (ElasticSearch), thereby test seeding and dynamic analysis of model-seeding are not applicable on crashes of this project, we excluded ElasticSearch crashes from JCrashPack. The diversity of crashes in this benchmark also suggests mitigation of this threat.

Verifiability
A replication package of our empirical evaluation is available at `https://github.com/STAMP-project/ExRunner-bash/tree/master`. The complete results and analysis scripts are also provided as a dataset in Zenodo [63] for long-term storage. Our extension of Botsing is released under a Apache-2.0 license and available at `https://github.com/STAMP-project/botsing`.

3.8 Future Work

We observed that one of the advantageous factors in model seeding, which helps the search process to reproduce more crashes, consists in using more multiple resources for collecting the call sequences. Further diversification of sources is worth considering. In our future work, we will consider other sources of information, like logs of the running environment, to collect relevant call sequences and additional information about the actual usage of the application.

Also, collecting additional information from the log files would enable using full-fledged behavioral usage models (*i.e.,* a transition system with probabilities on their transitions quantifying the actual usage of the application) to select and prioritize abstract object behaviors according to that usage as it is suggested by statistical testing approaches [144]. For instance, we can put a high priority for the most uncommon observed call sequences for the abstract object behavior selection. We observed that selecting the most dissimilar paths in model-seeding helps the search process through crash reproduction. However, there is no guarantee that this approach is the best one. In future studies, we examine this approach with the new abstract object behavior selection approaches that we gain by the new full-fledged behavioral usage models.

In this study, we focus on the impact of seeding during guided initialization by using different values for $Pr[pick\ init]$ and $Pr[clone]$ and setting $Pr[pick\ mut]$ to the default value (0.3). However, our results show that even with the default value 0.3, using seeded objects during the search process helps to reproduce several crashes. Our future work includes a thorough assessment of that factor. Furthermore, in the current version of model seeding, we noticed that the fixed size for the selected abstract object behaviors from the usage models could negatively impact the crash reproduction process. This set's size affects Botsing's performance and must be chosen carefully. If too small, abstract object behaviors may not cover the transition system sufficiently, missing out on important usage information. Too few abstract object behaviors can misguide the search process. In contrast, too many of them will lead to a time-consuming test concretization process. In future investigations, we will study the integration of the search process with the abstract object behavior selection from the models. This integration can guide the seeding (*e.g.,* the abstract object behavior selection) using the current status of the search process.

Finally, we hypothesize that this seeding strategy may be useful for other search-based software testing applications and we will evaluate this hypothesis in our future work.

3.9 Conclusion

Manual crash reproduction is labor-intensive for developers. A promising approach to alleviate them from this challenging activity is to automate crash reproduction using search-based techniques. In this chapter, we evaluate the relevance of using both test and behavioral model seeding to improve crash reproduction achieved by such techniques. We implement both test seeding and the novel model seeding in Botsing.

For practitioners, the implication is that more crashes can be automatically reproduced, with a small cost. In particular, our results show that behavioral model seeding outperforms test seeding and no seeding without a major impact on efficiency. The different behavioral model seeding configurations reproduce 6% more crashes compared to no seeding, while test seeding reduces the number of reproduced crashes. Also, behavioral model seeding can significantly increase the search initialization rate for 3 crashes compared to no seeding, while test seeding performs worse than no seeding in this aspect. We hypothesize that the improvements achieved by model seeding can be further extended by using more resources (*i.e.,* execution logs) for collecting the call sequences which are beneficial for the model generation.

From the research perspective, by abstracting behavior through models and taking advantage of the advances made by the model-based testing community, we can enhance search-based crash reproduction. Our analysis reveals that (1) using collected call sequences, together with (2) the dissimilar selection, and (3) prioritization of abstract object behaviors, as well as (4) the combined information from source code and test execution, enable more search processes to get started, and ultimately more crashes to be reproduced.

In our future work, we will explore whether behavioral model seeding has further ranging implications for the broader area of search-based software testing. Furthermore, we aim to study the effect of changing the seeding probabilities on the search process, explore other sources of data to generate the model and try different abstract object behavior selection strategies.

4
Improving Search-based Crash Reproduction With Helper Objectives

EvoCrash relies on a single-objective evolutionary algorithm (*Single-Objective Search* hereafter) that evolves test cases according to an objective (*Crash Distance* hereafter) measuring how far a generated test is from reproducing the crash. *Crash Distance* combines three heuristics: *line coverage* (how far is the test from executing the line causing the crash?), *exception coverage* (does the test throw the same exception as in the crash?), and *stack trace similarity* (how similar is the exception stack trace from the one reported in the crash?). Although Single-Objective Search performs well compared to the other crash reproduction approaches, a more extensive empirical study in Chapter 2 evidenced that it is not successful in reproducing complex crashes (*i.e.,* large stack traces). Hence, further studies to enhance the guidance of the search process are required.

Just like any other evolutionary-based algorithm, Single-Objective Search requires to maintain a balance between *exploration* and *exploitation* [163]. The former refers to the generation of completely new solutions (*i.e.,* test cases executing new paths in the code); the latter refers to the generation of solutions in the neighborhood of the existing ones (*i.e.,* test cases with similar execution paths). Single-Objective Search ensures exploitation through Guided Mutation, which guarantees that each solution contains the method call causing the crash (and reported in the stack trace) [28]. However, the low exploration of Single-Objective Search may lead to a lack of diversity, trapping the search in local optima [163].

To tackle this problem, our prior study [68] investigated the usage of *Decomposition-based Multi-Objectivization* (*De-MO*) to decompose the *Crash Distance* in three distinct (sub-)objectives. A target crash is reproduced when the search process fulfills all three sub-objectives at the same time. The empirical evaluation shows that *De-MO* slightly improves the efficiency for some crashes. However, since the sub-objectives are not conflicting, while conflicting objectives maintain more diversity in the population and guide the search process away from local optima [168], their combined usage can be detrimental for crash reproduction [68]. Our other previous study [169] also conjectured that increasing diversity via additional objective is a feasible yet unexplored research direction to follow. However, no systematic empirical study has been conducted to evaluate that hypothesis further.

In this study, we investigate a new strategy to Multi-Objectivize crash reproduction based on Helper-Objectives (*MO-HO*) rather than decomposition. More specifically, we add two additional helper-objectives to *Crash Distance* (first objective): *method sequence diversity* (second objective) and *test case length minimization* (third objective). The second objective aims to increase the diversity in the method sequences; more diverse sequences are more likely to cover diverse paths and, consequently, improve exploration. The third objective aims to address the *bloating effect* (*i.e.,* the generated test cases can become longer and longer after each generation until the all of the system memory is used), as diversity can lead to an unnecessary and counter-productive increase of the test case length [42, 170]. Since these three objectives are *conflicting*, we expect an improvement in the solutions' diversity and, hence, improving the effectiveness (crash reproduction ratio) and efficiency.

To assess the performance of *MO-HO* on crash reproduction, we use five multi-objective evolutionary algorithms (MOEAs): NSGA-II [171], SPEA2 [172], MOEA/D [173], PESA-II [174], and FEMO [175]. We apply them to 124 non-trivial crashes from JCrashPack (Chapter 2). Those crashes can only be reproduced by a test case that brings the software under test to a specific state and invokes the target method with one or more specific input parameters. We performed an internal assessment among *MO-HO* algorithms to find the best multi-objective evolutionary algorithm for this optimization problem. According to the results observed in this assessment, *SPEA2* outperforms other MOEAs in crash reproduction using *MO-HO* helper-objectives.

Furthermore, we compared the best-performing *MO-HO* (*MO-HO + SPEA2*) against two state-of-the-art approaches (Single-Objective Search [28] and *De-MO* [68]) from the perspectives of *crash reproduction ratio* and *efficiency*. Our results show that *MO-HO* outperforms the state-of-the-art in terms of crash reproduction ratio and efficiency. This algorithm improves the crash reproduction ratio by up to 100% and 93.3% (10% and 8%, on average) compared to Single-Objective Search and *De-MO*, respectively. Also, after five minutes of search, *MO-HO* reproduces five and six crashes (4% and 5% more crashes) that cannot be reproduced by Single-Objective Search and *De-MO*, respectively. In addition, *MO-HO* reproduces crashes significantly faster than Single-Objective Search and *De-MO* in 34.6% and 37.9% of the crashes, respectively.

A replication package, enabling the full-replication of our evaluation and data analysis of our results is available on Zenodo [64].

4.1 Background And Related Work

Several approaches have been introduced in the literature that aim to reproduce a given crash. Some of these techniques (*e.g.,* ReCore [32]) use runtime data (*i.e.,* core dumps). However, collecting the runtime data may induce a significant overhead and raises privacy concerns. In contrast, other approaches [29–31, 33] only require the *stack traces* of the unhandled exception causing the crash, collected from executions logs or reported issues. For Java programs, a stack trace includes the list of classes, methods, and code line

```
0   java.lang.ArrayIndexOutOfBoundsException: 4 (@@)
1     at [...].FastDateParser.toArray(FastDateParser.java:413) (@@)
2     at [...].FastDateParser.getDisplayNames([...]:381)
3     at [...].FastDateParser$TextStrategy.addRegex([...]:664) (@@)
4     at [...].FastDateParser.init([...]:138)
5     at [...].FastDateParser.<init>([...]:108)
6     [...] (@@)
```

Figure 4.1: LANG-9b crash stack trace [71]

numbers involved in the crash. As an example, Figure 4.1 shows a stack trace produced by a crash (due to a bug) in Apache Commons Lang. This stack trace contains the *type of the exception* (`ArrayIndexOutOfBoundsException`) and *frames* (lines 1-6) indicating the stack of active method calls during the crash.

Among the various approaches solely using a stack trace as input, STAR [30] and BugRedux [97] use backward and forward symbolic execution, respectively; MuCrash [33] mutates the existing test cases of the classes involved in the stack trace; JCharming [31, 134] applies model checking and program slicing for crash reproduction; and ConCrash [29] is designed to use pruning strategies to reproduce the crash-reproducing test case.

EvoCrash is an evolutionary-based approach that applies a Single-Objective Genetic Algorithm (Single-Objective Search) to generate a crash-reproducing test case for a given stack trace and a target frame (*i.e.,* the class under test for which the test case is generated). The generated test will trigger a crash with a stack trace that is identical to the original one, up to the target frame. For instance, for the stack trace in Figure 4.1 with a target frame at line 3, EvoCrash generates a test case that reproduces the first three frames of this stack trace (*i.e.,* identical from lines 0 to 3). A previous empirical evaluation [28] shows that EvoCrash performs better compared to other crash reproduction approaches relying on model checking and program slicing [31, 134], backward symbolic execution [30], or exploiting existing test cases [33]. The study also confirms that automatically generated crash-reproducing test cases help developers to reduce their debugging effort.

4.1.1 Single-Objective Search Heuristics

To evaluate the candidate tests, and consequently guide the search process, Single-Objective Search applies a fitness function called the *Crash Distance.* This fitness function contains three components: (i) the **line coverage distance**, indicating the distance between the execution trace and the target line (the line number pointed to by the target frame), (ii) the **exception type coverage**, indicating whether the target exception is thrown, and (iii) the **stack trace similarity**, indicating whether all frames (from the beginning up to the target frame) are included in the triggered stack trace.

Definition 4.1.1 (*Crash Distance* [28]) *For a given test case execution t, the Crash Distance (f) is defined as follows:*

$$f(t) = \begin{cases} 3 \times d_s(t) + 2 \times max(d_e) + max(d_{tr}) & \textit{if line not reached} \\ 3 \times min(d_s) + 2 \times d_e(t) + max(d_{tr}) & \textit{if line reached} \\ 3 \times min(d_s) + 2 \times min(d_e) + d_{tr}(t) & \textit{if exception thrown} \end{cases} \tag{4.1}$$

Where $d_s(t) \in [0,1]$ indicates how far the test t is from reaching the *target line* using two heuristics: *approach level* and *branch distance* [4]. The former measures the minimum number of control dependencies between the execution path of t and the target line; the latter indicates how far t is from satisfying the branch condition on which the target line is control dependent. And $d_e(t) \in \{0,1\}$ indicates whether an exception with the same type as the target exception is thrown (0) or not (1). Finally, $d_{tr}(t) \in [0,1]$ calculates the similarity between the stack trace produced by t and the expected one, based on classes, methods, and line numbers appearing in both stack traces. Functions $max(.)$ and $min(.)$ denote the maximum and minimum possible values for a function, respectively. Concretely, $d_e(t)$ and $d_{tr}(t)$ are only calculated upon the satisfaction of two *constraints*: *exception type coverage* and *stack trace similarity* are relevant only when we reach the target line (first constraint) and when we have the same type of exception (second constraint), respectively.

4.1.2 Single-Objective Search

The search process starts with a **guided initialization** during which an initial population of randomly generated test cases is created. The algorithm ensures that each test case calls the *target method* (pointed to by the target frame) at least once. In each generation, the fittest test cases are evolved by applying **guided mutation** and **guided crossover**. Guided mutation applies a classical mutation to the test cases while ensuring that the mutated test contains one or more calls to the target method. Similarly, guided crossover is a variant of the single-point crossover that preserves calls to the target methods in the offsprings. Accordingly, each generated test case contains at least one call to the target method (*i.e.,* the method triggering the crash) [28].

With those operators, Single-Objective Search improves the exploitation, but it penalizes exploration of new areas of the search space by not generating diverse enough test cases. As a consequence, the search process may get stuck in local optima.

4.1.3 Decomposition-based Multi-objectivization

To increase diversity during the search, a prior study [68] investigated the usage of Decomposition-based Multi-Objectivization (called *De-MO* hereafter) to decompose the *Crash Distance* in three distinct (sub-)objectives. *De-MO* on the *Crash Distance* (temporarily) decomposes the function in three distinct (sub-)objectives: $d_s(t)$, $d_e(t)$, and $d_{tr}(t)$. Then, *De-MO* uses a multi-objective evolutionary algorithm optimizing three objectives to generate one crash-reproducing solution. In the end, the global optimal solution is a test case in the Pareto front produced by MOEAs that satisfies all of the sub-objectives simultaneously. The empirical evaluation shows that *De-MO* increases the efficiency of the crash reproduction process for some specific cases compared to Single-Objective Search. However, it loses efficiency in some other cases.

In particular, in *Multi-objectivization*, search objectives should be conflicting to increase the diversity of generated solutions [168]. However, the three sub-objectives in *De-MO* [68] are tightly coupled and not conflicting: the stack trace similarity ($d_{tr}(t)$) cannot be computed for test case t without executing the target line ($d_s(t) = 0$) and throwing the correct type of exception ($d_e(t) = 0$). Also, the type of exception ($d_e(t)$) is not relevant, while test t does not cover the statement in the target line ($d_s(t) = 0.0$).

4.2 Multi-Objectivization with Helper-Objectives (MO-HO)

Decomposing the *Crash Distance* leads to a set of dependent sub-objectives, which reduces the effect of improving diversity through multi-objectivization [168]. In this study, we focus on using new helper-objectives in addition to the *Crash Distance*, rather than decomposing it. We define two helper-objectives called **method sequence diversity** and **test length minimization** that aim to (i) increase diversity in the population (*i.e.,* generated tests) and (ii) address the *bloating* effect [42, 147]. Then, we use five different evolutionary algorithms belonging to different categories of MOEAs (*e.g.,* decomposition-based and rank-based) to solve this optimization problem. In the remainder of this section, we first discuss the two helper-objectives. Next, we present the MOEAs used to solve this problem.

4.2.1 Helper-objectives

As suggested by Jensen *et al.* [168], adding helper-objectives to an existing single objective can help search algorithms escape from local optima. However, this requires that the helper objectives are in conflict with the primary one [168]. Therefore, defining proper helper-objectives is crucial.

Method Sequence Diversity. The first helper-objective seeks to maximize the diversity of the method-call sequences that compose the generated tests because more diverse tests might execute different paths or behaviors of the *target class*. Notice that each test case is a sequence of statements, where each statement belongs to one of the following five different categories [42]: *primitive* statements, *constructors*, *field statements*, *method calls*, or *assignments*. Furthermore, the length of a test case is variable, *i.e.,* it is not fixed a priori and can vary during the search.

In recent years, several functions have been introduced to measure test case diversity [147]. These functions measure the diversity between two test cases by using a binary encoding function to calculate the distance between the corresponding encoded vectors using the Levenshtein distance [176], Hamming distance [177], *etc.* For three or more test cases, the overall diversity corresponds to the average pairwise diversity of the existing test cases [147]. These metrics have been used in other testing tasks (*e.g.,* automated test selection), but not in crash reproduction.

To measure the value of this helper-objective for the generated solutions, we follow a similar procedure. Let us assume that $F = \{f_1, f_2, ..f_n\}$ is a set of public and protected methods in the target class (*i.e.,* method calls that can be called directly by the generated tests), and $T = \{t_1, t_2, ..t_m\}$ is a set of generated test cases. To calculate the diversity of T, we first need to encode each $t_k \in T$ into a binary vector. We use the same encoding function proposed by Mondal *et al.* [147]: each test case $t_k \in T$ corresponds to a binary vector v_k of length n (*i.e.,* the number of public and protected methods in the target class). Each element $v_k[i]$ of the binary vector denotes whether the corresponding method $f_i \in F$ is invoked by the test case t_k. More formally, for each method $f_i \in F$, the corresponding entry $v_k[i] = 1$ if t_k calls f_i; $v_k[i] = 0$ otherwise.

Then, we calculate the diversity for each pair of test cases t_k and t_i as the Hamming distance between the corresponding binary vectors v_k and v_i [177]. The Hamming distance (*Hamming*) between two vectors corresponds to the number of mismatches (The number of positions at which the corresponding bits are different) over the total length of the binary vectors. For instance, the Hamming distance between $A = \langle 1,1,0,1,0\rangle$ and $B = \langle 0,1,0,1,1\rangle$ equals to 2/5 = 0.4.

Definition 4.2.1 (Method Sequence Diversity) *Given an encoding function $V(.)$, the method sequence diversity (MSD) of a test $t \in T$ corresponds to the average Hamming distance of that test from the other test cases in T:*

$$MSD(t) = \frac{\sum_{t_i \in T \setminus \{t\}} Hamming(V(t), V(t_i))}{|T| - 1} \tag{4.2}$$

In our approach, *MSD* should be maximized to increase the chance of the generated test to execute new paths or behaviors in the *target class*. Since our tool (see Section 4.3.1) is designed for minimization problems, we minimize the method sequence similarity using the formula:

$$f_{MSD}(t) = 1 - MSD(t) \tag{4.3}$$

Test Length Minimization While increasing method sequence diversity can help to execute diverse paths of the target class, a previous study [170] also showed that *test diversity metrics* (such as call sequence diversity) can reduce coverage. This is due to the *bloating effect*, *i.e.,* diversity will also promote larger test cases over short ones. Let us assume that we have a set of short test cases with few method calls in our population (most of the elements in their binary vectors are 0). A lengthy test case t_L that calls all the methods of the target class will have a binary vector containing only 1 values. As a consequence, t_L will have a large Hamming distance from the existing test cases.

Larger tests introduce two potential issues: (i) they are likely more expensive to run (extra overhead), and (ii) they may contain spurious statements that do not help code coverage (which is a part of *Crash Distance*). In the latter case, mutation can become less effective as it may mutate spurious statements rather than the relevant part of the chromosomes. Therefore, test diversity is in conflict with *Crash Distance*. To avoid the *bloating* effect, our second helper-objective is *test length minimization*, which counts the number of statements in a given test:

Definition 4.2.2 (Test Length Minimization) *For a test case t with a length $|t|$, the fitness function is:*

$$f_{len}(t) = |t| \tag{4.4}$$

4.2.2 Multi-objective Evolutionary Algorithms

In this study, our goal is to solve a multi-objectivized problem by minimizing the three objective functions (*Crash Distance*, f_{MSD}, and f_{len}). In theory, we could consider various MOEAs, each coming with different advantages and disadvantages over different optimization problems (*e.g.,* multimodal, convex, *etc.*). However, we cannot establish upfront what type of MOEA works better for crash reproduction as the shape of the Pareto Front (*i.e.,* type of problem) for crash reproduction is unknown. Hence, we chose five MOEAs from different categories to determine the best algorithm for *MO-HO*: *NSGA-II* uses the non-dominated sorting procedure; *SPEA2* is an archive-based algorithm that selects the best solutions according to the fitness value; *PESA-II* divides the objective space to hyper-boxes and selects the solutions from the hyper-boxes with the lower density; *MOEA/D* decomposes the problem to multiple sub-problems; and *FEMO*, is a (1+1) evolutionary algorithm that evolves tests solely with mutation and without crossover.

We use the same stopping conditions for all search algorithms, which is a maximum search budget, or when the target crash is successfully reproduced, *i.e.,* a solution with a *Crash Distance* of 0.0 is found. Also, to increase *exploitation* during the search, all algorithms use the *guided crossover* and *guided mutation* operators.

In the following subsections, we briefly describe the selected search algorithms and their core characteristics.

Non-dominated Sorting Genetic Algorithm II (NSGA-II) [171].
In NSGA-II, offspring tests are generated, from given a population of size N, using genetic operators (crossover and mutation). Next, NSGA-II unions the offspring population with the parent population into a set of size $2N$ and applies a *non-dominated sorting* to select the N individuals for the next generation. This sorting is performed based on the *dominance* relation and *crowding distance*: the solutions are sorted into subsequent dominance fronts. The non-dominated solutions are in the first front ($Front_0$). These solutions have a higher chance of being selected. Furthermore, *crowding distance* is used to raise the chance of the most diverse solutions within the same front to be selected for the next generation. In each generation, parent test cases are selected for reproduction using the *binary tournament selection.*

Strength Pareto Evolutionary Algorithm 2 (SPEA2) [172].
Besides the current population, SPEA2 contains an external archive that collects the non-dominated solutions among all of the solutions considered during the search process. SPEA2 assigns a *fitness value* to each solution (test) in the archive. The *fitness value* of solution i is calculated by summing up two values: *Raw fitness* ($R(i) \in \mathbb{N}_0$), which represents the dominance relation of i; and *Strength value* ($S(i) \in [0, 1]$), which estimates the density of solutions in the same Pareto front (solutions that are not dominating each other). A solution with lower fitness value is "better" and has a higher chance of being selected. For instance, the non-dominated solutions have a $R(i) = 0$, and their fitness values are lower than 1.

The external archive has a fixed size, which is given at the beginning of the search process. After updating the archive in each iteration, the algorithm checks if the size of the archive exceeds this given size. If the size of the archive is smaller than the given size, SPEA2 fills the archive with the existing dominated solutions. In contrast, if the size of the archive is bigger than the given size, this algorithm uses a *truncation operator* to remove the solutions with a high *fitness value* from the archive. After updating the archive, SPEA2 applies *binary tournament selection* based on the calculated *fitness values*, selects parent solutions, and generates offspring solutions via *crossover* and *mutation.*

Pareto Envelop-based Selection Algorithm (PESA-II) [174].
Similar to SPEA2, PESA-II benefits from an external archive. In each generation, the archive is updated by storing the non-dominated solutions in the archive and the current population. However, the difference is in the selection strategy and archive truncation. In this algorithm, instead of assigning a fitness value to each of the solutions in the archive, the objective space is divided, based on the existing solutions, into hyper-boxes or grids.

Non-dominated solutions in a hyper-box with lower density have a higher chance of being selected and a lower chance of being removed.

Multi-objective Evolutionary Algorithm Based on Decomposition (MOEA/D) [173].

This algorithm decomposes the M-objectives problem into K single-objective sub-problems and optimizes them simultaneously. Each sub-problem has different weights for the optimization objectives. The K sub-problems $g(x|w_1),\ldots, g(x|w_K)$ are obtained using a scalarization function $g(x|w)$ and a set of uniformly-distributed weight vectors $W = \{w_1,\ldots, w_k\}$. The decomposition can be done with several techniques such as *weighted sum* [178], *Tchebycheff* [178], or *Boundary Intersection* [179, 180]. In each generation, MOEA/D maintains the best individuals for each subproblem $g(x|w_i)$, while the reproduction (based on crossover and mutation) is allowed only among solutions (tests) within the same neighborhood (*mating restriction*).

Fair Evolutionary Multi-objective Optimizer (FEMO) [175].
This algorithm is a local (1+1) evolutionary algorithm. It means that in each iteration, only one solution is evolved by the mutation operator to have only one offspring solution for the next generation. FEMO contains an archive. In the first iteration, it generates a random solution and places it in the archive. In the next generations, it selects one individual from the archive and evolves it by mutation operator to generate a new solution. Finally, if the new solution dominates at least one of the solutions in the archive, it adds the new solution to the archive and removes the dominated solutions.

Each solution in the archive has a weight (w) that indicates the number of times that a solution was selected from the archive. So, the initial weight of a newly generated test case is 0. During the selection, FEMO selects a solution randomly from the solutions in the archive that have the lowest w.

4.3 Empirical Evaluation

To assess the impact of *MO-HO* on crash reproduction, we performed an empirical evaluation and answered the following research questions.

RQ$_1$: *Which Multi-Objective algorithm performs better with MO-HO's search objectives in terms of crash reproduction?*

RQ$_2$: *What is the impact of the MO-HO algorithm on crash reproduction compared to Single-Objective Search and De-MO?*

RQ$_3$: *How does MO-HO's efficiency compare to Single-Objective Search and De-MO?*

4.3.1 Implementation

Since other crash reproduction approaches are not openly available, we implemented a new open-source evolutionary-based crash reproduction framework, called BOTSING.[1] BOTSING is well-tested and designed to be easily extensible for new techniques (new evolutionary algorithms, new genetic operators, *etc.*). It relies on EVOSUITE [6], an evolutionary-based unit test generation tool, for code instrumentation and for the internal representation of an individual (*i.e.,* a test case) by using `evosuite-client` as a dependency.

[1]Available at `https://github.com/STAMP-project/botsing`

For this study, we implemented the techniques used in previous studies for crash reproduction (Single-Objective Search and *De-MO*) in Botsing. Moreover, we implemented all of the *MO-HO* approaches, which include the two fitness functions for our new helper-objectives (*method sequence diversity* and *test length*) and the five MOEAs mentioned above.

4.3.2 Setup

Crash Selection

We selected our crashes from JCrashPack (Chapter 2). Based on the reported results in Chapter 2 and our prior studies on decomposition-based crash reproduction [68], we know that Single-Objective Search and *De-MO* face various challenges to reproduce many of the crashes in this benchmark. For this study, we apply our approach and state-of-the-art algorithms to 124 crashes from JCrashPack. These crashes stem from six open-source projects: JFreeChart, a framework for building interactive charts; Commons-lang, a library providing extra utilities to the `java.lang` API; Commons-math, a library for mathematical and statistical usages; Mockito, a testing framework for mocking objects; Joda-time, a library for date and time manipulation; XWiki, a large-scale enterprise wiki management system.

Algorithm Selection

We attempted to reproduce the selected crashes using seven evolutionary algorithms: Single-Objective Search, *De-MO*, and *MO-HO* with five MOEAs (NSGA-II, SPEA2, PESA-II, MOEA/D, and FEMO). For each crash, we ran each algorithm on each frame of crash stack traces. We repeated each execution 30 times to take randomness into account, for a total number of 199,710 independent executions. We ran the evaluation on servers with 40 CPU-cores, 128 GB memory, and 6 TB hard drive.

Evaluation procedure

In RQ$_1$, we perform an internal assessment of *MO-HO* by comparing all MOEAs to determine the best-performing one when optimizing the search objectives in *MO-HO*. Then, to answer RQ$_2$ and RQ$_3$, we use the best-performing *MO-HO* configuration (MOEA) to evaluate its effectiveness and efficiency against the state-of-the-art crash reproduction approaches.

Parameter Settings

We set the search budget to five minutes, as suggested by previous studies on evolutionary-based crash reproduction [28]. Also, we fixed the population size and archive size (if needed) to 50 individuals, as recommended in prior studies on test case generation [42]. For *MO-HO* with PESA-II, the number of bisections for gridding is set to the default value of five grids. In *MO-HO* with MOEA/D, the weight vectors are obtained using a variant *simplex-lattice design* [181] and using the *Tchebycheff approach* as the aggregation function. Finally, we set the *neighborhood selection probability* to 0.2 (set to the default value [182]) and the maximum number of *solutions that can be replaced* in each generation to 50. For all MOEAs, we use the *guided mutation* with mutation probability $p_m = 1/n$ (n is the length of the test case), and *guided crossover* with crossover probability $p_c = 0.8$ (the same parameters used for the suggested baselines).

4.3.3 Data Analysis

To evaluate the crash reproduction ratio (*i.e.,* the percentage of successful crash reproduction attempts in 30 rounds of runs) of different algorithms, we follow the same procedure as the study presented in Chapter 3: for each crash C, we find the highest frame that can be reproduced by at least one of the algorithms (r_{max}). We analyze the crash reproduction ratio of each algorithm for a target crash C targeting frame r_{max}.

To check whether the performance (reproduction ratio) of MOEAs significantly differs from one another, we use the Friedman test [183]. The Friedman test is a non-parametric version of the ANOVA test [184], *i.e.,* it does not make any assumption about the data distribution. It is a multiple-problem statistical test and has been widely used in the literature to compare randomized algorithms [167, 185]. Friedman's test allows to rank and statistically compare different MOEAs over multiple independent problems, i.e., crashes in our case. For Friedman's test, we use a level of significance $\alpha = 0.05$. If the p-values obtained from Friedman's test are significant (p-values $<= 0.05$), we apply pairwise multiple comparison using Conover's post-hoc procedure [186]. To correct for multiple comparison errors, we adjust the p-values from Conover's procedure using Holm-Bonferroni [187].

To answer RQ_2, we need to determine whether an algorithm reproduces a crash. Since we repeat each execution 30 times, we use the majority of outcomes for a crash reproduction result. In other words, if an algorithm could reproduce a crash in ≥ 15 runs (*i.e.,* reproduction ratio of $\geq 50\%$), we count that frame as *reproduced.*

To compare the number of reproduced crashes by each algorithm, we used the same procedure used by Almasi *et al.* [25] and Campos *et al.* [188]: we check crash reproduction status and reproduction ratio of the best-performing *MO-HO* algorithm (according to the results of RQ_1), Single-Objective Search, and *De-MO* at five time intervals: 1, 2, 3, 4 and 5 minute.

To evaluate the efficiency of the algorithms (RQ_3), we analyze the time spent by the best *MO-HO* algorithm, Single-Objective Search, and *De-MO* for generating a crash reproducing test cases. Since efficiency is only applicable to the reproduced crashes, we compare the efficiency of algorithms on the crashes that are reproduced at least once by one of the algorithms. If, for one execution, an algorithm was not able to reproduce the crash, it means that it consumed the maximum allowed time budget (5 minutes). To assess the effect size of differences between algorithms, we use the Vargha-Delaney $\hat{A}_{12}$ statistic [165]. A value of $\hat{A}_{12} < 0.5$ for a pair of factors (A, B) shows that A reproduced the target crash in a shorter time, while a value of $\hat{A}_{12} > 0.5$ indicates the opposite. Besides, $\hat{A}_{12} = 0.5$ means that there is no difference between the factors. To evaluate the significance of effect sizes ($\hat{A}_{12}$), we use the non-parametric Wilcoxon Rank Sum test, with $\alpha = 0.05$ for the Type I error.

A replication package of our evaluation is available on Zenodo [64]. It contains the selected crashes, the results and data analysis presented in this chapter, as well as the implementation of MOEAs in Botsing and a Docker-based infrastructure to enable the full-replication of our evaluation.

4.4 Results

This section presents the results of our empirical evaluation and answers, one by one, our research questions.

Table 4.1: MOEAs ranking (in *MO-HO*) in terms of crash reproduction ratio (Friedman's test) and results of the pairwise comparison (p-value ≤ 0.05)

Rank	MOEA	Rank value	Significantly better than
1	SPEA2	2.63	(2), (3), (4), (5)
2	PESA-II	2.86	(4), (5)
3	NSGA-II	2.90	(4), (5)
4	MOEAD	4.97	(5)
5	FEMO	5.05	

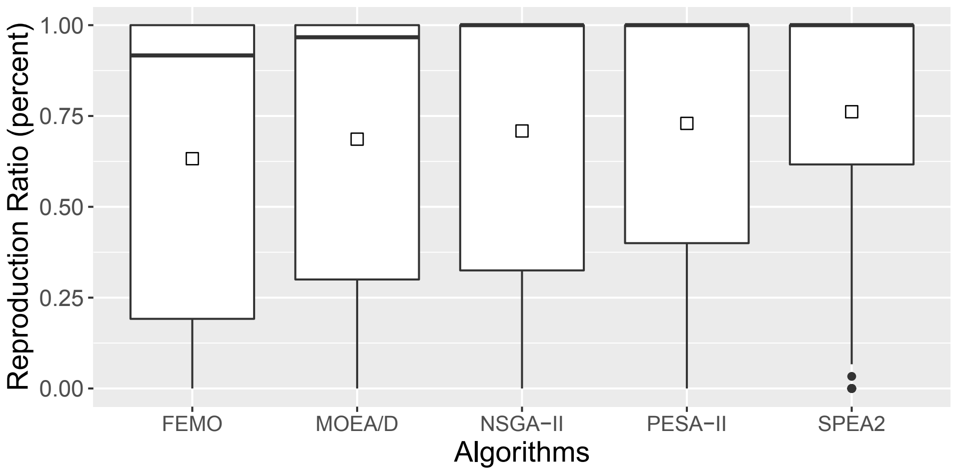


Figure 4.2: Crash reproduction ratio (out of 30 executions) of *MO-HO* algorithms. The upper and lower edge of each box present the upper and lower quartile, respectively. (□) denotes the arithmetic mean and (—) is the median.

4.4.1 Best MOEA for *MO-HO* (RQ1)

Figure 4.2 presents the crash reproduction ratio of the MOEAs applied to our *MO-HO* framework. For this analysis, we consider the number of times (in percentage) each MOEAs could reproduce a given crash across 30 runs and using a search budget of five minutes. On average (the squares in Figure 4.2), the best algorithm for *MO-HO* is *SPEA2*, with an average and median of 76% and 100% of successful reproductions, respectively. *SPEA2* is Followed by *PESA-II*, *NSGA-II*, and *MOEAD*. Also, this figure shows that the first quartile of the crash reproduction ratio of *SPEA2* is, at least, about 25% higher than other MOEAs.

According to Friedman's test, the differences in reproduction ratios are statistically significant (p-value ≤ 0.05). This means that some MOEAs are significantly better than others within our *MO-HO* framework. For completeness, Table 4.1 reports the ranking produced by the Friedman test. To better understand for which pairs of MOEAS the statistical significance holds, we applied the post-hoc Conover's procedure for the pairwise comparison. The results of the comparison are also reported in Table 4.1. According to this table, the best-performing algorithm is *MO-HO* + *SPEA2*, which has a significantly higher

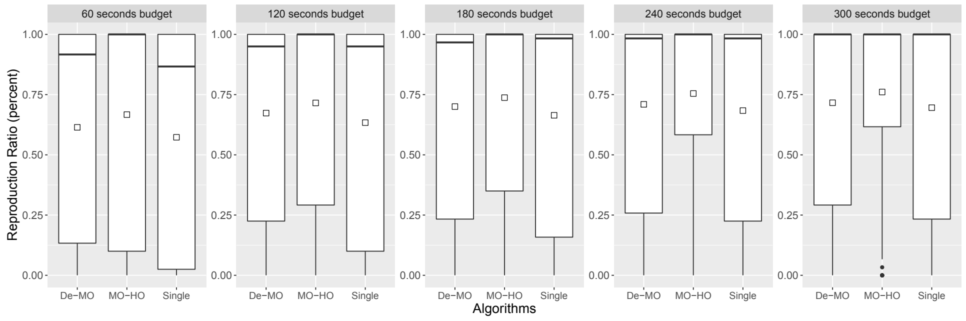


Figure 4.3: Crash reproduction ratio (out of 30 executions) of *MO-HO* against state-of-the-art in five different time intervals. (□) denotes the arithmetic mean and (—) is the median.

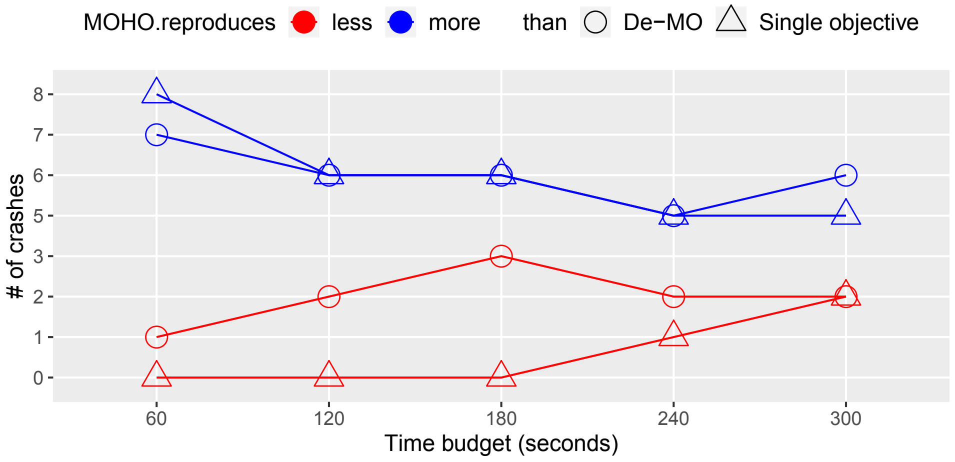


Figure 4.4: Number of crashes reproduced only by *MO-HO* or only by one of the state-of-the-art algorithms.

crash reproduction ratio compared to other *MO-HO* algorithms. The next algorithms are *MO-HO + PESA-II* and *MO-HO + NSGA-II*. These two algorithms are significantly better than *MO-HO + MOEAD* and *MO-HO + FEMO*. Finally, the worst algorithm in terms of crash reproduction is *FEMO*, which is significantly worse than other MOEAs.

Summary (RQ$_1$). *MO-HO + SPEA2 achieved the highest performance in terms of crash reproduction ratio compared to MO-HO + other MOEAs. The next best-performing MOEAs, in terms of crash reproduction, are PESA-II and NSGA-II.*

4.4.2 Crash Reproduction (RQ2)

Figure 4.3 depicts the crash reproduction ratio of the best-performing *MO-HO* configuration (i.e., with *SPEA2*), Single-Objective Search, and *De-MO* at five time intervals (search budgets). As indicated in this figure, the average crash reproduction ratio of *MO-HO* is higher than other algorithms at all of the time intervals. Also, the median crash reproduction ratio for this algorithm is always 100%. Furthermore, the maximum improvement achieved by *MO-HO* with the five-minutes search budget is in *XWIKI-14599* (with 100% improvement) and *MATH-3b* (with 93.3% improvement) compared to Single-Objective Search and *De-MO*, respectively. In contrast, the largest reduction in reproduction ratio by *MO-HO* (with the five-minutes budget) is in *XCOMMONS-1057* (with 30% drop) and *XWIKI-13616* (with 40% reduction) compared to Single-Objective Search and *De-MO*, respectively. We will explain the negative factors in *MO-HO*, which lead to negative results for this algorithm in some corner cases, in Section 4.4.4.

Moreover, we can see that *De-MO* is the second-best algorithm in all of the time intervals. In the first 60 seconds of the crash reproduction process, on average, its crash reproduction ratio is 4% better than Single-Objective Search. However, in contrast to the other two algorithms, the crash reproduction ratio of this algorithm changes only slightly after the first 120 seconds. Hence, at the end of the search process, the average crash reproduction ratio of *De-MO* is only 2% better than Single-Objective Search. In contrast, since the crash reproduction ratio of *MO-HO* keeps growing, on average, it remains more effective than Single-Objective Search (about 10%) even after 300 seconds. The other interesting point in Figure 4.3 is the first quantile of *MO-HO*. In the first 60 seconds, this value is lower than 12%, but it grows up to 62% after 300 seconds. This improvement is not observable in state-of-the-art algorithms.

Furthermore, *MO-HO* is more stable in crash reproduction after 300 seconds budget compared to the other algorithms. Figure 3 demonstrates that the interquartile range (i.e., the difference between first and third quartile) of crash reproduction ratio in *MO-HO* with the 300 seconds budget is 46% smaller than the interquartile range of other algorithms (being 38.3% for *MO-HO*, 76.6%. for Single-Objective Search, and 70.8% for *De-MO*).

Also, Figure 4.4 shows the number of crashes, which are reproduced by *MO-HO*, but not by the state-of-the-art algorithms and vice versa in different time intervals. As indicated in this figure, in all of the time intervals, the number of crashes that are reproduced by *MO-HO* is higher than the crashes that it cannot reproduce. In the best case (after 1 minute of search), *MO-HO* reproduces eight and seven new crashes that cannot be reproduced by Single-Objective Search and *De-MO*, respectively. In contrast, there is only one crash that can be reproduced by *De-MO* and not by *MO-HO*. Also, after five minutes, *MO-HO* still reproduces more crashes than the baselines: it reproduces five and six new crashes that cannot be reproduced by Single-Objective Search and *De-MO*, respectively.

The crashes that are reproduced by *MO-HO* after five minutes but not by Single-Objective Search are: `TIME-10b` frame 5, `XCOMMONS-928` frame 2, `XWIKI-14227` frame 2, `XWIKI-14475` frame 1, and `XWIKI-14599` frame 1. And the crashes that are reproduced by *MO-HO* after five minutes but not by *De-MO* are: `MOCKITO-16b` frame 4, `TIME-5b` frame 3, `XWIKI-13377` frame 3, `XWIKI-14227` frame 2, `MATH-3b` frame 1, and `MOCKITO-10b` frame 1.

Figure 4.5 shows the crash's stack trace reported in the issue `XWIKI-14227`. *MO-HO* is the only approach that can reproduce the first two frames of this stack trace. Here, the target method is `useMainStore` (Figure 4.6), which does not have any input argument. Hence, to reproduce this crash, the crash reproducing test generated by *MO-HO* (depicted in Figure 4.8) should invoke specific methods (*e.g.*, `setWiki`, `setWikiId`) to set different local variables in the `xwikiContext0` object, and then, pass this object to the class under test

```
java.lang.NullPointerException: null
        at [...].XWiki.getPlugin(XWiki.java:5619)
        at [...].ActivityStreamConfiguration.useMainStore([...]:85)
    [...]
```

Figure 4.5: XWIKI-14227 crash's stack trace.

```
public boolean useMainStore(){
    XWikiContext context = contextProvider.get();
    if (context.isMainWiki()) {return false;}
    ActivityStreamPlugin plugin = (ActivityStreamPlugin)
      context.getWiki().getPlugin[...] context); // <-- target line
}
```

Figure 4.6: Method `useMainStore` appears in the second frame of the `XWIKI-14227` crash's stack trace.

(here, `ActivityStreamConfiguration`). Since the crash reproducing test case generated by *MO-HO* does not add any plugin to the `xWiki0` object, the execution of this test indeed leads to a `NullPointerException` thrown at line 5619 of the `getPlugin` method in Figure 4.7. Generating such a specific test case requires a search process with high exploration ability, which can generate diverse test cases.

We do note that Single-Objective Search cannot even generate a test case covering the target line (line 85 of the `useMainStore` method). However, *De-MO* can cover the target line thanks to more test generation diversity delivered by the application of multi-objectivization.

Moreover, Single-Objective Search and *De-MO* reproduces two crashes that cannot be reproduced by *MO-HO* after five minutes. We will analyze these corner cases later in Section 4.4.4.

In addition, after five minutes of crash reproduction, *De-MO* reproduced six crashes, which are not reproduced by Single-Objective Search. Still, there are more crashes (seven) that can be reproduced by Single-Objective Search but not by *De-MO*. This result shows that despite the new crashes reproduced by *De-MO*, this algorithm was counter-productive with respect to the total number of reproduced crashes.

Summary (RQ$_2$). *On average, MO-HO has the highest crash reproduction ratio independently from the search budgets.*

4.4.3 Efficiency (RQ3)

Figure 4.9 shows the time (in seconds) needed by the *MO-HO* and the state-of-the-art algorithms to successfully reproduce the crashes in our benchmark. On average, the fastest algorithm is *MO-HO*, with an average search time of 71 seconds per crash replication. The median of its running time is lower than 10 seconds. The second fastest algorithm is *De-MO* that, on average, uses 84 seconds to reproduce the crashes. The slowest algorithm is Single-Objective Search, which demands, on average, about 100 seconds.

Moreover, the biggest improvements achieved by *MO-HO* in terms of efficiency are for *XWIKI-14599*, in which *MO-HO* requires only 3% of the time required by Single-Objective Search to achieve crash reproduction, and *MATH-3b*, in which *MO-HO* requires only

```
public XWikiPluginInterface getPlugin([...]){
    XWikiPluginManager plugins = getPluginManager();
    Vector<String> pluginlist = plugins.getPlugins();
    [...]
}
```

Figure 4.7: Method `getPlugin` appears in the first frame of the `XWIKI-14227` crash's stack trace.

```
public void test0() throws Throwable {
    ActivityStreamConfiguration ac0 = new ActivityStreamConfiguration();
    XWikiContext xWikiContext0 = new XWikiContext();
    XWiki xWiki0 = new XWiki();
    xWikiContext0.setWiki(xWiki0);
    xWikiContext0.setWikiId("4~YRlfI >.U{ib");
    Provider<XWikiContext> provider0 = (Provider<XWikiContext>)
      mock([...]);
    doReturn(xWikiContext0).when(provider0).get();
    Injector.inject(ac0,[...], "contextProvider", (Object) provider0);

    // Undeclared exception!
    ac0.useMainStore();
}
```

Figure 4.8: Crash-reproducing test case generated by *MO-HO* for the XWIKI-14227 crash.

7% of the time required by *De-MO* to finish the crash reproduction task. However, the biggest efficiency losses by *MO-HO* are in *MATH-81b* with 45 seconds drop (15% of time budget) and *XRENDERING-481* with 145 seconds drop (48% of time budget) compared to Single-Objective Search and *De-MO*, respectively.

Table 4.2 compares the budget consumption of the algorithms from a statistical point of view, *i.e.,* according to the effect sizes ($\hat{A}_{12} < 0.5$) and statistical significance (*p-value* < 0.5). According to this table, *MO-HO* is the fastest algorithm: it significantly reproduced 43 (34.6% of crashes) and 47 (37.9% of crashes) crashes faster than Single-Objective Search and *De-MO*, respectively. Most of these significant improvements have large effect sizes (35 against Single-Objective Search and 33 against *De-MO*). In cases that *MO-HO* improves efficiency, on average, this algorithm decreases the time required for crash reproduction by 47% and 58% compared to *De-MO* and Single-Objective Search, respectively.

Furthermore, Table 4.2 shows a few cases, in which *MO-HO* increases the consumed time compared to the state-of-the-art: 3 against Single-Objective Search and 5 against *De-MO*. In most of these cases (7 out of 8), the crash reproduction process needs to reproduce a crash with only one frame. Even the exceptional case is a stack trace with three frames. In contrast, in cases that *MO-HO* wins, we have many crashes with more frames (six frames, for instance). Also, this table shows that *De-MO* is significantly slower than Single-Objective Search in 11 crashes. Meanwhile, *MO-HO* is only slow in reproducing three crashes. Hence, our proposed algorithm reduces the cases in which the multi-objectivization search process is slower than the single objective search by 73%.

Summary (RQ$_3$). *The fastest crash reproduction algorithm is MO-HO with an average*

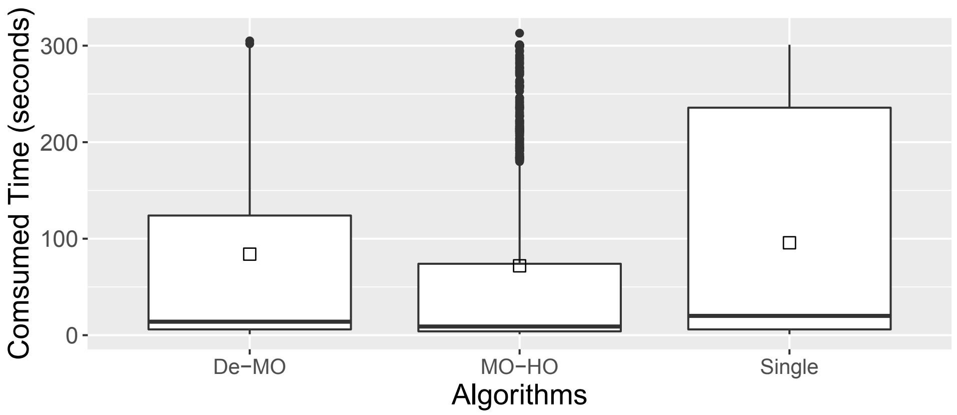


Figure 4.9: Overall budget consumption in seconds (log. scale). (□) denotes the arithmetic mean and (—) is the median.

Table 4.2: Pairwise comparison of the budget consumption with a small (S), medium (M), and large (L) effect size $\hat{A}_{12} < 0.5$ and a statistical significance < 0.05.

#($\hat{A}_{12}$ < 0.5)	**Single**			**De-MO**			**MO-HO**		
	L	M	S	L	M	S	L	M	S
Single	-	-	-	7	-	4	1	-	2
De-MO	13	7	2	-	-	-	3	2	-
MO-HO	35	6	2	33	10	4	-	-	-

improvement in running time in 34.6% of the crashes compared to the state of the art.

4.4.4 Corner Cases Analysis

Despite the notable improvements achieved by *MO-HO*, there are few specific cases, in which Single-Objective Search or *De-MO* outperform *MO-HO*. For instance, in Section 4.4.2, Single-Objective Search and *De-MO* reproduce two crashes that are not reproduced by *MO-HO*. Also, we observed in Section 4.4.3 that the efficiency of these two algorithms is higher than *MO-HO* in 8 crashes.

To understand why *MO-HO* is counter-productive in a few cases, we performed a manual analysis to analyze the factors in *MO-HO* that negatively impact the crash reproduction process. Results of our analysis point to two adverse factors: **extra overhead in calculating the objectives (fitness evaluation)** and **helper-objectives misguidance**.

Extra calculation in fitness evaluation

In some cases, crash reproduction is trivial, and the search process reproduces it in a few seconds. For instance, in `TIME-8b` [71], Single-Objective Search and *De-MO* reproduce the crash in about a second. The time required by *MO-HO* to reproduce this crash is three seconds (3 times more). This stems from the fact that fitness function evaluation in *MO-HO* is more time-consuming than the state-of-the-art: Single-Objective Search and *De-MO* need to calculate only the crash distance for each test case evaluation, while *MO-HO* needs

```
java.lang.ArrayIndexOutOfBoundsException: 2
    at org.apache.commons.math.linear.BigMatrixImpl.operate(BigMatrixImpl.java:997)
```

Figure 4.10: `MATH-98b` crash's stack trace [71].

```
public BigDecimal[] operate(BigDecimal[] v) {
    final int nRows = this.getRowDimension();
    final int nCols = this.getColumnDimension();
    final BigDecimal[] out = new BigDecimal[v.length];
    for (int row = 0; row < nRows; row++) {
        ...
        out[row] = sum; // <-- target line
    }
    ...
}
```

Figure 4.11: Method `operate` appears in the first frame of the `MATH-98b` crash's stack trace [71].

to calculate the call diversity, as well. This extra calculation lengthens the search process by a couple of seconds. In these cases, the increased crash reproduction time is lower than 5 seconds, and it is negligible in practice.

Helper-objectives misguidance

In some other cases, the scenario, which leads to crash reproduction, needs a simple sequence of methods calls to the target class. Still, the complexity of this scenario stems from the input arguments used for the method calls. In these cases, since crash reproduction does not need the call diversity, *method sequence diversity* objective misguides the search process. Alternatively, we need another objective for method input argument diversity (*i.e.,* improves the diversity of the input arguments for method calls). Adding new helper-objectives to consider other aspects of diversity is part of our future agenda.

As an example, let us analyze `MATH-98b` (Figure 4.10), in which MO-HO doubled the time consumed by the crash reproduction search process against state-of-the-art. This crash concerns an `ArrayIndexOutOfBoundsException`. Also, this crash has only one frame. For reproducing this crash, the generated test case needs to instantiate a class called `BigMatrixImpl` and call a method named `operate` (Figure 4.11) with precise input values. Method `getColumnDimension` used in `operate` returns the number of rows in the data variable, which has been set in the constructor. To reproduce this crash, the generated test case should pass an array with a size smaller than the passed size to the constructor. In this case, method argument diversity could help the search process, and the method call diversity is not helpful.

4.5 Discussion

4.5.1 Effectiveness And Applicability

Generally, *De-MO* reproduces some crashes that cannot be reproduced by Single-Objective Search due to its improved exploration ability, resulting from the multi-objectivization of the crash distance. However, since the decomposed objectives in this approach depend on one another (*e.g.*, the stack trace similarity is not helpful if the generated test does not throw the given type of exception), they may misguide the search process in various cases. For instance, as we saw in Section 4.4.2, Single-Objective Search reproduces six crashes that are not reproducible by *De-MO*.

In contrast, *MO-HO* has three *conflicting* search objectives. From the theory [168], the objective function must be conflicting to increase the overall exploration ability. Our results confirm the theory: the chance of the search process getting trapped in a local optimum is lower by using *MO-HO* objectives compared to the ones used in *De-MO*. As we observed in Section 4.4.2, after 1 minute of search, *MO-HO* reproduces 8 and 7 crashes more than Single-Objective Search and *De-MO*, respectively. Also, it continues outperforming with larger search budgets (2, 3, 4, and 5 minutes) until the end of the search process. It reproduces 5 and 6 crashes more than Single-Objective Search and *De-MO*, respectively, while it cannot reproduce only two crashes, reproduced by the other algorithms.

Note that reproducing each crash needs a particular test case which drives the software under test to a particular state, and then, it calls a method with proper input variables. To achieve this goal, each crash reproducing test case needs to create multiple complex objects. Hence, reproducing five new crashes (4% of crashes available in our benchmark) is a significant improvement for *MO-HO*.

4.5.2 Factors In The Benchmark Crashes That Impact The Success Of *MO-HO*

There are multiple factors/characteristics of the crashes in our benchmark that might impact the performance of our approach positively. We identify the following relevant factors: (1) the type of the exception (e.g., null pointer exception), (2) the size the stack frames, (3) the number of classes involved in the crashes, (4) the number of methods of the deepest class in the crash stack. To verify whether these factors influence the performance of our algorithm, we used the two-way permutation test [189]. The permutation test is a well-established non-parametric to assess the significance of factor interactions in multi-factorial analysis of variance (non-parametric ANOVA). We use a significance level $alpha$=0.05 and a very large number of iterations (1,000,000) to ensure the stability of the results over multiple executions of the procedure [189].

For the sake of our analysis, we considered the difference in crash reproduction rate between *MO-HO* and the baselines as the dependent variable, while the co-factors are our independent variables. According to the permutation test, the type of exception (p-value=0.006) and the number of crash stack frames (p-value=0.001) significantly impact the performance of *MO-HO* compared to Single Objective Search. We can also observe similar results when considering the improvements of *MO-HO* against *De-MO*: p-values=< 10^{-12} for both exception type and the number of frames). In other words, there are certain types of exceptions and stack trace sizes for which *MO-HO* is statistically better than the state-of-the-art approaches.

From a deeper analysis, we observe that for `NullPointerExcep-tion` and `org.joda.tim-e.IllegalFieldValueException`, *MO-HO* achieves a higher reproduction ratio than Single Objective Search when the stack traces contain up to three frames for `NPE` (+22% in reproduction rate) and up to five frames for `IllegalFieldValueEx-ception` (+50% in reproduction rate). Instead, for stack traces with more frames, the differences in reproduction ratio are negligible (±1% on average) or negative (-10% in reproduction ratio). Besides, *MO-HO* achieves better reproduction ratios for the following exceptions independently of the stack size: `XWikiExceptions` (+23% on average), `UnsupportedOperationException` (+6% on average), `MathRuntimeException` (+14% on average).

Finally, *MO-HO* outperforms *De-MO* when reproducing `NullPoin-terException` with 1-3 frames (+8% on average), `ClassCastExcep-tion` (+8% on average), `StringOutOfBound-sException` (+18% with more than 2 frames, on average), `IllegalFieldException` (+8% on average), `UnsupportedOperationException` (+23% on average), `MockitoException` (+83% for short traces, on average), and `MissingMethodInvocation` (+80% on average).

4.5.3 Crash Reproduction Cost

In this study, we observed that since *MO-HO* increases the diversity of the generated test cases, it can dramatically improve the efficiency of crash reproduction. This algorithm significantly improved the speed of the search process in more than 36% of crashes compared to Single-Objective Search and *De-MO*. In cases in which *MO-HO* had a significant impact, it improves the crash reproduction speed by more than 47%.

Our prior study (Chapter 2) suggested 5 minutes as the search budget because the search process cannot reproduce more after 5 minutes. However, we observed that despite the high efficiency of *MO-HO*, this algorithm continues to reproduce more crashes in the second half of the time budget. Section 4.4.2 shows that *MO-HO* keeps increasing the crash reproduction ratio even in the last minutes of the search process, while the previous multi-objectivization approach (*De-MO*) changes only slightly after the first 2 minutes of crash reproduction. Hence, increasing the search budget for *MO-HO* can lead to a higher crash reproduction ratio.

4.5.4 Extendability

The improvement achieved by the proposed helper-objectives shows the impact of suitable objectives on increasing the diversity of the generated test cases and result in improving the effectiveness and efficiency of the crash reproduction search process. Hence, we hypothesize that this approach can be extended by adding new relevant helper-objectives.

4.6 Threats To Validity

Internal validity. We cannot ensure that our implementation of Botsing is without bugs. However, we mitigated this threat by testing our tool and manually analyzing some samples of the results. We used a previously defined benchmark for crash reproduction, which contains 124 non-trivial crashes from six open-source projects and applications. Moreover, we explained how we parametrized the evolutionary algorithms in Section 4.3.2. We used the default values of these algorithms in the other open-source implementations like EvoSuite and *JMetal*. The effect of these values for crash reproduction is part of our future work. Finally, to take the randomness of the search process into account, we followed the guidelines of the related literature [110] and executed each evolutionary crash reproduction algorithm for 30 times.

External validity. We report our results for only 124 crashes introduced by JCrash-

Pack (Cahpter 2), which is an open-source crash reproduction benchmark collected from six open-source projects. However, we recall here that we cannot guarantee that our results are generalizable to all crashes. Evaluation *MO-HO* on a larger benchmark from more projects is part of our future work.

Reproducibility. We provide Botsing as an open-source publicly available tool. Also, the data and the processing scripts used to present the results of this chapter, including the subjects of our evaluation (inputs), the evolution of the best fitness function value in each generation of each execution, and the produced test cases (outputs), are openly available as a docker image [64].

4.7 Conclusion And Future Work

Crash reproduction can ease the process of debugging for developers. Evolutionary approaches have been successfully used to automate this process. Existing evolutionary-based approaches use one single objective (*i.e., Crash Distance*) to guide the search and rely on guided genetic operators. Later strategies applied multi-objectivization via decomposition (*De-MO*) in an attempt to improve diversity (and, therefore, exploration). However, the latter strategy may misguide the search process because the sub-objectives are not strongly conflicting.

In this study, we apply a new approach called Multi-Objectivization using Helper-Objectives (*MO-HO*) to tackle the problems of the former techniques. In *MO-HO*, multi-objectivization is performed by adding two helper-objectives that are in conflict with *Crash Distance*. We evaluated *MO-HO* with five MOEAs, which are selected from different categories of multi-objective algorithms. Our results indicate that *MO-HO* is the most efficient algorithm, significantly outperforming Single-Objective Search and *De-MO*. Also, this algorithm is able to reproduce 8 and 5 more crashes in 1 and 5 minutes, respectively, compared to the state-of-the-art. Moreover, in contrast to the previous multi-objectivized crash reproduction approach (*De-MO*), the crash reproduction ability of *MO-HO* increases with large search budgets (*i.e.,* above two minutes).

We performed an additional analysis to find the correlation between the different aspects of the crashes and the ability of *MO-HO* in reproducing them. The result of this analysis shows that two factors in crashes significantly impact the performance of *MO-HO*: (i) type of exception and (ii) the number of crash stack frames.

Furthermore, we observed that Single-Objective Search and *De-MO* could outperform *MO-HO* but only in a few cases. We performed a manual analysis to characterize the negative factors leading to the adverse results in these cases. Our analysis reveals that two negative factors are at play in these cases: (i) extra calculations in fitness evaluation and (ii) helper-objectives misguidance. We also showed in Section 4.4.4 that while the differences in *extra calculations in fitness evaluation* are significant, they are often negligible in practice.

The contributions of the chapter are as follows:

1. An open-source implementation of seven crash reproduction techniques (Section 4.3.1).
2. An empirical comparison of seven search-based crash reproduction approaches (Section 4.3).

3. An analysis of the benefits of multi-objectivization with helper objectives in terms of reproduction ratio and efficiency (Section 4.4).
4. The identification of the special situations in which *MO-HO* can be counter-productive (Section 4.4.4).
5. The identification of a strong correlation between the ability of *MO-HO* in improving the efficiency and effectiveness of crash reproduction for combinations of exception types and the number of frames in the stack trace of the target crash (Section 4.5.2).

In our future work, we will investigate additional helper-objectives for crash reproduction. For instance, the current helper-objectives in *MO-HO* concern the test length and method sequence diversity. However, further objectives can be added, such as test input/data diversity. Increasing the number of objectives will require to evaluate their performance using different many-objective evolutionary algorithms. We will also analyze the evolution of the fitness values of existing and new objective to further investigate the root causes of good and bad performances of *MO-HO* and other objectives for different crashes and different MOEAs.

Moreover, the search objectives introduced by *De-MO* is only optimized by *NSGA-II* MOEA. As future work, we will investigate the impact of utilizing other MOEAs for optimizing *De-MO* objectives.

5 Basic Block Coverage for Search-Based Crash Reproduction

Various search-based techniques have been introduced to automate different white-box test generation activities (*e.g.,* unit testing [6, 41], system-level testing [190], *etc.*). Depending on the testing level, each of these approaches utilizes dedicated fitness functions to guide the search process and produce a test suite satisfying given criteria (*e.g.,* line coverage, branch coverage, *etc.*).

Fitness functions typically rely on *control flow graphs (CFGs)* to represent the source code of the software under test [4]. Each node in a CFG is a *basic block* of code (*i.e.,* maximal linear sequence of statements with a single entry and exit point without any internal branch), and each edge represents a possible *execution flow* between two blocks. Two well-known heuristics are usually combined to achieve high line and branch coverages: the *approach level* and the *branch distance* [4]. The former measures the distance between the execution path of the generated test and a target basic block (*i.e.,* a basic block containing a statement to cover) in the CFG. The latter measures, using a set of rules, the distance between an execution and the coverage of a *true* or *false* branch of a particular predicate in a branching basic block of the CFG.

Both *approach level* and *branch distance* assume that only a limited number of basic blocks (*i.e., control dependent* basic blocks [39]) can change the execution path away from a target statement (*e.g.,* if a target basic block is the true branch of an conditional statement). However, basic blocks are not atomic due to the presence of **implicit branches** [191] (*i.e.,* branches occurring due to the exceptional behavior of instructions). As a consequence, any basic block between the entry point of the CFG and the target basic block can impact the execution of the target basic block. For instance, a generated test case may stop its execution in the middle of a basic block with a runtime exception thrown by one of the statements of that basic block. In these cases, the search process does not benefit from any further guidance from the approach level and branch distance.

Fraser and Arcuri [192] introduced testability transformation, which instruments the code to guide the unit test generation search to cover implicit exceptions happening in the class under test. However, this approach does not guide the search process in scenarios where an implicit branch happens in the other classes called by the class under test. This is because of the extra cost added to the process stemming from the calculation and monitoring of the implicit branches in all of the classes, coupled with the class under test. For instance, the class under test may be heavily coupled with other classes in the project, thereby finding implicit branches in all of these classes can be expensive.

However, for some test case generation scenarios, like **crash reproduction**, we aim to cover a limited number of paths, and thereby we only need to analyse a limited number of basic blocks [28, 30, 32, 33, 134]. Current crash reproduction approaches rely on information about a reported crash (*e.g.,* stack trace, core dump *etc.*) to generate a **crash reproducing test case (CRT)**

Among these approaches, search-based crash reproduction [28, 32] takes as input a **stack trace** to guide the generation process. More specifically, the statements pointed by the stack trace act as target statements for the approach level and branch distance. Hence, current search-based crash reproduction techniques suffer from the lack of guidance in cases where the involved basic blocks contain implicit branches (which is common when trying to reproduce a crash).

This chapter introduces a novel secondary objective called **Basic Block Coverage (*BBC*)** to address this guidance problem in crash reproduction. *BBC* helps the search process to compare two generated test cases with the same distance (according to approach level and branch distance) to determine which one is closer to the target statement. In this comparison, *BBC* analyzes the coverage level, achieved by each of these test cases, of the basic blocks in between the closest covered control dependent basic block and the target statement.

We assessed the impact of *BBC* secondary objective on two fitness functions in search-based crash reproduction: *Crash Distance* and *STDistance*. The former guides the search process in EvoCrash [28]. The latter was introduced as one of the heuristics in the fitness function, which guides the search process in ReCore [32] to reproduce a crash using the stack trace and core dump produced during the crash occurring. *STDistance* is the only heuristic in ReCore that measures the distance of the generated solution from the given stack trace and relies on approach level and branch distance more compared to *Crash Distance*. We empirically compared these two fitness functions' performance with and without using *BBC* (4 configurations in total). We applied these four crash reproduction configurations to 124 hard-to-reproduce crashes introduced as JCrashPack (Chapter 2). We compare the performances in terms of *effectiveness in crash reproduction ratio* (*i.e.,* percentage of times that an approach can reproduce a crash) and *efficiency* (*i.e.,* time required by for reproducing a crash).

Our results show that *BBC* significantly improves the crash reproduction ratio over the 30 runs in our experiment for respectively 5 and 1 crashes when compared to using *STDistance* and *Crash Distance* without any secondary objective. Also, *BBC* helps these two fitness functions to reproduce 6 (for *STDistance*) and 1 (for *Crash Distance*) crashes that they could not be reproduced without secondary objective. Besides, on average, *BBC* increases the crash reproduction ratio of *STDistance* by 4%. Applying *BBC* also significantly reduces the time consumed for crash reproduction guided by *STDistance* and *Crash Distance* in 33 (26.6% of cases) and 14 (13.7% of cases) crashes, respectively, while it was significantly counter productive in only one case. In cases where *BBC* has a significant impact on efficiency, this secondary objective improves the average efficiency of *STDistance* and *Crash Distance* by 40.6% and 44.3%, respectively.

5.1 Background

5.1.1 Coverage Distance Heuristics

Many structural-based search-based test generation approaches mix the *branch distance* and *approach level* heuristics to achieve a high line and branch coverage [4]. These heuristics measure the distance between a test execution path and a specific statement or a specific branch in the software under test. For that, they rely on the coverage information of control dependent basic blocks, *i.e.,* basic blocks that have at least one outgoing edge leading the execution path toward the target basic block (containing the targeted statement) and at least another outgoing edge leading the execution path away from the target basic block. As an example, Listing 5.1 shows the source code of method `fromMap` in XWIKI[1], and Figure 5.1 contains the corresponding CFG. In this graph, the basic block `409` is control dependent on the basic block `407-408` because the execution of line `409` is dependent on the satisfaction of the predicate at line `408` (*i.e.,* line `409` will be executed only if elements of array `formvalues` are `String`).

The *approach level* is the number of uncovered control dependent basic blocks for the target basic block between the closest covered control dependent basic block and the target basic block. The *branch distance* is calculated from the predicate of the closest covered control dependent basic block, based on a set of predefined rules. Assuming that the test *t* covers only line `403` and `417`, and our target line is `409`, the approach level is 2 because two control dependent basic blocks (`404-406` and `407-408`) are not covered by *t*. The branch distance the predicate in line `403` (the closest covered control dependency of node `409`) is measured based on the rules from the established technique [4].

To the best of our knowledge, there is no related work studying the extra heuristics helping the combination of approach level and branch distance to improve the coverage. Most related to our work, Panichella *et al.* [42] and Rojas *et al.* [193] introduced two heuristics called *infection distance* and *propagation distance*, to improve the weak mutation score of two generated test cases. However, these heuristics do not help the search process to improve the general statement coverage (*i.e.,* they are effective only after covering a mutated statement).

In this chapter, we introduce a new secondary objective to improve the statement coverage achieved by fitness functions based on the approach level and branch distance, and analyze the impact of this secondary objective on **search-based crash reproduction**.

Example 5.2: XWIKI-13377 crash stack trace (collected in Chapter 2)

```
java.lang.ClassCastException: [...]
    at [...].BaseStringProperty.setValue(BaseStringProperty.java:45)
    at [...].PropertyClass.fromValue(PropertyClass.java:615)
    at [...].BaseClass.fromMap(BaseClass.java:413)
    [...] (@@)
```

[1]https://github.com/xwiki

Example 5.1: Method `fromMap` from XWIKI version 8.1 (collected in Chapter 2)

```
402     public BaseCollection fromMap(Map<[...]> map, BaseCollection object){
403         for (PropertyClass property : (Collection<[...]>) getFieldList()) {
404             String name = property.getName();
405             Object formvalues = map.get(name);
406             if (formvalues != null) {
407                 BaseProperty objprop;
408                 if (formvalues instanceof String[]) {
409                     [...]
410                 } else if (formvalues instanceof String) {
411                     objprop = property.fromString(formvalues.toString());
412                 } else {
413                     objprop = property.fromValue(formvalues);
414                 }
415                 [...]
416             }}
417         return object;}
```

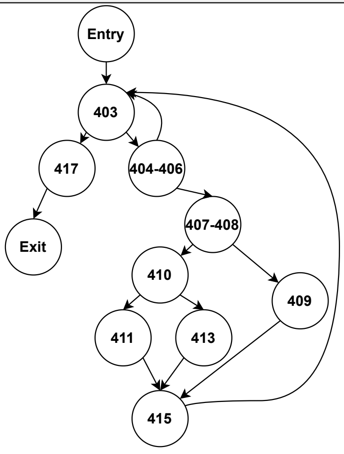


Figure 5.1: CFG for method `fromMap`

5.1.2 Search-based Crash Reproduction

After a crash is reported, one of the essential steps of software debugging is to write a **Crash Reproducing Test case (CRT)** to make the crash observable to the developer and help them in identifying the root cause of the failure [57]. Later, this CRT can be integrated into the existing test suite to prevent future regressions. Despite the usefulness of a CRT, the process of writing this test can be labor-intensive and time-taking [28]. Various techniques have been introduced to automate the reproduction of a crash [28, 30, 32, 33, 134], and search-based approaches (EvoCrash [28] and ReCore [32]) yielded the best results [28].

EvoCrash. This approach utilizes a single-objective genetic algorithm to generate a CRT from a given stack trace and a *target frame* (*i.e.,* a frame in the stack trace that its class will be used as the class under test). The CRT generated by EvoCrash throws the same stack trace as the given one up to the target frame. For example, by passing the stack trace in Listing 5.2 and target frame 3 to EvoCrash, it generates a test case reproducing the first three frames of this stack trace (*i.e.,* thrown stack trace is identical from line 0 to 3).

EvoCrash uses a fitness function, called *Crash Distance*, to evaluate the candidate test cases. *Crash Distance* is the sum scalarization of three components: (i) the **target**

line coverage (d_s), which measures the distance between the execution trace and the *target line* (*i.e.,* the line number pointed to by the target frame) using *approach level* and *branch distance*; (ii) the **exception type coverage** (d_e), determining whether the type of the triggered exception is the same as the given one; and (iii) the **stack trace similarity** (d_{tr}), which indicates whether the stack trace triggered by the generated test contains all frames (from the most in-depth frame up to the target frame) in the given stack trace.

Definition 5.1.1 (*Crash Distance* [28]) *For a given test case execution t, the Crash Distance (ws) is defined as follows:*

$$ws(t) = \begin{cases} 3 \times d_s(t) + 2 \times max(d_e) + max(d_{tr}) & \textit{if line not reached} \\ 3 \times min(d_s) + 2 \times d_e(t) + max(d_{tr}) & \textit{if line reached} \\ 3 \times min(d_s) + 2 \times min(d_e) + d_{tr}(t) & \textit{if exception thrown} \end{cases} \tag{5.1}$$

Where $d_s(t) \in [0,1]$ indicates how far t is from reaching the target line and is computed using the normalized approach level and branch distance: $d_s(t) = \|approachLevel_s(t) + \|branchDistance_s(t)\|\|$. Also, $d_e(t) \in \{0,1\}$ shows if the type of the exception thrown by t is the same as the given stack trace (0) or not (1). Finally, $d_{tr}(t) \in [0,1]$ measures the stack trace similarity between the given stack trace and the one thrown by t. $max(f)$ and $min(f)$ denote the maximum and minimum possible values for a function f, respectively. In this fitness function, $d_e(t)$ and $d_{tr}(t)$ are only considered in the satisfaction of two *constraints*: (i) *exception type coverage* is relevant only when we reach the target line and (ii) *stack trace similarity* is important only when we both reach the target line and throw the same type of exception.

As an example, when applying EvoCrash on the stack trace from Listing 5.2 with the target frame 3, *Crash Distance* first checks if the test cases generated by the search process reach the statement pointed to by the target frame (line 413 in class `BaseClass` in this case). Then, it checks if the generated test can throw a `ClassCastException` or not. Finally, after fulfilling the first two constraints, it checks the similarity of frames in the stack trace thrown by the generated test case against the given stack trace in Listing 5.2.

EvoCrash uses **guided** initialization, mutation and single-point crossover operators to ensure that the target method (*i.e.,* the method appeared in the target frame) is always called by the different tests during the evolution process.

According to a recent study, EvoCrash outperforms other non-search-based crash reproduction approaches in terms of *effectiveness in crash reproduction* and *efficiency* [28]. This study also shows the helpfulness of tests generated by EvoCrash for developers during debugging.

In this chapter, we assess the impact of *BBC* as the secondary objective in the EvoCrash search process.

ReCore. This approach utilizes a genetic algorithm guided by a single fitness function, which has been defined according to the core dump and the stack trace produced by the system when the crash happened. To be more precise, this fitness function is a sum scalarization of three sub-functions: (i) **TestStackTraceDistance**, which guides the search process according to the given stack trace; (ii) **ExceptionPenalty**, which indicates whether the same type of exception as the given one is thrown or not (identical to ExceptionCoverage in EvoCrash); and (iii) **StackDumpDistance**, which guides the search process by the given core dump.

Definition 5.1.2 (*TestStackTraceDistance* [32]) *For a given test case execution t, the TestStackTraceDistance (STD) is defined as follows:*

$$STD(R,t) = |R| - lcp - (1 - StatementDistance(s)) \tag{5.2}$$

Where $|R|$ is the number of frames in the given stack trace. And lcp is the longest common prefix frames between the given stack trace and the stack trace thrown by t. Concretely, $|R| - lcp$ is the number of frames not covered by t. Moreover, $StatementDistance(s)$ is calculated using the sum of the approach level and the normalized branch distance to reach the statement s, which is pointed to by the first (the utmost) uncovered frame by t: $StatementDistance(s) = approachLevel_s(t) + \|branchDistance_s(t)\|$.

Since using runtime data (such as core dumps) can cause significant overhead [30] and leads to privacy issues [134], the performance of ReCore in crash reproduction was not compared with EvoCrash in prior studies [28]. Although, two out of three fitness functions in ReCore use only the given stack trace to guide the search process. Hence, this study only considers *TestStackTraceDistance + ExceptionPenalty* (called *STDistance* hereafter).

As an example, when applying ReCore with *STDistance* on the stack trace in Listing 5.2 with target frame 3, first, *STDistance* determines if the generated test covers the statement at frame 3 (line 413 in class `BaseClass`). Then, it checks the coverage of frame 2 (line 615 in class `PropertyClass`). After covering the first two frames by the generated test case, it checks the coverage of the statement pointed to by the deepest frame (line 45 in class `BaseStringProperty`). For measuring the coverage of each of these statements, *STDistance* uses the approach level and branch distance. After covering all of the frames, this fitness function checks if the the generated test throws `ClassCastException` in the deepest frame.

In this study, we perform an empirical evaluation to assess the performance of crash reproduction using *STDistance* with and without *BBC* as the secondary objective in terms of *effectiveness in crash reproduction* and *efficiency*.

5.2 Basic Block Coverage

5.2.1 Motivating Example

During the search process, the fitness of a test case is evaluated using a fitness function, either *Crash Distance* or *STDistance*. Since the search-based crash reproduction techniques model this task to a minimization problem, the generated test cases with lower fitness values have a higher chance of being selected and evolved to generate the next generation. One of the main components of these fitness functions is the coverage of specific statements pointed by the given stack trace. The distance of the test case from the target statement is calculated using the approach level and branch distance heuristics. As we have discussed in Section 5.1.1, the approach level and branch distance cannot guide the search process if the execution stops because of implicit branches in the middle of basic blocks (*e.g.,* a thrown `NullPointerException` during the execution of a basic block). As a consequence, these fitness functions may return the same fitness value for two tests, although the tests do not cover the same statements in the block of code where the implicit branching happens.

For instance, assume that the search process for reproducing the crash in Figure 5.2 generates two test cases T_1 and T_2. The first step for these test cases is to cover frame

3 in the stack trace (line 413 in BaseClass). However, T_1 stops the execution at line 404 due to a `NullPointerException` thrown in method `getName`, and T_2 throws a `NullPointerException` at line 405 because it passes a null value input argument to `map`. Even though T_2 covers more lines, the combination of approach level and branch distance returns the same fitness value for both of these test cases: approach level is 2 (nodes 407-408 and 410) and branch distance is measured according to the last predicate. This is because these two heuristics assume that each basic block is atomic, and by covering line 404, it means that lines 405 and 406 are covered, as well.

5.2.2 Secondary Objective

The goal of the Basic Block Coverage (*BBC*) secondary objective is to prioritize the test cases with the same fitness value according to their coverage within the basic blocks between the closest covered control dependency and the target statement. At each iteration of the search algorithm, test cases with the same fitness value are compared with each other using *BBC*. Algorithm 1 presents the pseudo-code of the *BBC* calculation. Inputs of this algorithm are two test cases T_1 and T_2, which both have the same fitness value (calculated either using *Crash Distance* or *STDistance*), as well as line number and method name of the target statement. This algorithm compares the coverage of basic blocks on the path between the entry point of the CFG of the given method and the basic block that contains the target statement (called *effective blocks* hereafter) achieved by T_1 and T_2. If *BBC* determines there is no preference between these two test cases, it returns 0. Also, it returns a value < 0 if T_1 has higher coverage compared to T_2, and vice versa. A higher absolute value of the returned integer indicates a bigger distance between the given test cases.

Algorithm 1: *BBC* secondary objective computation algorithm

```
   Input: test $T_1$, test $T_2$, String method, int line
   Output: int
 1 $FCB_1 \leftarrow$ fullyCoveredBlocks($T_1$,method,line);
 2 $FCB_2 \leftarrow$ fullyCoveredBlocks($T_2$,method,line);
 3 $SCB_1 \leftarrow$ semiCoveredBlocks($T_1$,method,line);
 4 $SCB_2 \leftarrow$ semiCoveredBlocks($T_2$,method,line);
 5 if $\underline{FCB_1 \subset FCB_2 \wedge SCB_1 \subset SCB_2) \vee (FCB_2 \subset FCB_1 \wedge SCB_2 \subset SCB_1 n}$ then
 6     return size($FCB_2 \cup SCB_2$) - size($FCB_1 \cup SCB_1$);
 7 else if $\underline{FCB_1 = FCB_2 \wedge SCB_1 = SCB_2}$ then
 8     closestBlock $\leftarrow$ closestSemiCoveredBlocks($SCB_1$, method, line);
 9     coveredLines1 $\leftarrow$ getCoveredLines($T_1$,closestBlock);
10     coveredLines2 $\leftarrow$ getCoveredLines($T_2$,closestBlock);
11     return size(coveredLines2) - size(coveredLines1);
12 else
13     return 0;
14 end
```

In the first step, *BBC* detects the effective blocks fully covered by each given test case (*i.e.,* the test covers all of the statements in the block) and saves them in two sets called FCB_1 and FCB_2 (lines 1 and 2 in Algorithm 1). Then, it detects the effective blocks semi-covered by each test case (*i.e.,* blocks where the test covers the first line but not the last line) and stores them in $\texttt{SCB}_1$ and $\texttt{SCB}_2$ (lines 3 and 4). The semi-covered blocks indicate the presence of implicit branches. Next, *BBC* checks if both fully and semi-covered blocks of one of the tests are subsets of the blocks covered by the other test (line 5). In this case, the test case that covers the most basic blocks is the winner. Hence, *BBC* returns the number of blocks only covered by the winner test case (line 6). If *BBC* determines T_2 wins over T_1, the returned value will be positive, and vice versa.

If none of the test cases subsumes the coverage of the other one, *BBC* checks if the blocks covered by T_1 and T_2 are identical (line 7). If this is the case, *BBC* checks if one of the tests has a higher line coverage for the semi-covered blocks closest to the target statement (lines 8 to 11). If this is the case, *BBC* will return the number of lines in this block covered only by the winning test case. If the lines covered are the same for T_1 and T_2 (*i.e.,* `coveredLines1` and `coveredLines2` have the same size), there is no difference between these two test cases and *BBC* returns value 0 (line 11). Finally, if each of the given tests has a unique covered block in the given method (*i.e.,* the tests cover different paths in the method), *BBC* cannot determine the winner and returns 0 (lines 12 and 13) because we do not know which path leads to the crash reproduction.

Example. When giving two tests with the same fitness value (calculated by the primary objective) T_1 and T_2 from our motivation example to *BBC* with target method `fromMap` and line number 413 (according to the frame 3 of Figure 5.2), this algorithm compares their fully and semi-covered blocks with each other. In this example both T_1 and T_2 cover the same basic blocks: the fully covered block is `403` and the semi-covered block is `404-406`. So, *BBC* checks the number of lines covered by T_1 and T_2 in block `404-406`. Since T_1 stopped its execution at line 404, the number of lines covered by this test is 1. In contrast, T_2 managed to execute two lines (404 and 405). Hence, *BBC* returns $size(coveredLines2) - size(coveredLines1) = 1$. The positive return value indicates that T_2 is closer to the target statement and therefore, it should have higher chance to be selected for the next generation.

Branchless Methods. *BBC* can also be helpful for branchless methods. Since there are no control dependent nodes in branchless methods, approach level and branch distance cannot guide the search process in these cases. For instance, methods from frames 1 and 2 in Figure 5.2 are branchless. So, we expect that *BBC* can help the current heuristics to guide the search process toward covering the most in-depth statement.

5.3 Empirical Evaluation

To assess the impact of *BBC* on search-based crash reproduction, we perform an empirical evaluation to answer the following research questions:

RQ_1: *What is the impact of BBC on crash reproduction in terms of effectiveness in crash reproduction ratio?*

RQ_2: *What is the impact of BBC on the efficiency of crash reproduction?*

In these two RQs we want to evaluate the effect of *BBC* on the existing fitness functions, namely *STDistance* and *Crash Distance*, from two perspectives: effectiveness on crash reproduction ratio and efficiency.

5.3.1 Setup

Implementation. Since ReCore and EvoCrash are not openly available, we implement *BBC* in Botsing, an extensible, well-tested, and open-source search-based crash reproduction framework already implementing the *Crash Distance* fitness function and the guided initialization, mutation, and crossover operators. We also implement *STDistance* (ReCore fitness function) in this tool. Botsing relies on EvoSuite [6], an open-source search-based tool for unit test generation, for code instrumentation and test case generation by using `evosuite-client` as a dependency. We also implement the *STDistance* fitness function used as baseline in this chapter.

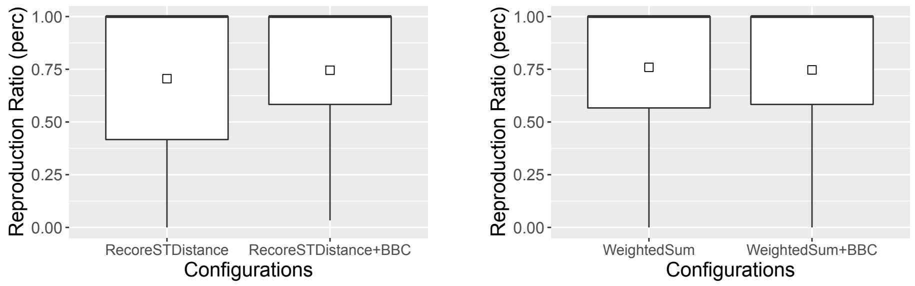
Crash selection. We select crashes from JCrashPack (Chapter 2), a benchmark containing hard-to-reproduce Java crashes. We apply the two fitness functions with and without using *BBC* as a secondary objective to 124 crashes, which have also been used in Chapter 3. These crashes stem from six open-source projects: JFreeChart, Commons-lang, Commons-math, Mockito, Joda-time, and XWiki. For each crash, we apply each configuration on each frame of the crash stack traces. We repeat each execution 30 times to take randomness into account, for a total number of 114,120 independent executions. We run the evaluation on two servers with 40 CPU-cores, 128 GB memory, and 6 TB hard drive.

Parameter settings. We run each search process with five minutes budget and set the population size to 50 individuals, as suggested by previous studies on search-based test generation [42]. Moreover, as recommended in prior studies on search-based crash reproduction [28], we use the *guided mutation* with a probability $p_m = 1/n$ (n = length of the generated test case), and the *guided crossover* with a probability $p_c = 0.8$ to evolve test cases. We do note that prior studies do not investigate the sensitivity of the crash reproduction to these probabilities. Tuning these parameters should be undertaken as future works.

5.3.2 Data Analysis

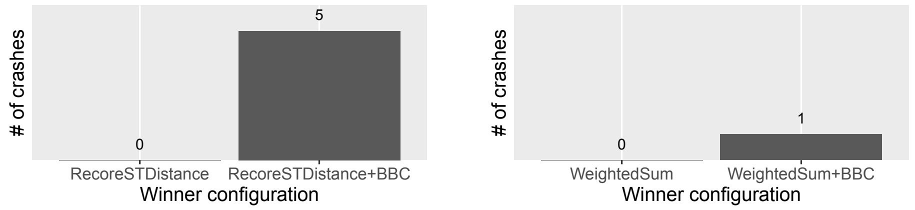
To evaluate the crash reproduction ratio (*i.e.,* the ratio of success in crash reproduction in 30 rounds of runs) of different assessed configurations (**RQ**$_1$), we follow the same procedure as Chapter 3: for each crash C, we detect the highest frame that can be reproduced by at least one of the configurations (r_{max}). We examine the crash reproduction ratio of each configuration for crash C targeting frame r_{max}. Since crash reproduction data has a dichotomic distribution (*i.e.,* an algorithm reproduces a crash C from its r_{max} or not), we use the Odds Ratio (OR) to measure the impact of each algorithm in crash reproduction ratio. A value $OR > 0$ in a comparison between a pair of factors (A, B) indicates that the application of factor A increases the crash reproduction ratio, while $OR < 0$ indicates the opposite. Also, a value of $OR = 0$ indicates that both of the factors have the same performance. We apply Fisher's exact test, with $\alpha = 0.01$ for the Type I error, to assess the significance of results.

To evaluate the efficiency of different configurations (**RQ**$_2$), we analyze the time spent by each configuration on generating a crash reproducing test case. We do note that the extra pre-analysis and basic block coverage in *BBC* is considered in the spent time. Since measuring efficiency is only possible for the reproduced crashes, we compare the efficiency of algorithms on the crashes that are reproduced at least once by one of the algorithms.



(a) *STDistance* (b) *Crash Distance*

Figure 5.2: Crash reproduction ratio (out of 30 executions) of fitness functions with and without *BBC*. (□) denotes the arithmetic mean and the bold line (—) is the median.



(a) *STDistance* (b) *Crash Distance*

Figure 5.3: Pairwise comparison of impact of *BBC* on each fitness function in terms of crash reproduction ratio with a statistical significance < 0.01.

In executions that an algorithm failed to reproduce a crash, we assume that it reached the maximum allowed budget (5 minutes).

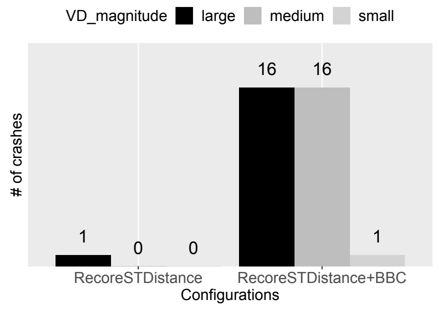
In this study, we use the Vargha-Delaney $\hat{A}_{12}$ statistic [165] to examine the effect size of differences between using and not using *BBC* for efficiency. For a pair of factors (A, B) a value of $\hat{A}_{12} > 0.5$ indicates that A reproduces the target crash in a longer time, while a value of $\hat{A}_{12} < 0.5$ shows the opposite. Also, $\hat{A}_{12} = 0.5$ means that there is no difference between the factors. In addition, to assess the significance of effect sizes ($\hat{A}_{12}$), we utilize the non-parametric Wilcoxon Rank Sum test, with $\alpha = 0.01$ for the Type I error.

A replication package of this study has been uploaded to Zenodo [65].

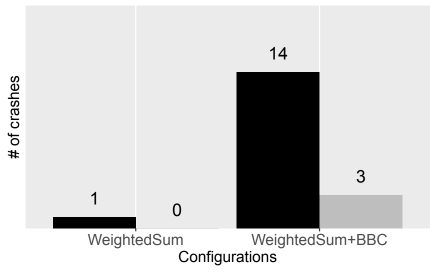
5.4 Results

Crash reproduction effectiveness (RQ$_1$)

Figure 5.2 presents the crash reproduction ratio of the search processes guided by *STDistance* (Figure 5.2a) and *Crash Distance* (Figure 5.2b), with and without *BBC* as a secondary objective. This figure shows that the crash reproduction ratio of *Crash Distance* improves slightly when using *BBC*. However, on average, the crash reproduction ratio achieved by *STDistance + BBC* is 4% better than *STDistance* without *BBC*. Also, the lower quartile of



(a) *STDistance*



(b) *Crash Distance*

Figure 5.4: Pairwise comparison of impact of *BBC* on each fitness function in terms of efficiency with a small, medium, and large effect size $\hat{A}_{12} < 0.5$ and a statistical significance < 0.01.

crash reproduction ratio using *STDistance* has been improved by about 30% by utilizing *BBC*.

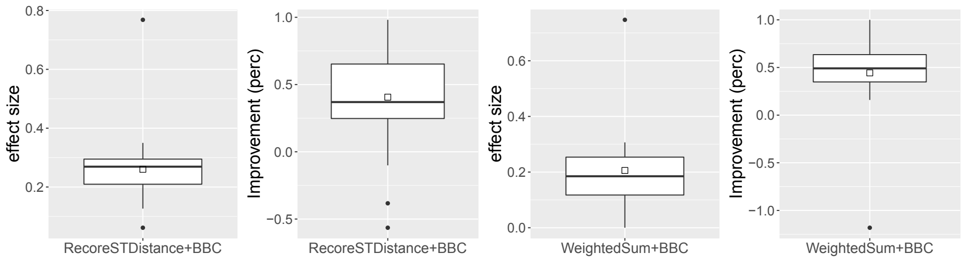
Figure 5.3 depicts the number of crashes, for which *BBC* has a significant impact on the effectiveness of crash reproduction guided by *STDistance* (Figure 5.3a) and *Crash Distance* (Figure 5.3b). *BBC* significantly improves the crash reproduction ratio in 5 and 1 crashes for fitness functions *STDistance* and *Crash Distance*, respectively. Importantly, the application of this secondary objective does not have any significant negative effect on crash reproduction. Also, *BBC* helps *STDistance* and *Crash Distance* to reproduce 6 and 1 new crashes, respectively (in at least one out of 30 runs), that could not be reproduced without *BBC*.

Summary. *BBC* slightly improves the crash reproduction ratio when using the *Crash Distance* fitness function. However, on average, *BBC* achieves a higher improvement when used as a secondary objective with the *STDistance* function.

Crash reproduction efficiency (RQ$_2$)

Figure 5.4 illustrates the number of crashes, in which *BBC* significantly affects the time consumed by the crash reproduction search process. As Figure 5.4b shows, *BBC* significantly improves the speed of crash reproduction guided by *Crash Distance* in 17 crashes (13.7% of cases), while it lost efficiency in the reproduction of only one crash. In cases that *BBC* significantly improves the efficiency of *Crash Distance*, on average, the efficiency is improved for about 40%. Moreover, Figure 5.4a shows that *BBC* has a higher positive impact on the efficiency of the search process guided by *STDistance*. It significantly reduces the time consumed by the search process in 33 crashes (26.6% of cases), while it had an adverse impact on the reproduction efficiency of only one crash. In cases that *BBC* significantly improves the efficiency of *STDistance*, on average, the efficiency is improved for about 53%.

Figure 5.5 depicts the average improvements in the efficiency and effect sizes for crashes where the difference in the consumed budget when using *BBC* or not was significant. According to the right-side plot in Figure 5.5a, *BBC* reduces the time consumed by the search process guided by *STDistance* up to 98% (being 40.6% on average). Also, the



(a) *STDistance* (b) *Crash Distance*

Figure 5.5: The effect size and the average improvement achieved by *BBC* on each of the fitness functions in cases that *BBC* makes a significant difference in terms of efficiency.

left-side plot indicates that the average effect size of differences between *STDistance* and *STDistance* +*BBC* (calculated by Vargha-Delaney) is 0.26 (lower than 0.5 indicates that *BBC* improved the efficiency). Figure 5.5b shows that the average improvement (right-side plot) achieved by using *BBC* as the second objective of *Crash Distance* is 44.3%, and the average effect size (left-side plot), in terms of the crash reproduction efficiency, is 20.5.

Summary. *BBC* improves the efficiency of the search process with both of the crash reproduction fitness functions.

5.5 Discussion

Generally, using *BBC* as secondary objective leads to a better crash reproduction ratio and higher efficiency in search-based crash reproduction. This improvement is achieved thanks to the additional ability to guide the search process when facing implicit branches during the search. Combining *BBC* with *STDistance* shows an important improvement compared to the combination of *BBC* with *Crash Distance*. This result was expected, since only one (out of three) component in *Crash Distance* is allocated to line coverage, and thereby most parts of the fitness function do not use the approach level and branch distance heuristics. In contrast, *STDistance* uses the approach level and branch distance to cover each of the frames in the given stack trace incrementally.

Our results show that *BBC* helps the crash reproduction process to reproduce new crashes. For instance, the crash that we used in this study (XWIKI-13377) can be reproduced only by *STDistance* + *BBC*. Considering our results, we believe that the usage of approach level and branch distance can be improved in other areas of search-based test generation (*e.g.,* unit testing) by taking the *implicit branches* into account. However, it can be expensive to apply this secondary objective in cases where the search process tries to cover multiple paths. Assessing the impact of *BBC* on other search-based test generation techniques is part of our future research agenda.

5.6 Threats to validity.

We cannot guarantee that our implementation of Botsing is bug-free. However, we mitigated this threat by testing our tool and manually examining some samples of the results. We cannot ensure that our results are generalizable to all crashes. However, we used an earlier established benchmark for crash reproduction containing 124 hard-to-reproduce crashes provoked by real bugs in a variety of open-source applications. Moreover, by following the guidelines of the related literature [110], we executed each configuration 30 times to take the randomness of the search process into account. Finally, we provide Botsing as an open-source tool. Also, the data and the processing scripts used to present the results are available as a replication package on Zenodo[65].

5.7 Conclusion and Future Work

Approach level and branch distance are two well-known heuristics, widely used by search-based test generation approaches to guide the search process towards covering target statements and branches. These heuristics measure the distance of a generated tests from covering the target using the coverage of control dependencies. However, these two heuristics do not consider implicit branches. For instance, if a test throws an exception during the execution of a non-branch statement, approach level and branch distance cannot guide the search process to tackle this exception. In this chapter, we introduced a secondary objective called *BBC* to address this issue. To assess *BBC*, we used it for search-based crash reproduction due to the high chance of implicit branch occurrence and the limited number of basic blocks that should be covered. Our results show that *BBC* helps *STDistance* and *Crash Distance* to reproduce 6 and 1 new crashes, respectively. Also, *BBC* significantly improves the efficiency in 26.6% and 13.7% of the crashes using *STDistance* and *Crash Distance*, respectively.

In our future work, we will investigate the application of *BBC* for other search-based test generation techniques (such as unit and integration).

6 Generating Class-Level Integration Tests Using Call Site Information

Search-based approaches have been applied to a variety of white-box testing activities [135], among which test case and data generation [4]. In white-box testing, most of the existing work has focused on the unit level, where the goal is to generate tests that achieve high structural (e.g., branch) coverage. Prior work has shown that search-based unit test generation can achieve high code coverage [20, 25, 194], detect real-bugs [22, 192], and help developers during debugging activities [23, 37].

Despite these undeniable results, researchers have identified various limitations of the generated unit tests [22, 26, 195]. Prior studies have questioned the effectiveness of the generated unit tests with high code coverage in terms of their capability to detect real faults or to kill mutants when using mutation coverage. For example, Gay *et al.* [26] have highlighted how traditional code coverage could be a poor indicator of test effectiveness (in terms of fault detection rate and mutation score). Shamshiri *et al.* [22] have reported that around 50% of faults remain undetected when relying on generated tests with high coverage. Similar results have also been observed for large industrial systems [25].

Gay *et al.* [26] have observed that traditional unit-level adequacy criteria only measure whether certain code elements are reached, but not *how* each element is covered. The quality of the test data and the paths from the covered element to the assertion play an essential role in better test effectiveness. As such, they have advocated the need for more reliable adequacy criteria for test case generation tools. While these results hold for generated unit tests, other studies on hand-written unit tests have further highlighted the limitation of unit-level code coverage criteria [195, 196].

In this chapter, we explore the usage of the integration code between coupled classes as guidance for the test generation process. The idea is that, by exercising the behavior of a class under test E (the calleE) through another class R (the calleR) calling its methods, R will handle the creation of complex parameter values and exercise valid usages of E. In other words, the caller R contains integration code that (1) enables the creation of better test data for the callee E, and (2) allows to better validate the data returned by E.

Integration testing can be approached from many different angles [197, 198]. Among others, *dataflow analysis* seeks to identify possible interactions between the definition and usage (def-use) of a variable. Various coverage criteria based on intra- (for class unit testing) and inter-class (for class integration testing) def-uses have been defined over the years [199–204]. Dataflow analysis faces several challenges, including the scalability of the algorithms to identify def-use pairs [205] and the number of test objectives that is much larger for dataflow criteria compared to *control flow* ones like branch and branch pair coverage [199, 204].

In our case, we focus on **class integration testing** between a caller and a callee [206]. Class integration testing aims to assess whether two or more classes work together properly by thoroughly testing their interactions [206]. Our idea is to complement unit test generation for a class under test by looking at its integration with other classes using **control flow analysis**. To that end, we define a novel structural adequacy criterion that we call **Coupled Branches Coverage** (CBC), targeting specific integration points between two classes. Coupled branches are pairs of branches $\langle r, e \rangle$, with r a branch of the caller, and e a branch of the callee, such that an integration test that exercises branch r also exercises branch e.

Furthermore, we implement a search-based approach that generates integration-level test suites leveraging the CBC criterion. We coin our approach CLING (for class integration testing). CLING uses a state-of-the-art many-objective solver that generates test suites maximizing the number of covered coupled branches. For the guidance, CLING uses novel search heuristics defined for each pair of coupled branches (the search objectives).

We conducted an empirical study on 140 well-distributed (in terms of complexity and coupling) pairs of caller and callee classes extracted from five open-source Java projects. Our results show that CLING can achieve up to 99% CBC scores, with an average CBC coverage of 49% across all classes. We analyzed the benefits of the integration-level test cases generated by CLING compared to unit-level tests generated by EVOSUITE [6], the state-of-the-art generator of unit-level tests, and RANDOOP [207], a random-based unit test case generator. In particular, we assess whether integration-level tests generated by CLING can kill mutants and detect faults that would remain uncovered when relying on generated unit tests.

According to our results, on average, CLING kills 7.7% (resp. 13.5%) of mutants per class that remain undetected by unit tests generated using EVOSUITE (resp. RANDOOP) for both the caller and the callee. The improvements in mutation score are up to 50% for certain classes. Our analysis indicates that many of the most frequently killed mutants are produced by integration-level mutation operators. Finally, we have found 27 integration faults (*i.e.,* faults due to wrong assumptions about the usage of the callee class) that were detected only by the integration tests generated with CLING (and not through unit testing with EVOSUITE or RANDOOP).

The remainder of the chapter is organized as follows. Section 6.1 summarizes the background and related work in the area. Section 6.2 defines the Coupled Branches Criteria and introduces CLING, our integration-level test case generator. Section 6.3 describes our empirical study, while Section 6.4 reports the empirical results. Section 6.5 discusses the practical implication of our results. Section 6.6 discusses the threats to validity. Finally, Section 6.7 concludes the chapter.

6.1 Background And Related Work

McMinn [4] defined search-based software testing (SBST) as *"using a meta-heuristic optimizing search technique, such as a genetic algorithm, to automate or partially automate a testing task"*. Within this realm, test data generation at different testing levels (such as *unit testing*, *integration testing*, *etc.*) has been actively investigated [4]. This section provides an overview of earlier work in this area.

6.1.1 Search-based Approaches For Unit Testing

In Section 1.1.1 we have explained how the general structural-based unit testing approaches work. We also showed an example with Listing 1.1 to demonstrate how these techniques produce CFGs of the class under test and use two heuristics (called *approach level* and *branch distance*) to achieve a high line and branch coverage.

In search-based unit testing, each generated test case is a sequence of method calls to a target class. This call sequence can be generated randomly, or it can be generated using existing resources. Goffi *et al.* [208] leverage existing documentation in this process, but for various reasons it does not allow to detect all bugs [209–211].

Rojas *et al.* [124] collect the usages of classes in the existing test cases to generate the call sequences. To reach that goal, they need to execute each of the existing tests to find the call sequences; this may be a time taking process.

In this chapter, we analyze how a class is used/invoked by the other classes within the same system. For this purpose, we merge the Class-level Control Flow Graph (CCFG) of target callee and caller classes.

6.1.2 Search-Based Approaches For Integration Testing

Integration testing aims at finding faults that are related to the interaction between components. We discuss existing integration testing criteria and explain the search-based approaches that use these criteria to define fitness functions for automating integration-level testing tasks.

Integration testing criteria

Jin *et al.* [197] categorize the connections between two procedures into four types: *call couplings* (type 1) occur when one procedure calls another procedure; *parameter couplings* (type 2) happen when a procedure passes a parameter to another procedure; *shared data couplings* (type 3) occur when two procedures refer to the same data objects; *external device coupling* (type 4) happens when two procedures access the same storage device. They introduce integration testing criteria according to the data flow graph (containing the definitions and usages of variables at the integration points) of procedure-based software. Their criteria, called *coupling-based testing criteria*, require that the tests' execution paths cover the last definition of a parameter's value in the CFG of a procedure (the *caller procedure*), a node (the *call site*) calling another procedure with that parameter, and the first use of the parameter in the *callee* (and in the caller after the call if the parameter is a call-by-reference).

Harrold *et al.* [200] introduced data flow testing for a single class focusing on method-integration testing. They define three levels of testing: *intra-method testing*, which tests an individual method (*i.e.,* the smallest possible unit to test); *inter-method testing*, in which

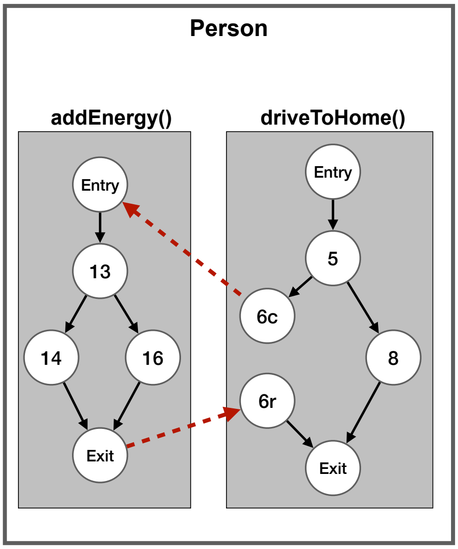


Figure 6.1: Class-level CFG for class `Person`

a public method is tested that (in)directly calls other methods of the same class, and *intra-class testing*, in which the various sequences of public methods in a class are tested. For data flow testing of inter-method and intra-class testing, they defined a *Class-level Control Flow Graph* (CCFG). The CCFG of class C is a directed graph $CCFG_C = (N_{Cm}, E_{Cm})$ which is a composition of the control flow graphs of methods in C; the CFGs are connected through their call sites to methods in the same class [200]. This graph demonstrates all paths that might be crossed within the class by calling its methods or constructors.

Let us consider again the class *Person* in Listing 1.1. The CCFG of class *Person* is created by merging the CFGs of its method, as demonstrated in Figure 6.3. For example, in the CFG of the method `Person.driveToHome()`, the *node 6c* is a call site to `Person.addEnergy()`. In the approach introduced by Harrold *et al.* [200], they detect the def-use paths in the constructed CCFGs and try to cover those paths.

Denaro *et al.* [205] revisited previous work on data flow analysis for object-oriented programs [200, 201] to define an efficient approach to compute *contextual def-use* coverage [201] for class integration testing. The approach relies on *contextual data flow analysis* to take state-dependent behavior of classes that aggregate other classes into account. Compared to def-use paths, contextual def-use include the chain of method calls leading to the definition or the use.

A special case is represented by the polymorphic interactions that need to be tested. Alexander *et al.* [202, 203] used the data flow graph to define testing criteria for integrations between classes in the same hierarchy tree.

All of the mentioned approaches are using data-flow analysis to define integration testing criteria. However, generating data-flow graphs covering the def-uses involved

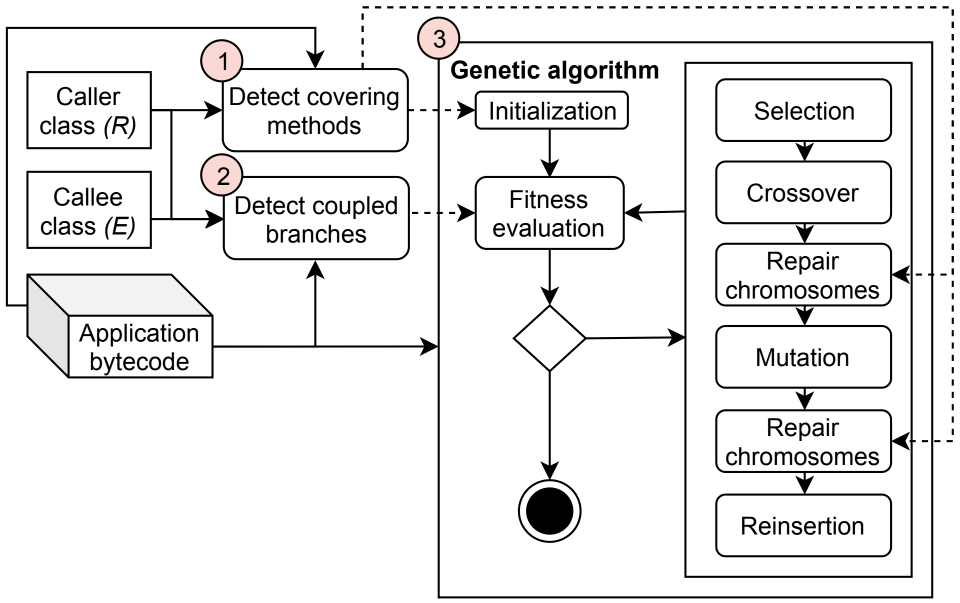


Figure 6.2: General overview of CLING

in between classes is expensive and not scalable in complex cases [199]. Vivanti *et al.* [204] shows that the average number of def-use paths in a single class in isolation is three times more than the number of branches. By adding def-use paths between the non-trivial classes, this number grows exponentially.

In search-based approaches, the number of search objectives matters as too many objectives leads to the search process misguidance. Compared to previous work, our approach does not try to cover def-use paths. Instead, we use a *control flow analysis* to identify from a CCFG a restricted number of pairs of branches (in a caller and a callee) that are not trivially executed together. For instance, the couple of branches $\langle 13, 16\rangle$ and $\langle b8, b9\rangle$ in Figure 6.3. Those pairs of branches are then used to define the search objectives of our test case generator. Section 6.2 details the analysis of the CCFG to identify such pairs of branches, including for special cases of interaction (namely inheritance and polymorphism), and the definition of the objectives and search algorithm.

CCFGs have been used previously for other usages. For instance, Wang *et al.* [212] merge the CFGs of methods of classes in the dependencies of software under test to identify the dependency conflicts.

6.2 Class Integration Testing

The main idea of our class integration testing (hereinafter referred to as CLING) is to test the integration of two classes by leveraging the usage of one class by another class. More specifically, we focus on the calls between the former, the callee (E), and the latter, the caller (R). By doing so, we benefit from the additional context setup by R before calling E

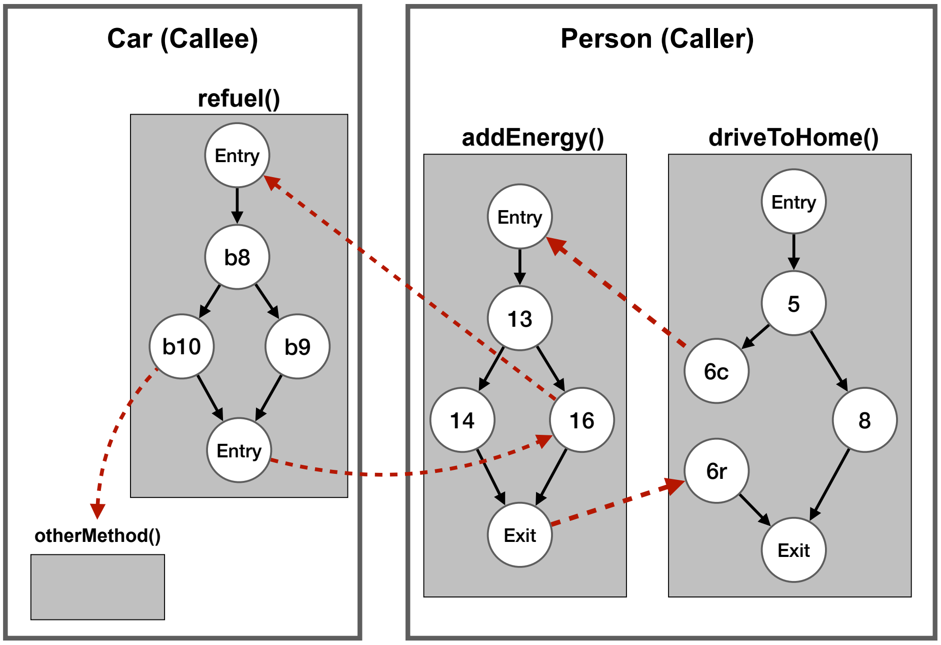


Figure 6.3: Merging CCFGs of two classes: `Person` (caller) and `Car` (callee)

(*e.g.*, initializing a complex input parameter), and the additional post-processing after E returns (*e.g.*, using the return value later on in R), thus (implicitly) making assumptions on the behavior of E.

Figure 6.2 presents the general overview of CLING. CLING takes as input a pair of caller-callee $\langle R, E \rangle$ classes with at least one call (denoted **call site** hereafter) from R to E. Since the goal of CLING is to generate test cases covering E by calling methods in R, the first step (①) statically collects the list of *covering methods* in R that, when called, may directly or indirectly cover statements in E. This list is later used during the generation process to ensure that test cases contain calls to covering methods. The second step (②) statically analyzes the CCFGs of R and E to identify the coupled branches between R and E used later on to guide the search. The CCFGs are statically built from the CFGs of the methods (including inherited ones) in R and E. Finally, the generation of the test cases (③) uses a genetic algorithm with two additional *repair* steps, ensuring that the crossover and mutation only produce test cases able to cover lines in E. The result is a test suite for E, whose test cases invokes methods in R that cover the interactions between R and E.

The remainder of this section describes our novel underlying Coupled Branches Criterion, the corresponding search-heuristics, and test case generation in CLING.

6.2.1 Coupled Branch Testing Criterion

To test the integration between two classes E and R, we need to define a coverage criterion that helps us to measure how thoroughly a test suite T exercises the interaction calls between the two classes (E and R). One possible coverage criterion would consist of testing all possible paths (*inter-class path coverage*) that start from the entry node of the caller R, execute the integration calls to E and terminate in one of the exit points of R. However, such a criterion will be affected by the *path explosion problem* [213]: the number of paths increases exponentially with the cyclomatic complexity of E and R, and thus the number of interaction calls between the two classes.

To avoid the *path explosion problem*, we define an integration-level coverage criterion, namely the Coupled Branch Criterion (CBC), where the number of coverage targets remains polynomial to the cyclomatic complexity of E and R. More precisely, CBC focuses on call coupling between caller and callee classes. Intuitively, let $s \in R$ be a call site, i.e., a call statement to a method of the class E. Our criterion requires to cover all pairs of branches (b_r, b_e), where b_r is a branch in R that leads to s (the method call), and b_e is a branch of the callee E that is not trivially covered by every execution of E. So, in the worst case, the number of coverage targets is quadratic in number of branches in the caller and callee classes.

Target caller branches

Among all branches in the caller class, we are interested in covering the branches that are not trivially (always) executed, and they always lead to the integration call site (*i.e.,* calling the callee class) when covered. We refer to these branches as *target branches* for the caller.

Definition 6.2.1 (Target branches for the caller) *For a call site s in R, the set of target branches $B_R(s)$ for the caller R contains the branches having the following characteristics: (i) the branches are outgoing edges for the node on which s is control dependent (i.e., nodes for which s post-dominates one of its outgoing branches but does not post-dominate the node itself); and (ii) the branches are post-dominated by s, i.e., branches for which all the paths through the branch to the exit point pass through s.*

To understand how we determine the target branches in the caller, let us consider the example of the caller and the callee in Figure 6.3. The code for the class `Person` is reported in Listing 1.1. The class `Person` contains two methods, `addEnergy()` and `driveToHome()`, with the latter invoking the former (line 6 in Listing 1.1). The method `Person.addEnergy()` invokes the method `refuel()` of the class `Car` (line 16 in Listing 1.1). The method `Person.d-riveToHome()` invokes the method `Car.drive()` (line 8 in Listing 1.1). Therefore, the class *Person* is the caller, while `Car` is the callee.

Figure 6.3 shows an excerpt of the Class-level Control Flow Graphs (CCFGs) for the two classes. In the figure, the names of the nodes are labelled with the line number of the corresponding statements in the code of Listing 1.1. Node 16 in `Person.addEnergy()` is a call site to `Car.refuel()`; it is also control dependent on nodes 5 (`Person.driveTo-Home()`) and 13 (`Person.addEnergy()`). Furthermore, node 16 only post-dominates branch $\langle 13, 16\rangle$. Instead, the branch $\langle 5, 6c\rangle$ is not post-dominated by node 16 as covering $\langle 5, 6c\rangle$ does not always imply covering node 16 as well. Therefore, the branches in the caller `Person.addEnergy()` that always lead to the callee are $B_{\texttt{Person}}(\texttt{Car.refuel()}) = \{\langle 13, 16\rangle\}$.

Hence, among all branches in the caller class (`Person` in our example), we are interested in covering the branches that, when executed, always lead to the integration call site (i.e., calling the callee class). We refer to these branches as *target branches* for the caller.

Target callee branches

Like the target branches of the caller, the target branches of the callee are branches that are not trivially (always) executed each time the method is called.

Definition 6.2.2 (Target branches for the callee) *The set of target branches $B_E(s)$ for the callee E contains branches satisfying the following properties: (i) the branches are among the outgoing branches of branching nodes (i.e., the nodes having more than one outgoing edge); and (ii) the branches are accessible from the entry node of the method called in s.*

Let us consider the example of Figure 6.3 again. This time, let us look at the branches in the callee (`Car`) that are directly related to the integration call. In the example, executing the method call `Car.refuel()` (node 16 of the method `Person.addEnergy()`) leads to the execution of the branching node $b8$ of the class `Car`. Hence, the set of branches affected by the interaction calls is $B_{\texttt{Car}}(\texttt{Car.refu-el}()) = \{\langle b8, b9\rangle; \langle b8, b10\rangle\}$. In the following, we refer to these branches as *target branches* for the callee. Note that, for a call site s in R calling E, the set of target branches for the callee also includes branches that are trivially executed by any execution of s.

Coupled branches

Given the sets of target branches for both the caller and callee, an integration test case should exercise at least one target branch for the caller (branch affecting the integration call) and one target branch for the callee (i.e., the integration call should lead to covering branches in the callee). In the following, we define pairs of target branches $(b_r \in B_R(s), b_e \in B_E(s))$ as *coupled branches* because covering b_r can lead to covering b_e as well.

Definition 6.2.3 (Coupled branches) *Let $B_R(s)$ be the set of target branches in the caller class R; let $B_E(s)$ be the set of target branches in the callee class E; and let s be the call site in R to the methods of E. The set of coupled branches $CB_{R,E}(s)$ is the cartesian product of $B_R(s)$ and $B_E(s)$:*

$$CB_{R,E}(s) = CB_{R,E}(s) = B_R(s) \times B_E(s) \tag{6.1}$$

In our example of Figure 6.3, we have two coupled branches: the branches $(\langle 13, 16\rangle, \langle b8, b9\rangle)$ and the branches $(\langle 13, 16\rangle, \langle b8, b10\rangle)$.

Definition 6.2.4 (Set of coupled branches) *Let $S = (s_1, \ldots, s_k)$ be the list of call sites from a caller R to a callee E, the set of coupled branches for R and E is the union of the coupled branches for the different call sites S:*

$$CB_{R,E} = \cup_{s \in S} CB_{R,E}(s)$$

Example 6.1: Class GreenPerson

```
1  Class GreenPerson extends Person{
2      private HybridCar car = new HybridCar();
3      @override
4      public void addEnergy(){
5          if(this.lazy){
6              takeBus();
7          }else if (chargerAvailable()){
8              car.recharge()
9          }else{
10             car.refuel();
11         }
12     }
13
14     private void chargerAvailable(){
15         if(ChargingStation.takeavailableStations().size > 0){
16             return true;
17         }
18         return false;
19     }
20 }
```

Coupled Branches Criterion (CBC)

Based on the definition above, the CBC criterion requires that for all the call sites S from a caller R to a callee E, a given test suite T covers all the coupled branches:

$$CBC_{R,E} = \frac{|\{(r_i, e_i) \in CB_{R,E} | \exists t \in T : t \ covers \ r_i \ and \ e_i\}|}{|CB_{R,E}|}$$

We do note that this formula is only relevant if there are indeed call interactions between caller and callee. As for classical branch and branch-pair coverage, $CB_{R,E}$ may contain unreachable branch-pairs. However, detecting and filtering those incompatible pairs is an undecidable problem. Hence, in this study, we target all coupled branches.

Inheritance and polymorphism

In the special case where the caller and callee classes are in the same inheritance tree, we use a different procedure to build the CCFG of the super-class and find the call sites S. The CCFG of the super-class is built by merging the CFGs of the methods that are not overridden by the sub-class. As previously, the CCFG of the sub-class is built by merging the CFGs of the methods defined in this class, including the inherited methods overridden by the sub-class (other non-overridden inherited methods are not part of the CCFG of the sub-class).

For instance, the class `GreenPerson` in Listing 6.1, representing owners of hybrid cars, extends class `Person` from Listing 1.1. For adding energy, a green person can either refuel or recharge her car (lines 7 to 11). `GreenPerson` overrides the method `Person.addEnergy()` and defines an additional method `GreenPerson.chargerAvailable()` indicating whether the charging station is available. Only those two methods are used in the CCFG of the class `GreenPerson` presented in Figure 6.4, inherited methods are not included in the CCFG; the CCFG of the super-class `Person` does not contain the method `Person.addEnergy()`, redefined by the sub-class `GreenPerson`.

The call sites S are identified according to the CCFGs, depending on the caller and the callee. If the caller R is the super-class, S will contain all the calls in R to methods that have

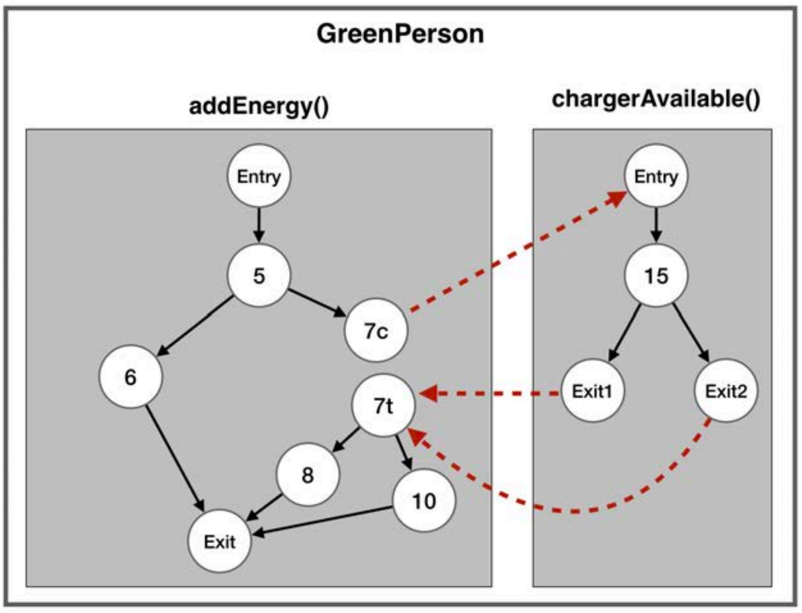


Figure 6.4: CCFG of *GreenPerson* as subclass

been redefined by the sub-class. For instance, nodes 6 and 13 in Figure 6.3 with `Person` as caller. If the caller R is the sub-class, S will contain all the calls in R to methods that have been inherited but not redefined by R. For instance, node 6 in Figure 6.4 with `GreenPerson` as caller.

6.2.2 Cling

Cling is the tool that we have developed to generate integration-level test suites that maximize the proposed CBC adequacy criterion. The inputs of Cling are the (1) *application's bytecode*, (2) a *caller class R*, and (3) *callee class E*. As presented in Figure 6.2, Cling first detects the covering methods (step ①) and identifies the coupled branches $CB_{R,E}(s)$ for the different call sites (step ②), before starting the search-based test case generation process (detailed in the following subsections). Cling produces a test suite that maximizes the CBC criterion for R and E.

Satisfying the CBC criterion is essentially a many-objective problem where integration-level test cases have to cover pairs of coupled branches separately. In other words, each pair of coupled branches corresponds to a search objective to optimize. The next subsection describes our search objectives.

Search objectives

In our approach, each objective function measures the distance of a generated test from covering one of the coupled branch pairs. The value ranges between $[0, +\infty)$ (zero denoting that the objective is satisfied). Assuming that $CB_{R,E} = \{c_1, c_2, \ldots, c_n\}$ is the set of coupled branches $\langle r_i, e_i \rangle$ between R and E. Then, the fitness for a test case t is defined as follows:

$$Objectives = \begin{cases} d(c_1, t) = D(r_1, t) \oplus D(e_1, t) \\ \ldots \\ d(c_n, t) = D(r_n, t) \oplus D(e_n, t) \end{cases} \tag{6.2}$$

where $D(b, t) = al(b, t) + bd(b, t)$ computes the distance between the test t to the branch b using the classical approach level $al(b, t)$ (*i.e.,* the minimum number of control dependencies between b and the execution path of t) and normalized branch distance $bd(b, t)$ (*i.e.,* the distance, computed based on a set of rules, to the branch leading to b in the closest node on the execution path of t) [4]; and $D(r_i, t) \oplus D(e_i, t)$ is defined as $D(r_i, t) + 1$ if $D(r_i, t) > 0$ (*i.e.,* the caller branch is not covered) and $D(e_i, t)$ otherwise (*i.e.,* the caller branch is covered).

Search algorithm

To solve such a many-objective problem, we tailored the Many-Objective Sorting Algorithm (MOSA) [214] to generate test cases through class integration. MOSA has been introduced and assessed in the context of unit test generation [214] and security testing [215]. Besides, previous studies [35, 214] showed that MOSA is very competitive compared with alternative algorithms when handling hundreds and thousands of testing objectives. Interested readers can find more details about the original MOSA algorithm in Panichella *et al.* [214]. Although a more efficient variant of MOSA has been recently proposed [42], such a variant (DynaMOSA) requires to have a hierarchy of dependencies between coverage targets that exists only at the unit level. Since targets in unit testing are all available in the same control flow graph, the dependencies between objectives can be calculated (*i.e.,* control dependencies). In contrast, CLING's objective is covering combinations of targets in different control flow graphs. Since covering one combination does not depend on the coverage of another combination, DynaMOSA is not applicable to this problem.

Therefore, in CLING, we tailored MOSA to work at integration level, targeting pairs of coupled branches rather than unit-level coverage targets (e.g., statements). In the following, we describe the main modifications we applied to MOSA to generate integration-level test cases.

Initial population

The search process starts by generating an initial population of test cases. A random test case is a sequence of statements (*object instantiations*, *primitive statements*, *method calls*, and *constructor calls to the class under test*) of variable lengths. More precisely, the random test cases include *method calls* and *constructors* for the caller *R*, which directly or indirectly invoke methods of the callee *E* (*covering methods*). Although CLING generates these test cases randomly, it extends the initialization procedure used for search-based crash reproduction [36]. In particular, the initialization procedure in CLING gives a higher priority to methods in the caller class *R* that invoke methods of the callee class *E*. While calls to other methods of *R* are also inserted, their insertion has a lower probability. This prioritization ensures to generate tests covering call sites to the callee class. In the original MOSA algorithm, all methods of the class under test are inserted in each random test case with the same probability without any prioritization. The execution time of the initialisation procedure is part of the search budget.

Mutation and crossover

Cling uses the traditional single-point crossover and mutation operators (adding, changing and removing statements) [6] with an additional procedure to repair broken chromosomes. The initial test cases are guaranteed to contain at least one *covering method* (a method of R that directly or indirectly invokes methods of E). However, mutation and crossover can lead to generating *offspring* tests that do not include any *covering method*. We refer to these chromosomes as *broken chromosomes*. To fix the broken chromosomes, the *repair procedure* works in two different ways, depending on whether the broken chromosome is created by the crossover or by the mutation.

If the broken chromosome is the result of the mutation operator, then the repair procedure works as follows: let t be the broken chromosome and let M be the list of covering methods; then, Cling applies the mutation operator to t in an attempt to insert one of the covering methods in M. If the insertion is not successful, then the mutation operator is invoked again within a loop. The loop terminates when either a covering method is successfully injected in t or when the number of unsuccessful attempts is greater than a threshold (50 by default). In the latter case, t is not inserted in the new population for the next generation.

If the broken chromosome is generated by the crossover operator, then the broken child is replaced by one of its parents.

Polymorphism

If the caller and callee are in the same hierarchy and the caller is the super-class, Cling cannot generate tests for the caller class that will cover the callee class (since the methods to cover are not defined in the super-class). This is the case for instance if the super-class (caller) calls abstract methods defined in the sub-class (callee). In this particular case, Cling generates tests for the callee class. However, it selects the covering methods only from the inherited methods which are not overridden by the callee (sub-class). A covering method should be able to cover calls to the methods that have been redefined by the sub-class. With this slight change, Cling can improve the CBC coverage, as described in Section 6.2.1.

6.3 Empirical Evaluation

Our evaluation aims to answer three research questions. The first research question analyzes the levels of CBC coverage achieved by Cling. For this research question, we first analyze the coupled branches covered by Cling in each of the cases:

RQ1.1 *What is the CBC coverage achieved by Cling?*

As explained in Section 6.1.2, to the best of our knowledge, there is no class-integration test case generator available for comparison. We thus compare Cling to the state-of-the-art unit test generators in terms of CBC coverage:

RQ1.2 *How does the CBC coverage achieved by Cling compares to automatically generated unit-level tests?*

Since the test cases generated by Cling aim to cover coupled branches between two classes, we need to determine the effectiveness of this kind of coverage compared to test suites generated for high branch coverage in unit testing:

RQ2 *What is the effectiveness of the integration-level tests compared to unit-level tests?*

Finally, we want to see whether the tests generated by Cling can make any difference in practice. Hence, we analyzed the integration faults captured by these tests:

RQ3 *What integration faults does Cling detect?*

6.3.1 Implementation

We implemented Cling as an open-source tool written in Java.[1] The tool relies on the EvoSuite [6] library as an external dependency. It implements the code instrumentation for pairs of classes, builds the CCFGs at the byte-code level, and derives the coverage targets (pairs of branches) according to the CBC criterion introduced in Section 6.2.1. The tool also implements the search heuristics, which are applied to compute the objective scores as described in Section 6.2. Besides, Cling implements the repair procedure described in Section 6.2.2, which extends the interface of the genetic operators in EvoSuite. Moreover, we customized the many-objective MOSA algorithm [42], which is implemented in EvoSuite, for our test case generation problem in Cling.

Baseline Selection

The goal of this evaluation is to explore the impact of the tests generated by Cling on the results of the search-based unit testing in various aspects. To achieve this purpose, we run our tool against EvoSuite, which is currently the best tool in terms of achieving branch coverage [216–220], and Randoop [207], a random-based unit test case generator. We configured EvoSuite to use *DynaMOSA* (`-Dalgorithm=DynaMOSA`), which has the best outcome in structural and mutation coverage [42] and branch coverage (`-Dcriterion=BRANCH`). For Randoop, we used the default parameter values.

6.3.2 Study Setup

Subjects Selection

The subjects of our studies are five Java projects, namely *Closure compiler*, *Apache commons-lang*, *Apache commons-math*, *Mockito*, and *Joda-Time*. These projects have been used in prior studies to assess the coverage and the effectiveness of unit-level test case generation [22, 42, 70, 71], program repair [72, 73], fault localization [74, 75], and regression testing [76, 77].

To sample the classes under test, we first extract pairs of `caller` and `callee` classes (*i.e.,* pairs with interaction calls) in each project. Then, we remove pairs that contain trivial classes, *i.e.,* classes where the caller and callee methods have no decision point (*i.e.,* with cyclomatic complexity equal to one). This is because methods with no decision points can be covered with single method calls at the unit testing level. Note that similar filtering based on code complexity has been used and recommended in the related literature [42, 194, 218]. From the remaining pairs, we sampled 140 distinct pairs of classes from the five

[1]Available at `https://github.com/STAMP-project/botsing/tree/master/cling`

Table 6.1: Projects in our empirical study. # indicates the number of caller-callee pairs. `CC` indicates the cyclomatic complexity of the caller and callee classes. `Calls` indicates the number of calls from the caller to the callee. `Coupled branches` indicates the number of coupled branches.

Project	#	**Caller**		**Callee**		**Calls**		**Coupled branches**			
		$\overline{cc}$	σ	$\overline{cc}$	σ	$\overline{count}$	σ	min	$\overline{count}$	σ	max
closure	26	1,221.3	1,723.0	377.2	472.5	70.3	101.0	4	10,542	17,080	60,754
mockito	20	115.3	114.4	127.8	113.2	39.5	64.9	0	1,185	1,974	6,929
time	51	68.7	84.0	87.2	92.3	23.9	50.5	0	494	1,093	5,457
lang	18	145.0	177.8	235.3	242.7	12.4	14.6	2	409	598	1,826
math	25	79.2	88.4	57.5	64.4	18.8	34.5	2	294	613	2,682
All	140	301.1	859.5	160.6	257.7	32.4	62.8	0	2,412	8,294	60,754

projects in total, which offers a good balance between generalization (*i.e.,* the number of pairs to consider) and statistical power (*i.e.,* the number of executions of each tool against each class or pair of classes). We performed the sampling to have classes with a broad range of complexity and coupling. In our sampling procedure, each selected class pair includes either the classes with the highest cyclomatic complexity or the mosts coupled classes. The numbers of pairs selected from each project are reported in Table 6.1. The least and most complex classes in the selected class pairs have one and 5,034 branching nodes, respectively. Also, the caller class of the least and most coupled class pairs contain one and 453 call sites to the callee class, respectively. Each pair of caller and callee classes represents a target for CLING.

Our replication package[2] contains the list of class pairs sampled for our study, their detailed statistics (*i.e.,* cyclomatic complexity and the number of interaction calls), and the project versions.

Evaluation Procedure

To answer the research questions, we run CLING on each of the selected class pairs. For each class pair targeted with CLING, we run RANDOOP and EVOSUITE with the caller and the callee classes as target classes under test to compare the class integration test suite with unit level test suites for the individual classes. This results in having five test suites:

1. T_{CLING}, the integration-level test suite generated by CLING;
2. T_{RanR}, the unit-level test suite generated by RANDOOP for the caller;
3. T_{RanE}, the unit-level test suite generated by RANDOOP for the callee;
4. T_{EvoR}, the unit-level test suite generated by EVOSUITE for the caller;
5. T_{EvoE}, the unit-level test suite generated by EVOSUITE for the callee.

To address the random nature of the three tools, we repeat each run 20 times (140 pairs of classes × 5 executions × 20 repetitions = 140,000 executions). Moreover, each CLING run is configured with a search budget of five minutes, including two minutes of search initialization timeout. To allow a fair comparison, we run RANDOOP and EVOSUITE for five minutes on each caller/callee class, including default initialization timeout (14,000 × 5 minutes ≃ 48.6 days execution time for test case generation).

[2]`https://github.com/STAMP-project/Cling-application`

For **RQ1**, we analyzed the average (median) CBC coverage scores achieved by T_{Cling} and compared them with the CBC coverages of T_{RanR} and T_{EvoR} across the 20 independent runs.

For **RQ2**, we measure the effectiveness of the generated test suite using both *line coverage* and *mutation analysis* on the callee classes E (considered as the class under test in our approach). Mutation analysis is a high-end coverage criterion, and mutants are often used as substitutes for real faults since previous studies highlighted its significant correlation with fault-detection capability [221, 222]. Besides, mutation analysis provides a better measure of the test effectiveness compared to more traditional coverage criteria [196] (*e.g.,* branch coverage).

We compute the line coverage and mutation scores achieved by T_{CLING} for the callee class in each target class pair. Then, we compare them to the line coverage and mutation scores achieved by the unit-level test suites (T_{RanR}, T_{RanE}, T_{EvoR}, and T_{EvoE}) for the callee class. Moreover, we analyze the orthogonality of the sets of mutants in the callee that are strongly killed by T_{CLING}, and those killed by the unit-level tests individually. In other words, we look at whether T_{CLING} allows killing mutants that are not killed at unit-level (strong mutation). Also, we analyze the type of the mutants which are only killed by T_{CLING}.

For line coverage and mutation analysis, we use PIT [223], which is a state-of-the-art mutation testing tool for Java code, to mutate the callee classes. PIT also collects and reports the line coverage of the test suite on the original class before mutation. PIT has been used in literature to assess the effectiveness of test case generation tools [70, 217–220, 224], and it has also been applied in industry[3]. In our study, we use PIT v.1.4.9 with all mutation operators activated (*i.e.,* the `ALL` mutators group).

For **RQ3**, we analyze the exceptions triggered by both integration and unit-level test suites. In particular, we extract unexpected exceptions causing crashes, *i.e.,* exceptions that are triggered by the test suites but that are (i) not declared in the signature of the caller and callee methods using `throws` clauses, (ii) not caught by a `try-catch` blocks, and (iii) not documented in the `Javadoc` of the caller or callee classes. Then, we manually analyze unexpected exceptions that are triggered by the integration-level test cases (*i.e.,* by CLING), but not by the unit-level tests.

The test suites generated by CLING and EVOSUITE may contain **flaky tests**, *i.e.,* test cases that exhibit intermittent failures if executed with the same configuration. To detect and remove flaky tests, we ran each generated test suite five times. Then, we removed tests that fail in at least one of the independent runs. Hence, the test suites used to answer our three research questions likely do not contain flaky tests.

To keep the execution time (which includes test generation, flaky test detection, and mutation and coverage analysis) manageable, we used a cluster (with 20 CPU-cores, 384 GB memory, and 482 GB hard drive) to parallelize the execution for our evaluation (50 simultaneous executions). With this parallelization, the automated execution of the whole evaluation took about three days (one day for test generation and two days for flaky test detection and mutation and line coverage measurement).

[3]http://pitest.org/sky_experience/

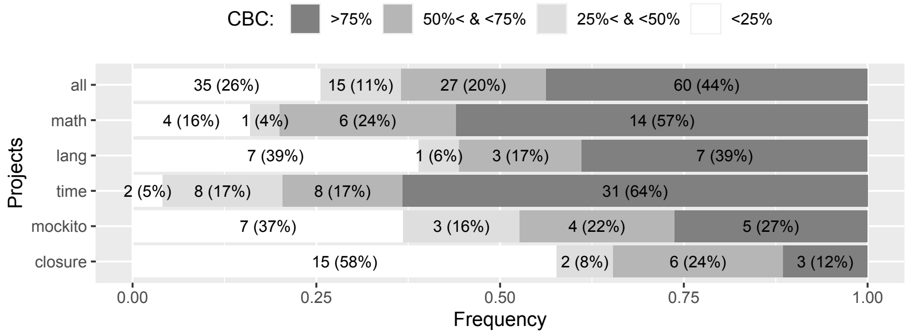


Figure 6.5: Distribution of CLING's CBC coverage for the different class pairs.

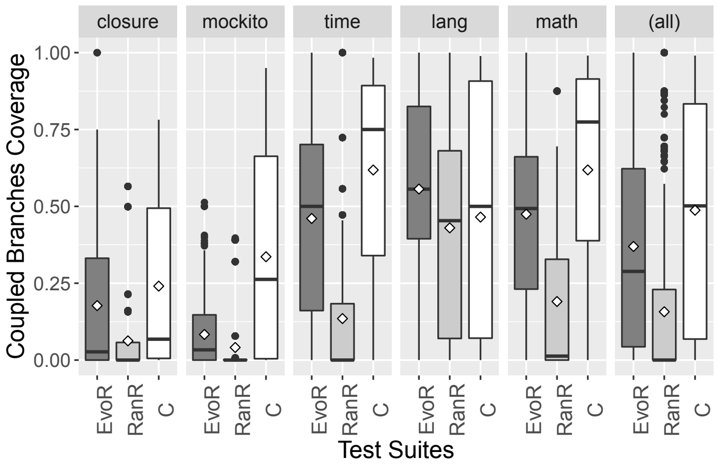


Figure 6.6: Total coupled-branches coverage achieved by T_{CLING} (C), T_{RanR} (RanR) and T_{EvoR} (EvoR). (◇) denotes the arithmetic mean and (—) is the median.

6.4 Evaluation Results

This section presents the results of the evaluation and answers the research questions.

6.4.1 CBC achieved by CLING (RQ1.1)

As reported in Table 6.1, CLING did not identify any coupled-branches for three pairs of classes (one in `mockito` and two in `time`). This is due to the absence of target branches in either the caller or the callee, resulting in no couple of branches to cover. Those three pairs have been excluded from the results presented in this section. Figure 6.5 gives the distribution of the CBC coverage achieved by CLING for the remaining 137 pairs of classes. In total, CLING could generate at least one test suite achieving a coupled-branches coverage of at least 50% for 87 out of 137 class pairs. Figure 6.6 presents the coupled-branches coverage of T_{CLING} in all of the projects. On average (the diamonds in Figure 6.6) the test

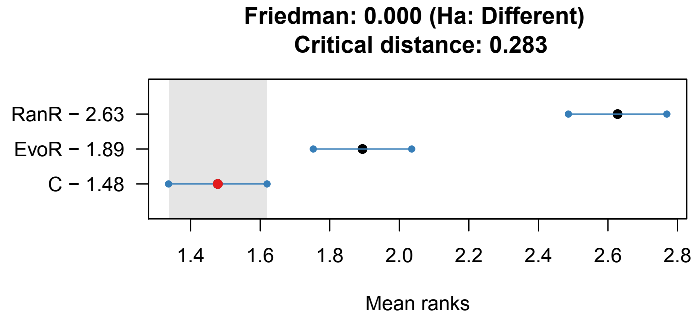


Figure 6.7: Non-parametric multiple comparisons in terms of CBC score for T_{Cling} (C), T_{EvoR} (EvoR), and T_{RanR} (RanR) using Friedman's test with Nemenyi's post-hoc procedure.

suites generated by Cling cover 48.7% of coupled-branches.

The most covered couples are in the `math` project (61.9% on average), followed by `time` (61.8% on average) and `lang` (46.5% on average). The least covered couples are in the `closure` (24% on average) and `mockito` projects (33.6% on average), which are also the projects with the highest number of coupled-branches in Table 6.1 (10,542 coupled-branches on average for all the class pairs in `closure` and 1,185 coupled-branches on average in `mockito`).

For 9 caller-callee pairs, Cling could not generate a test suite able to cover at least one coupled branch out of 20 executions: 3 pairs from `math`, 3 pairs from `mockito`, 2 pairs from `closure`, and 1 from `lang`. In the class pair from `lang`, Cling could not cover any coupled branch because the callee class (`StringUtils`) misleads the search process (we detail the explanation in Section 6.4.2). The remaining 8 pairs cannot be explained solely by the complexities of the caller (with a cyclomatic complexity ranging from 8 to 5,034 for those classes) and the callee (with a cyclomatic complexity ranging from 1 to 2,186) or the number of call sites (ranging from 1 to 177). This calls for a deeper understanding of the interactions between caller and callee around the call sites. In our future work, we plan to refine the caller-callee pair selection (for which we currently looked at the global complexity of the classes) to investigate the local complexity of the classes around the call sites.

Summary (RQ1.1). On average, the generated tests by Cling cover 48.7% of coupled-branches. In 87 out of 140 (59.2%) of the pairs, these test suites achieve a CBC higher than 50%.

6.4.2 CBC achieved by Cling vs. unit tests (RQ1.2)

Since T_{RanE} and T_{EvoE} cover only branches in the callee class (*i.e.*, it does not call any methods in the caller class), the coupled-branches coverages achieved by these tests are always zero. Hence, for this research question, we compare the tests generated by Cling (T_{Cling}) against the tests generated by Randoop and EvoSuite applied to the caller class (T_{RanR} and T_{EvoR}) w.r.t. coupled-branches coverage.

Figure 6.6 presents the coupled-branches coverage of T_{Cling}, T_{RanR}, and T_{EvoR} for all the projects. The number of covered coupled-branches by $T_{\textsc{Cling}}$ is higher in total (*all* in Figure 6.6). On average (the diamonds in Figure 6.6), the test suites generated by Cling (48.7%) cover more coupled-branches compared to 37% for T_{EvoR}, and 15.7% for T_{RanR}. On average, the coupled-branches coverage achieved by unit tests is lower than the one achieved by Cling in all of the projects except `lang`. The average coupled-branches coverage of EvoSuite in this project is 55.6%, compared to 48.9% for Cling. We also observe a wider distribution of the CBC coverage for $T_{\textsc{Cling}}$ (with a median of 51.0% and an IQR of 78.2%) compared to T_{EvoR} (with a median of 30.7% and an IQR of 59.0%) and T_{RanR} (with a median < 1.0% and an IQR of 25.0%).

We further compare the different test suites using Friedman's non-parametric test for repeated measurements with a significance level $\alpha = 0.05$ [225] . This test is used to test the significance of the differences between groups (treatments) over the dependent variable (CBC coverage in our case). We further complement the test for significance with Nemenyi's post-hoc procedure [226, 227]. Figure 6.7 provides a graphical representation of the ranking (*i.e.,* mean ranks with confidence interval) of the different test suites. According to the Friedman test, the different treatments (i.e., Cling, EvoSuite, and Randoop) achieve significantly different CBC coverage (p-values < 0.001). According to Figure 6.7, the average rank of Cling is much smaller than the average ranks of the two baselines. Furthermore, the differences between the average rank of $T_{\textsc{Cling}}$ and the average rank of the two baselines are larger than the critical distance $CD = 0.283$ determined by Nemenyi's post-hoc procedure. This indicates that $T_{\textsc{Cling}}$ achieves a significantly higher CBC coverage than T_{EvoR} and T_{RanR}.

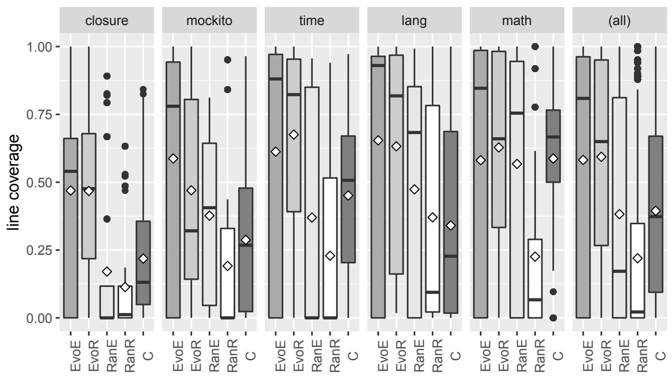
Finally, we have manually analyzed the search progress of Cling for pairs of classes where the number of covered coupled-branches is low (*i.e.,* lower than 10). We noticed that Cling is counter-productive for specific class pairs where the callee class is `StringUtils`. In those cases, the test cases generated during the search initialization throw a `NoSuchFieldError` in the callee class (`StringUtils` here). Since these test cases achieve small approach levels and branch distances from the callee branches, they are fitter (*i.e.,* their fitness value is lower) than other test cases. Therefore, these test cases are selected for the next generation and drive the search process in local optima.

Summary (RQ1.2). On average, the generated test suites by Cling cover 11.7% more coupled-branches compared to EvoSuite and 33% more coupled-branches compared to Randoop.

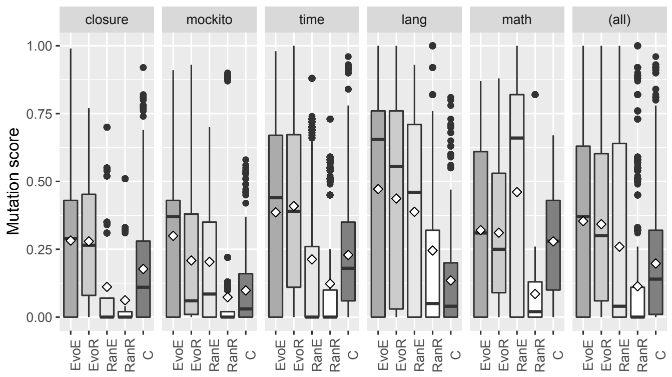
6.4.3 Line Coverage and Mutation Scores (RQ2)

Figure 6.8a shows line coverage of the callee classes (E) for the test suites generated by the different approaches. On average, Cling covers 39.5% of the lines of the callee classes. While this is lower compared to unit-level tests generated using EvoSuite (58.2% for T_{EvoE} and 59.4% for T_{EvoR}), Cling still achieves a better line coverage than Randoop (38.2% for T_{RanE} and 22% for T_{RanR}).

To understand the fault revealing capabilities of Cling compared to unit-level test suites, we first show in Figure 6.8b the overall mutation scores when mutating class E, and apply the test suite T_{EvoE}, T_{EvoR}, T_{RanE}, T_{RanR}, and $T_{\textsc{Cling}}$. Similar to line coverage, test suites optimized for overall branch coverage achieve a total higher mutation score (35.4% for T_{EvoE} and 34.2% for T_{EvoR} on average), simply because a mutant that is on a



(a) Line coverage of the callee (E)



(b) Mutation score of the callee (E)

Figure 6.8: Effectiveness of T_{Cling} (C), T_{EvoE} (EvoE), T_{EvoR} (EvoR), T_{RanE} (RanE), and T_{RanR} (RanR). (◇) denotes the arithmetic mean and (—) indicates the median.

line that is never executed cannot be killed. Randoop achieves on average a mutation score of 25.9% for T_{RanE} and 11.4% for T_{RanR}. T_{Cling} scores lower (20.0% on average), since Cling searches for dedicated interaction pairs, but does not try to optimize overall line coverage. Note that T_{Cling} achieves the highest average mutation score for classes in `math`, while it achieves the lowest mutation score for classes in the `mockito` project.

Our results are consistent with the design and objectives of the three tools: EvoSuite seeks to cover all the branches of the class under test; Cling targets specific pairs of branches between the caller and callee classes; and Randoop performs (feedback-directed) random testing. Our results also confirm previous observations that EvoSuite achieves a better structural coverage and mutation score than Randoop [216–220].

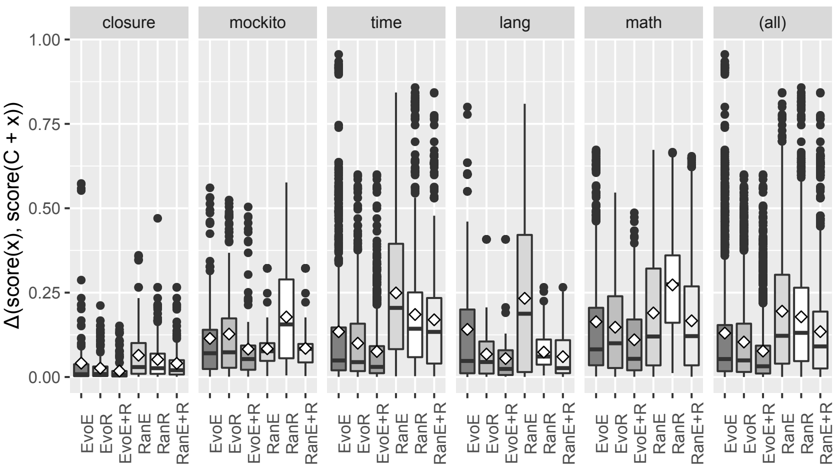


Figure 6.9: Increases (Δ) of the mutation score when combining T_{CLING} with unit test suites T_{EvoE}, T_{EvoR}, T_{RanE}, T_{RanR} and their unions $T_{EvoE+EvoR}$, and $T_{RanE+RanR}$. (⊠) denotes the arithmetic mean and (—) is the median.

Combined Mutation Analysis

Figure 6.8b shows that unit test suites do not kill almost half of the mutants. Cling targets more mutants, including those that remain alive with unit tests. In Figure 6.9, we report the *improvement* (Δ) in the mutation score when executing T_{CLING} in addition to unit test suites (T_{EvoE}, T_{EvoR}, T_{RanE}, and T_{RanE}), and their unions ($T_{EvoE+EvoR}$, $T_{RanE+RanR}$).

On average, 13%, resp. 19.5%, of the mutants are killed only by T_{CLING}, compared to T_{EvoE}, resp. T_{RanE}, the unit test suites optimized for the class under test (E). This difference decreases to 10.3%, resp. 17.8%, if we use T_{EvoR}, resp. T_{RanR}, the unit test suites exercising E via the caller class R (as more class interactions are executed). The difference with traditional unit testing is still 7.7%, resp. 13.5%, when comparing Cling with the combined unit test suites $T_{EvoE+EvoR}$, resp. $T_{RanE+RanR}$, exercising E directly as much as possible as well as indirectly via call sites in R.

The outliers in Figure 6.9 are also of interest: for 20, resp. 18, classes (out of 140), Cling was able to generate a test suite where more than half of the mutants were killed only by T_{CLING}, compared to T_{EvoE}, resp. T_{RanE}, (i.e., +50% of mutation score). When compared to $T_{EvoE+EvoR}$, resp. $T_{RanE+RanR}$, there are 4, resp. 9, classes for which T_{CLING} kills more than half of the mutants that are killed by neither T_{EvoE}, resp. T_{RanE}, nor T_{EvoR}, resp. T_{RanE}. This further emphasizes the complementarity between the unit and integration testing.

Table 6.2 presents the status of the mutants that are killed by T_{CLING} but not by unit-level test cases. What stands out is that many mutants are in fact covered, but not killed by unit-level test suites. Here, Cling leverages the context of caller, not only to reach a mutant, but also to propagate the (modified) values inside the caller's context, so that the mutants can be eventually killed.

Table 6.2: Status (for $T_E voR$, $T_E voE$, $T_R anR$, and $T_R anE$) of the mutants killed only by T_{CLING}. `Not-covered` denotes the number of mutants killed by T_{CLING}, which are not covered by EvoSuite (or Randoop) test suites, and `survived` denotes the number of mutants killed by T_{CLING}, which are covered by EvoSuite (or Randoop) tests but not killed. The numbers between parentheses denote the percentage of mutants.

Test Suite	**closure**		**lang**		**math**		**mockito**		**time**	
	not-covered	survived	not-covered	survived	not-covered	survived	not-covered	survived	not-covered	survived
T_{EvoE}	1,988 (<1%)	881 (<1%)	3,247 (1%)	403 (<1%)	6,178 (5%)	1,747 (1%)	5,604 (4%)	2,414 (2%)	10,905 (3%)	5,920 (1%)
T_{EvoR}	2,480 (<1%)	780 (<1%)	2,797 (<1%)	851 (<1%)	5,310 (4%)	2,558 (2%)	4,867 (4%)	3,144 (2%)	7,431 (2%)	9,150 (2%)
T_{RanE}	11,256 (1%)	1,002 (<1%)	1,862 (1%)	348 (<1%)	8,483 (7%)	1,117 (1%)	6,278 (5%)	2,387 (2%)	42,503 (10%)	7,124 (2%)
T_{RanR}	11,034 (1%)	1,172 (<1%)	1,384 (1%)	829 (<1%)	8,367 (7%)	1,238 (1%)	5,313 (4%)	3,352 (2%)	44,192 (10%)	5,545 (1%)

Mutation Operators

We analyzed the mutation operators that generate mutants that are exclusively killed by T_{CLING}. We categorize the mutation operators implemented in Pit into *integration-level* and *non-integration-level.* For this categorization, we rely on the definition of mutation operators for integration testing provided by Delamaro *et al.* [228]. We observed that ten of the mutation operators implemented in Pit inject integration-level faults. These operators can be mapped to two integration-level operators defined by Delamaro *et al.* [228]: *RetStaRep*, which replaces the return value of the called method, and *FunCalDel*, which removes the calls to void method calls and replaces the non-void method calls by a proper value.

Table 6.3 lists the number of mutants killed exclusively by T_{CLING} and grouped by mutation operators. Integration-level operators are indicated in bold with the mapping to either *RetStaRep* or *FunCalDel* between parenthesis. As we can see in this table, the most frequently killed mutants are produced by an integration-level operator, and other integration-level operators also produce frequently killed mutants. We can see that all of the ten integration-level mutation operators generate mutants that can be killed using Cling.

Furthermore, some of the most frequently killed mutants are not produced by integration-level operators. For instance, operator *NegateConditionalsMutator*, which mutates the conditions in the target class, produces the second most frequently killed mutants. These mutants are not killed but also not covered by tests generated by EvoSuite.

Lets look at an example of a mutant killed only by T_{CLING}. Figure 6.11 illustrates one of the mutants in method `evaluateStep` in class `SwitchState` (callee class) from the *Apache commons-math* project. This mutant is produced by an integration-level mutation operator (*RetStaRep*) that replaces a boolean return value by `true`. Method *evaluateStep* is called from the method `evaluateStepC` (Figure 6.10) declared in `SwitchingFunctionsHandler` (caller class). Method `evaluateStepC` must return false if it calls the callee class in a certain situation: (i) the variable `first` in the caller class is null, and (ii) the callee method returns false because of the execution of line 12 in Figure 6.11.

The unit test suites generated by EvoSuite targeting `SwitchState` (T_{EvoE}) or class `SwitchingFunctionsHandler` (T_{EvoR}) both cover the mutant but do not kill it. T_{EvoE} easily cover the mutant statement, but it does not have any assertion to check the return value. T_{EvoR} also covers this statement by calling the right method in `SwitchingFunctionsHandler`. However, as it depicted by Figure 6.11, both methods in caller and callee class have multiple branches. So, T_{EvoR} covers the mutant from another path, which does not reveal

Table 6.3: Number of mutants killed only by T_{CLING} and grouped by mutation operators. Integration-level operators are highlighted in **bold** face and the corresponding integration-level mutation operator defined by Delamaro *et al.* [228] is indicated between parenthesis.

Against	EvoSuite		Randoop	
Mutation operator	**Rank**	**#kills**	**Rank**	**#kills**
NonVoidMethodCallMutator (*RetStaRep*)	1	1,983	1	1,809
NegateConditionalsMutator	2	1,638	2	1,505
InlineConstantMutator	3	1,201	4	971
ReturnValsMutator (*RetStaRep*)	4	1,195	6	944
RemoveConditionalMutator_EQUAL_IF	5	1,110	3	1,063
RemoveConditionalMutator_EQUAL_ELSE	6	1,015	5	962
NullReturnValsMutator (*RetStaRep*)	7	578	8	428
ArgumentPropagationMutator (*FunCalDel*)	8	518	7	432
MathMutator	9	513	10	377
MemberVariableMutator	10	458	9	379
ConstructorCallMutator (*FunCalDel*)	11	379	11	344
RemoveConditionalMutator_ORDER_IF	12	375	13	278
VoidMethodCallMutator (*FunCalDel*)	13	374	12	321
RemoveConditionalMutator_ORDER_ELSE	14	348	14	248
ConditionalsBoundaryMutator	15	322	15	227
PrimitiveReturnsMutator (*RetStaRep*)	16	309	16	217
NakedReceiverMutator	17	264	18	174
IncrementsMutator	18	143	19	119
BooleanTrueReturnValsMutator (*RetStaRep*)	19	142	21	102
RemoveIncrementsMutator	20	106	22	82
RemoveSwitchMutator	21	89	17	198
EmptyObjectReturnValsMutator (*RetStaRep*)	22	71	20	108
BooleanFalseReturnValsMutator (*RetStaRep*)	23	63	23	49
InvertNegsMutator	24	38	24	31
SwitchMutator	25	16	25	28

Example 6.2: CLING test case killing mutant in Figure 6.11.

```
1    public void test07()  throws Throwable  {
2        [...]
3        boolean boolean1 = switchingFunctionsHandler0.evaluateStepC(stepInterpolator0);
4        assertTrue(boolean1 == boolean0);
5        assertFalse(boolean1);
6    }
```

the change in the boolean return value.

In contrast, this mutant is killed by T_{CLING}, targeting `SwitchingFunctionsHandler` and `SwitchState` as the caller and callee classes, respectively (Listing 6.2). According to the assertion in line 5 of this test case, `switchingFunctionsHandler0.evaluateStep` must return false. However, the mutant changes the returned value in line 7 of the caller class (Figure 6.10), and thereby the true branch of the condition in line 7 is executed. This true branch changes the value of variable `first` from null to a non-null value. Hence, the `evaluateStep` method in the caller class returns true in line 12. So, the assertion in the last line of the method in Listing 6.2 kills this mutant.

```
boolean evaluateStepC(StepInterpolator interpolator){
    if (functions.isEmpty()){[...]}
    if (! initialized) {[...]}
    for ([...]) {
        [...];
        // calling the callee class in the next line.
        if (state.evaluateStep(interpolator)){
            // Changing variable first
            [...]
        }
    }
    return first != null;
}
```

Figure 6.10: Method *evaluateStep* in caller class *SwitchingFunctionsHandler*.

```
boolean evaluateStep(final StepInterpolator interpolator){
    [...]
    for([...]){
        if([...]){
            [...];
        }
        if([...]){
            [...];
        }
    }
    [...];
    ~~return false;~~ return true; //mutant
}
```

Figure 6.11: Method *evaluateStep* in callee class *SwitchsState*.

Example 6.3: Exception captured only by Cling in Closure

```
    java.lang.NullPointerException
    com.google.javascript.rhino.head.Decompiler.appendString(Decompiler.java:226)
    com.google.javascript.rhino.head.Decompiler.addName(Decompiler.java:156)
    com.google.javascript.rhino.head.IRFactory.transformName(IRFactory.java:833)
    com.google.javascript.rhino.head.IRFactory.transform(IRFactory.java:157)
```

Summary (RQ2). The test suite generated by Cling for a caller *R* and callee *E*, can kill *different* mutants than unit test suites for *E*, *R* or their union, increasing the mutation score on average by 13.0%, 10.4%, and 7.7%, respectively, for EvoSuite, and 19.5%, 17.8%, and 13.5%, respectively, for Randoop, with outliers well above 50%. Our analysis indicates that many of the most frequently killed mutants are produced by integration-level mutation operators.

6.4.4 Integration Faults Exposed by Cling (RQ3)

In our experiments, Cling generates 50 test cases that triggered an unexpected exception in one of the subject systems. None of those unexpected exceptions were observed during the execution of the unit test cases generated by EvoSuite and Randoop.

The first and second authors independently performed a manual root cause analysis for all 50 unexpected exceptions to check if they are stemming from a real integration-level fault. For this analysis, we check the API documentation to see if the generated test cases break any precondition. We indicated a test case as a fault revealing test if it does

Example 6.4: Cling test case triggering the crash in Listing 6.3

```
public void testFraction() {
    IRFactory iRFactory0 = new IRFactory();
    Name name0 = new Name(65536, 65536);

    // Undeclared exception!
    iRFactory0.transform(name0);
}
```

not violate any precondition according to the documentation, and it truly exposes an issue about the interaction between the caller and callee class. We found that out of the 50 test cases generated by Cling, 27 are **fault revealing**. The detailed description of the analysis is available in our replication package [59].[4]

To illustrate the type of problem detected by Cling, consider the test case it has generated in Figure 6.4 and the induced stack trace (for a `NullPointerException`) in Figure 6.3. This test invokes the method `transform`, in class `IRFactory` from the Closure Compiler. This method requires an object from another class, called `Name`, as an input parameter. This class has various constructors, in which multiple local variables are set. One of these local variables is a String named `identifier`. Most of the constructors in the class `Name` set a value for this String. However, one of the specific constructors (`Name(int pos, int len)`) keeps the value of `identifier` to null. The test generated by Cling uses this specific constructor to instantiate an object from `Name` and passes this object to method `transform`. This method gets `identifier` by calling the getter method in `Name` and passes it to another class (`Decompiler`) without checking it. Finally a method in `Decompiler`, called `appendString`, uses this passed `null` String without checking it and this leads to a `NullPointerException`. We do note that the documentation available in the involved classes does not limit the occurrence of this scenario.

In this particular example, a constructor in class `Name` assumes that having a null value for `identifier` is not a problem. In contrast, class `IRFactory` assumes that this variable can never be null and passes it as a proper String to another class to use it. The Cling integration testing approach brought these conflicting assumptions together, triggering the stack trace of Figure 6.3.

As is typical for integration faults, this problem can be fixed in multiple ways. The most consistent would be to adjust the `transform` method in class `IRFactory`, to check the value of `identifier` in the passed `Name` object. This then would ensure that it does not use a null value for calling methods in `Decompiler`.

Summary (RQ3). Cling-based automated testing of ⟨caller, callee⟩ class pairs exposes actual problems that are not found by unit testing either the caller or callee class individually. These problems relate to conflicting assumptions on the safe use of methods across classes (*e.g.*, due to undocumented exception throws, implicit assumptions on parameter values, *etc.*).

[4]Also available online at `https://github.com/STAMP-project/Cling-application/blob/master/data_analysis/manual-analysis/failure-explanation.md`.

6.5 Discussion

6.5.1 Applicability

Cling considers pairs of classes and exercises the integration between them. We did not propose any procedure for selecting pairs of classes to give in input to Cling. Since the technique requires pairs of classes to test, it would be time-consuming and tedious for developers to manually collect and provide the class pairs. Hence, we suggest using an automated process for class pair selection, as well. In this study, we implemented a tool that automatically analyzes each class pair to find the ones with high cyclomatic complexity and coupled branches (according to the CBC criterion defined in this chapter). This procedure is explained in Section 6.3.2.

Besides, our approach can be further extended by incorporating automated integration test prioritization approaches and selecting classes to integrate according to a predefined ordering [43, 45–48, 50–54]. So, the end-to-end process of generating test for class integrations can be automated to require a minimal manual effort from the developer.

6.5.2 Test generation cost

One of the challenges in automated class integration testing is detecting the integration points between classes in a SUT. The number of code elements (*e.g.,* branches) that are related to the integration points increases with the complexity of the involved classes. Finding and testing a high number of integration code targets increases the time budget that we need for generating integration-level tests.

With CBC, the number of coupled branches to exercise is upper bounded to the cartesian product between the branches in the caller R and the callee E. Let B_R be the set of branches in R and B_E the set of branches in E, the maximum number of coupled branches $CB_{R,E}$ is $B_R \times B_E$. In practice, the size of $CB_{R,E}$ is much smaller than the upper bound as the target branches in the caller and callee are subsets of R and E, respectively. Besides, CBC is defined for pairs of classes and not for multiple classes together. This substantially reduces the number of targets we would incur when considering more than two classes at the same time.

While a fair amount of the test generation process can be automated, multiple instances of this approach can be executed simultaneously, and thereby, this approach can be used to generate test suites for a complete project at once in a reasonable amount of time. For instance, in this study, we managed to test each of the 140 class pairs with Cling for 20 times in less than a day thanks to a parallelization of the executions.

Finally, we have used a five minutes time budget to test each class pair's interactions. Since Cling considers each coupled branch as an objective for the search process, we could have defined a different search budget per pair, depending on the number of objectives. Same as for EvoSuite and Randoop, the outcome of Cling may differ depending on the given time budget. Defining the best trade-off between the search-budget and effectiveness of the tests generated using Cling is part of our future work.

6.5.3 Effectiveness

To answer **RQ2**, we analyzed the set of mutants that are killed by Cling (integration tests), but not by the unit-test suites for the caller and callee separately (boxes labeled with T_{EvoE+R} and T_{RanE+R} in Figure 6.9). The test suite T_{Cling} was generated using a search budget of five minutes. Similarly, the unit-level suites were generated with a search budget of five minutes for each caller and callee class separately. Therefore, the total search budget for unit test generation (T_{EvoE+R} and T_{RanE+R}) is twice as large (10 minutes for each tool). Despite the larger search budget spent on unit testing, there are still mutants and faults detected only by Cling and in less time.

Cling is not an alternative to unit testing. In fact, integration test suites do not subsume unit-level suites as the two types of suites focus on different aspects of the system under test. Our results (**RQ2**) confirm that integration and unit testing are complementary. Indeed, some mutants can be killed exclusively by unit-test suites: *e.g.*, the overall mutation scores for the unit tests $TEvoE$, and $TEvoR$ are larger than the overall mutation scores of Cling. This higher mutation score is expected due to the larger unit-level branch coverage achieved by the unit tests (coverage is a necessity but not a sufficient condition to kill mutant).

Instead, Cling focuses on a subset of the branches in the units (caller and callee), but exercises the integration between them more extensively. In other words, the search is less broad (fewer branches), but more in-depth (the same branches are covered multiple times within different pairs of coupled branches). This more in-depth search allows killing mutants that could not be detected by satisfying unit-level criteria. Our results further indicate that it also allows us finding bugs that are not detectable by unit tests.

6.6 Threats to Validity

Internal validity. Our implementation of Cling may contain bugs. We mitigated this threat by reusing standard algorithms implemented in EvoSuite, a widely used state-of-the-art unit test generation tool. And by unit testing the different extensions (described in Section 6.3.1) we have developed.

To take the randomness of the search process into account, we followed the guidelines of the related literature [110] and executed Cling, EvoSuite, and Randoop 20 times to generate the different test suites (T_{Cling}, T_{EvoE}, T_{EvoR}, T_{RanE}, and T_{RanR}) for the 140 caller-callee classes pairs. We have described how we parametrize Cling, EvoSuite, and Randoop in Sections 6.2.2 and 4.3. We left all other parameters to their default value, as suggested by related literature [166, 214, 229].

External validity. We acknowledge that we report our results for only five open-source projects. However, we recall here their diversity and broad adoption by the software engineering community. The identification and categorization of the integration faults done in **RQ3** have been performed by the first and second authors independently.

Reproducibility. We provide Cling as an open-source publicly available tool as the data and the processing scrips used to present the results of this chapter.[5] Including the subjects of our evaluation (inputs) and the produced test cases (outputs). The full replication package has been uploaded on Zenodo for long-term storage [59].

[5]`https://github.com/STAMP-project/Cling-application`

6.7 Conclusion And Future Work

In this chapter we have introduce a testing criterion for integration testing, called the *Coupled Branches Coverage* (CBC) criterion. Unlike previous previous work on class integration testing focusing on (costly) data-flow analysis, CBC relies on a (lighter) control flow analysis to identify couples of branches between a caller and a callee class that are not trivially executed together, resulting in a lower number of test objectives.

Previous studies have introduced many automated unit and system-level testing approaches for helping developers to test their software projects. However, there is no approach to automate the process of testing the integration between classes, even though this type of testing is one of the fundamental and labor-intensive tasks in testing. To automate the generation of test cases satisfying the CBC criterion, we defined an evolutionary-based class integration testing approach called Cling.

In our investigation of 140 branch pairs, collected from 5 open source Java projects, we found that Cling has reached an average CBC score of 48.7% across all classes, while for some classes we reached 90% coverage. More tangibly, if we consider mutation coverage and compare automatically generated unit tests with automatically generated integration tests using the Cling approach, we find that our approach allows to kill 7.7% (resp. 13.5%) of mutants per class that cannot be killed (despite a larger search budget) by unit tests generated with EvoSuite (resp. Randoop). Finally, we identified 27 faults causing system crashes that could be evidenced only by the generated class-integration tests.

The results indicate a clear potential application perspective, more so because our approach can be incorporated into any integration testing practice. Additionally, Cling can be applied in conjunction with with other automated unit and system-level test generation approaches in a complementary way.

From a research perspective, our study shows that Cling is not an alternative for unit testing. However, it can be used for complementing unit testing for reaching higher mutation coverage and capturing additional crashes which materialize during the integration of classes. These improvements of Cling are achieved by the key idea of using existing usages of classes in calling classes in the test generation process.

For now, Cling only tests the call-coupling between classes. In our future work, we will extend our approach to explore how other types of coupling between classes (*e.g.*, parameter coupling, shared data coupling, and external device coupling) can be used to refine the couples of branches to target. Indeed, our study indicates that despite the effectiveness of Cling in complementing unit tests, lots of objectives (coupled branches) remain uncovered during our search process. Hence, in future studies, we will enhance the detection of infeasible branches to remove them from the search objectives and perform a fitness landscape analysis of the search process to identify potential bottlenecks.

Finally, this chapter mostly focuses on examining the results of this approach on coupled branches coverage, mutation coverage, and detected faults. In our future work, we will explore how Cling can be effectively integrated with a development lifecycle (for instance, in a continuous integration process) and how automatically generated class integration tests can help developers to detect potential faults and debug their software.

7 Commonality-Driven Unit Test Generation

Despite several advances, search-based unit test generation still faces many challenges. Among those are (i) the crafting of complex objects and values used during test generation [25], and (ii) the indirect coverage of encapsulated elements (*e.g.*, private methods and class attributes) through the invocation of specific paths in public methods [27]. Various approaches address those challenges by relying on dynamic symbolic execution to generate complex objects and values using constraint solvers [230–233]; seeding to identify objects and values from the application source and test code that are later reused during the search [124]; or class usages, learned from static analysis of the source code [156] and dynamic execution of the existing tests (like behavioral model seeding introduced in Chapter 3), and used to generate realistic objects.

However, if complex objects and values can indeed lead to an improvement in the coverage, it does not always succeed in covering all the elements of a class under test. For instance, if the indirect coverage of a private method requires specific executions paths in a public method, the current fitness functions will not be able to provide sufficient guidance to the search process [27].

In this chapter, we hypothesize that common and uncommon execution paths, observed during the actual operation of the system, can lead to better guidance of the search process, and hence, better coverage. Complementing previous work on seeding [124], which is aimed at triggering different execution paths in the methods under test, we consider the commonality of those execution paths. For that, we approximate commonality using weights for the different code blocks, and define a secondary objective called the commonality score, denoting how close an execution path is from common or uncommon executions of the software.

We implemented the commonality score in EvoSuite [6] and evaluated it on 150 classes from JabRef, an open-source bibliography reference manager, for common and

The author of this thesis partially contributed to this chapter as one of the supervisors, writers (reviewing the first draft of the thesis and one of the paper's main writers), and the resources provider (*e.g.*, experiment runner infrastructure).

uncommon behaviors. We compare the commonality score (**RQ.1**), the structural coverage (**RQ.2**), and the fault-finding capabilities (**RQ.3**) of the thus generated tests to tests generated by the standard EvoSuite implementation. Our results are mixed but show that this secondary objective significantly improves the number of covered common paths in 32.6% of the classes. Although the average structural coverage remains stable, the commonality score significantly improves the line (resp. branch) coverage in three (resp. four) classes, but also negatively impacts the coverage for eight (resp. nine) classes. Finally, the commonality score impacts the number of killed mutants for 22 classes (11 positively and 11 negatively). Our implementation is openly available at `https://github.com/STAMP-project/evosuite-ramp`, and the replication package of our evaluation and data analysis have been uploaded to Zenodo [67, 234].

7.1 Background And Related Work

7.1.1 Search-Based Unit Test Generation

Search-based unit test generation has been extensively investigated by prior studies [6, 21, 42]. These studies have confirmed that it achieves a high level of coverage [21, 42], detects faults in real-world applications [25, 235], and reduces the debugging costs [37]. Most search-based unit test generation approaches abstract the source code of a method to a control flow graph:

Definition 7.1.1 (Control Flow Graph (CFG) [39]) *A control flow graph for a method m is a directed graph $G = (B, E)$, where B is the set of basic blocks (i.e., sequences of statements that do not have any branch before the last statement), E is the set of control flow edges connecting the basic blocks.*

7

For instance, for the method with the pseudo-code presented in Figure 7.1a, the corresponding CFG for this method is depicted in Figure 7.1b.

Search-based software unit test generation approaches use meta-heuristics to evolve a set of test cases. These techniques start with generating an initial population of randomly produced test cases. The fitness of each individual in the population is evaluated using a fitness function, which is usually defined according to the coverage in the CFGs of the target class. Next, a subset of the fittest individuals is selected for evolution and leads to the generation of a new population. The evolving process contains three steps: (i) crossover, which randomly mixes two selected individuals to generate new offspring; (ii) mutation, which randomly changes, adds, or removes a statement in an individual; and (iii) insertion, which reinserts the modified individuals into the population for the next iteration of the algorithm. This process continues until either satisfactory individuals are found, or the time budget allocated for the search is consumed. Among the different approaches, EvoSuite [6] uses genetic algorithms to evolve Java test suites in order to cover a class under test.

MOSA [214] and *DynaMOSA* [42] are two new many-objectives genetic algorithms proposed for unit test generation. These algorithms consider test cases as individuals and incorporate separate fitness functions for separate coverage goals (*e.g.*, covering each branch in the CFGs will be an independent search objective). They use non-dominated fronts to generate test cases in the direction of multiple coverage goals in parallel, and thereby generate tests aiming to cover specific goals, while not letting the test generation be trapped for covering a single goal. Panichella *et al.* [214] show that *MOSA* outperforms the original EvoSuite approach in terms of structural coverage and mutation score.

In this chapter, we use *MOSA* to automatically generate test cases according to the collected logs during the production phase. Future work includes the evaluation of our approach with other multi and many-objectives algorithms, like *DynaMOSA*.

7.1.2 Usage-based Test Generation

The majority of search-based test generation techniques aim to achieve high coverage for various metrics (*e.g.,* line coverage, branch coverage, or more recently mutation coverage). Despite their considerable achievements, they do not consider the execution patterns observed in production use for automatic generation of unit tests. Hence, Wang *et al.* [236] investigated how developer-written tests and automatically generated tests represent typical execution patterns in production. Their study confirms that these tests are not a proper representation of real-world execution patterns.

The behavior of actual users may reveal faults, which are not detected by the existing test cases. For instance, a piece of code in the software under test that is not often used in practice may be left relatively untested because it is rarely exercised in production. A recent method from Wang *et al.* [237], based on symbolic execution, recreates users behaviors using log data from a system run in production, which has allowed to find the same faults in a system encountered by a user. This chapter aims to expand upon generating tests based on the actual usage of a system at the unit level. In contrast to Wang *et al.* [237], where the aim is to replicate a full behavior executed by a user by using symbolic execution, we aim to guide the search process in a genetic algorithm towards executing common or uncommon behaviors. In the same vein as Wang et al. [237], log data is used to determine the execution counts of code branches.

Other approaches consider user feedback [238], or usage models of the application and statistical testing [144, 151, 239, 240] to generate and prioritize test cases at the system level. A usage model consists in a state machine where transitions have been labelled with a probability of being executed. Unlike those approaches, we consider test case generation at the unit level.

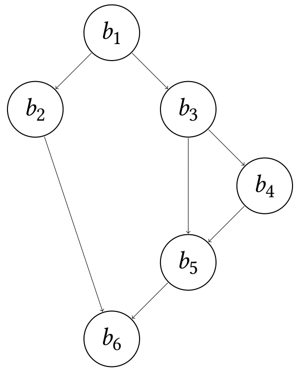
7.2 Test Generation For Common And Uncommon Behaviors

Intuitively, commonality describes to what extent a test exercises code branches that are executed often during the normal operation of the system under test. If a test executes branches that are often (respectively rarely) executed in practice, it will have a high (respectively low) commonality score. The commonality score has a value between 0 and 1 and is computed based on an annotated control flow graph [39]:

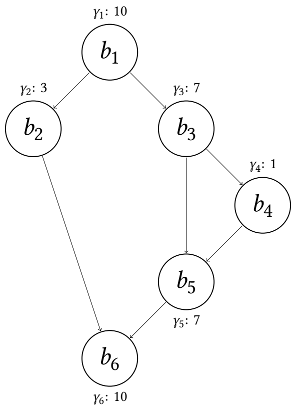
Definition 7.2.1 (Annotated Control Flow Graph) *An annotated control flow graph is a directed graph $G = (B, E, \gamma)$, where $G = (B, E)$ is a control flow graph, and $\gamma : B \longrightarrow \mathbb{R}$ is a labelling function giving for the basic blocks in B an execution weight denoting how often the block is executed during operations.*

```
/* Branch 1 */
if condition1 then
    /* Branch 2 */
    ...;
else
    /* Branch 3 */
    ...;
    if condition2 then
        /* Branch 4
           */
        ...;
    /* Branch 5 */
    ...;
/* Branch 6 */
...;
```

(a) Pseudo-code.



(b) CFG.



(c) ACFG.

Figure 7.1: Example of pseudo-code and its corresponding annotated control flow graph. The γ_i indicate to the execution weight of the node.

The execution weights can be derived from the operation logs of the system, an instrumented version of the system (like in our evaluation), or assigned manually.

Let us define the commonality score. For a test case, its commonality score depends only on the branches it covers and on the highest and lowest execution weights in the class under test. Branches without execution weights are ignored and branches covered multiple times (*e.g.*, in a loop) are counted only once.

Definition 7.2.2 (Commonality score) *For a test case t executing n basic blocks b_i labelled by a function γ, the highest execution weight in the class under test h, the lowest execution weight in the class under test l, the commonality score of t, denoted $c(t)$ is defined as:*

$$c(t) = \frac{\sum_{i=1}^{n} \left(\gamma b_i - l\right)}{n \times (h - l)}$$

The commonality score for a test suite s is defined as the average of the commonality scores of its test cases: $c(s) = \left(\sum_{t_i \in s} c(t_i)\right) / |s|$.

For instance, considering a class containing a single method with the pseudo-code presented in Figure 7.1a, the corresponding annotated control flow graph in Figure 7.1c, and a test suite containing three test cases t_1 covering (b_1, b_2, b_6), t_2 covering (b_1, b_3, b_5, b_6), and t_3 covering (b_1, b_3, b_4, b_5, b_6). The commonality scores are:

$c(t_1) = ((10-1)) + (3-1) + (10-1)) / (3 \times (10-1)) = 20/27 \approx 0.741$, $c(t_2) = 5/6 \approx 0.833$, and $c(t_3) = 2/3 \approx 0.667$.

7.2.1 Commonality As A Secondary Objective

Secondary objectives are used to choose between different test cases in case of a tie in the main objectives. For instance, the default secondary objective used by MOSA [214] minimizes the test case length (*i.e.*, the number of statements) when two test cases satisfy the same main objectives (*e.g.*, cover the same branches). Using test case length minimization as a secondary objective addresses the bloating effect [241] by preventing the search process from always generating longer test cases. Since this is a desirable property, we combine the test case length minimization with the commonality of the test case using a weighted sum when comparing two test cases.

Definition 7.2.3 (Commonality secondary objective) *For two test cases t_1, t_2 with lengths l_1, l_2, the comparison between the two test cases is done using the following formula:*

$$common(t_1, t_2) = \frac{\left(\alpha\left(\frac{l_1}{l_2}\right) + \beta\left(\frac{1-c(t_1)}{1-c(t_2)}\right)\right)}{(\alpha + \beta)}$$

If $common(t_1, t_2) \leq 1$, then t_1 is kept, otherwise t_2 is kept.

Similarly, for the uncommonality between two test cases, we will have the following definition.

Definition 7.2.4 (Uncommonality secondary objective) *For two test cases t_1, t_2 with lengths l_1, l_2, the comparison between the two test cases is done using the following formula:*

$$uncommon(t_1, t_2) = \frac{\left(\alpha\left(\frac{l_1}{l_2}\right) + \beta\left(\frac{c(t_1)}{c(t_2)}\right)\right)}{(\alpha + \beta)}$$

If $uncommon(t_1, t_2) \leq 1$, then t_1 is kept, otherwise t_2 is kept.

In our evaluation, we use commonality and uncommonality with MOSA to answer our different research questions.

7.3 Empirical Evaluation

To assess the usage of commonality as a secondary objective for test case generation, we performed an empirical evaluation using 150 classes from JabRef[1], an open source bibliography reference manager, to answer the following research questions:

RQ.1 How does the commonality score of the generated tests compare when using the *common*, *uncommon*, and *default* secondary objectives?

RQ.2 How does the line and branch coverage of the generated tests compare when using the *common*, *uncommon*, and *default* secondary objectives?

RQ.3 How does the mutation score of the generated tests compare when using the *common*, *uncommon*, and *default* secondary objectives?

[1] `https://www.jabref.org`

We implemented the secondary objectives from Section 7.2 in EvoSuite [6], a state-of-the-art white-box unit test generator tool for Java. Our implementation is openly available at `https://github.com/STAMP-project/evosuite-ramp`, and the replication package of our evaluation and data analysis have been uploaded to Zenodo [67, 234].

7.3.1 Subject And Execution Weights

Collecting execution weights For our evaluation, we choose JabRef (46 KLOC), an open-source Java bibliography reference manager with a graphical user interface working with BibTex files. To determine the execution weights of the different branches, we instrumented JabRef using Spoon [78] and added log statements producing a message with a unique identifier each time a branch is executed. These identifiers are then mapped to a source code location, identified by the class name, the method name, and the line number. Furthermore, the number of occurrences of the identifier in the log messages is established. We then asked five people (including the first author) to use our modified JabRef implementation to perform various tasks (adding a reference, updating a reference, removing a reference, *etc.*) and collected the produced logs. In an industrial context, operations logs can be analyses and traced back to the source code to identify the execution weights [242].

Classes under test We sampled 150 classes. We excluded classes from the `org.jabref.gui` and `org.jabref.logic.importer.fileformat` packages as they respectively work with JavaFX and perform input-output operations. From the remaining classes and following the best practices of the search-based unit testing community [243], we selected 75 classes with the highest cyclomatic complexity, as classes with a higher cyclomatic complexity are harder to process for unit test generation tools and 38 classes with the largest number of lines of code. Additionally, we selected 37 classes that were executed the most by our modified JabRef implementation.

7

Configuration parameters We ran EvoSuite with the default coverage criteria (line, branch, exception, weak mutation, input, output, method, method without exceptions, and context branch) and three different secondary objectives: (i) *default*, minimizing the test case length, (ii) *commonality*, as described in Definition 7.2.3, and (iii) *uncommonality*, as described in Definition 7.2.4. We executed EvoSuite on each class under test 30 times with the MOSA algorithm [214] and a search budget of three minutes, offering a good compromise between runtime and coverage [20, 21]. All other configuration parameters were left to their default value.

7.3.2 Data Analysis

For each of the 13,500 execution (150 classes × 30 repetitions × 3 configurations), we collected the commonality score and structural coverage information from EvoSuite. Additionally, we performed a mutation analysis of the generated test suites using Pit [244]. For 46 classes (out of 150), EvoSuite could not complete 30 executions using our different configurations. We excluded those classes to keep the comparison fair and performed our analysis on the 104 remaining classes.

To compare the commonality score, the structural coverage, and the mutation score, we used the non-parametric Wilcoxon Rank Sum test, with $\alpha = 0.05$ for Type I error, and the Vargha-Delaney statistic $\widehat{A}_{12}$ [245] to evaluate the effect size between two configurations. An $\widehat{A}_{12}$ value lower than 0.5 for a pair of configurations (A,B) indicates that A increases the score or coverage compared to B, and a value higher than 0.5 indicates the opposite. The Vargha-Delaney magnitude measure also allows partitioning the results into three categories having large, medium, and small impact [245].

7.4 Results

7.4.1 Commonality Score (RQ1)

In this section we answer the question: How does the commonality score of the generated tests compare when using the *common*, *uncommon*, and *default* secondary objectives?

Figure 7.2 illustrates the impact of using *commonality* and *uncommonality*, as the secondary objective, on the commonality score of the generated test cases. Figure 7.2a shows that the average and median of the commonality score is improved by 8% and 12%, respectively, compared to *default* when using *commonality* as secondary objective. In parallel, using *uncommonality* as secondary objective reduces the commonality score by, on average, 5% (2.5% for median) compared to *default*. Moreover, Figure 7.2c presents the number of cases (*i.e.,* classes used as the target class for unit testing), in which the application of *commonality* and *uncommonality* significantly (*p-value* < 0.05) changes the commonality score with effect size magnitude of large, medium, or small. As we can see in this figure, utilizing *commonality* always leads to a significant improvement in the commonality score (blue bars), and in contrast, using *uncommonality* always reduces this score (red bars). In total, *commonality* significantly improves the commonality score in 34 cases (32.6% of classes), and *uncommonality* significantly reduces this score in 21 classes (20.1% of cases). Figure 7.2b depicts the effect sizes of differences observed in these cases. Consistent with the previous figures, the average effect size ($\widehat{A}_{12}$) achieved by *commonality* is higher than 0.5 (*i.e.,* commonality score has been improved). However, this value is lower than 0.5 for *uncommonality*.

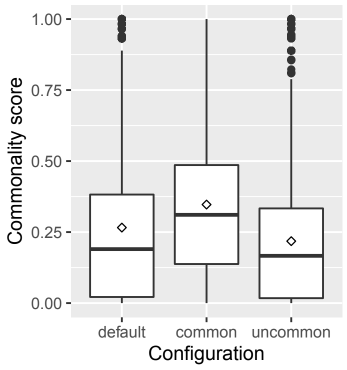
Summary Using *commonality* as secondary objective in the EvoSuite search-based test case generation process leads to test cases that exhibit an improved commonality score. In parallel, the application of *uncommonality* leads to the reduction of the commonality score.

7.4.2 Structural Coverage (RQ2)

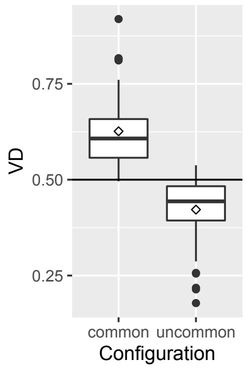
In this section we provide an answer to the following research question: How does the line and branch coverage of the generated tests compare when using the *common*, *uncommon*, and *default* secondary objectives?

Figure 7.3 shows the line and branch coverage achieved by using *commonality* and *uncommonality* as secondary objectives compared to *default*. Figure 7.3a indicates that the average coverage is the same for all of the assessed configurations.

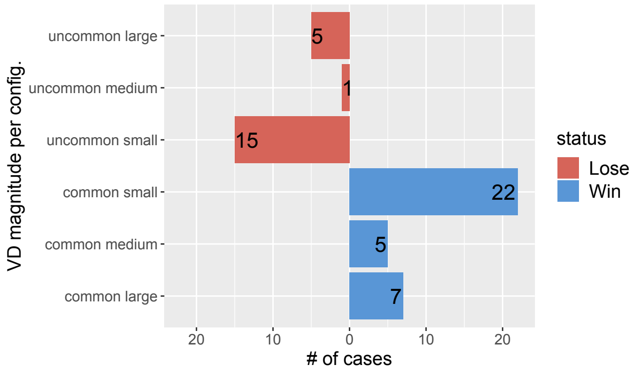
Looking at the comparison of the structural coverage values achieved by each secondary objective in each class, we can see that the line and branch coverage is signifi-



(a) Test cases commonality scores.



(b) Effect sizes $\widehat{A}_{12}$.

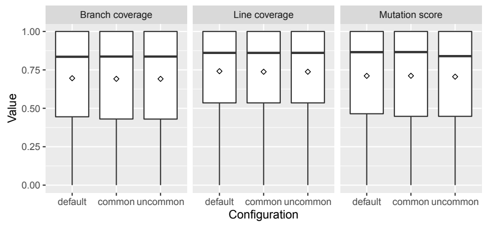


(c) Effect sizes $\widehat{A}_{12}$ magnitudes.

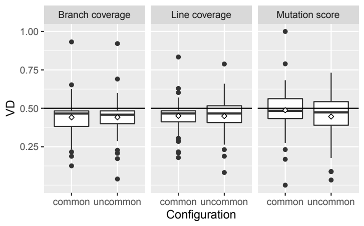
Figure 7.2: Test cases commonality values and comparison to *default*. Diamonds indicate mean values and horizontal bars (–) indicate median values.

cantly impacted by *commonality* and *uncommonality* in some cases. Figure 7.3c presents the number of cases that these secondary objectives significantly (*p-value* < 0.05) reduce ($\widehat{A}_{12} < 0.5$) or increase ($\widehat{A}_{12} > 0.5$) the line and branch coverage with effect size magnitude small, medium, or large. According to this figure, in general, utilizing *commonality* leads to a significant improvement for line and branch coverage in three and four classes, respectively. Nevertheless, this secondary objective reduced the line and branch coverage in eight and nine classes, respectively.

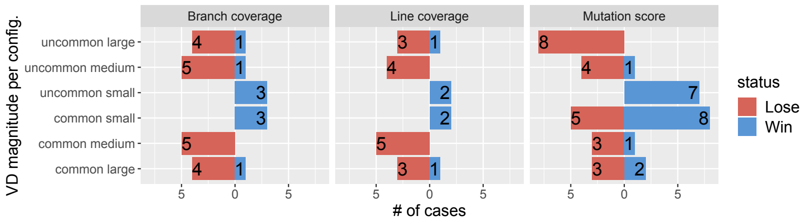
Also, we can see a similar result for *uncommonality*: significant improvements in three and five classes and significant reductions in seven and nine cases for line and branch coverage. Since the number of cases in which *commonality* and *uncommonality* lead to a significantly lower structural coverage is higher than the the number of cases in which we see a significant improvement in coverage, the average effect size of differences (Figure 7.3b) is slightly less than 0.5 for both line (0.47 for both secondary objectives) and



(a) Test suites coverage and mutation score.



(b) Effect sizes $\widehat{A}_{12}$.



(c) Effect sizes $\widehat{A}_{12}$ magnitudes.

Figure 7.3: Test suites coverage and mutation score, and comparison to *default*. Diamonds indicate mean values and horizontal bars (–) indicate median values.

branch coverage (0.46 for both).

Summary On average, using *commonality* or *uncommonality* does not impact the line and branch coverage. However, these two secondary objectives can significantly impact the structural coverage in specific cases.

7.4.3 Mutation Analysis (RQ3)

In the final research question we reflect on How does the mutation score of the generated tests compare when using the *common*, *uncommon*, and *default* secondary objectives?

Figure 7.3a depicts the mutation score achieved by using *commonality* and *uncommonality* compared to *default*. Like line and branch coverage, the average mutation scores achieved by these secondary objectives is similar to the one achieved by *default*. However, Figure 7.3c shows that *commonality* and *uncommonality* can significantly (*p-value* < 0.05) impact the mutation score achieved by unit test generation. The *commonality* secondary objective significantly increases the number of mutants killed for 11 classes but, at the same time, also decreases the mutation score in another 11 cases. Moreover, *uncommonality* significantly changes the mutation score in 20 cases (8 wins against 12 losses). Figure 7.3b shows the effect size of differences in these cases for both *commonality* and *uncommonality* secondary objectives. According to this Figure, the average $\widehat{A}_{12}$ estimations are 0.49 and 0.47. Since these values are lower than 0.5, on average, the difference achieved by these two secondary objectives is negative. However, the outliers in this figure show us that the effect sizes of *commonality* above 0.75 in some specific cases. Hence, the graphs in Figure 7.3c indicate that using *commonality* and *uncommonality* can improve the mutation score in specific cases.

Summary On average, using *commonality* or *uncommonality* does not have any effect on the mutation score achieved by the generated test suites. However, these two secondary objectives can significantly change the killed mutants in some cases.

7.5 Discussion

7.5.1 Execution Weights

In our evaluation, we collected execution weights using an instrumented version of JabRef distributed to five different users. As a result, a large number of log messages allowed us to have execution weights for many different classes. Such data collection is not realistic in an industrial setting as the collection and analysis of log data is challenging for large applications [246]. It is likely that the collected data will not cover the complete system but only a subset of its classes.

However, we believe that the development of scalable software analytics practices [247] represents an opportunity to include information from the software operations environment in various development activities, including testing [246, 248]. For instance, Winter *et al.* [242] recently brought information about the number of times a log statement is executed to the developer's IDE to raise awareness about the load it represented during the operations of the system. Similar information can be collected for seeding (like Chapter 3) or annotating a control flow graph, like in our approach, and allow developers to generate new tests from their IDE.

Finally, our current approach considers execution weights individually to approximate usages, which allows us to compute a commonality score quickly. Different definitions of the commonality score based, for instance, on full and partial execution paths identified from the operations logs are possible. Those finer grained definitions of commonality would allow to take multiple executions of the same code blocks (like loops) into account, at the expense of a higher computational cost. Exploration and evaluation of such definitions are left for future work.

7.5.2 Impact On Mutation Analysis

After manual investigation of the generated tests[2], we saw that the classes on which *commonality* performs relatively well are often executed. The class on which the performance was especially good compared to *default* was an enum class (`org.jabref.logic.util.StandardFileType`). In a large majority of the cases (25 out of 30), the tests generated using the *commonality* secondary objective contain a method sequence that is not present in the tests generated by *default*. We inspected the execution counts of individual branches stemming from the operational use of the system. From this inspection, we found a single branch in the code of method `getExtensions` that has been executed as part of the usage scenarios from all participants.

This method is consistently involved in the test cases that kill the mutants that the tests generated by *default* fail to kill most of the time. This supports our initial assumption

[2]The results and complete manual analysis are available online [67, 234, 249].

that using *commonality* can drive the search process to cover the code in a different way, possibly finding different kinds of faults.

7.5.3 Usefulness For Debugging

The end-goal of any test suite is to identify faults in source code and help the developer during her debugging activities. Our evaluation measured the fault-finding capabilities of the generated test suites, but did not investigate their usefulness for debugging. Previous research has confirmed that automatically generated tests can find faults in real software [25] and are useful for debugging [23].

However, there remain several challenges, like the understandability of the generated tests [25, 109]. Since the *commonality* and *uncommonality* secondary objectives aim to influence how the lines in a method are covered, we expect that it will also have an impact on the understandability of the generated tests. Future research will include the assessment of the debugging capabilities of the generated tests (*e.g.,* understandability, performance, readability, *etc.*).

7.6 Threats To The Validity

Internal validity We repeated each execution of EvoSuite 30 times to take the randomness of the generation process into account in our data analysis. We have tested our implementation of the *commonality* and *uncommonality* secondary objectives to reduce bugs' chance of influencing our results.

External validity We gathered execution data needed from a small number of people in a relatively structured manner. We cannot guarantee that those executions are representative of all the usages of JabRef. However, we believe that the diversity of the tasks performed by our users is enough for this evaluation. Also, the evaluation was performed using only one case study. Future research includes the repetition of the assessment on other Java applications.

Construct validity We relied on the reports produced by EvoSuite for structural coverage and the reports produced by Pit for mutation analysis to compare our different secondary objectives. The usage of those standard metrics allows the comparison of our results with other search-based unit test generation approaches.

Conclusion validity Our conclusions were only drawn based on statistically significant results with $\alpha = 0.05$. We used the standard non-parametric Wilcoxon Rank Sum test for significance and the Vargha-Delaney statistic for the effect size.

7.7 Conclusion And Future Work

In this chapter, we introduced the commonality score, denoting how close an execution path is from common or uncommon executions of the software in production, and the *commonality* and *uncommonality* secondary objectives for search-based unit test generation. We implemented our approach in EvoSuite and evaluated it on JabRef using execution data from real usages of the application. Our results are mixed. The *commonality* secondary objective leads to an increase of the commonality score, and the *uncommonality* secondary objective leads to a decrease of the score, compared to the *default* secondary objective (**RQ.1**). However, results also show that if the commonality score can have a positive impact on the structural coverage (**RQ.2**) and mutation score (**RQ.3**) of the generated test suites, it can also be detrimental in some cases. Future research includes a replication of our evaluation on different applications and using different algorithms (*e.g., DynaMOSA*) to gain a deeper understanding of when to apply *commonality* and *uncommonality* secondary objectives; the exploration and assessment of different definitions of commonality; and an assessment of the generated tests regarding their usefulness for debugging.

8 Conclusion

8.1 Research Questions Revisited

This section revisits the three research questions defined in Chapter 1.

Research Question 1: *What are the challenges in search-based crash reproduction?*

With this research question, we tried to detect and categorize the challenges in search-based crash reproduction. We could not provide a thorough answer only based on existing results since previous evaluations of these techniques contained a limited number of subjects (at most 50 cases). Hence, to take the first step towards answering this research question, we have designed a benchmark, called JCrashPack, which includes 200 real-world Java crashes come from seven open-source projects. We have also proposed a new tool, called ExRunner, to conduct extensive experiments on these crashes.

We have made both JCrashPack and ExRunner publicly available for two purposes: (i) ease the comparison of the existing crash reproduction techniques; and (ii) help researchers in analyzing and identifying the impacting factors in search-based crash reproduction techniques.

We have utilized ExRunner to apply the state-of-the-art automated crash reproduction tool (EvoCrash) on all crashes in JCrashPack. The outcome of this experiment shows that EvoCrash is not successful in reproducing more than 50% of the crashes in JCrashPack. Hence, we have performed an extensive manual analysis to identify the challenges leading to unsuccessful crash reproduction in this search-based approach. We have characterized the identified challenges into 13 categories. Some of the identified challenges (*e.g.*, complex input generation, complex code, *etc.*) are related to general search-based test generation issues, and the others are specific to crash reproduction (*e.g.*, exceptions are captured by try/catch, and nested private calls in the given stack trace).

This research question confirms that there is still space for improvements in search-based crash reproduction and test generation techniques. Despite the undeniable achievements by search-based crash reproduction, it cannot reproduce about 50% of crashes in JCrashPack. We have categorized the challenges this technique has encountered in unsuccessful cases. This categorization indicates that two of the most common search-based crash reproduction issues stem from the general search-based crash reproduction difficulties: Input Data Generation and Complex Code. These challenges indicate the complexity of the problem in search-based crash reproduction and test generation. Hence, establishing that using pure randomness, without using any other sources of information, is not enough for the search process to generate solutions for complex scenarios (crashes and object states).

Research Question 2: *Based on the identified challenges, how can we leverage the existing knowledge, carved from information sources, to steer the crash reproduction search process?*

With this research question, we sought to address two of the most common challenges in search-based crash reproduction, identified in Research Question 1: Input Data Generation and Complex Code. For this purpose, this thesis has introduced new techniques, which utilize the pieces of knowledge carved from different information sources such as source code and hand-written tests. Chapters 3 and 4 address the first challenge by introducing a novel seeding strategy (behavioral model seeding) and designing two search helper-objectives (*MO-HO*), respectively. Chapter 5 considers the second challenge by providing a new secondary objective (*BBC*) to complement the guidance provided by approach level and branch distance, which are the most well-known heuristics for guiding a test generation test process to cover all of the statements in a target class. The results, delivered from this research question, confirm the relevance of these two challenges.

In Chapter 3, we have introduced a new seeding technique for search-based crash reproduction, called behavioral model seeding, to address complex input data generation challenge. This seeding strategy monitors and abstracts the usages of objects in source code and existing test suites. Then, it uses these models to generate test cases during the search process. We have assessed the relevance of using this seeding strategy in search-based crash reproduction. We have also compared the behavioral model seeding with the state-of-the-art seeding strategy (called test seeding), which only seeds the existing test cases to the search process. We have witnessed that the behavioral model seeding outperforms search-based crash reproduction without seeding and with test seeding in terms of effectiveness and efficiency. According to the results, model seeding increases the reproduced crashes by a minimum of 6% compared to test seeding and no seeding.

Chapter 4 introduces a novel technique, called *MO-HO*, which combines pieces of information from sequences of method calls in the generated test cases and their length to improve the exploration ability of the search process in crash reproduction. Improving the exploration in this search-based technique leads to more diverse test cases during the search process. By generating more diverse test cases, the search process can generate more complex solutions with more complex input data [168]. Hence, *MO-HO* can help the search process to address the complex input generation challenge, as we have identified in Chapter 2. The results presented in this chapter show that using multiple information sources to define new helper objectives for crash reproduction improves the single-objective crash reproduction's effectiveness and efficiency in 10% and 34.6% of crashes, respectively.

Finally, Chapter 5 addresses the complex code challenge, which we have identified in Research Question 1, by introducing a novel secondary objective called *Basic Block Coverage* (*BBC*). This objective carves additional information about the coverage of basic blocks by test cases generated in the search process, and thereby it brings more guidance to the search process when the existing primary objective cannot guide. Our results show that *BBC* reduces the chance of getting trapped in local optima during the search process, and consequently, aids the search-based crash reproduction to reach higher effectiveness and efficiency. This secondary objective improves the efficiency of the state-of-the-art techniques in at least 13.7% of crashes. It also improves the crash reproduction ability of the state-of-the-art in 6 crashes.

Research Question 3: *How can we leverage the existing knowledge, carved from information sources, to design search-based test generation approaches for unit and class integration testing?*

As we observed in Research Question 1, some of the search-based crash reproduction challenges are connected to the limitations in general search-based test generation. Since we have addressed two of these challenges in Research Question 2 by leveraging the knowledge carved from different information sources, in this research question, we try to extend our study to investigate if using additional knowledge from other sources can help the search-based test generation in fulfilling other criteria such as class integration testing (Chapter 6) and unit testing (Chapter 7).

Chapter 6 introduces a new approach called Cling, which uses the call-site information to generate test cases for covering different interactions between two classes. Since Cling is the first white-box search-based technique for testing class integration, we have evaluated it against the state-of-the-art unit test generation approach. The results, presented in this chapter, confirm that the tests generated by Cling complement automatically generated unit tests for higher mutation scores (on average, 7.7% more killed mutants) and detects class-integration faults not detected by EvoSuite. This research question also shows that Cling can detect integration-level faults that remained unrevealed in unit testing, thanks to the call-site information.

Chapter 7 introduces a new metric for the generated tests during the search process called commonality score. This metric measures how close the execution path of a test case is from the common or uncommon execution patterns observed in production. We have also introduced *commonality* and *uncommonality* as secondary objectives for search-based unit test generation according to the commonality score. The former helps the search process to generate tests with the highest similarity to the common execution paths. In contrast, the latter aids the search process to generate unit tests, which are covering the most uncommon execution paths. The results of the evaluation of these two secondary objectives are mixed. We have observed that using these two helper objectives can help the search process to achieve a higher mutation score in some cases, while we see the opposite in some other cases.

8.2 Implications

This section, first, discusses the implications for research. Then, it presents an outline of how developers have used or could use our implemented tools for search-based crash reproduction (Botsing), class integration test generation (Cling), and commonality-driven unit test generation (using commonality score secondary objectives).

8.2.1 Implications for research

All of the studies presented in this thesis are fully reproducible thanks to the provided replication packages (see Section 1.7). Any independent group can repeat the same executions easily using each of these artifacts. Also, ExRunner, which is implemented for assessing the crash reproduction (answering research questions 1 and 2), enables usability by providing a documented and well-structured infrastructure, facilitating the reuse and repurposing.

8.2.2 Implications for developers

Botsing

Writing a test case reproducing a crash reported to developers aids them in understanding the scenario in which the crash happened and thereby helps them in bug fixing and debugging practices [57]. However, as indicated by one of our industrial partners in STAMP[1], reproducing a reported crash manually needs a knowledgable developer. Since crash reproduction is a time-consuming and labor-intensive task, an automated crash reproduction technique reduces companies' debugging costs.

This thesis introduces Botsing, an open-source automated crash reproduction framework using search-based test generation techniques. Initially, we implemented the best-performing automated crash reproduction approach [28] in Botsing. A previous study confirmed that this approach helped developers in debugging practices.

Chapter 2 of this thesis shows that this approach still has some limitations in reproducing complex and non-trivial crashes. Hence, we implemented novel techniques (introduced in Chapters 3 to 5) to improve this algorithm's effectiveness and efficiency. So, Botsing is currently the best automated crash reproduction tool, openly available.

Botsing is useful in any issue tracking system in which the reproted crashes contain a stack trace.

Botsing in Continuous Integration Pipelines

Botsing is useful in any issue tracking system in which the reported issues contain a stack trace. Developers can integrate Botsing in their continuous integration systems to get a test case, which reproduces the scenario in which the reported crash happened.

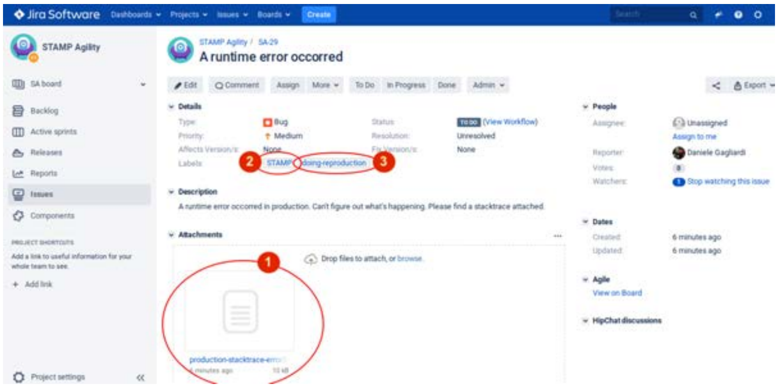
Botsing Jira Plugin[2]: In STAMP, our industrial partner implemented the Botsing Jira plugin[3]. With this plugin, any software project using the Jira issue tracking system can automatically generate crash reproducing test cases for their reported crashes. The plugin automatically detects any issue with an attached stack trace (1 in Figure 8.1), `doing-reproduction` label and `STAMP` label (2 and 3 in Figure 8.1) and initiates a Botsing instance in a remote server, linked in the configurations (Figure 8.2), for reproducing the attached crash.

The remote server attaches the crash reproducing test case to the issue after finishing the Botsing execution (Figure 8.3). By clicking on the attachment, developers can see the crash reproducing test case (Figure 8.4).

[1]STAMP: Software testing amplification for DevOps, an H2020 European project containing multiple industrial and academic partners (including Delft University).

[2]This section uses the figures from STAMP's final deliverable: `https://github.com/STAMP-project/docs-forum/blob/master/docs/d44_final_api_public_version_services_courseware.pdf`

[3]`https://github.com/STAMP-project/botsing-jira-plugin`



STAMP Agility / SA-29

A runtime error occurred

Details

Type: Bug
Priority: Medium
Affects Version/s: None
Labels: STAMP doing-reproduction
Status: TO DO (View Workflow)
Resolution: Unresolved
Fix Version/s: None

Description

A runtime error occurred in production. Can't figure out what's happening. Please find a stacktrace attached.

Attachments

Drop files to attach, or browse.

production-stacktrace-erro

People

Assignee: Unassigned
Assign to me
Reporter: Daniele Gagliardi
Votes: 0
Watchers: 1 Stop watching this issue

Dates

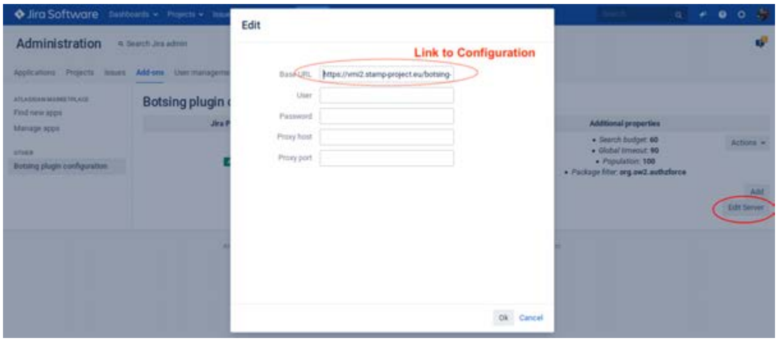
Created: 6 minutes ago
Updated: 6 minutes ago

Agile

View on Board

HipChat discussions

Figure 8.1: Triggering automatic crash reproduction with the Botsing Jira plugin.



Edit

Base URL: https://vmi2.stamp-project.eu/botsing-
User:
Password:
Proxy host:
Proxy port:

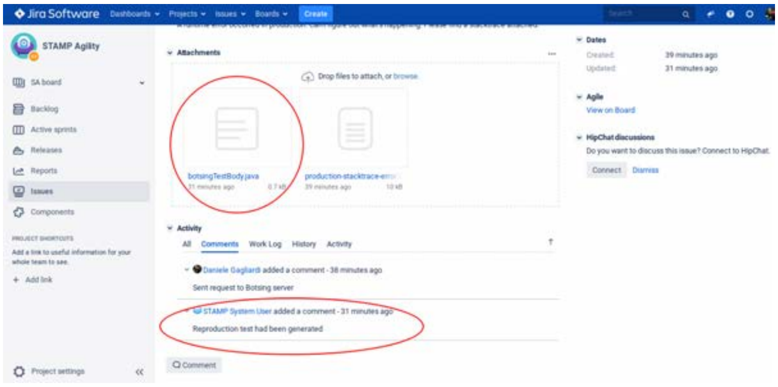
Ok Cancel

Botsing plugin c

Additional properties

- Search budget: 60
- Global timeout: 90
- Population: 100
- Package filter: org.ow2.authzforce

Figure 8.2: Botsing remote server configuration in Jira



Attachments

Drop files to attach, or browse.

botsingTestBody.java 31 minutes ago 0.7 kB
production-stacktrace-erro 39 minutes ago 10 kB

Activity

Daniele Gagliardi added a comment - 38 minutes ago
Sent request to Botsing server

STAMP System User added a comment - 31 minutes ago
Reproduction test had been generated

Dates

Created: 39 minutes ago
Updated: 31 minutes ago

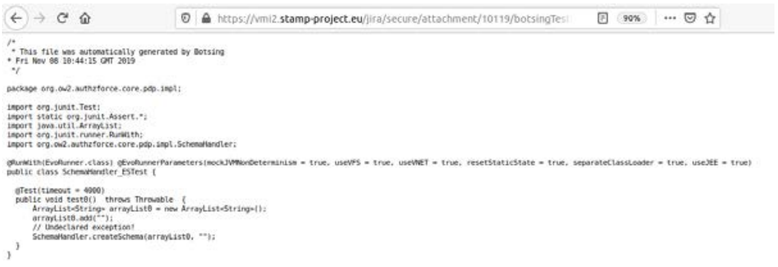
Agile

View on Board

HipChat discussions

Do you want to discuss this issue? Connect to HipChat.

Figure 8.3: Crash reproducing test attached to the issue.


https://vmi2.stamp-project.eu/jira/secure/attachment/10119/botsingTest
90%

```
/*
 * This file was automatically generated by Botsing
* Fri Nov 08 10:44:15 GMT 2019
 */

package org.ow2.authzforce.core.pdp.impl;

import org.junit.Test;
import static org.junit.Assert.*;
import java.util.ArrayList;
import org.junit.runner.RunWith;
import org.ow2.authzforce.core.pdp.impl.SchemaHandler;

@RunWith(EvoRunner.class) @EvoRunnerParameters(mockJVMNonDeterminism = true, useVFS = true, useVNET = true, resetStaticState = true, separateClassLoader = true, useJEE = true)
public class SchemaHandler_ESTest {

  @Test(timeout = 4000)
  public void test0()  throws Throwable  {
      ArrayList<String> arrayList0 = new ArrayList<String>();
      arrayList0.add("");
      // Undeclared exception!
      SchemaHandler.createSchema(arrayList0, "");
  }
}
```

Figure 8.4: A crash reproducing test case by Botsing Jira plugin.

Cling

Cling is an open-source class-integration test generation tool for Java. It gets two classes, which are calling each other, and tries to generate tests covering different interactions between these two given classes. Section 6.4 of this thesis has shown that this approach actually managed to detect the integration-level faults.

In the same way as EvoSuite, which has been used widely on industry [25] for generating unit-level tests, Cling has the potential to be used by software projects for generating integration-level tests complementing the tests generated by EvoSuite. For this reason, we intend to implement different plugins for Maven, IntelliJ, and Jenkins to ease the application of Cling on different projects.

Commonality Score For Unit Test Generation

As mentioned in Section 7.5, we still need to perform a more in-depth investigation about the usefulness of commonality score in finding faults. However, since the introduced secondary objectives based on this metric aim to change how the statements are covered, we believe they can impact the test cases' understandability generated by the search process

All tools unitedly

The growths of the techniques introduced in this thesis will lead to a set of tools, alleviating developers from doing much groundwork in software testing. A developer can use each of these tools to generate a set of test cases/suites as a starting point for testing unit classes, class integrations, or for debugging reported crashes. Since these approaches are automated, they take machine time, meaning that precious developer time can be spent on other tasks. Moreover, by new implementations and executions of the system under test, these approaches will have more data to improve their test generation and search guidance. Thereby, the benefits stemming from these approaches will increase as the software grows.

8.3 Recommendations For Future Work

This thesis shows that there are still many ways to improve the search-based test generation techniques for various criteria. Hence, this section gives some recommendations for future work.

8.3.1 Search-based Crash Reproduction

R1: *Crash Distance* improvements

As described in section 2.1.2, the *Crash Distance* fitness function is used in search-based crash reproduction to measure the distance of each generated test from throwing the same crash as the given one. The three elements in this fitness function may lead to flat landscapes in the search space, which is undesirable since it provides no guidance to the search. For instance, **the exception coverage** heuristic is a binary value, which indicates whether the same type of exception (as the given one) is thrown or not. Devising fitness functions that leverage all values between 0 and 1 instead, would offer more guidance.

R2: Other secondary and helper objectives

Chapters 4 and 5 show how adding secondary objectives and conflicting helper-objectives help the existing *Crash Distance* fitness function in reproducing more crashes. However, there are still many possibilities to improve search objectives. First of all, the *Crash Distance* fitness function itself can be modified to make sure that we have less flat landscapes in the search space. Also, more helper objectives can be combined with this fitness function to improve the exploration of the search process.

8.3.2 Search-based Integration Testing

R3: Multiple class integration

In Chapter 6, we introduced a new search-based technique for testing interactions between two coupled classes. In the next step, this approach can be extended to handle more classes. It might even be beneficial to design an approach to test the integration of two modules that contain multiple classes.

8.3.3 Carving Knowledge For Search-based Test Generation

This thesis showed the benefits of using knowledge collected from different sources in search-based test generation. Hence, this section describes how the novel techniques introduced in this thesis can be extended.

R4: Using more resources

This thesis introduced various strategies to utilize the carved information from source code (*e.g.,* Chapter 6), existing test cases (*e.g.,* Chapter 3), and execution logs (Chapter 7) in search-based test generation. However, other resources such as commits addressing previously detected faults can be used to generate more realistic test cases during the search process. Moreover, other useful information can be carved from the application's documentation.

R5: Carving more information

This thesis has used carved information from method call sequences (*e.g.,* Chapters 3 and 4), call-sites (Chapter 6), and test execution patterns (Chapter 7). Other data, such as information regarding the input parameters, can be carved from the existing sources for white-box search-based test generation to continue this research path. For example, some applications need strings, following a specific grammar and pattern (*e.g.,* XML, JSON), to be used in their testing. These kinds of information can be carved from source code and documentation.

Bibliography

References

[1] Boris Cherry, Xavier Devroey, and Pouria Derakhshanfar. Crash reproduction difficulty, an initial assessment. In Mike Papadakis and Maxime Cordy, editors, Proceedings of the 19th Belgium-Netherlands Software Evolution Workshop (BENEVOL '20), Luxembourg, Luxembourg, dec 2020.

[2] Pouria Derakhshanfar. Well-informed test case generation and crash reproduction. In 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), pages 424–426. IEEE, 2020.

[3] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In Future of Software Engineering (FOSE'07), pages 85–103. IEEE, 2007.

[4] Phil McMinn. Search-based software test data generation: A survey. Software Testing Verification and Reliability, 14(2):105–156, 2004.

[5] M. Harman and B. Jones. Seminal software engineering using metaheuristic innovative algorithms. In Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001. IEEE Comput. Soc.

[6] 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, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM.

[7] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. Combining symbolic execution and search-based testing for programs with complex heap inputs. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017, pages 90–101, New York, NY, USA, 2017. ACM.

[8] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. Sushi: a test generator for programs with complex structured inputs. In 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), pages 21–24. IEEE, 2018.

[9] IS Wishnu B Prasetya. T3, a combinator-based random testing tool for java: benchmarking. In International Workshop on Future Internet Testing, pages 101–110. Springer, 2013.

[10] Andrea Arcuri. RESTful API automated test case generation with Evomaster. ACM Transactions on Software Engineering and Methodology, 28(1):1–37, 2019.

[11] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In Presented as part of the 21st USENIX Security Symposium (USENIX Security 12), pages 445–458, Bellevue, WA, 2012. USENIX.

[12] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. pages 329–340, 2019.

[13] Michael Beyene and James H Andrews. Generating string test data for code coverage. In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, pages 270–279. IEEE, 2012.

[14] David Coppit and Jiexin Lian. Yagg: an easy-to-use generator for structured test inputs. In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pages 356–359. ACM, 2005.

[15] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In ACM Sigplan Notices, volume 43, pages 206–215. ACM, 2008.

[16] Salah Ghamizi, Maxime Cordy, Martin Gubri, Mike Papadakis, Andrey Boystov, Yves Le Traon, and Anne Goujon. Search-based adversarial testing and improvement of constrained credit scoring systems. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, November 2020.

[17] Raveendra Kumar Medicherla, Raghavan Komondoor, and Abhik Roychoudhury. Fitness guided vulnerability detection with greybox fuzzing. In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, pages 513–520, 2020.

[18] Bogdan Korel and Ali M Al-Yami. Assertion-oriented automated test data generation. In Proceedings of IEEE 18th International Conference on Software Engineering, pages 71–80. IEEE, 1996.

[19] Nigel Tracey, John Clark, Keith Mander, and John McDermid. Automated test-data generation for exception conditions. Software: Practice and Experience, 30(1):61–79, 2000.

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

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

[22] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges. Proceedings - 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, pages 201–211, 2016.

[23] Mariano Ceccato, Alessandro Marchetto, Leonardo Mariani, Cu D. Nguyen, and Paolo Tonella. Do automatically generated test cases make debugging easier? an experimental assessment of debugging effectiveness and efficiency. ACM Trans. Softw. Eng. Methodol., 25(1):5:1–5:38, December 2015.

[24] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. Deploying Search Based Software Engineering with Sapienz at Facebook. In Search-Based Software Engineering. SSBSE 2018., volume 11036 of LNCS. Springer, 2018.

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

[26] Gregory Gay, Matt Staats, Michael Whalen, and Mats PE Heimdahl. The risks of coverage-directed test case generation. IEEE Transactions on Software Engineering, 41(8):803–819, 2015.

[27] Alireza Salahirad, Hussein Almulla, and Gregory Gay. Choosing the fitness function for the job: Automated generation of test suites that detect real faults. STVR, 29(4-5):e1701, jun 2019.

[28] Mozhan Soltani, Annibale Panichella, and Arie Van Deursen. Search-Based Crash Reproduction and Its Impact on Debugging. IEEE Transactions on Software Engineering, 2018.

[29] Francesco A. Bianchi, Mauro Pezzè, and Valerio Terragni. Reproducing concurrency failures from crash stacks. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017, pages 705–716. ACM Press, 2017.

[30] Ning Chen and Sunghun Kim. STAR: Stack trace based automatic crash reproduction via symbolic execution. IEEE Trans. on Software Engineering, 41(2):198–220, 2015.

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

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

[33] Jifeng Xuan, Xiaoyuan Xie, and Martin Monperrus. Crash reproduction via test case mutation: Let existing test cases help. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015, pages 910–913, New York, New York, USA, 2015. ACM Press.

[34] Phil McMinn. Search-based software testing: Past, present and future. In Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, ICSTW '11, pages 153–163, Washington, DC, USA, 2011. IEEE Computer Society.

[35] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. An empirical evaluation of evolutionary algorithms for unit test suite generation. Information and Software Technology, 104(August):207–235, 2018.

[36] Mozhan Soltani, Annibale Panichella, and Arie van Deursen. A guided genetic algorithm for automated crash reproduction. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pages 209–220, Buenos Aires, Argentina, may 2017. IEEE.

[37] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C. Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. Proceedings - International Conference on Software Engineering, 14-22-May-2016:547–558, 2016.

[38] A. Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J. Hellendoorn. Revisiting test smells in automatically generated tests: Limitations, pitfalls, and opportunities. In 36th IEEE The International Conference on Software Maintenance and Evolution (ICSME 2020). IEEE, August 2020.

[39] Frances E. Allen. Control flow analysis. ACM SIGPLAN Notices, 5(7):1–19, jul 1970.

[40] Annibale Panichella, José Campos, and Gordon Fraser. Evosuite at the sbst 2020 tool competition. In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, pages 549–552, 2020.

[41] Gordon Fraser and Andrea Arcuri. Whole test suite generation. IEEE Transactions on Software Engineering, 39(2):276–291, feb 2013.

[42] 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, 2018.

[43] Zhengshan Wang, Bixin Li, Lulu Wang, Meng Wang, and Xufang Gong. Using Coupling Measure Technique and Random Iterative Algorithm for Inter-Class Integration Test Order Problem. In 2010 IEEE 34th Annual Computer Software and Applications Conference Workshops, volume 1, pages 329–334. IEEE, jul 2010.

[44] Michael Steindl and Juergen Mottok. Optimizing Software Integration by Considering Integration Test Complexity and Test Effort. In Proceedings of the 10th International Workshop on Intelligent Solutions in Embedded Systems, pages 63–68. IEEE, 2012.

[45] N.L. Hashim, H.W. Schmidt, and Sita Ramakrishnan. Test Order for Class-based Integration Testing of Java Applications. In Fifth International Conference on Quality Software (QSIC'05), volume 2005, pages 11–18. IEEE, 2005.

[46] Silvia Regina Vergilio, Aurora Pozo, João Carlos Garcia Árias, Rafael da Veiga Cabral, and Tiago Nobre. Multi-objective optimization algorithms applied to the class integration and test order problem. International Journal on Software Tools for Technology Transfer, 14(4):461–475, aug 2012.

[47] Priti Bansal, Sangeeta Sabharwal, and Pameeta Sidhu. An investigation of strategies for finding test order during Integration testing of object Oriented applications. In 2009 Proceeding of International Conference on Methods and Models in Computer Science (ICM2CS), pages 1–8. IEEE, dec 2009.

[48] S. Jiang, Miao Zhang, Yanmei Zhang, Rongcun Wang, Qiao Yu, and Jacky Wai Keung. An Integration Test Order Strategy to Consider Control Coupling. IEEE Transactions on Software Engineering, 5589(c):1–1, 2019.

[49] Lars Borner and Barbara Paech. Integration Test Order Strategies to Consider Test Focus and Simulation Effort. In 2009 First International Conference on Advances in System Testing and Validation Lifecycle, pages 80–85. IEEE, sep 2009.

[50] Thainá Mariani, Giovani Guizzo, Silvia R Vergilio, and Aurora T.R. Pozo. Grammatical Evolution for the Multi-Objective Integration and Test Order Problem. In Proceedings of the 2016 on Genetic and Evolutionary Computation Conference - GECCO '16, pages 1069–1076, Madrid, Spain, 2016. ACM Press.

[51] Giovani Guizzo, Gian Mauricio Fritsche, Silvia Regina Vergilio, and Aurora Trinidad Ramirez Pozo. A Hyper-Heuristic for the Multi-Objective Integration and Test Order Problem. In Proceedings of the 2015 on Genetic and Evolutionary Computation Conference - GECCO '15, pages 1343–1350, Madrid, Spain, 2015. ACM Press.

[52] Aynur Abdurazik and Jeff Offutt. Using Coupling-Based Weights for the Class Integration and Test Order Problem. The Computer Journal, 52(5):557–570, aug 2009.

[53] Rafael da Veiga Cabral, Aurora Pozo, and Silvia Regina Vergilio. A Pareto Ant Colony Algorithm Applied to the Class Integration and Test Order Problem. In IFIP International Conference on Testing Software and Systems, volume 6435 LNCS of ICTSS 2010, pages 16–29. Springer, 2010.

[54] L.C. Briand, Yvan Labiche, and Yihong Wang. An investigation of graph-based class integration test order strategies. IEEE Transactions on Software Engineering, 29(7):594–607, jul 2003.

[55] Shaukat Ali Khan and Aamer Nadeem. Automated Test Data Generation for Coupling Based Integration Testing of Object Oriented Programs Using Evolutionary Approaches. In 2013 10th International Conference on Information Technology: New Generations, pages 369–374. IEEE, apr 2013.

[56] Shaukat Ali Khan and Aamer Nadeem. Automated Test Data Generation for Coupling Based Integration Testing of Object Oriented Programs Using Particle Swarm Optimization (PSO). In Jeng-Shyang Pan, Pavel Krömer, and Václav Snášel, editors,

Proceedings of the Seventh International Conference on Genetic and Evolutionary Computing, ICGEC 2013, volume 238 of Advances in Intelligent Systems and Computing, pages 115–124, Cham, 2014. Springer International Publishing.

[57] Andreas Zeller. Why Programs Fail, Second Edition: A Guide to Systematic Debugging. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2009.

[58] Daniele Gagliardi, Mael Audren De Kerdrel, Luca Andreatta, Nicola Bertazzo, Ciro Formisano, Daniele Gagliardi, Jesús Gorroñogoitia Cruz, Caroline Landry, and Ricardo Tejada. STAMP Deliverable D4.4: Final public version of API and implementation of services and courseware. Technical report, STAMP project, 2019.

[59] Pouria Derakhshanfar, Xavier Devroey, Gilles Perrouin, Andy Zaidman, and Arie Deursen. Search-based crash reproduction using behavioural model seeding. STVR, 30(3):e1733, may 2020.

[60] Hevner, March, Park, and Ram. Design Science in Information Systems Research. MIS Quarterly, 28(1):75, 2004.

[61] Wikipedia contributors. Open science — Wikipedia, the free encyclopedia, 2020. [Online; accessed 21-September-2020].

[62] Pouria Derakhshanfar and Xavier Devroey. Jcrashpack: A java crash reproduction benchmark, April 2020.

[63] Pouria Derakhshanfar, Xavier Devroey, Gilles Perrouin, Andy Zaidman, and Arie van Deursen. Replication package of "Search-based Crash Reproduction using Behavioral Model Seeding", October 2019.

[64] Pouria Derakhshanfar, Xavier Devroey, Andy Zaidman, Arie van Deursen, and Annibale Panichella. Replication package of "Good Things Come In Threes: Improving Search-based Crash Reproduction With Helper Objectives", August 2020.

[65] Pouria Derakhshanfar and Xavier Devroey. Replication package of Basic Block Coverage for Search-Based Crash Reproduction.

[66] Pouria Derakhshanfar and Xavier Devroey. Replication package of generating class-level integration tests using call site information, December 2020.

[67] Björn Evers, Pouria Derakhshanfar, Xavier Devroey, and Andy Zaidman. Unit test generation for common and uncommon behaviors: dataset, June 2020.

[68] 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 Thelma Elita Colanzi and Phil McMinn, editors, Symposium on Search-Based Software Engineering. SSBSE 2018., volume 11036 of LNCS, pages 325–340, Montpellier, France, 2018. Springer.

[69] Mael Audren, Mohamed Boussaa, Lars Thomas Boye, Pierre-Yves Gibello, Jesús Gorroñogoitia, Vincent Massol, Fernando Mendez, Assad Montasser, and Pedro Velho. STAMP WP5 - D5.7 - Use Cases Validation Report V3. https://www.stamp-project.eu/view/main/deliverables, 2019.

[70] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramler. Grt: Program-analysis-guided random testing (t). In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 212–223. IEEE, 2015.

[71] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014, pages 437–440, San Jose, CA, USA, 2014. ACM Press.

[72] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pages 532–543. ACM, 2015.

[73] Matias Martinez and Martin Monperrus. Astor: A program repair library for java. In Proceedings of the 25th International Symposium on Software Testing and Analysis, pages 441–444. ACM, 2016.

[74] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In Proceedings of the 39th International Conference on Software Engineering, pages 609–620. IEEE Press, 2017.

[75] Tien-Duy B Le, David Lo, Claire Le Goues, and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants. In Proceedings of the 25th International Symposium on Software Testing and Analysis, pages 177–188. ACM, 2016.

[76] Tanzeem Bin Noor and Hadi Hemmati. A similarity-based approach for test case prioritization using historical failure data. In 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), pages 58–68. IEEE, 2015.

[77] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. How does regression test prioritization perform in real-world software evolution? In 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pages 535–546. IEEE, 2016.

[78] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of java source code. Software: Practice and Experience, 46:1155–1179, 2015.

[79] Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan De Halleux. Precise identification of problems for structural test generation. In Software Engineering (ICSE),

2011 33rd International Conference on, pages 611–620, Waikiki, Honolulu , HI, USA, 2011. IEEE, ACM.

[80] Lionel Briand, Domenico Bianculli, Shiva Nejati, Fabrizio Pastore, and Mehrdad Sabetzadeh. The Case for Context-Driven Software Engineering Research: Generalizability Is Overrated. IEEE Software, 34(5):72–75, 2017.

[81] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. On the dichotomy of debugging behavior among programmers. In Proceedings of the International Conference on Software Engineering (ICSE), pages 572–583. ACM, 2018.

[82] Shay Artzi, Sunghun Kim, and Michael D. Ernst. Recrash: Making software failures reproducible by preserving object states. In Proceedings of the 22Nd European Conference on Object-Oriented Programming, ECOOP '08, pages 542–565, Berlin, Heidelberg, 2008. Springer-Verlag.

[83] James Clause and Alessandro Orso. A technique for enabling and supporting debugging of field failures. In Proceedings of the 29th International Conference on Software Engineering, ICSE '07, pages 261–270, Washington, DC, USA, 2007. IEEE Computer Society.

[84] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In Proceedings of the 32Nd Annual International Symposium on Computer Architecture, ISCA '05, pages 284–295, Washington, DC, USA, 2005. IEEE Computer Society.

[85] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. jrapture: A capture/replay tool for observation-based testing. In Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '00, pages 158–167, New York, NY, USA, 2000. ACM.

[86] María Gómez, Romain Rouvoy, Bram Adams, and Lionel Seinturier. Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring. In Proceedings of the International Conference on Mobile Software Engineering and Systems, MOBILESoft '16, pages 88–99, New York, NY, USA, 2016. ACM.

[87] Jonathan Bell, Nikhil Sarda, and Gail Kaiser. Chronicler: Lightweight recording to reproduce field failures. In Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pages 362–371, Piscataway, NJ, USA, 2013. IEEE Press.

[88] Yu Cao, Hongyu Zhang, and Sun Ding. Symcrash: Selective recording for reproducing crashes. In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, pages 791–802. ACM, 2014.

[89] Tingting Yu, Tarannum S Zaman, and Chao Wang. Descry: reproducing system-level concurrency failures. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pages 694–704. ACM, 2017.

[90] Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, ASPLOS XV, pages 155–166, New York, NY, USA, 2010. ACM.

[91] Andreas Leitner, Ilinca Ciupa, Manuel Oriol, Bertrand Meyer, and Arno Fiva. Contract driven development = test driven development - writing test cases. In Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE '07, pages 425–434, New York, NY, USA, 2007. ACM.

[92] 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.

[93] 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 Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, ASE'13, pages 604–609, Piscataway, NJ, USA, 2013. IEEE Press.

[94] Fitsum Meshesha Kifetew, Wei Jin, Roberto Tiella, Alessandro Orso, and Paolo Tonella. Reproducing field failures for programs with complex grammar-based input. In Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation, ICST '14, pages 163–172, Washington, DC, USA, 2014. IEEE Computer Society.

[95] Apache. Ant. `http://ant.apache.org/`, 2017. [Online; accessed 25-January-2018].

[96] Cristian Zamfir and George Candea. Execution synthesis: A technique for automated software debugging. In Proceedings of the 5th European Conference on Computer Systems, EuroSys '10, pages 321–334, New York, NY, USA, 2010. ACM.

[97] Wei Jin and Alessandro Orso. BugRedux: reproducing field failures for in-house debugging. In 2012 34th International Conference on Software Engineering (ICSE), pages 474–484. IEEE, jun 2012.

[98] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14, pages 249–265, Berkeley, CA, USA, 2014. USENIX Association.

[99] 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 Proceedings of the 12th Working Conference on Mining Software Repositories, MSR '15, pages 134–145, Piscataway, NJ, USA, 2015. IEEE Press.

[100] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys (CSUR), 45(1):11, 2012.

[101] Susan Elliott Sim, Steve Easterbrook, and Richard C Holt. Using Benchmarking to Advance Research: A Challenge to Software Engineering. In Proceedings of the 25th International Conference on Software Engineering, ICSE '03, pages 74–83, Portland, Oregon, USA, 2003. IEEE Computer Society.

[102] Apache. Commons Collections. `https://commons.apache.org/proper/commons-collections/`, 2017. [Online; accessed 25-January-2018].

[103] Apache. Log4j. `https://logging.apache.org/log4j/2.x/`, 2017. [Online; accessed 25-January-2018].

[104] XWiki. The Advanced Open Source Enterprise and Application Wiki. `http://www.xwiki.org/`, 2018. [Online; accessed 25-January-2018].

[105] Elastic. Elasticsearch: RESTful, Distributed Search and Analytics. `https://www.elastic.co/products/elasticsearch`, 2018. [Online; accessed 25-January-2018].

[106] Java Design Patterns. Design patterns implemented in Java. `http://java-design-patterns.com`, 2018. [Online; accessed 25-January-2018].

[107] Dubbo. A high-performance, java based, open source RPC framework. `http://dubbo.io`, 2018. [Online; accessed 25-January-2018].

[108] RxJava. Reactive Extensions for the JVM. `https://github.com/ReactiveX/RxJava`, 2018. [Online; accessed 25-January-2018].

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

[110] 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.

[111] Andrea Arcuri and Gordon Fraser. On Parameter Tuning in Search Based Software Engineering. In Population English Edition, pages 33–47. 2011.

[112] Barbara Liskov and John Guttag. Program development in JAVA: abstraction, specification, and
Pearson Education, London, England, UK, 2000.

[113] Andrea Arcuri, Gordon Fraser, and René Just. Private API access and functional mocking in automated unit test generation. In Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on, pages 126–137, Tokyo, Japan, 2017. IEEE, IEEE Computer Society.

[114] Gordon Fraser and Andrea Arcuri. Automated test generation for java generics. In International Conference on Software Quality, pages 185–198, Vienna, Austria, 2014. Springer, Springer.

[115] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. IEEE Transactions on Software Engineering, 30(1):3–16, 2004.

[116] Jan Malburg and Gordon Fraser. Combining search-based and constraint-based testing. In Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on, pages 436–439, Lawrence, KS, USA, 2011. IEEE, IEEE Computer Society.

[117] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. Automated unit test generation for classes with environment dependencies. In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14, pages 79–90, Vasteras, Sweden, 2014. ACM Press.

[118] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014, pages 643–653, 2014.

[119] Oracle. What's New in JDK 8. `https://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html`, 2019. Accessed: 2019-05-14.

[120] JDK. Stack trace has invalid line numbers. `https://bugs.openjdk.java.net/browse/JDK-7024096`, 2016. [Online; accessed 25-January-2018].

[121] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. Experimentation in Software Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[122] Mathieu Nayrolles and Abdelwahab Hamou-Lhadj. BUMPER: A Tool for Coping with Natural Language Searches of Millions of Bugs and Fixes. In 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pages 649–652, Suita, Osaka, Japan, mar 2016. IEEE.

[123] Andrea Arcuri. RESTful API Automated Test Case Generation. In 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), pages 9–20. IEEE, jul 2017.

[124] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. Seeding strategies in search-based unit test generation. Softw. Test. Verif. Reliab., 26(5):366–401, 2016.

[125] James Roche. Adopting DevOps practices in quality assurance. Communications of the ACM, 56(11):38–43, 2013.

[126] Bruno Cabral and Paulo Marques. Exception Handling: A Field Study in Java and .NET. In ECOOP 2007 – Object-Oriented Programming, volume 4609, pages 151–175. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[127] Roberta Coelho, Lucas Almeida, Georgios Gousios, Arie van Deursen, and Christoph Treude. Exception handling bug hazards in Android. Empirical Software Engineering, 22(3):1264–1304, jun 2017.

[128] Abdou Maiga, Abdelwahab Hamou-Lhadj, Mathieu Nayrolles, Korosh Koochekian Sabor, and Alf Larsson. An empirical study on the handling of crash reports in a large software company: An experience report. In 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 342–351, Bremen, Germany, sep 2015. IEEE.

[129] Yanchuan Li and Gordon Fraser. Bytecode testability transformation. In International Symposium on Search Based Software Engineering, pages 237–251, Szeged, Hungary, 2011. Springer, Springer.

[130] Mark Harman, Lin Hu, Robert Hierons, André Baresel, and Harmen Sthamer. Improving evolutionary testing by flag removal. In Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation, pages 1359–1366, New York, USA, 2002. Morgan Kaufmann Publishers Inc., Morgan Kaufmann.

[131] André Baresel, David Binkley, Mark Harman, and Bogdan Korel. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In ACM SIGSOFT Software Engineering Notes, volume 29, pages 108–118, Boston, Massachusetts, USA, 2004. ACM, ACM.

[132] Mark Utting and Bruno Legeard. Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, 2007.

[133] Martin White, Mario Linares Vásquez, Peter Johnson, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Generating reproducible and replayable bug reports from android application crashes. In Andrea De Lucia, Christian Bird, and Rocco Oliveto, editors, Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC 2015, Florence/Firenze, Italy, May 16-24, 2015, pages 48–59. IEEE Computer Society, 2015.

[134] Mathieu Nayrolles, Abdelwahab Hamou-Lhadj, Sofiene 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, mar 2015.

[135] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering. ACM Computing Surveys, 45(1):1–61, nov 2012.

[136] Gordon Fraser and Andrea Arcuri. The Seed is Strong: Seeding Strategies in Search-Based Software Testing. In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, pages 121–130. IEEE, apr 2012.

[137] Tao Chen, Miqing Li, and Xin Yao. On the Effects of Seeding Strategies: A Case for Search-based Multi-Objective Service Composition. In Proceedings of the Genetic and Evolutionary Computation Conference on - GECCO '18, pages 1419–1426. ACM Press, 2018.

[138] Roberto E Lopez-Herrejon, Javier Ferrer, Francisco Chicano, Alexander Egyed, and Enrique Alba. Comparative analysis of classical multi-objective evolutionary algorithms and seeding strategies for pairwise testing of Software Product Lines. In 2014 IEEE Congress on Evolutionary Computation (CEC), pages 387–396. IEEE, jul 2014.

[139] Phil McMinn, Muzammil Shahbaz, and Mark Stevenson. Search-Based Test Input Generation for String Data Types Using the Results of Web Queries. In IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST '12), pages 141–150. IEEE, apr 2012.

[140] Luca Della Toffola, Cristian-Alexandru Staicu, and Michael Pradel. Saying 'Hi!' is not enough: Mining inputs for effective test generation. In 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 44–49. IEEE, oct 2017.

[141] Nadia Alshahwan and Mark Harman. Automated web application testing using search based software engineering. In 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pages 3–12. IEEE, nov 2011.

[142] Christel Baier and Joost-Pieter Katoen. Principles of Model Checking. MIT Press, 2007.

[143] Jan Tretmans. Model based testing with labelled transition systems. Formal methods and testing, pages 1–38, 2008.

[144] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Hamza Samih, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Statistical prioritization for software product line testing: an experience report. Software & Systems Modeling, 16(1):153–171, feb 2017.

[145] Emanuela G Cartaxo, Patrícia D L Machado, and Francisco G Oliveira Neto. On the use of a similarity function for test case selection in the context of model-based testing. Software Testing, Verification and Reliability, 21(2):75–100, 2011.

[146] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. Achieving scalable model-based testing through test case diversity. ACM Transactions on Software Engineering and Methodology, 22(1):1–42, feb 2013.

[147] Debajyoti Mondal, Hadi Hemmati, and Stephane Durocher. Exploring test suite diversification and code coverage in multi-objective test case selection. In 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), ICST '15, pages 1–10. IEEE, apr 2015.

[148] Paul Jaccard. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. Bulletin del la Société Vaudoise des Sciences Naturelles, 37:547–579, 1901.

[149] Steffen Herbold, Patrick Harms, and Jens Grabowski. Combining usage-based and model-based testing for service-oriented architectures in the industrial practice. International Journal on Software Tools for Technology Transfer, 19(3):309–324, jun 2017.

[150] Maikel Leemans, Wil M. P. van der Aalst, and Mark G. J. van den Brand. The Statechart Workbench: Enabling scalable software event log analysis using process mining. In 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 502–506. IEEE, mar 2018.

[151] Sara E Sprenkle, Lori L Pollock, and Lucy M Simko. Configuring effective navigation models and abstract test cases for web applications by analysing user behaviour. Software Testing, Verification and Reliability, 23(6):439–464, 2013.

[152] Sara Sprenkle, Lori Pollock, and Lucy Simko. A Study of Usage-Based Navigation Models and Generated Abstract Test Cases for Web Applications. In Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on, pages 230–239. IEEE, mar 2011.

[153] Paolo Tonella, Roberto Tiella, and Cu Duy Nguyen. Interpolated n-grams for model based testing. In Proceedings of the 36th International Conference on Software Engineering - ICSE 2014, pages 562–572. ACM Press, 2014.

[154] Sicco Verwer and Christian A Hammerschmidt. flexfringe: A Passive Automaton Learning Package. In L O'Conner, editor, 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 638–642. IEEE, sep 2017.

[155] Paolo Tonella, Alessandro Marchetto, Cu Duy Nguyen, Yue Jia, Kiran Lakhotia, and Mark Harman. Finding the optimal balance between over and under approximation of models inferred from execution logs. In Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, pages 21–30. IEEE, 2012.

[156] Gordon Fraser and Andreas Zeller. Exploiting Common Object Usage in Test Case Generation. In 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, pages 80–89. IEEE, mar 2011.

[157] Ivo Krka, Yuriy Brun, Daniel Popescu, Joshua Garcia, and Nenad Medvidovic. Using dynamic execution traces and program invariants to enhance behavioral model inference. In Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE '10, pages 179–182. ACM, 2010.

[158] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In Proceedings of the 13th international conference on Software engineering - ICSE '08, page 501. ACM Press, 2008.

[159] Carlo Ghezzi, Mauro Pezzè, Michele Sama, and Giordano Tamburrelli. Mining Behavior Models from User-intensive Web Applications. In Proceedings of the 36th International Conference on Software Engineering, ICSE '14, pages 277–287, Hyderabad, India, 2014. ACM Press.

[160] S.J Prowell and J.H Poore. Computing system reliability using Markov chain usage models. Journal of Systems and Software, 73(2):219–225, oct 2004.

[161] Yuanyuan Zhang, Mark Harman, Yue Jia, and Federica Sarro. Inferring Test Models from Kate's Bug Reports Using Multi-objective Search. In Márcio Barros and Yvan Labiche, editors, Search-Based Software Engineering, SSBSE '15, pages 301–307. Springer International Publishing, 2015.

[162] W. Dulz and Fenhua Zhen. MaTeLo - statistical usage testing by annotated sequence diagrams, Markov chains and TTCN-3. In Third International Conference on Quality Software, 2003. Proceedings., pages 336–342. IEEE, 2003.

[163] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. Exploration and exploitation in evolutionary algorithms: A survey. ACM Comput. Surv., 45(3), 2013.

[164] Xavier Devroey, Gilles Perrouin, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Search-based Similarity-driven Behavioural SPL Testing. In Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems - VaMoS '16, pages 89–96, Salvador, Brazil, jan 2016. ACM Press.

[165] 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.

[166] Andrea Arcuri and Gordon Fraser. Parameter tuning or default values? An empirical investigation in search-based software engineering. Empirical Software Engineering, 18(3):594–623, jun 2013.

[167] Annibale Panichella and Urko Rueda Molina. Java unit testing tool competition - Fifth round. Proceedings - 2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing, SBST 2017, pages 32–38, 2017.

[168] Mikkel T Jensen. Helper-objectives: Using multi-objective evolutionary algorithms for single-objective optimisation. Journal of Mathematical Modelling and Algorithms, 3(4):323–347, 2004.

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

[170] Nasser M Albunian. Diversity in search-based unit test suite generation. In International Symposium on Search Based Software Engineering, pages 183–189. Springer, 2017.

[171] 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.

[172] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm. TIK-report, 103, 2001.

[173] Qingfu Zhang and Hui Li. Moea/d: A multiobjective evolutionary algorithm based on decomposition. IEEE Transactions on evolutionary computation, 11(6):712–731, 2007.

[174] David W. Corne, Nick R. Jerram, Joshua D. Knowles, and Martin J. Oates. Pesa-ii: Region-based selection in evolutionary multiobjective optimization. In Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation, GECCO 01, pages 283–290, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[175] Marco Laumanns, Lothar Thiele, Eckart Zitzler, Emo Welzl, and Kalyanmoy Deb. Running time analysis of multi-objective evolutionary algorithms on a simple discrete optimization problem. In International Conference on Parallel Problem Solving from Nature, pages 44–53. Springer, 2002.

[176] Vladimir Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In Soviet Physics Doklady, volume 10, pages 707–710, 1966.

[177] R. W. Hamming. Error Detecting and Error Correcting Codes. Bell System Technical Journal, 29(2):147–160, apr 1950.

[178] Kaisa Miettinen. Nonlinear Multiobjective Optimization: Kaisa Miettinen. Springer US, 1st edition, 1999.

[179] Indraneel Das and J. E. Dennis. Normal-boundary intersection: A new method for generating the pareto surface in nonlinear multicriteria optimization problems. SIAM J. on Optimization, 8(3):631––657, March 1998.

[180] Achille Messac, Amir Ismail-Yahaya, and Christopher A Mattson. The normalized normal constraint method for generating the pareto frontier. Structural and multidisciplinary optimization, 25(2):86–98, 2003.

[181] Yan-Yan Tan, Yong-Chang Jiao, Hong Li, and Xin-Kuan Wang. A modification to moea/d-de for multiobjective optimization problems with complicated pareto sets. Information Sciences, 213:14–38, 2012.

[182] Juan J Durillo, Antonio J Nebro, and Enrique Alba. The jmetal framework for multi-objective optimization: Design and architecture. In IEEE congress on evolutionary computation, pages 1–8. IEEE, 2010.

[183] Salvador García, Daniel Molina, Manuel Lozano, and Francisco Herrera. A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: a case study on the cec'2005 special session on real parameter optimization. Journal of Heuristics, 15(6):617, May 2008.

[184] Ronald A Fisher. The use of multiple measurements in taxonomic problems. Annals of Eugenics, 7(2):179–188, 1936.

[185] Sadeeq Jan, Annibale Panichella, Andrea Arcuri, and Lionel Briand. Automatic Generation of Tests to Exploit XML Injection Vulnerabilities in Web Applications. IEEE Transactions on Software Engineering, (i):1–27, 2017.

[186] W. J. Conover and Ronald L. Iman. Rank transformations as a bridge between parametric and nonparametric statistics. The American Statistician, 35(3):124–129, 1981.

[187] S. Holm. A simple sequentially rejective multiple test procedure. Scandinavian Journal of Statistics, 6:65–70, 1979.

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

[189] Fortunato Pesarin and Luigi Salmaso. Permutation tests for complex data: theory, applications and software. John Wiley & Sons, 2010.

[190] Andrea Arcuri. RESTful API automated test case generation with evomaster. ACM Transactions on Software Engineering and Methodology (TOSEM), 28(1):1–37, 2019.

[191] Paulo Borba, Ana Cavalcanti, Augusto Sampaio, and Jim Woodcook. Testing techniques in software engineering: Second pernambuco summer school on software engineering, PSSE 2007, Recife, Brazil, December 3-7, 2007, Revised Lectures, volume 6153. Springer, 2010.

[192] Gordon Fraser and Andrea Arcuri. 1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite. Empirical Software Engineering, 20(3):611–639, 2015.

[193] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. Combining multiple coverage criteria in search-based unit test generation. In International Symposium on Search Based Software Engineering, pages 93–108. Springer, 2015.

[194] José Campos, Yan Ge, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. An empirical evaluation of evolutionary algorithms for test suite generation. In Tim Menzies and Justyna Petke, editors, Symposium on Search Based Software Engineering (SSBSE '17), volume 10452 of LNCS, pages 33–48, Cham, 2017. Springer International Publishing.

[195] Amanda Schwartz, Daniel Puckett, Ying Meng, and Gregory Gay. Investigating faults missed by test suites achieving high code coverage. Journal of Systems and Software, 144:106–120, 2018.

[196] Yi Wei, Bertrand Meyer, and Manuel Oriol. Is branch coverage a good measure of testing effectiveness? In Empirical Software Engineering and Verification, pages 194–212. Springer, 2012.

[197] Zhenyi Jin and A. Jefferson Offutt. Coupling-based criteria for integration testing. Software Testing, Verification and Reliability, 8(3):133–154, sep 1998.

[198] A.J. Offutt, Aynur Abdurazik, and R.T. Alexander. An analysis tool for coupling-based integration testing. In Proceedings Sixth IEEE International Conference on Engineering of Complex Computer Systems. ICECCS 2000, pages 172–178. IEEE Comput. Soc, 2000.

[199] Ting Su, Ke Wu, Weikai Miao, Geguang Pu, Jifeng He, Yuting Chen, and Zhendong Su. A Survey on Data-Flow Testing. ACM Computing Surveys, 50(1):1–35, apr 2017.

[200] Mary Jean Harrold and Gregg Rothermel. Performing data flow testing on classes. SIGSOFT Softw. Eng. Notes, 19(5):154–163, December 1994.

[201] A.L. Souter and L.L. Pollock. The construction of contextual def-use associations for object-oriented systems. IEEE Transactions on Software Engineering, 29(11):1005–1018, nov 2003.

[202] R.T. Alexander and A.J. Offutt. Criteria for testing polymorphic relationships. In Proceedings 11th International Symposium on Software Reliability Engineering. ISSRE 2000, pages 15–23. IEEE Comput. Soc, 2000.

[203] Roger T Alexander, Jeff Offutt, and Andreas Stefik. Testing coupling relationships in object-oriented programs. Software Testing, Verification and Reliability, 20(4):291–327, dec 2010.

[204] Mattia Vivanti, Andre Mis, Alessandra Gorla, and Gordon Fraser. Search-based data-flow test generation. In 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), pages 370–379. IEEE, 2013.

[205] Giovanni Denaro, Alessandra Gorla, and Mauro Pezzè. Contextual Integration Testing of Classes. In Fundamental Approaches to Software Engineering (FASE '08), volume 4961 of LNCS, pages 246–260. Springer, 2008.

[206] Ambler Scott. Building object applications that work, your step-by-step handbook for developing robust systems using object technology, 1997.

[207] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-Directed Random Test Generation. In 29th International Conference on Software Engineering (ICSE'07), pages 75–84. IEEE, may 2007.

[208] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. Automatic generation of oracles for exceptional behaviors. pages 213–224, 2016.

[209] Martin P Robillard and Robert Deline. A field study of api learning obstacles. Empirical Software Engineering, 16(6):703–732, 2011.

[210] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 242–253. ACM, 2018.

[211] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. Analyzing apis documentation and code to detect directive defects. In Proceedings of the 39th International Conference on Software Engineering, pages 27–37. IEEE Press, 2017.

[212] Ying Wang, Ming Wen, Rongxin Wu, Zhenwei Liu, Shin Hwei Tan, Zhiliang Zhu, Hai Yu, and Shing-Chi Cheung. Could i have a stack trace to examine the dependency conflict issue? In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pages 572–583. IEEE, 2019.

[213] Christel Baier and Joost-Pieter Katoen. Principles of model checking. MIT Press, 2008.

[214] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating Branch Coverage as a Many-Objective Optimization Problem. In ICST'15, pages 1–10. IEEE, apr 2015.

[215] Sadeeq Jan, Annibale Panichella, Andrea Arcuri, and Lionel Briand. Search-based multi-vulnerability testing of xml injections in web applications. Empirical Software Engineering, pages 1–34, 2019.

[216] Urko Rueda, René Just, Juan P Galeotti, and Tanja EJ Vos. Unit testing tool competition—round four. In 2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST), pages 19–28. IEEE, 2016.

[217] Annibale Panichella and Urko Rueda Molina. Java unit testing tool competition-fifth round. In 2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST), pages 32–38. IEEE, 2017.

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

[219] Fitsum Kifetew, Xavier Devroey, and Urko Rueda. Java unit testing tool competition: seventh round. In Proceedings of the 12th International Workshop on Search-Based Software Testing, pages 15–20. IEEE Press, 2019.

[220] Xavier Devroey, Sebastiano Panichella, and Alessio Gambi. Java Unit Testing Tool Competition - Eighth Round. In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, pages 545–548. ACM, jun 2020.

[221] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 654–665. ACM, 2014.

[222] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In Proceedings of the 27th international conference on Software engineering, pages 402–411. ACM, 2005.

[223] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. PIT: a practical mutation testing tool for Java (demo). In Proceedings of the 25th International Symposium on Software Testing and Analysis, pages 449–452. ACM, jul 2016.

[224] Qianqian Zhu, Annibale Panichella, and Andy Zaidman. A systematic literature review of how mutation testing supports quality assurance processes. Softw. Test. Verification Reliab., 28(6), 2018.

[225] Salvador García, Daniel Molina, Manuel Lozano, and Francisco Herrera. A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: A case study on the CEC'2005 special session on real parameter optimization. Journal of Heuristics, 15(6):617–644, December 2009.

[226] N. Japkowicz and M. Shah. Evaluating Learning Algorithms: A Classification Perspective. Cambridge University Press, 2011.

[227] Annibale Panichella. A systematic comparison of search-based approaches for lda hyperparameter tuning. Information and Software Technology, 130:106411, 2021.

[228] Marcio Eduardo Delamaro, JC Maidonado, and Aditya P. Mathur. Interface mutation: An approach for integration testing. IEEE transactions on software engineering, 27(3):228–247, 2001.

[229] Sina Shamshiri, José Miguel Rojas, Luca Gazzola, Gordon Fraser, Phil McMinn, Leonardo Mariani, and Andrea Arcuri. Random or evolutionary search for object-oriented test suite generation? Software Testing, Verification and Reliability, 28(4):e1660, jun 2018.

[230] Kobi Inkumsah and Tao Xie. Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution. In ASE'08, pages 297–306. IEEE, sep 2008.

[231] Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan de Halleux. FloPSy - Search-Based Floating Point Constraint Solving for Symbolic Execution. In Alexandre Petrenko, A Simão, and J. C. Maldonado, editors, ICTSS'10, volume 6435 of LNCS, pages 142–157. Springer, 2010.

[232] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In ISSRE'13, pages 360–369. IEEE, nov 2013.

[233] S.-D. Gouraud, A Denise, M.-C. Gaudel, and B Marre. A new way of automating statistical testing methods. In ASE '01, pages 5–12. IEEE, nov 2001.

[234] Björn Evers, Pouria Derakhshanfar, Xavier Devroey, and Andy Zaidman. Unit test generation for common and uncommon behaviors: replication package, June 2020.

[235] Gordon Fraser and Andrea Arcuri. Achieving scalable mutation-based generation of whole test suites. EMSE, 20(3):783–812, jun 2015.

[236] Qianqian Wang, Yuriy Brun, and Alessandro Orso. Behavioral execution comparison: Are tests representative of field behavior? In ICST '17. IEEE, mar 2017.

[237] Qianqian Wang and Alessandro Orso. Mimicking user behavior to improve in-house test suites. In ICSE '19. IEEE, may 2019.

[238] Giovanni Grano, Adelina Ciurumelea, Sebastiano Panichella, Fabio Palomba, and Harald C. Gall. Exploring the integration of user feedback in automated testing of Android applications. In SANER '18, pages 72–83. IEEE, mar 2018.

[239] Paolo Tonella and Filippo Ricca. Statistical testing of Web applications. Journal of Software Maintenance and Evolution: Research and Practice, 16(1-2):103–127, jan 2004.

[240] Chaitanya Kallepalli and Jeff Tian. Measuring and modeling usage and reliability for statistical Web testing. TSE, 27(11):1023–1036, nov 2001.

[241] Sara Silva, Stephen Dignum, and Leonardo Vanneschi. Operator equalisation for bloat free genetic programming and a survey of bloat control methods. Genetic Programming and Evolvable Machines, 13(2):197–238, jun 2012.

[242] Jos Winter, Maurício Aniche, Jürgen Cito, and Arie van Deursen. Monitoring-aware IDEs. In ESEC/FSE'19, pages 420–431. ACM, 2019.

[243] Urko Rueda Molina, Fitsum Kifetew, and Annibale Panichella. Java unit testing tool competition - Sixth Round Urko. In SBST '18, pages 22–29. ACM, 2018.

[244] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. PIT: a practical mutation testing tool for Java. In ISSTA 2016, pages 449–452. ACM, 2016.

[245] 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, jun 2000.

[246] Jeanderson Candido, Maurício Aniche, and Arie van Deursen. Contemporary Software Monitoring: A Systematic Literature Review. 2019.

[247] T. Menzies and T. Zimmermann. Software analytics: So what? IEEE Software, 30(4):31–37, 2013.

[248] Boyuan Chen, Jian Song, Peng Xu, Xing Hu, and Zhen Ming (Jack) Jiang. An automated approach to estimating code coverage measures via execution logs. In ASE'18, number 3, pages 305–316. ACM Press, 2018.

[249] Björn Evers. Unit test generation for common and uncommon behaviors. master thesis, Delft University of Technology, 2020.

Glossary

AIOOBE ArrayIndexOutOfBoundException, a type of exception in Java.

API Application Program Interface, a computing interface which defines interactions between multiple software intermediaries.

BBC Basic Block Coverage, a proposed secondary-objective complementing the classical line coverage fitness functions in search-based software testing. More details are available in chapter 5.

CBC Coupled Branch Coverage, a testing criterion for class integration testing.

CCE ClassCastException, a type of exception in Java.

CCFG Class-level Control Flow Graph, a representation, using graph notation, of all paths that might be traversed through a class during its execution.

CCN Cyclomatic Complexity Number, a metric indicating the complexity of a program.

CFG Control Flow Graph, a representation, using graph notation, of all paths that might be traversed through a method during its execution.

CLING CLass INtegration test Generation, a proposed approach for generating test cases for class integration. More details are available in chapter 6.

CRT Crash Reproducing Test, tests generated by automated crash reproducing test generation approaches that can throw the same crash as the reported one.

CSV Comma-Separated Values, a delimited text file that uses a comma to separate values.

DE-MO DEcomposition-based Multi-Objectivization in crash reproduction, a proposed approach to enhance the diversity of test generation in crash reproduction. More details can be found in chapter 4.

HTTP Hypertext Transfer Protocol, an application layer protocol for distributed, collaborative, hypermedia information systems.

IAE IllegalArgumentException, a type of exception in Java.

IDE Integrated development environment, a software application that provides comprehensive facilities to computer programmers for software development.

ISE IllegalStateException, a type of exception in Java.

JAR Java ARchive, a file type for packaging Java class files.

JSON JavaScript Object Notation, a lightweight data-interchange format.

KNCSS Thousands of Non-Commenting Sources Statements.

MO-HO Multi-Objectivization using Helper-Objectives in crash reproduction, a proposed approach to enhance the diversity of test generation in crash reproduction. More details can be found in chapter 4.

MOEA Multi-objective Evolutionary Algorithm, techniques that use evolutionary-based techniques to solve a multi-objective optimization problem involves several conflicting objectives. These algorithm have a set of Pareto optimal solutions.

NCSS Non-Commenting Source Statements , a metric that counts the number of all statements excluding comments.

NPE NullPointerException, a type of exception in Java.

SBST Search-Based Software Testing, application of metaheuristic search techniques to software testing problems.

SDK Software Development Kit, a collection of software development tools in one installable package.

SIOOBE StringIndexOutOfBoundException, a type of exception in Java.

SUT System Under Test, a system that is being tested for correct operation.

Curriculum Vitæ

Pouria Derakhshanfar

26/08/1992	Date of birth in Tehran, Iran

Education

8/2017-4/2021	Ph.D. Student, Software Engineering Research Group, Delft University of Technology, The Netherlands, *Carving Information Sources to Drive Search-Based Crash Reproduction and Test Case Generation* Supervisor: Dr. A. Panichella, Dr. X. Devroey Promotors: Prof. Dr. A. van Deursen, Prof. Dr. A.E. Zaidman
9/2014-9/2016	M.Sc. Computer Software Engineering, Sharif University of Technology, Iran
9/2010-8/2014	B.Sc. Computer Software Engineering, Kharazmi University, Iran

Academic Service

Reviewer	Software Quality Journal, 2020 Transactions on Software Engineering and Methodology (TOSEM), 2020 Empirical Software Engineering (EMSE), 2021
Program Committee Member	Tool Demo track of the 37th International Conference on Software Maintenance and Evolution (ICSME 2021)

Co-Supervisor	Björn Evers's Master Thesis "Unit test generation for common and uncommon behaviors", Delft University of Technology, 2019-2020
	Shang Xiang's Master Thesis "Fit2Crash: Specialising Fitness Functions for Crash Reproduction", Delft University of Technology, 2019-2020
	Sven Popping's Master Thesis "Automated crash fault localization", Delft University of Technology, 2019-2020
	Boris Cherry's Master Thesis "JCrashPack2.0: Search-based crash reproduction hardness analysis", University of Namur, 2020
	B. Dikker, C. Paulsen, L. Leibbrandt, N. Nijkamp, S. Op den Orth, S. Walboomers Bachelor's Context Project "Enhanced guidance for C++ fuzzing", Delft University of Technology, 2019
	W. Tutuarima, P. van Egmond, M. Halvemaan, N. Alwani, G. Vegelien Bachelor's Context Project "IntelliJ plugin for enhanced stack trace visualisation", Delft University of Technology, 2020

Talks

Industry Talks	*Search-based Crash Reproduction* at Tellu, Oslo, Norway, 2018
	Botsing: A Search-based Crash Reproduction Tool at OW2con'19, Paris, France, 2018
	Search-based Crash Reproduction at AI for Fintech Research, Amsterdam, The Netherlands, 2020

Academic Talks	*Single-objective versus Multi-Objectivized Optimization for Evolutionary Crash Reproduction* at SSBSE18, Montpellier, France, 2018
	Botsing Tool Tutorial at SSBSE19, Tallinn, Estonia, 2019.
	Crash Reproduction using helper objectives at GECCO2020, online, 2020
	A benchmark-based evaluation of search-based crash reproduction at ICSE2020, online, 2020
	Good Things Come In Threes: Improving Search-based Crash Reproduction With Helper Objectives at ASE2020, online, 2020
	It is not Only About Control Dependent Nodes: Basic Block Coverage for Search-Based Crash Reproduction at SSBSE2020, online, 2020

Implemented Tools

2018-current	**Botsing**[4]: An open-source search-based crash reproduction framework.
2018-current	**ExRunner**[5]: A bash-based infrastructure for running multiple instances of crash reproduction tools on 200 hard-to-reproduce crashes. ExRunner can be used to perform empirical evaluations of different crash reproduction techniques.
2019-current	**Cling**[6]: An open-source search-based test generation tool for class integration testing.

Projects

2017-2019	*Software Testing AMPlification (STAMP)*[7]: A collaborative effort between 10 organizations, from 6 European countries to advance the state of the art in DevOps and automatic software testing
2021-current	*COSMOS*: A European project between 12 organizations from 8 countries to study DevOps for Complex Cyber-physical Systems.

[4]`https://github.com/STAMP-project/botsing`
[5]`https://github.com/STAMP-project/ExRunner-bash`
[6]`https://github.com/STAMP-project/botsing/tree/master/cling`
[7]`https://www.stamp-project.eu/view/main/`

List of Publications

1. Mozhan Soltani, **Pouria Derakhshanfar**, Xavier Devroey, and Arie van Deursen: A benchmark-based evaluation of search-based crash reproduction. Empirical Software Engineering (EMSE), 2020. doi: 10.1007/978-3-319-99241-9_1. This paper was also presented at the 42nd International Conference on Software Engineering (ICSE) as Journal First Paper in 2020.

2. **Pouria Derakhshanfar**, Xavier Devroey, Gilles Perrouin, Andy Zaidman, and Arie van Deursen: Search-based crash reproduction using behavioural model seeding. Software Testing, Verification and Reliability (STVR), 2020. doi: 10.1002/stvr.1733

3. **Pouria Derakhshanfar**, Xavier Devroey, Andy Zaidman, Arie van Deursen, and Annibale Panichella: Good Things Come In Threes: Improving Search-based Crash Reproduction With Helper Objectives. 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2020. doi: 10.1145/3324884.3416643

4. **Pouria Derakhshanfar**, Xavier Devroey, and Andy Zaidman: It is not Only About Control Dependent Nodes: Basic Block Coverage for Search-Based Crash Reproduction. 12th International Symposium of Search-Based Software Engineering (SSBSE), 2020. doi: 10.1007/978-3-030-59762-7_4

5. Björn Evers, **Pouria Derakhshanfar**, Xavier Devroey, and Andy Zaidman: Commonality-Driven Unit Test Generation. 12th International Symposium of Search-Based Software Engineering (SSBSE), 2020. doi: 10.1007/978-3-030-59762-7_9

6. Mozhan Soltani, **Pouria Derakhshanfar**, Annibale Panichella, Xavier Devroey, Andy Zaidman, and Arie van Deursen: Single-objective Versus Multi-objectivized Optimization for Evolutionary Crash Reproduction. 10th International Symposium of Search-Based Software Engineering (SSBSE), 2018. doi: 10.1007/978-3-319-99241-9_18

7. **Pouria Derakhshanfar**, Xavier Devroey, Andy Zaidman, Arie van Deursen, and Annibale Panichella: Crash reproduction using helper objectives. Genetic and Evolutionary Computation Conference Companion (GECCO), Poster Paper, 2020. doi: 10.1145/3377929.3390077

8. **Pouria Derakhshanfar**: Well-informed Test Case Generation and Crash Reproduction. 13th IEEE International Conference on Software Testing, Validation and Verification (ICST), Doctoral Symposium, 2020. doi: 10.1109/ICST46399.2020.00054

9. **Pouria Derakhshanfar**, Xavier Devroey, Annibale Panichella, Andy Zaidman, and Arie van Deursen: Botsing, a Search-based Crash Reproduction Framework for Java. 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), Tool Demonstrations, 2020. doi: 10.1145/3324884.3415299

10. Mitchell Olsthoorn, **Pouria Derakhshanfar**, and Xavier Devroey: An Application of Model Seeding to Search-Based Unit Test Generation for Gson. 12th International Symposium of Search-Based Software Engineering (SSBSE), Challenge Paper, 2020. doi: 10.1007/978-3-030-59762-7_17

11. Boris Cherry, Xavier Devroey, **Pouria Derakhshanfar**, and Benoît Vanderose: Crash reproduction difficulty, an initial assessment. 19TH Belgium-Netherlands Software Evolution Workshop (BENEVOL), 2020.

🗎 Included in this thesis.
🏆 Won a best paper award.

Titles in the IPA Dissertation Series since 2018

A. Amighi. Specification and Verification of Synchronisation Classes in Java: A Practical Approach. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. Verification of Program Parallelization. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. Coequations and Eilenberg-type Correspondences. Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broştean. Active Model Learning for the Analysis of Network Protocols. Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. From Concurrent State Machines to Reliable Multi-threaded Java Code. Faculty of Mathematics and Computer Science, TU/e. 2018-05

H. Basold. Mixed Inductive-Coinductive Reasoning Types, Programs and Logic. Faculty of Science, Mathematics and Computer Science, RU. 2018-06

A. Lele. Response Modeling: Model Refinements for Timing Analysis of Runtime Scheduling in Real-time Streaming Systems. Faculty of Mathematics and Computer Science, TU/e. 2018-07

N. Bezirgiannis. Abstract Behavioral Specification: unifying modeling and programming. Faculty of Mathematics and Natural Sciences, UL. 2018-08

M.P. Konzack. Trajectory Analysis: Bridging Algorithms and Visualization. Faculty of Mathematics and Computer Science, TU/e. 2018-09

E.J.J. Ruijters. Zen and the art of railway maintenance: Analysis and optimization of maintenance via fault trees and statistical model checking. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-10

F. Yang. A Theory of Executability: with a Focus on the Expressivity of Process Calculi. Faculty of Mathematics and Computer Science, TU/e. 2018-11

L. Swartjes. Model-based design of baggage handling systems. Faculty of Mechanical Engineering, TU/e. 2018-12

T.A.E. Ophelders. Continuous Similarity Measures for Curves and Surfaces. Faculty of Mathematics and Computer Science, TU/e. 2018-13

M. Talebi. Scalable Performance Analysis of Wireless Sensor Network. Faculty of Mathematics and Computer Science, TU/e. 2018-14

R. Kumar. Truth or Dare: Quantitative security analysis using attack trees. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-15

M.M. Beller. An Empirical Evaluation of Feedback-Driven Software Development. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2018-16

M. Mehr. Faster Algorithms for Geometric Clustering and Competitive Facility-Location Problems. Faculty of Mathematics and Computer Science, TU/e. 2018-17

M. Alizadeh. Auditing of User Behavior: Identification, Analysis and Understanding of Deviations. Faculty of Mathematics and Computer Science, TU/e. 2018-18