The University Of Sheffield.

The University of Sheffield
Department of Computer Science

Automated Unit Testing of Evolving Software

Sina Shamshiri

Supervisors: Gordon Fraser & Phil McMinn

This dissertation is submitted for the degree of
Doctor of Philosophy

Faculty of Engineering October 2016

Abstract

As software programs evolve, developers need to ensure that new changes do not affect the originally intended functionality of the program. To increase their confidence, developers commonly write unit tests along with the program, and execute them after a change is made. However, manually writing these unit-tests is difficult and time-consuming, and as their number increases, so does the cost of executing and maintaining them.

Automated test generation techniques have been proposed in the literature to assist developers in the endeavour of writing these tests. However, it remains an open question how well these tools can help with fault finding in practice, and maintaining these automatically generated tests may require extra effort compared to human written ones.

This thesis evaluates the effectiveness of a number of existing automatic unit test generation techniques at detecting real faults, and explores how these techniques can be improved. In particular, we present a novel multi-objective search-based approach for generating tests that reveal changes across two versions of a program. We then investigate whether these tests can be used such that no maintenance effort is necessary.

Our results show that overall, state-of-the-art test generation tools can indeed be effective at detecting real faults: collectively, the tools revealed more than half of the bugs we studied. We also show that our proposed alternative technique that is better suited to the problem of revealing changes, can detect more faults, and does so more frequently. However, we also find that for a majority of object-oriented programs, even a random search can achieve good results. Finally, we show that such change-revealing tests can be generated on demand in practice, without requiring them to be maintained over time.

Acknowledgements

First and foremost, my deep gratitude goes to my supervisors Gordon Fraser and Phil McMinn for their tireless support, mentorship, and patience throughout the course of this PhD. In particular, the achievements of this work may have not been possible without their wisdom, expertise and thoughtful discussions. I also want to thank Guy Brown for his advice and encouragement during both my undergraduate and postgraduate studies.

Special thanks are due to my collaborators José Campos, José Miguel Rojas, René Just, Andrea Arcuri, and Alessandro Orso. I would like to also thank all members of my research group and laboratory (Verification and Testing) for their moral and personal support. In addition, I am thankful to Chris Wright, Ciprian Dragomir, and Mathew Hall for their help and the enjoyable conversations, during the early part of my PhD.

Finally, I am grateful and eternally indebted to my parents for their love, encouragement, and endless support during all the years.

Declaration and Publications

This thesis contains original work undertaken at the University of Sheffield between October 2012 and September 2016, parts of which have been published or are currently under review.

- Sina Shamshiri, Gordon Fraser, Phil McMinn, and Alessandro Orso (2013). Search-Based Propagation of Regression Faults in Automated Regression Testing. In *Proceedings of the ICST International Workshop on Regression Testing*, pages 396–399. IEEE

- Sina Shamshiri, José Miguel Rojas, Gordon Fraser, and Phil McMinn (2015). Random or Genetic Algorithm Search for Object-Oriented Test Suite Generation?. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 1367–1374 . ACM ***(Winner of the Best Paper award for the SBSE-SS track)***

- Sina Shamshiri (2015). Automated unit test generation for evolving software. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), Doctoral Symposium*, pages 1038–1041. ACM

- Sina Shamshiri, Rene Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri (2015). Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 201–211. IEEE ***(Winner of the ACM Distinguished Paper award)***

- Sina Shamshiri, José Campos, Gordon Fraser, and Phil McMinn (2017). Disposable Testing: Avoiding Maintenance of Generated Unit Tests by Throwing Them Away. In *Proceedings of the International Conference on Software Engineering (ICSE), Poster Track*, IEEE

Sina Shamshiri
October 2016

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Overview

"Software is eating the world.", claims Marc Andreessen [7]. After the computer revolution, over the past decades more and more industries and businesses rely on software programs to provide service to their customers. With the ubiquity of personal computers and smartphones – which is predicted to reach over 6 Billion people by 2020 – people around the world rely on modern programs every day. Even if people try to avoid having any digital devices, their lives will nevertheless be affected by software if they travel, bank, visit a hospital, and so forth. Therefore, ensuring the reliability of software programs is essential to our everyday lives.

Rarely is it the case that a program is written once, and then forgotten. Software programs evolve over time as developers introduce many changes throughout the life-cycle of the program. These changes often range from small refactorings to the addition of large new features. However, some of these changes may affect the originally intended functionality of the software, by unintentionally introducing bugs – also known as *regression faults* or *regression bugs*. To avoid regressions in the functionality, engineers write tests along with the software, and after making changes they execute the tests to assess whether the intended functionality of the software is intact. This practice is better known as *regression testing* and is commonly used in industry.

1.2 Regression Testing

While regression testing can help with early detection of regression faults, developers face several challenges when applying the technique. As the number of tests grows, execution of all tests after every single change can become expensive and impractical. This problem has been well studied in the literature [198] and many techniques such as test selection, prioritization and minimization have been proposed to address the problem.

The challenges however are not limited to the growing cost of regression testing. Even if all tests are always executed, three main problems remain: 1) an existing set of tests is required, 2) the tests are often written without foreseeing future changes, and 3) the effectiveness of the tests in finding regression faults depends on the quality of the written tests. These challenges give way to techniques (preferably automated), that can help developers with their regression testing efforts by improving the existing set of tests and/or generating new ones.

In order to reveal a fault, a test has to first execute the code that contains a fault; secondly the execution should result in a change in the internal state of the program (infection), which then leads to an observable faulty behaviour (propagation) [174]. While several techniques exist for augmenting existing test suites to improve them (e.g., [148, 196]) and generating regression tests (e.g., [163, 164, 123, 23]), they suffer drawbacks such as focusing on reaching the fault, yet the number of paths to propagate the infected state to the output can explode, which may impose a limit on the scalability of the approach [28]. Other techniques also exist which either rely on existing tests [101] or do not generate test inputs with sequences of method calls [27].

In this thesis, we investigate techniques for automatically testing programs as they evolve, without requiring access to an existing set of tests. We explore different techniques – and propose our own – for automatically generating unit tests such that they can reveal changes across two versions of a program (i.e., after a developer makes a change to a program, two versions of the program can be considered: the one before the change, and the one after the change was applied). In the following section, we motivate how such techniques can work in practice, and how they can benefit developers.

1.2.1 Motivation for Automated Regression Testing

Consider a self-driving car which automatically goes to the factory once it requires maintenance. The engineers of this car have designed to it to visit the factory once the car has driven at least 5000 miles since the last checkup, and also that the maintenance-free period has ended – that is, the current year has passed the year the car will require maintenance (e.g., the engineers of the car may think that a car built in 2020 may not need maintenance until 2024). Therefore, the engineers have written the code below to check for whether or not the car should go to the factory for maintenance.

```
1 public class AutomatedMaintenance {
2   private CentralDatabase db = new CentralDatabase();
3   public boolean shouldGotoFactory(int mileage, int year){
4     if (mileage >= 5000 && year >= db.get("year_maintenance_starts"))
5         return true;
6
7     return false;
8   }
9 }
```

After a while, a new developer joins the team and decides to refactor the long conditional statement on line 4 into separate conditions to increase the clarity of the code.

```
1 public class AutomatedMaintenance {
2    private CentralDatabase db = new CentralDatabase();
3    public boolean shouldGotoFactory(int mileage, int year){
4        if(year  <=  db.get("year_maintenance_starts"))
5            return false;
6
7        if (mileage >= 5000)
8            return true;
9
10       return false;
11   }
12 }
```

However, as highlighted above, the developer makes a mistake when inverting the condition on line 4, and uses a <= operator instead of using <. And so, this mistake results in a regression in the functionality such that the car will potentially visit the factory one year late. To prevent such regressions, developers of the program should have written tests to validate the behaviour of the `shouldGotoFactory()` method. However, despite the simplicity of the toy example above, even if the developers have written tests covering all statements and branches in the program – which are proxy measures commonly used by developers in the industry to assess the quality of their tests – the tests may not be adequate to find the fault. That is, in this case, developers should have tested for boundary values as well.

Although testing for boundary values along with many other testing techniques are well-studied and are even taught in most software testing textbooks, putting additional constraints on testing requirements will only increase the time and cost of creating, maintaining, and executing test sets. For instance, in our example, testing for boundary values will result in the following tests (`assertTrue` and `assertFalse` are assertions that check whether the output of the method is respectively `true` or `false`, and prompt a failure if the check does not hold):

```
1  AutomatedMaintenance am = new AutomatedMaintenance();
2  int maintenanceStarts = db.get("year_maintenance_starts");
3  assertFalse(am.shouldGotoFactory(4999, maintenanceStarts));
4  assertFalse(am.shouldGotoFactory(4999, maintenanceStarts + 1));
5  assertFalse(am.shouldGotoFactory(4999, maintenanceStarts - 1));
6  assertTrue(am.shouldGotoFactory(5000, maintenanceStarts));
7  assertTrue(am.shouldGotoFactory(5000, maintenanceStarts + 1));
8  assertFalse(am.shouldGotoFactory(5000, maintenanceStarts - 1));
9  assertTrue(am.shouldGotoFactory(5001, maintenanceStarts));
10 assertTrue(am.shouldGotoFactory(5001, maintenanceStarts + 1));
11 assertFalse(am.shouldGotoFactory(5001, maintenanceStarts - 1));
```

Of the large number of checks above, when executed on the faulty program, only two of the assertions will fail (lines 6 and 9). This means that developers have to create comprehensive and sensitive test suites (i.e., set of tests), continuously execute them after making changes, and maintain these tests over time. For

some, this leads to the question of whether the effort can be automated – fully or even partially – such that developers can spend more time on the development of the software.

For example, a partially automated technique could take the first 3 lines of the test above as input, and augment it by deriving lines 4–11. A fully automated technique on the other hand could take the class under test (`AutomatedMaintenance`) as input, and generate the whole test above. Alternatively, in a regression testing scenario where no tests are available, a program could take the two versions of the class under test as input, and return with the following small test case, highlighting the observed behavioural difference between the two programs:

```
1 AutomatedMaintenance am = new AutomatedMaintenance();
2 int maintenanceStarts = db.get("year_maintenance_starts");
3 // [change detected] Original version: true | New version: false
4 assertTrue(am.shouldGotoFactory(5000, maintenanceStarts));
```

Using the test case above, the developer can then investigate whether or not the new behaviour is as expected, and if not, proceed to fix the new program. In fact, creating – either automatically or manually – and maintaining regression unit tests may not even be necessary, if such tests as above can be generated on demand reliably and automatically. In this thesis we review and investigate techniques that can help developers automatically test their programs as they evolve.

1.2.2 Summary

Given that as systems grow more complex, writing good regression tests becomes increasingly more difficult and time consuming, it would be inherently desirable if such tests could be generated automatically. Even when developers have unlimited resources and can afford to rigorously test their programs – i.e., the difficulty and time consumption are not considered as problems – the number of tests will grow over time, such that the cost of maintaining and executing these tests may eventually exceed the cost of damages caused by not testing at all. As we will discuss in depth in Chapter 2, researchers have been actively working on tackling many problems in regression testing. Automated techniques have been developed to augment existing set of tests, or even generate new

ones. Nevertheless, these techniques come with limitations such as requiring existing tests, not being suitable for finding regression faults, or having further implications on maintenance costs.

Therefore, in this work, we look at the problem of automatically generating regression test suites, without requiring an existing set of tests. We evaluate several existing state-of-the-art techniques and algorithms for test generation and detection of real faults, and propose our own techniques. We also look at practical implications and the effectiveness of using these techniques over time when used in addition to/instead of existing manual testing efforts.

1.3 Structure and Contributions of this Thesis

This section outlines the structure of this thesis, alongside presenting the key novel contributions of this work:

Chapter 2: "Literature Review" presents a survey of the literature relating to the topics explored in this thesis. The chapter begins with describing a set of terminologies and definitions on software testing relevant to this work. Manual and automated testing techniques are then described. In particular, we broadly look at automated test generation approaches, as well as automated regression-testing techniques in the literature. A number of challenges faced by existing work are also presented in this chapter. One important gap we identified was the lack of a clear comparison between the automated techniques for detecting real faults, which enables the evaluation of the rest of the work presented in this thesis. The next chapter aims to address this.

Chapter 3: "Evaluating Automated Unit Test Generation Tools Using Real Faults" investigates the effectiveness of three state-of-the-art automated test generation tools for detecting real-faults. While many different automated test generation tools and techniques exist, these techniques have often been evaluated on different sets of artefacts using different methodologies. We present a methodology and framework for generating, filtering (e.g., removing false positives), and evaluating the effectiveness of generated tests at detecting changes.

To achieve the objective of this chapter, a large-scale empirical evaluation was conducted using three state of the art test generation tools – two open source and one commercial – on 357 real faults taken from open-source repositories. Using the result of our evaluation, we then present a detailed analysis of the effectiveness of the generated tests at revealing the faults. To guide the development and research in automated test generation tools, we present the result of our in-depth qualitative investigation into the shortcomings and individual advancements of existing state-of-the-art techniques.

Contribution 1: *Developing a methodology and framework for evaluating the effectiveness of tests generated using automated unit test generation techniques at detecting faults, along with insights into the current state-of-the-art and how it can be further improved*

Equipped with insights into shortcomings of existing approaches, and also a framework to evaluate future techniques with, next we propose our own alternative technique.

Chapter 4: "Differential Testing Using a Search-based Approach" proposes a search-based unit-test generation tool with the aim of finding behavioural differences – which can be either intended changes or regression faults – given two versions of a program. We present the first search-based technique for automatically generating regression unit test suites. We achieve this using a novel multi-objective fitness function for generating regression tests which simultaneously maximises code coverage, state differences between the two programs, and divergence in the control-flow of the program.

Using the framework described in Chapter 3, we then conduct a large-scale empirical evaluation of our search-based regression test generation technique at detecting real faults, as well as evaluating individual components of the fitness function. We also compare our results with the ones obtained by the state-of-the-art tools in Chapter 3.

Contribution 2 *Developing a novel search based approach for generating differential regression test suites, accompanied with with a large-scale empirical evaluation using real faults*

During the evaluation of our search-based approach, we made an interesting observation that for a majority of classes, using a complex search algorithm may not be necessary, and simple random search may be good enough. To clarify whether this is limited to differential testing, Chapter 5 provides an in-depth investigation to find the underlying reasons.

Chapter 5: "Comparing Random and Genetic Algorithm Search for Generating High-coverage Test Suites" conducts an investigation based on an observation we made in Chapter 4. In particular, for a large number of subjects, random-testing and search-based testing performed equally well, while at the same time, for a small number of subjects random testing was significantly more effective than our proposed GA. In this section we explore whether our observation was limited to the domain of differential/regression testing, or whether it is a broader behaviour which applies to search-based testing in general when testing object-oriented programs. In particular, we conduct a large-scale empirical comparison of GA and random search for coverage-driven test suite generation using 1,000 real-world Java classes

The types of branching conditions found in object-oriented programs – as opposed to procedural programs – can have a direct impact on the effectiveness of search based techniques. That is, given that search-based approaches often guide the search using insights from covered or uncovered branches, can we extract useful information from all branches of different natures to guide the search? We present an in-depth investigation on the influence of the nature of branches in object-oriented programs, on search-based test generation techniques. We also investigate the impact of the search budget on the outcome of the techniques, when generating coverage-driven tests using GA and random search.

Contribution 3 *A large-scale empirical comparison of GA and random search for coverage-driven test suite generation, with an in-depth investigation into the challenges of the search landscape created by the types of branches prevalent in object-oriented programs*

Using the insights gained so far, in Chapter 6 we propose a practical approach for using automated test generation tools to prevent regression faults.

Chapter 6: "Disposable Testing: Detecting Changes during Software Evolution without Keeping and Maintaining Generated Test Suite" presents *disposable testing* as an alternative to the traditional generate-and-maintain approaches where tests are generated and then kept and maintained along with program code to be executed as the program evolves. Instead, this approach suggests generating change-revealing tests – such as when applying differential testing – and throwing the tests away after inspecting them.

We evaluate this approach using differential-testing – based on the most effective technique evaluated in Chapter 4 – as a means to apply this approach in practice. As such, we present the first large-scale evaluation of coverage-driven (as an example of a traditional generate-and-maintain approach) and differential testing at finding real faults. Moreover, we answer whether or not tests generated using differential testing should be kept or disposed of by investigating their effectiveness at revealing subsequent changes, using mutants and real faults.

Although disposable-testing removes the cost of maintaining the tests, the inspection cost remains – that is, as with any other regression test suite, a developer needs to inspect whether a test failure is due to a bug (regression) or an intended change. As such, we investigate whether or not applying disposable testing has a negative implication on the overall maintenance efforts (i.e., do developers need to inspect more tests as a result?).

Contribution 4 *Creating and empirically evaluating a new approach named "disposable testing" as a practical way to use automated unit test generation techniques during software development*

We conclude this work in the final chapter.

Chapter 7: "Conclusions and Future Work" summarises our findings and the contributions of this thesis. We also present future research directions in the area based on the work presented in the prior chapters.

Chapter 2

Literature Review

2.1 Software Testing

According to Beizer [20], software maintenance often accounts for as much as two-thirds of the software production cost. With programs becoming more complex, the possibility of having software bugs increases, and therefore programs need to be tested in order to ensure that the functionality of the program is as intended. This activity has become so important that over 50% of the total development time and cost of software projects often gets allocated to testing [119].

> Software testing is the activity of executing a given program P with sample inputs selected from the input space for P, to try to reveal failures in the program. (Harrold et al. [79])

Following this definition, developers create tests that evaluate the behaviour of the program given certain *test inputs*. They also use *test oracles* – which are checks validating the output/behaviour of the program, implemented normally using *assertions* – to assess whether the program is working as intended. These set of inputs and oracles are often implemented together in a *test case*, where the program is first invoked using the test inputs, and then the resulting output is validated. If executing any of these test cases leads to an unexpected outcome, it can indicate a fault (also referred to as a *bug* or *defect*). A defect is often revealed by a test through a *failure* – a program behaviour contrary to its specification

is detected, e.g., a test oracle observes an unexpected output – or through an *error* – an unexpected behaviour occurring within the system bounds, such as reaching an unexpected system state, often not detectable by a simple assertion.

To simplify the terminology used in this thesis, we may use *failure* to refer to both cases of errors and failures. While the term *bug* is often known to have multiple definitions referring to both a fault (defect) and failure, we normally use this term interchangeably with the former. Moreover, we may use the terms *test* and *test case* interchangeably, in which case we normally intend the latter. Furthermore, this thesis mainly focuses on the concept and method of "unit-testing", where a test case often involves a sequence of program invocations with test inputs, to test the correctness of a program unit. Each "unit" normally refers to a small – or even the smallest – separate testable module of the program, although conflicting ideas exist among developers regarding the scale of each unit [145]. In procedural programming a unit may span from a single procedure to an entire module, while in object-oriented programming it can range from testing a single method to an entire class. While we do review techniques that target testing procedural programs, the focus of this thesis is more on the latter case.

After we as developers start to write such tests, we may face the question of when our testing is sufficient? How can we measure the quality of the tests we have written so far? If the input space is large (or practically infinite), how can we know that the right inputs are being used to test the program? How can we make sure that the inputs used are more likely to reveal a fault? Over the past decades, much work has been done to answer these questions. The next section looks at some of these works and the existing challenges.

2.1.1 Test Adequacy Criteria

Researchers have proposed many different criteria to measure the quality of a test suite (i.e., a set of tests), such as adherence of the program to a certain model, coverage of different characteristics of the program (e.g., program statements, data-flow dependencies, behavioural response [161], structure, execution paths, etc.), and mutation score. We discuss the two most common criteria below.

2.1.1.1 Coverage Criteria

Given that the input space to a program can often be infinitely large, developers seek a quantifiable way to determine the effectiveness of their test suites. Coverage criteria provide an actionable metric that can provide a target to the developers. The simplest and most common of these criteria are the *structural coverage* criteria which set the target to covering the whole structure of the program [5]. A structural coverage criterion defines some structural items of the source code, and each of these items needs to be executed by at least one test, and then the ratio of covered items is the *coverage value* (or coverage ratio). A simple yet common example of these criteria is *statement coverage*, requiring all statements in the program to be executed. However, simply executing all statements in the program does not necessarily cover the structure of the program. For instance, only covering one side of a conditional statement may be enough to cover all statements.

To improve on the effectiveness of statement coverage, *branch coverage* – also known as decision coverage, but not to be confused with condition coverage – aims to cover all branches in the program resulted from conditional statements. Therefore, by covering all branches in the program, the code under test will be covered. To further improve upon this, *conditional coverage* requires all sub-expressions of a conditional statement to be covered. For instance, in a conditional statement `if(a>b || b<c)`, if either `(a>b)` or `(b<c)` evaluate to `true`, the `true` branch of the program will be covered. Conditional coverage requires both sides of these sub expressions – 4 possible outcomes in total – should be covered. Other coverage criteria also exist that necessitate more rigorous testing of the program [202]. For instance, *data-flow coverage* requires the values to be carried from their points of definition, to all points they were used, which can be particularly important for testing object-oriented programs.

2.1.1.2 Propagation Problem

Achieving coverage alone however is believed to be insufficient for detecting faults [83]. Marinescu et al. [104] even show that bugs are often introduced on real open-source projects despite the existence of high-coverage test suites. According to the PIE model [174], in order to detect a fault in a program, a

test should be able to first, reach and execute the fault. Secondly, the execution should result in a change, thus infecting the state of the program. Finally, it should propagate to an observable result in the output. Therefore, simply executing the program under test does necessarily mean that the fault can be propagated to the output. Developers generally have to manually write test cases in order to test software programs, however, creating test suites (sets of test cases) that can cover the structure of the program and achieve such propagations is known to be difficult and time-consuming.

The problem of propagation in particular has been deeply studied in the literature, and still remains as a challenge in software testing. In 1991, Freedman [57] proposed domain testability, as a measure of input-output consistency. Voas and Miller [175] later suggested the domain/range ratio (DRR), which is the ratio of the possible inputs to the range of possible outputs. They proposed that by increasing the DRR, there is a higher chance of detecting faults during the testing process. These approaches focused on the testability of programs in general, both from the implementation aspect as well as the program specifications. While the approach by Voas and Miller focused more on increasing the testability of the program, Freedman centred the work around modifications that can make the behaviour of the software observable and controllable. Woodward and Al-Khanjari [184] in a study focused on the connection that exists between these two approaches which aim to help with increasing software testability, and established a mathematical link between them.

Since these studies focused on faults remaining hidden to the testing process, many studies focused on increasing testability and ensuring fault propagation to the output. Clark and Hierons [34] proposed a measure named *squeeziness*, as an information theoretic measure to prevent fault masking. They suggest an approach of ranking execution paths with the aim of increasing the dependencies. The results of their study suggested that there exists a strong correlation between their proposed measurement and the likelihood of the fault being identified. To tackle the problem of fault finding with coverage metrics, recent ideas have been proposed to guide coverage criteria using databases of real faults [111].

2.1.1.3 Mutation Testing and Analysis

Given that, as discussed in the previous section, coverage criteria may be insufficient to determine the actual effectiveness of a set of tests, an alternative is to actually simulate the possible faults that may occur. This helps developers quantify – to some extent — the effectiveness of their tests at revealing bugs. Mutation testing is a technique that makes such a quantification possible, and has been commonly used for evaluating the adequacy of the existing tests.

Since its first introduction in the 1970s, much research has been done in the area, with the aim of making it an approach that can be practically used in the industry. Principally, mutation testing works by creating mutated versions of a program, where mutations are faults that are deliberately added to the program, that are similar to the ones that developers can make. After creating these mutated versions (also known as mutants), the existing tests are executed against the mutants, and their adequacy is assessed by their ability to identify the faults in the mutated versions. The outcome of this process is the ratio of the number of detected faults over the number of mutations, which is known as the mutation score.

Mutation testing is widely used in other applications such as test data generation, regression testing, and also to assist with test suite minimisation [87]. The main underlying reason for the big appeal of mutation testing is the fact that mutants can be automatically and systematically generated, and exercised, with the aim of representing a large number of potential faults that can happen in the program.

In contrast to the previously discussed coverage criteria, mutation testing aims to provide a proxy to the set of real faults that a test suite can discover. However, this may lead to the question of whether these sets of mutants can – at least to some extent – represent real faults [8]. In a study by Just et al. [90], the authors aim to answer this. In particular, they conduct a large-scale study using 357 real faults, and measure the correlation between detection of mutants and detection of real faults. Their results show that on the one hand there exists a general correlation between finding mutants and real faults, on the other hand, they observe that for 27% of bugs (i.e., real faults), such a correlation does not exist, thus pointing to the limitations of the technique. The

authors however note that a number of these cases can be addressed by creating stronger or new mutation operators (i.e., a mutation operator applies a specific mutation to a program, for instance, negating all equality operators == to !=). Nevertheless, for 17% of the bugs, they observe that the faults cannot be coupled to mutants. These cases include faults that relate to incorrect implementation of algorithms, extra code that should have been removed, incorrect method calls due to similarity of the names of the methods, understanding of the context, and so forth.

Although the results of the study by Just et al. were supportive of using mutation testing as a proxy for a majority of faults, the challenges with using the technique are not limited to the limitations discussed earlier. The mutation score for instance may not always be actionable. This is mainly due to the fact that a) a large number of mutants need to be generated and evaluated, b) some mutants can be equivalent. An *equivalent* mutant is created when the functionality of the program remains semantically the same. For instance, consider the simple example below:

```
1 public boolean isMathsWorking(){
2       return (2 + 2 == 4);
3 }
```

The same program can be written as follows, by changing the addition operator + to a multiplication operator ∗:

```
1 public boolean isMathsWorking(){
2       return (2 * 2 == 4);
3 }
```

In this case, although a mutation has been applied to the program, the two programs are semantically equivalent. Therefore, simply achieving a certain mutation ratio does not necessitate that the uncovered mutants are actually feasible. Similarly, not achieving a high mutation ratio does not necessitate that the uncovered mutants were infeasible to kill. Although some equivalent mutants can be detected using mathematical models and symbolic techniques, the problem of detecting equivalent mutants is undecidable [5]. In addition to equivalent mutants, generating and exercising a large number of mutants can be expensive, and may therefore limit the adoption of the technique in practice.

2.1.2 Automated Test Generation

Despite software testing taking a majority of the costs of software development, in 2002 it was reported that the lack of robust infrastructure for software testing has an annual cost of $22.5–$59.5 billion to the US economy [166]. Techniques for automating the generation of software tests aim to lower the cost of the testing process, while at the same time trying to be equally or more effective than the traditional approaches. Although many approaches exist for automated testing (or to aid with software testing efforts), they generally fall under the categories of a) random techniques: randomly generated test data is used to test the program, b) static techniques: test inputs are generated without executing the program; developers can then use these test inputs to test their program, and c) dynamic techniques: the program is executed during the testing process.

The techniques are however not limited to three categories; for instance, model-based test generation techniques use a model or specification to validate the system under test, and combinatorial testing aims to test all input combinations while at the same time reducing the input space. Algebraic and exhaustive approaches – as opposed to random ones – have also been proposed, which aim to test the program systematically using a combination of automatically inferred specifications and hints provided by developers (e.g., TACCLE [30], JWalk [160]). JWalk [160] for instance incrementally explores and learns algebraic properties and program specifications lazily with the help of the developer. This enables the approach to search for maximally changed object states and to reach new states during the testing process. These approaches however require – or significantly benefit from – user interactions with the tool to either validate the states or to guide the exploration, and so, are not fully automated or the expected manual inspection effort makes them less practical.

In the following sections, we review three of the most common techniques for test generation: random techniques, techniques relying on symbolic execution, and search based techniques.

2.1.3 Random Testing

The most basic form of test generation is random testing. By generating random inputs and passing them to the program and observing the output, the program can tested. This can be useful if the specification is incomplete, or when the developers are looking for unexpected security problems (such as in fuzz testing [62]). In the context of unit testing of object oriented classes, where test cases involve sequences of calls on interacting objects, a popular approach is to generate these sequences randomly (e.g., [124, 35, 132, 121, 9, 103, 147]). An example of this is RANDOOP [124], which is commonly used in studies and industry.

RANDOOP takes a slightly different spin on pure random testing by using a simple feedback-directed approach. The tool operates by incrementally generating method sequences – where new methods calls are selected randomly – and then executing such method sequences. The execution results are then used as feedback for the test generation to a) avoid illegal (e.g., causing runtime exceptions) or redundant statements, b) detect contract violations, and c) for generating assertions that can help detect future changes (i.e., when performing regression testing). An example of a test case generated by RANDOOP for the code snippet in Section 2.1.4 is shown below. Notice that RANDOOP generates the assertions (Line 6-8) based on the execution results of the `foo()` method (Line 3-5).

```
1 public void test001() throws Throwable {
2     Bar bar0 = new Bar();
3     boolean b2 = bar0.foo((int) (short) -1);
4     boolean b4 = bar0.foo((int) (byte) 0);
5     boolean b6 = bar0.foo((int) '4');
6     org.junit.Assert.assertTrue(b2 == false);
7     org.junit.Assert.assertTrue(b4 == false);
8     org.junit.Assert.assertTrue(b6 == true);
9 }
```

To improve random testing, researchers suggest that the randomly generated test inputs should be evenly spread across the input domain. As a result, *adaptive* random testing (ART) was proposed [32] to guide the test generation. ART has been studied as a popular testing methods and has been shown as an effective

alternative to random testing. However, using a large empirical study, Arcuri and Briand [12] argue that ART may not be cost effective in comparison to random testing. Overall, while a large number of random tests can be generated cheaply, executing and manually evaluating them can become impractical, and the tests may struggle to reach parts of the code that require complex scenarios to be generated, thus limiting the practicality of these approaches [15].

2.1.4 Testing Using Symbolic Execution

Symbolic execution [93] is another popular approach, which uses symbolic input values – as opposed to concrete input values – and executes the program symbolically [6]. To better illustrate the idea, consider the following code:

```
1 public boolean foo(int x){
2   x = x + 2;
3   if( x > 7)
4     return true;
5   else
6     return false;
7 }
```

In the code above, if we consider the input value `x` as a symbolic value, after the execution of line 2, the value of `x` in the conditional statement in line 3 is `x + 2`. Therefore, the evaluated condition on line 3 is `if` $(x > 5)$. Using this information, we simply need to generate two values – a task which is often done using a constraint solver such as Z3 [38] – for `x` such that the condition evaluates to `true` and `false` (e.g., respectively for input values 5 and 6). Concolic testing (also referred to as dynamic symbolic execution), combines the static approach we observed above, with dynamic testing. This can be necessary since the program may interact with external dependencies, or that sometimes the conditional constraint cannot be solved.

Some approaches relying on symbolic execution include the use of dynamic symbolic execution on the basis of developer-written parameterised unit tests [169], or derived techniques that aim to explore relevant sequences of calls [168, 188]. In Section 2.2 we look at some of these techniques that are more closely related to the topic of this thesis.

Although techniques based on symbolic execution can be effective at generating high-coverage tests, as well as deriving inputs to solve mathematical constraints [26], when applied to the problem of unit-testing, particularly when generating a sequence of statements invoking and interacting with the program, these techniques face a number of challenges – especially when applied to object-oriented programs. For example, if the state of the program needs to be modified by a sequence of calls to the public API rather than by setting the state using the input values directly, this may be particularly challenging to achieve using symbolic execution techniques. Additionally, scalability of symbolic techniques creates another challenge, given that as the complexity of the program increases (e.g., increase in the number of branches), the number of paths to explore increases with it [28]. Therefore, these challenges make symbolic execution less suitable for generating sequences of calls when unit-testing object-oriented programs. For example, symbolic techniques excel well when a method is called with a certain input value that is expected to reach a faulty area of code (so that it can generate a certain input that would lead to the fault), as opposed to when a method call with no input values has to be called several times for a faulty behaviour to be propagated to the output (see Chapter 4).

2.1.5 Testing Using Search Based Software Engineering

Use of metaheuristic search techniques is one of the most recent dynamic techniques for automated test generation [40]. These techniques aim to apply a search-based optimisation to generate test inputs [76].

In 2001, Harman and Jones [75] claimed that a new field is emerging in software engineering research, named search-based software engineering. This area of software engineering mainly looks at the use of metaheuristic search techniques that can be applied to a large number of areas. The simplest form of a search based algorithm is random search. However, without a guidance, the findings produced by random searching are poor [110]. Such guidance is often provided by a *fitness function* – further discussed in the rest of this section – which evaluates individual solutions based on their closeness to the ultimate solution. The set of all fitness values form a landscape of possible solutions and their respective fitness score. Guided search-based techniques

employ different strategies for exploring this landscape. Three main metaheuristic search algorithms exist that use guidance measurements, and are widely used in different areas. These techniques are presented in the following sections.

2.1.5.1 Guided Algorithms

Hill Climbing

Hill climbing techniques choose a random position in the search space at the start. Based on the position of the candidate, during each iteration, the candidate will be compared to its neighbours. In case any of the neighbours are found to have better fitness values, a move will be made to that neighbour. Therefore, as a result, the candidate solution is expected to improve over time, and as the search continues, the chosen candidate will have better fitness. Once no better neighbours can be found, then the search has found an optimum solution/fitness (i.e., potentially the best solution).

A problem associated with hill climbing techniques is the fact that the search space is not always linear or unimodal. Therefore, for instance, the hill climbing may improve the chosen candidate until it hits a maximum, and it would be expected that the candidate is the fittest individual in the search space. However, the chosen candidate may be within a local maximum and may be significantly poorer compared to the global maximum in the search space. One way used by researchers to address this issue is to restart the search multiple times at different initial locations in order to create a better understanding of the search space [70].

While using hill climbing is not best suited to find global optima, they can be ideal candidates for local search. A popular and well-studied example of this is Korel's Alternating Variable Method [95], for test data generation. This technique performs the search by adjusting input values and using different values in turn, that is, an exploratory move is made and the fitness of the solution is calculated. This continues until no variable changes improve the fitness value. The technique was shown to be effective at performing local search, and also to improve the effectiveness of other search-based techniques [76]. However, when applied in isolation, it may not be effective at global search.

Simulated Annealing

With the starting position having a high impact on the outcome of hill climbing techniques, simulated annealing aims to rely less on the initial position. To achieve this, this techniques works in a similar manner compared to hill climbing, however with a difference that it can probabilistically accept poorer solutions. Simulated annealing is given a control parameter named temperature, and the higher this control parameter be, the more freedom the currently chosen solution will have to move around the search space.

Inspired by annealing in nature, the search first starts with a high-temperature. Then, as the search continues, the temperature is cooled down (i.e., decreased), thus giving the selected subject less and less chance of movement. This enables the search to initially explore a large portion of the search space and afterwards follow a more linear hill-climbing approach around the chosen local search space. However, if the cool-down is done at a rapid pace, the search may only explore a small portion of the search space, and thus get stuck in a local optimum similar to hill climbing techniques [109].

Genetic Algorithms

Genetic algorithms use a different approach compared to the previous techniques. Inspired by natural selection, this technique simulates evolution as a search strategy. This technique considers solutions in the search space as individuals or "chromosomes". Similar to natural evolution, a population of these chromosomes is first created (often randomly) and during the search, mutation and recombinations of these individuals will occur. With iterations being called "generations", and the population of each generation having chromosomes as their members, on each generation, the chromosomes evolve using the following search operators: a) mutation operator, which randomly modifies the chromosome, and b) crossover operator, which recombines certain members of the population, or creates offspring chromosomes by taking two parent chromosomes. In order to make sure that only *better* individuals in the population make it to the next generation, and to guide the evolution, a "fitness function" is defined. The ultimate goal of the fitness function is to measure how close each individual is from a perfect solution. Individuals with better fitness get a higher chance of being included in the next generation, and as a result, the search will be guided

by the fitness function. Genetic algorithms are by far the most widely applied search technique in search based software engineering [109, 70].

2.1.5.2 Application in Testing

One of the application areas of search-based software engineering is search-based software testing (SBST). This area focuses on using search-based software engineering to automate software testing. To achieve this, different optimisation processes are defined in the form of a fitness function which aims to solve specific challenges, often simultaneously. This approach can be used to generate test inputs with multiple objectives, such as achieving a high code coverage across the whole program [110], which is a powerful aspect of search-based techniques in general. It has also been shown that multi-objective optimisation techniques can be even used to better tackle single objective problems [77]. This section discusses some of the application areas of this approach.

Temporal testing: This area aims to find best-case and worst-case execution times. Using a search-based approach, the search can look for the input data that can result in the shortest and longest execution times. Finding such input values using static analysis cannot be done accurately, since execution of the code with the given input values may be necessary [110].

Functional Testing: Testing the logical behaviour of the system according to its specification is another area of search based testing. In this area, the use of search-based techniques in testing the overall functionality of the system is explored. Different studies have been performed, namely a study of an automated parking system, in which a search based approach was used to automatically park a car [24]. Given the success of their technique, the authors aim to additionally expand the use of evolutionary algorithms in other aspects of vehicles such as the emergency brake system and intelligent speed controls.

Structural Testing: The main and most active area of search-based techniques is structural testing. This area targets the execution paths in the software, and input data is generated to target specific structural testing aspects. These

aspects often include branch and data flow coverage. Common approaches targeting branch coverage aim to generate test inputs with the objective of maximising branch coverage [110].

2.1.5.3 Search-Based Unit Test Generation

Search-based techniques search for candidates that most closely represent the ultimate solution. As such, the representation of the problem can vary. This characteristic makes them particularly suited for the problem of generating unit tests. In this section we look at different approaches in the literature that applied these techniques.

Random Search for Test Generation

One strategy for finding branch-covering test cases is simply to generate sequences of statements to the class under test at random, coupled with randomly-generated inputs. If a randomly-generated test case covers new branches that have not been executed before, it is added to the CUT's test suite, else it can be discarded. One disadvantage of this approach is the size of the resulting test suite, which can be very large and therefore carry a high execution cost.

A further problem is finding inputs that need to be certain "magic" values required to execute certain branches, such as constant values, specific strings, etc. that are unlikely to be generated by chance. One way of circumventing this problem is to enhance the algorithm through *seeding*.

Seeding

The process of *seeding* involves biasing the search process towards certain input values that are likely to improve the chances of executing more coverage goals [3, 49, 112]. Fraser and Arcuri [49] propose obtaining seeds both statically and dynamically, and present an implementation of it in the tool EvoSuite. The *static* approach takes place before test generation: EvoSuite collects all literal primitive and string values that appear in the Java bytecode of the class under test. Then, while tests are being generated, literals that are encountered at runtime may also be *dynamically* added to the pool of seeds. Some of these

seeds are specially computed, according to a set of predefined rules. For instance, if the test case includes the statement "`foo.startsWith(bar)`", involving the strings `foo` and `bar`, the concatenation `bar + foo` will be added to the seed pool. During the search process, EvoSuite will then choose to use a seed from the pool instead of generating a fresh value, according to a certain probability.

To distinguish between random search with and without seeding enabled, we refer to the enhanced version of random search incorporating seeding as *Random+*, and the basic implementation without seeding as *Pure Random*.

Evolutionary Search for Test Suite Generation

While random search relies on encountering solutions by chance, guided searches aim to find solutions more directly by using a problem-specific "fitness function". A fitness function scores better potential solutions to the problem with better fitness values. A good fitness function will provide a gradient of fitness values so that the search can follow a "path" to increasingly better solutions that are increasingly fit for purpose. With a good fitness function, guided search-based approaches are capable of finding suitable solutions in extremely large or infinite search spaces (such as the space of all possible test cases for a class as considered in this work).

Genetic Algorithms (GAs) are one example of a directed search technique that uses simulated natural evolution as a search strategy. GAs evolve solutions to a problem based on their fitness, often by evolving several candidate solutions at once in a "population". The initial population of candidate solutions is generated randomly. Each iteration of the algorithm seeks to adapt these solutions to ones with an increased fitness by using search functions: "Crossover" works to splice two solutions to form a new "offspring", while "mutation" randomly changes a component of a solution. The new solutions generated are taken forward to the next iteration depending on their fitness.

Algorithm 1 is an example of using a GA for search-based testing – as used in EvoSuite [51]. As shown, the GA first creates an initial population of solutions randomly (Line 2). Then, using rank selection, it selects two parents P_1 and P_2 (Line 6) and crosses them over (Line 7-10). With a certain probability, the GA applies the mutation operator on the resulting offspring O_1 and O_2 (Line 11),

Algorithm 1 A genetic algorithm as used in search-based tool EvoSuite [51].

```
1: seeds ← initialize seeds with collected static literals from bytecode
2: current_population ← generate random population using seeds
3: repeat
4:     Z ← elite of current_population
5:     while |Z| ≠ |current_population| do
6:         P1, P2 ← rank selection from current_population
7:         if crossover probability then
8:             O1, O2 ← crossover P1, P2
9:         else
10:            O1, O2 ← P1, P2
11:        mutate O1 and O2                    ▷ The seeds pool may be used
12:        fP = min(fitness(P1), fitness(P2))
13:        fO = min(fitness(O1), fitness(O2))
14:        seeds ← update seeds with collected dynamic seeds from fitness
       evaluations
15:        if fO ≤ fP then
16:            Z ← Z ∪ {O1, O2}
17:        else
18:            Z ← Z ∪ {P1, P2}
19:    current_population ← Z
20: until solution found or maximum resources spent
```

and then compares the fitness value of parents and their offspring to determine which one will be carried over to the next generation Z (Line 15-18). The GA repeats this process until a solution is found or the search budget is exhausted. During this process, to enhance the effectiveness of the generated solutions, the GA collects seeds statically and dynamically and uses them for the generation of new or mutated solutions (Line 1 and 14).

Early work in this area used genetic algorithms for generating test data, with the aim of achieving statement and branch coverage [127] . In this case, the fitness function guided the search to reach (i.e., cover) certain structural points by maximising the number of branching nodes that were executed as intended (i.e., been on the correct path to the intended point in the structure). As such, there was no guidance as to *how* any of these branches should be covered. Other work such as the one by Baresel et al. [17] have tried to improve upon the fitness function used for evolutionary test data generation, by for instance, using branch distance – that is, how far a conditional branch is from becoming

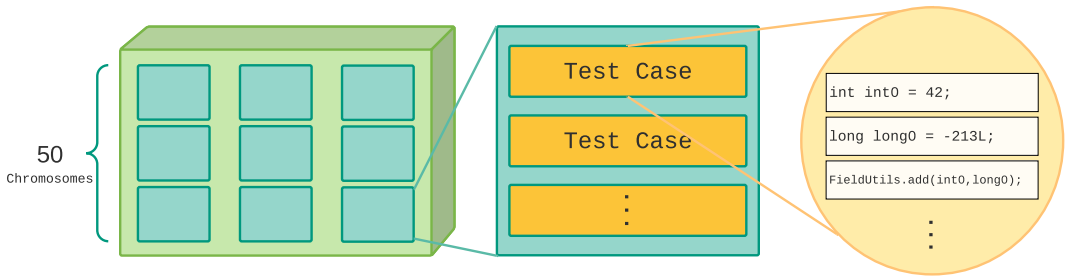


Fig. 2.1 A population of test suite chromosomes for whole test suite optimisation (i.e., as used in EvoSuite). The test statements on the right hand side are generated randomly.

`true` or `false` – to better guide the search. Much of these approaches however focused on procedural code, where the chromosomes are simplistic: they often involve a sequence of input values that are passed to the program. Tonella [170] applied a genetic algorithm for automated generation of unit tests, for testing object-oriented classes. The accompanied tool named *eToc* generated targeted tests for covering coverage goals (e.g., branches) individually, and then combined them into a single test suite. Therefore, the size of the test suite, as well as the test-generation time may grow out of hand for large subjects. Additionally, assertions were required to be added manually to the tests as part of the test process.

Fraser and Arcuri [51] propose using a GA for generating and optimizing *whole test suites*, and present an implementation of it in their tool EvoSuite. With EvoSuite's GA, a "solution" (i.e., chromosome) is a whole test suite, consisting of a series of test cases, which in turn contain a number of test statements. From a natural evolution point of view, these statements form the genome of the individuals, which are then executed on the class under tests. Therefore, the phenotype is the resulting execution traces which are then passed to the fitness function [143].

After the search budget is exhausted – unless an optimum solution is found earlier – the generated tests are embedded with assertions based on the existing behaviour (i.e., output) of the program. That is, the tool assumes that the current behaviour of the program is correct, and the developers can check whether the program is behaving correctly by validating the generated oracles. Since many different types of assertions can be created based on the different type of output

values, to increase test readability, different approaches have been proposed to reduce the number of produced assertions (e.g., [55, 56]). The following code snippet shows an example of a test-suite generated by EvoSuite for the sample program in Section 2.1.4:

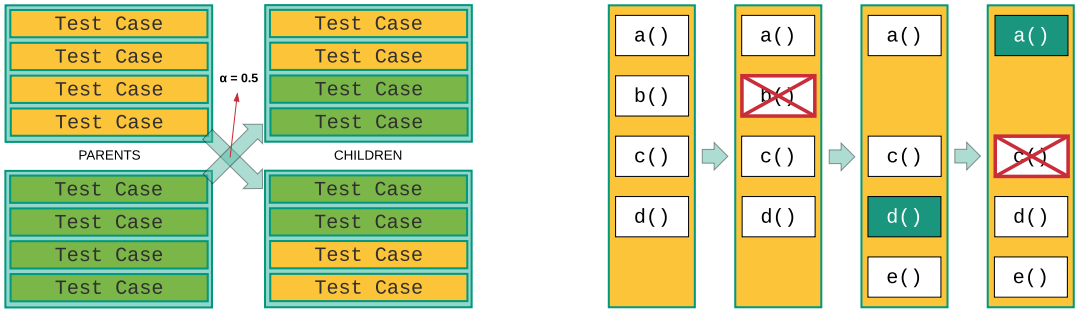
```
public void test0() throws Throwable {
    Bar bar0 = new Bar();
    boolean boolean0 = bar0.foo(5);
    assertFalse(boolean0);
}

public void test1() throws Throwable {
    Bar bar0 = new Bar();
    boolean boolean0 = bar0.foo(7);
    assertTrue(boolean0);
}

public void test2() throws Throwable {
    Bar bar0 = new Bar();
    boolean boolean0 = bar0.foo(1);
    assertFalse(boolean0);
}
```

Whenever a new test case is generated – as with the construction of the initial population, illustrated in Figure 2.1 – it is done so randomly as described in the previous section, with seeding enabled. An initial population of 50 chromosomes (test suites) is first generated. The population is then evolved using the two search functions crossover and mutation, as illustrated in Figure 2.2 and described in Algorithm 1. Crossover involves recombining test cases across two test suites while mutation works at two levels: at a test case level and the test suite level. At the test case level, the mutation operator either randomly adds new statements, removes existing ones, or modifies them and their parameters. At the test suite level, it adds a fresh, randomly-generated test case to an existing test suite.

To guide the search towards achieving a high coverage test suite, the fitness value can be calculated based on the number of covered goals. However, as mentioned earlier, a fitness function based solely on the number of *covered* goals provides no guidance to goals that remain *uncovered*. As with previous works



(a) Test Suite Crossover **(b) Test Case Mutation**

Fig. 2.2 The search operators crossover and mutation, as applied in EvoSuite (a) single-point crossover is performed: an α is chosen randomly between [0,1] to divide the parent test suites, and the opposite portions of the test suites are combined to form the offspring (children), (b) a number of mutations are applied to a test suite, such as statement removal (indicated by a cross), statement addition, or statement mutation (highlighted).

in search-based test generation [109], EvoSuite incorporates branch distance metrics, which indicate how "far" a branch is from being executed. For example, if a conditional "`if (a == b)`" is to be executed as true, the "raw" distance can be computed as "$|\texttt{a} - \texttt{b}|$". In this way, the closer the values of `a` and `b` are to one another, the lower the branch distance is, and the closer the search is to covering the goal.

Since EvoSuite aims to evolve test suites where each test case covers as many branches as possible, the fitness function involves adding the distance value $d(b,T)$ for each branch b within a test suite T, computed as follows [51]:

$$d(b,T) = \begin{cases} 0 & \text{if the branch has been covered,} \\ \nu(d_{min}(b,T)) & \text{if the predicate has been executed at least twice,} \\ 1 & \text{otherwise.} \end{cases} \tag{2.1}$$

where $d_{min}(b,T)$ is the minimum raw distance value for the b for T, and ν is a function that normalises a distance value between 0 and 1. Since the test suite must cover both the true and false outcomes of each individual branch, a distance value is not computed until the conditional is executed to return

both *true* and *false* by the test suite. This is so that the initial execution of the predicate, with some specific true/false evaluation, is not lost in the process of pursuing the alternative outcome.

As longer test suites require more memory and execution time, controlling the length of the test suite can improve search performance [50]. Therefore, when deciding which test suites should proceed into the population for the next iteration of the search, EvoSuite prefers shorter test suites to test suites with the same fitness but are composed of a higher number of statements.

Rojas et al. [138] also find that the effectiveness and performance of whole test suite generation can be increased by keeping a record of testing goals, and focusing the search on the ones that are still uncovered. Other work has also been done by creating hybrid approaches combining GAs effectiveness on global search with local search techniques to increase effectiveness on covering branches [18, 53], or by using symbolic execution to generate input data for uncovered conditional paths that are difficult to solve by the GA [58]. Nevertheless, to what extent these advanced evolutionary techniques are beneficial to generate a high-coverage test suite is unknown – see Chapter 5.

2.1.6 Summary

In this section we broadly looked at software testing. In particular, we discussed several basic concepts and definitions in this domain, in addition to a brief survey of the literature on techniques for automatically generating tests for software programs.

In the following section we look at a specific type of software testing related to software evolution. That is, we provide answers from the literature to questions such as: How do developers test their program as they evolve? How can they make sure that making changes does not break existing functionality? How can they find unintended consequences caused by the modifications they have made to the program? When new tests are added as the program evolves, how can the growing cost of testing be reduced? Specifically, we look at several approaches and techniques in the literature which aim to tackle these problems.

2.2 Regression Testing

As software programs evolve, developers add, remove, or enhance the software, and along with these changes, they may introduce unintended side-effects (i.e., bugs) that affect the original functionality of the program. Therefore, developers need to ensure that their changes have not affected the functionality of the program. To gain confidence that the originally intended functionality of the program is intact, developers create tests and execute the set of tests – also known as the *regression test suite* – after changes are made. Any failing tests can indicate a regression in the functionality, unless the test is obsolete and needs to be repaired or removed. This practice, also referred to as regression testing [198], helps developers to detect regressions in the functionality of the program before releasing the new changes to the software, and has been widely adopted in the industry (e.g., [120], [43]).

However, simply by adding new tests and re-executing all tests after each change, the cost of regression testing keeps growing, which has lead a majority of research in the literature on regression testing to look at reducing this cost. Over the past decades, a large number of studies and experiments have been conducted in this field. Initially, due to the high cost of having to re-run existing tests after every modification, regression test selection solutions were suggested in order to run a partial subset of the test suites. Some other approaches in the area considered adding new test cases – or improving existing ones – to cover the code elements that were affected by the changes, or to increase the effectiveness of the test suite at finding faults. Newer approaches in the area look at automating the regression testing process using external (e.g., third-party) or randomly generated test suites. In this section, a survey of the previous approaches in the area is presented and discussed.

It is worth noting that the regression problems and work in this area are not limited to those discussed in this chapter, but also expand to Graphical User Interface (GUI) regression problems (e.g., [115, 178, 114, 113, 68]), web-related problems with interfaces, sessions, services, dynamic behaviour, and so forth (e.g, [191, 71, 39, 134]), software-component related problems (e.g., [122]), and so forth. We also do not cover model-driven regression testing approaches (e.g., [98, 97, 190]) since in this thesis we assume that developers may not have the time

and resources to produce such models, especially considering that maintenance of the model along with the program can exacerbate the problem, particularly for rapidly evolving programs.

2.2.1 Test Minimisation, Selection and Prioritisation

Much research has been done for making a better use of existing test suites. The following sections look at different approaches towards running test suites more efficiently, and ways to find the regression faults more quickly.

2.2.1.1 Minimisation Techniques

One of the main arguments against rerunning all the tests on the System Under Test (SUT), is that some tests are redundant, and all regression faults they may be able to identify can also be found by other test cases. Test suite minimisation techniques propose identifying such tests within the test suite and removing them. As such, these techniques lower the cost of regression testing by reducing the size of the test suite. The biggest challenge associated with these techniques is to not discard a test case that could detect a new fault if it was not removed [198].

If developers rely on other means to generate the tests such as random test generation (Section 2.1.3), often individual test cases contain statements that are duplicate or unnecessary to detecting a fault. When such a test fails, statements unrelated to the failure can hinder the debugging effort. Therefore, techniques have also been proposed to minimise the length of individual tests – as opposed to minimising the size of the test suite. For example, the work by Leitner et al. [99] which targeted shortening individual test cases while maintaining their ability to reproduce the failure.

Due to minimisation being an NP-complete problem, many different heuristics were suggested in order to minimise test suites. While most approaches relied on reaching high level of code or structural coverage, few others looked at alternative metrics. Marré and Bertolino [107] created a compact form of Control Flow Graph (CFG) and their minimisation technique relied on finding a set of tests that can span over their graph. Other alternative approaches in the area included

using a Model-based test suite minimisation, by analysing dependencies and automatically modelling the modifications [96].

To measure the effect of such techniques, a large number of empirical studies were conducted. In two major studies conducted by Wong et el. referred to as WHLM [182] and WHLP [180], test suite minimisation was effective in reducing the size of the test suite by a rate of 1% up to 44%, in test suites with respectively 50% and above coverage. However, the effectiveness of the test suites in detecting the regression faults were reduced by an average of less than 7.28%. On the other hand, studies conducted by Rothermel et al. [140, 141] contradicted the previous results, and found a reduction in effectiveness of over 50% and sometimes even reaching 100%. The reason for this contradiction is considered to be due to using a larger case study and having coverage-adequate test suites. It is suggested that minimisation techniques that allow higher levels of redundancy such as vector-based reduction have negligible effect on the effectiveness of the test suite [199].

2.2.1.2 Regression Test Selection

Similar to test suite minimisation, test case selection techniques try to reduce the size of the test-suite that needs to be executed for regression testing purposes (i.e., creating a subset). The main difference in this case is that, selection techniques use the changes made to the SUT in order to make the selection – that is, select a subset of the tests that are most relevant to the change to detect regressions. Minimisation techniques in comparison generally target code or structural coverage of the SUT, rather than taking advantage of the changes [198].

A wide range of test case selection approaches were suggested over the past decades. A technique suggested by Harrold and Souffa [78] analysed the data-flow of the program in order to identify changes in the program, and selected test cases that covered these changes. An issue associated with data-flow analysis techniques, is the lack of support for considering changes that do not use variables or do not cause data-flow changes. A different approach used by Yau and Kishimoto [197] applied symbolic execution in order to select a subset of the test suite. Their complex algorithm initially analysed the code to identify input partitions, afterwards it generated and symbolically executed test cases

based on the modified parts of the code. Finally, it matched the CFG of the execution with the real test cases, and selected the tests based on the matchings. The technique however was found to be very expensive, and therefore insufficient to tackle the problem of reducing the cost of executing the whole test suite.

Given that approaches such as data-flow analysis and test suite minimisation could lead to omitting fault-revealing tests, researchers proposed *safe* techniques [65]. Despite being less effective in reducing the size of the test suite, these safe regression testing approaches select every test case that is able to reveal a regression fault. Harrold et al. [79] proposed a control-flow based approach, which generated a CFG of the classes under test and identified modified edges in the graph. Later, it selected all the test cases that exercised the identified modifications. Their approach achieved a varied success rate across test suites from different object-oriented programs. Other test selection approaches include, but are not limited to dynamic execution slicing of the program, textual comparison of the SUTs, and model-based comparison techniques [198].

2.2.1.3 Prioritisation Techniques

In contrast to techniques that aim to reduce the size of the test suite, some researchers suggested reordering the test cases, so that the test suite execution would reveal the faults quicker and provide faster feedback. Many techniques in this area focused on prioritising test cases according to structural coverage, and therefore aim to reach the highest coverage as soon as possible. Rothermel et al. [142] and Elbaum et al. [42] described several objectives and techniques for prioritisation of test cases. Some of the objectives included prioritising tests with highest rate of fault detection, and structural coverage. These techniques included different coverage criteria, such as prioritising based on coverage of branches and statements, as well as prioritising based on probability of fault exposure. The result of the study found significant improvement in rate of fault detection in all the heuristics.

Although it is not the objective of prioritisation techniques to lower the cost of test execution (as opposed to minimisation and selection approaches), it tries to maximise the cost of overall testing efforts. For instance, an earlier detection of a fault could save the time that a developer would normally spend waiting

for the tests to be executed. Moreover, an early detection (a failing test) could be used as a halting point for test execution. Walcott et al. [176] proposed a time-aware prioritisation technique that would reorder the test cases in order to achieve the highest rate of fault detection while not exceeding a time budget. The technique relied on using a genetic algorithm, with the fitness function optimising for coverage while considering the time constraints. The study found significant improvement in terms of effectiveness over the time, compared to other techniques. However, a significant issue associated with this technique is that more time is required to perform this prioritisation than the execution of the whole test suite. Although, it can be argued that the prioritised test suite can be re-used in the future.

2.2.2 Test Suite Augmentation

While much work in the area of regression testing has focused on making a better use of existing tests by reducing the execution costs and increasing efficiency, when software changes involve new behaviour, there may be simply *no* tests to exercise the new behaviour [159]. As a result, even considering the case of rerunning all the existing test cases, some modified or new behaviour of the software may not be visible to these tests [148]. Extending existing test sets with new or updated tests after the program under test has been modified is known as *test suite augmentation*. Test suite augmentation techniques aim to address this problem, so that new tests are added – or existing ones are improved – in order to increase the effectiveness of the test suite. In this section, we look at a number of different approaches taken towards augmenting existing test suites.

2.2.2.1 Assisting Manual Test Suite Augmentation

Apiwattanapong et al. [11] proposed an approach named MATRIX for test suite augmentation which created a set of test requirements based on the analysis of the changes between two versions of a program. Using these set of requirements, developers could assess the effectiveness of their existing testing efforts, in order to decide how to further improve the tests. The analysis performed by MATRIX involved multiple steps and components. In the first step, the approach computed the differences between the two program versions by finding the new, modified

and deleted statements. Secondly, based on the change information, a set of requirements were generated for the modified part of the programs. In the next step, the existing test suite minimised by a safe regression test selection tool would be executed on the new version of the software, while being guided by the previously produced requirements. Afterwards, the result of the execution and the level of coverage would be analysed and compared with the requirements. Finally, manual test generation was required to provide test cases that can satisfy the unsatisfied requirements. The process would then repeat from the execution step with the augmented test suite, and would continue until either a specified percentage or all of the requirements would be satisfied. The limitation of MATRIX however was within the analysis step, which required symbolic execution to derive information related to the program state, and therefore a certain depth limit had to be imposed. Additionally, only a single change could be handled by the technique at a time.

Aiming to further improve and explore the technique implemented in MATRIX, Santelices et al. [148] proposed a full implementation of the technique, further enhancements in the approach, as well as an empirical evaluation of the technique using few open source projects. The tool named *MATRIX Reloaded* used symbolic execution of the pair of program versions, to gather information such as control and data dependencies. Improving on the original approach, this technique analysed propagation chains and compared symbolic states of the corresponding positions between the two versions of the program, at the end of each propagation chain. Afterwards, the state requirements for the chains would be computed, and the test suite would be analysed with regards to the generated requirements. Similar to MATRIX, this approach did not generate new tests to augment the test suite. Furthermore, due to using symbolic execution, it still suffered from scalability limitations. However, this limitation was different from the previous approach, as it was limited to the complexity of the changes and not the complexity of the programs. Because of this difference, it was more effective and flexible than the earlier technique. Nevertheless, these approaches required manual effort from developers.

2.2.2.2 Augmentation by Generating New Tests

Xu and Rothermel [196] introduced DTSA as a test suite augmentation solution, which was able to add new test cases to the existing test suite in order to create a coverage-adequate test suite for the modified version of the program. To achieve this, their approach initially used a safe regression test selection tool named Dejavu [139] to identify uncovered code by the existing test suite, and used dynamic symbolic execution to generate test cases that can reach those parts of the code. The approach was found to have two main limitations of a) limited scalability due to using dynamic symbolic execution; and b) the generation of new test cases relying heavily on existing test cases. Aiming to resolve such limitations, Xu et al. [195] conducted a further study into the factors that can affect the effectiveness and efficiency of such techniques. Additionally, they explored the effectiveness and possible benefits of using a new genetic-algorithm based test generation method. Their large empirical study had several findings in terms of the ordering of test cases, the benefit of using old test cases, and the effectiveness of different test generation techniques. It was found that overall, prioritising some of the existing test cases would not affect the effectiveness of the approach, while it may increase the efficiency of genetic-algorithm based test generation. In terms of the benefit of using old test cases in addition to the newly generated ones, it was found to have a significant improvement on the effectiveness of the approach, whereas the trade-off being a significant increase in costs. Comparing the effectiveness of genetic-algorithm and dynamic symbolic execution test generation techniques, dynamic symbolic execution was found to be effective on simple subjects, while genetic algorithms were more flexible and managed to be more effective in almost every subject in their study.

2.2.2.3 Continuous Test Suite Augmentation

Whereas automated test generation techniques often look at a single version of the program, there typically exist many versions throughout software evolution. Accordingly, as mentioned earlier, test sets need to be maintained, tests may need to be updated [37, 116, 117], and new tests need to be added to satisfy the requirements placed on the test set (e.g., level of code coverage). Approaches to automatically generate new tests for test suite augmentation have been proposed

using dynamic symbolic execution [196], search-based approaches [195, 193], as well as hybrid approaches [194]. Most of these approaches aim to maintain code coverage throughout software evolution, although some approaches try to complement existing tests with new behaviour related to past interesting tests [105]. However, should techniques start from scratch each time when these techniques are being applied to the same program repeatedly? Can we use knowledge gained during this process to lower the cost and improve effectiveness?

A tight integration of this coverage-based test generation with continuous integration has been proposed in the context of *continuous test generation* (CTG) [29] and continuous test suite augmentation [192], where test generation is triggered at every commit to a repository, and executed on a continuous integration server. This repeated application increases the effectiveness of the test generator, and allows it to distribute computational resources across multiple invocations. Applying test suite augmentation with new tests in practice is not limited to one class of the program under test, but rather involves a whole program. Moreover, simply applying test suite augmentation after each commit to a repository does not take advantage of the knowledge gained during augmentation in earlier commits. Campos et al. [29] aim to address these through their CTG technique, in which during continuous integration after a change is made to the program, they take advantage of information learned from earlier commits.

In their proposed approach, they first attempt to better allocate resources (e.g., search budget) to classes of the program that are more difficult to test. Second, the order in which classes of the program are tested can be important, given that a test case generated for one class may help with testing another class. And finally, tests generated for earlier commits can provide a better starting point for testing the newly changed program. The authors evaluated their technique on several open source and industrial projects and report a gain of up to 58% in the level of coverage achieved, while at the same time reducing the time spent on test generation by up to 83%. This suggests that such optimizations based on information learned during the lifetime of CTG can not only reduce the time required for testing, but also increase the effectiveness of the approach. Continuous augmentation of test suites has also been applied in other domains such as in a software product line (SPL) [192]. The authors propose a technique

where rather than generating tests for products in the product line individually, they perform this in an order that can take advantage of tests generated for products which were tested earlier. Similar to the findings of Campos et al. an evaluation of their approach showed it to be more effective and efficient than augmentation based on individual products.

2.2.2.4 Propagating Regression Faults

Most approaches in this area focus on reaching and executing the changes, yet reaching a fault does not guarantee the propagation of the fault to the output. Previous techniques for addressing fault-propagation (such as MATRIX), used symbolic execution which has limited scalability due to computation costs. Symbolic execution requires dynamically calculating all execution paths leading to the fault, as well as finding propagation paths after reaching the fault. Hence, as the complexity of the program increases, the number of possible execution paths may exponentially grow, therefore, a boundary has to be defined. Santelices and Harrold [149] proposed an approach for propagation-based regression testing. The main objective of their study was finding an optimised solution to the significant limitation of symbolic execution in the previous approaches. To achieve this, their approach analysed dependencies and state differences during the execution (dynamically), as opposed to statically computing test requirements in advance. This enabled them to avoid generating dependency requirements using symbolic execution and resulted in a more efficient test suite augmentation approach, where the end result was more effective than earlier techniques at revealing differences. The result of the study also shows that propagation-based strategies are more effective for revealing differences, as opposed to coverage-based strategies. Their technique however was not fully automated, and test inputs needed to be provided by testers.

2.2.2.5 Enhancing Fault-finding Ability

When developers create comprehensive and well-written manual tests, these tests are often written with certain scenarios in mind which have perhaps the highest business priority. Therefore, given the time constraints of developers, they often test using a limited set of inputs, resulting in a limited set of execution paths.

Marinescu and Cadar [105] argue that these tests can be taken advantage of by further enhancing their ability at revealing future faults. Their approach, implemented in a tool called *ZESTI*, explores additional execution paths around sensitive parts of the system under test using a light-weight symbolic execution approach. The approach considers the execution of all possible input values that could be provided to sensitive instructions. While the technique can be applied automatically, and was able to detect previously unknown bugs in open source programs, there remain a few shortcomings. First, execution of the technique can be expensive and scalability of the approach may be limited, and second, an existing set of high-quality tests should be provided as input.

2.2.2.6 Test Suite Augmentation – Summary

Test suite augmentation techniques reviewed so far looked at providing requirements for testers to augment their test suites, and some considered automatically generating such tests. While most approaches focused on adding test cases to achieve better coverage and reaching the fault, others aimed to tackle the propagation of these faults. However, the approaches often require existing tests, suffer from either scalability limitations of symbolic execution, or do not consider faults that may result from executing a sequence of method calls. As such, while these approaches aim to address the effectiveness of regression suites, they have shortcomings in terms of efficiency, and add additional analysis costs to the existing "re-test all" testing costs. Moreover, augmentation techniques often rely on an existing set of tests – preferably written manually – which can be time consuming to create. Even with access to such tests, the quality of the tests would affect the outcome of these techniques. Thus, in the next section we look at automated techniques for generating regression tests from scratch.

2.2.3 Automated Regression Test Generation

While augmentation techniques aim to augment an existing test suite, such tests are expensive to create, and may be simply not available. Another common assumption is that high quality tests already exist that can be enhanced to test new functionality. However, access to such tests can often be a luxury, and can take a significant portion of the development time. Automated test generation

techniques aim to tackle this problem, and when applied in a regression testing scenario, they aim to reveal potential regressions. In this section we summarise work in the area which aim to generate regression tests automatically.

2.2.3.1 Use of Third-party Random Test Generators

Orso and Xie [123] presented BERT as an automated test generation tool for regression testing. The technique aimed to identify behavioural differences between two versions of a software. BERT operated in three stages of initially generating a large number of test cases, secondly running the tests over both the old and new versions of the class under the test while capturing their outcome, and finally analysing the observed behavioural differences and presenting the result back to the user. The initial step was achieved by using third-party test generation tools such as Agitar's JUnit Factory[2], and Randoop [124]. Afterwards, for the second stage, BERT ran the test cases created in the first stage on both versions of the software, and for each executed test case it logged three main outcomes: 1) State of the instance, 2) Return values and 3) Output of the test. While logging the outcomes, BERT compared the values and recorded any found differences. In the end, at the third stage, the refined and analysed output would be displayed to the developer. Despite the limited empirical study on only one subject, the results of the study were encouraging, and BERT managed to find actual regression faults between two versions of the subject program. The approach is however limited, due to relying on the quality of the automatically generated tests. Since no guidance exists for the test production, the quality of the provided tests would ultimately determine the power of BERT – this further motivates our approach in Chapter 4. The technique however informs the user when it detects a difference in state differences between two versions of a program, even if the generated tests result in the same output. The approach used by this tool also relates closely to differential testing approaches that we expand on later in Section 2.2.4.

A similar approach taken by Taneja and Xie [163] named DiffGen, synthesised its own test driver that would try to execute the structure of both versions of the program. The method added missing branches between the versions to make the branches equally testable. To generate this test driver, initially a textual differentiation analysis between the versions would choose the methods that have

been modified. In the next step, for each changed method, a method would be added to the test driver class, which would compare the execution result of the changed method. To perform this comparison, the test driver method received the result of running the method on the old version of the software, and would run and store the result of the execution of the changed method. After executing the new version of the method, the execution state and objects would be compared. Afterwards, it used automated test generated tools in order to create a high coverage test suite for the synthesised test driver. Observing the state differences during the execution, the differences were recorded and were reported back to the developers. While the approach showed promising results in terms of fault finding, DiffGen has internal limitations, such as being unable to correctly compare object states if a field is modified, added or removed in the new version of the software. Additionally, any refactoring on method names or their signature would make them incomparable.

2.2.3.2 Generating Tests Systematically

While the approaches reviewed in the previous section attempted to find regression faults between the two versions automatically, they mainly relied on randomly generated or externally provided test suites. Therefore the quality of the provided tests would be the determinant of their power in regression testing.

Qi et al. [133] proposed an automated test generation tool that aimed to execute the changes as well as propagating these changes to the output. To achieve this, dynamic symbolic execution was exercised to find the execution path to reach the change. Afterwards, test cases were generated to reach these changes. Finally, the approach looks at modifying and tuning the inputs of the generated tests so that they can propagate the state infections caused by the changes to the output.

The test generation phase of this technique relies on first, running an existing or generated test suite created for the old version of the program on the new version and collects the execution paths. If the execution successfully reaches the change, the corresponding test would be returned. Otherwise, the execution path of the test case will be modified to have a closer reaching point to the change, and this step will continue until the test becomes able to reach the change. Once

the change is reached by the test, the technique will aim to propagate the test result to an observable difference in the output between the two versions. This step first checks whether the currently generated test input propagates to the output. If not, the propagation path of the previous execution will be analysed and the program computes a propagation path to decide how to propagate the change. Afterwards the input will be modified based on the analysis so that the test reaches the change and propagates it to the output. Several limitations were associated with this approach including being limited to independent changes in the software, as well as failing to address scenarios where a sequence of method calls are required to propagate the changes to the output.

2.2.4 Differential Testing

In 1998, McKeeman [108] suggested using a form of random testing named *differential testing*, as a way to complement existing regression testing practices. The technique requires taking at least two comparable programs that are available to the tester, and the author proposes that if an input propagates to different outcomes (e.g., different output, crash or infinite loop on one version, etc.) between two of the program, it can indicate a potential regression in the functionality. Given that a majority of large software systems are already accompanied with an exhaustive set of tests, these tests can be used as a starting point for input data that can reach deep areas in the program.

The author measured the effectiveness of the technique on C compilers. To achieve this, they used the existing set of test suites and generated mutated versions of the tests in order to have neighbourhood test suites, and executed them on different compilers. Although the author reported on several bugs they managed to identify using the technique, he discusses several challenges faced when using the technique in practice. For instance, an important barrier towards the adoption of the technique is the overhead that may be imposed on the developers. If the developers already have a backlog of real bugs, an automated system reporting *potential* bugs may not be desirable among the developers. Other challenges include creating high quality tests, minimization of the size of these tests, and presentation of the differential testing outcome. As a

result, while the technique was shown to be an effective way to find real faults, adoption of the technique in practice faces several challenges.

However, in spite of the challenges, developers used differential testing in the industry at DIGITAL and Compaq (later acquired by Hewlett-Packard) to detect bugs in their C compiler, and they report that the technique helped them to identify and fix several bugs. While it was originally applied to C compilers, McKeeman suggests that same technique can be extended to testing Java API across different platforms. Although differential testing was originally proposed to test comparable programs to check whether they achieve the same goal (e.g., different compilers of the C language, Java API across different platforms), or to test the same program across different environments, Evans and Savoia [45] propose a similar technique under the same terminology, for testing different versions of the same program, in order to identify changes.

The authors propose a simple technique for applying differential testing in the context of regression testing. Given an original and modified version of a program, taking advantage of automated test generation techniques, the approach is to generate unit tests based on each version of the program, and to execute it on the other version of the software. This enables developers to identify unexpected functional changes across the two versions of the program. The technique was evaluated on 3 versions of an open source program JTopas, where the authors seeded 38 faults in each version, and performed differential testing between the original and faulty programs. The results showed that 21-34% of the seeded faults were detected using the technique. Given that the technique can be applied completely automatically, it has potential to be used by developers to detect regression faults automatically. However, as the technique only reveals changes across the two versions of a program, it becomes the duty of the developer to decide whether the change in the functionality was expected, or is an unwanted side-effect.

So far we have looked at differential testing applied in the context of testing different-but-comparable programs (e.g., compilers of C language) or the same program in different environments [108], or for the purpose of unit-test generation [45, 123]. Other approaches have looked at annotations [189] and instrumentation [163] to execute different versions of program methods simultaneously with the help of third-party test generation tools, while some techniques

have looked at solutions to guide the execution towards paths that can lead to the changes [164]. Compared to the earlier techniques, a different and interesting application of differential testing in the industry is the technique proposed by Groce et al. [69], using which they aimed to detect and address possible but rare hardware failures in space missions. Given the hostile condition in space such as the high level of radiation, rare hardware failures – for instance on the storage device of a Mars Rover – are more likely to occur. In order to increase the tolerance of their software for detecting of such changes, the authors injected faults into the hardware simulation to create a faulty version of the software. Using this, they compared execution results of randomly selected POSIX operations on the two version of the hardware, and kept track of cases in which the hardware failure resulted in unexpected output or produced different output. Using this technique the authors automatically generated 255 tests within the first 25 weeks of testing, where each test revealed a different fault. While the authors mention that this type of continuous random testing does not prove the correctness of the software, it can help with establishing confidence.

2.2.4.1 Automated Testing of Code Patches

Software programs undergo new changes over time. In general, each change to a program's code can be considered as a set of additions and a set of deletions – while modifications involve deletion and additions. This information on the set of applied changes can be unified into a single *patch*, and can be applied on the pre-change version of the program to create the new version. This means that all changes applied to a program over its lifetime can be considered as code-patches, and one can construct the final version of the program by taking the first version of this program and apply all intermediate changes. An example of a patch in the form of a unified-diff is shown below:

```
1 @@ -745,2 +745,2 @@
2 -         return len > 0;
3 +         return len > 0 && s.charAt(0) != '0';
```

Code patches that are applied during the development life-cycle of the software program can also contain a bugfix or a security fix that users can take and directly apply to their programs. While software patches are mostly aimed

at fixing existing bugs, or extending the current functionality, they can result in unexpected side-effects, and therefore result in regressions.

Marinescu and Cadar [106] propose a technique which uses symbolic execution and dynamic analysis to automatically test code patches. Their technique implemented in a tool named *KATCH* aims at reaching the code of the patch to cover the changes. To achieve this, in their technique they first use the set of manually-written test suites by the developers and execute the program using the test inputs. Then, the inputs closest to the location of the change (patch) are selected (as seed) for the symbolic execution stage. Using different heuristics, the technique then aims to reach the uncovered parts of the patch. On a high-level, the technique explores paths that lead to the patch, and does so cleverly by choosing the sides of branches that are more likely to lead to the changed code (based on the distance to the change). If during this exploration, a selected path is infeasible or data-dependency exists, an alternative path is regenerated. During this process, for branches that cannot be solved symbolically, concrete execution is used. The technique also takes advantage of definitions in the program structure to generate alternative input definitions that can help with reaching the desired targets.

Similar to several other techniques relying on symbolic execution presented earlier, while KATCH was found to be effective on real world programs (e.g., GNU `diffutils`, `binutils`, `findutils`), it mainly targets reaching the change and does not target propagation. Additionally, the technique relies on an existing set of unit tests and the quality of tests can directly affect the effectiveness of the technique.

While most of the aforementioned symbolic execution based techniques aim at generating tests that reach (cover) the patch, Palikareva et al. [125] look at generating test inputs that can lead to behavioural differences across the two versions of the program (i.e., taking a differential testing approach towards this problem). The authors present a technique named Shadow, which executes two versions of a program (i.e., the old and the new version) in the same symbolic execution environment, with the objective of generating inputs that propagate to different behaviours in the output, and to create divergences between the control flow of the two versions of the program. To achieve this, first the two versions of the program should be unified into one version using manual annotations. Then,

after executing the existing test suite of the program, the test inputs that reach the patch area are selected. If the inputs do not reach the patch, the closest inputs to the patch are selected and using *KATCH* (as described earlier), they generate inputs that can reach the patch.

Using inputs that reach the patch as starting point, the technique then uses dynamic symbolic execution to create divergences in the control flow across the two versions. Additionally, it aims to propagate the divergences to the output using bounded symbolic execution. Finally, the output values/behaviours are compared across the two versions. The authors evaluated the technique on 18 regression faults taken from the CoreUtils project, and demonstrate that the technique can be effective at creating both divergences and detecting some of the faults. However, Shadow suffers few limitations such as requiring an existing set of tests, and additionally a manual overhead is imposed on the developers to annotate the patches.

2.2.5 Regression Verification

In contrast to differential testing approaches, where the objective is to reveal differences, *regression verification* tries to verify that two similar programs are *equivalent*: ensure that the new version of the program is at least "as correct" as the older version [63]. This area tries to combine regression testing and formal verification, to validate the equivalence of similar programs, given a definition of equivalence.

An example of this is the work by Böhme et al. [23]. They suggest an approach where subsets of the input space (also known as input partitions) are separately and gradually verified. This can be beneficial, given that verifying the equivalence of two programs using all inputs from the input space can be simply infeasible. Moreover, if the execution of the technique is stopped at any point in time, one can be certain that for the partitions that have been verified so far, no different behaviour exists. The approach named PRV starts with random inputs, and then using symbolic execution it explores the common input space between the two versions of the program that lead to the same behaviour (either the behaviour of the two programs is identical, or a behavioural difference is observed). The approach was evaluated on a small number of open-source

programs, and was shown to be effective at revealing differences. Compared to other symbolic approaches, PRV relies on random inputs and re-execution of the program in order to continue its exploration, which may limit the scalability of the tool on large subjects.

2.2.6 The Oracle Problem

> The ugliest problem in testing is evaluating the result of a test.
>
> (McKeeman [108])

After a test is executed, as McKeeman states, determining whether the behaviour of the program was correct is a challenging problem. This is mainly due to the fact that an oracle needs to validate the behaviour of the program. Test oracles (assertions) are created to validate the behaviour of the program based on the expected behaviour. This expectation can be taken from the specifications of the program, or derived from other sources such as documentation, earlier versions, the system under test itself, and so forth. The oracles may also be implicit, such that a *segfault* for instance is most likely a fault. Using these, developers can manually add oracles in their test, or sanity checks/validations inside the program code to automate the decision on whether or not the program is behaving as intended.

However, during debugging, adding such oracles is not the only time consuming part of the process. When a failure occurs, a developer needs to make the final judgement on whether or not the program is behaving correctly. The next challenge is deciding what parts of the program state and behaviour need to be validated. Should developers write all possible assertions to validate the state of the program? While developers can often decide the important aspects of the program to test and validate, this can be significantly more challenging for automated techniques. Without knowing the original intention of the developers, and without knowing the correct behaviour of the software, constructing a sensitive – and yet not excessive – set of oracles is difficult for an automated tool. Although researchers have proposed different techniques such as keeping only tests that are more effective at killing mutants, this remains an open problem [19].

In the context of regression testing, one way to approach this problem is to assume that the behaviour of the current version of the program is correct, and set the objective to avoid behavioural changes by validating the new version's behaviour against the old one (e.g., as is assumed by some of the techniques reviewed earlier in Section 2.2.3 and Section 2.2.4). Xie [186] propose such an approach to augment existing regression tests with more assertions to increase their effectiveness at detecting future behavioural changes. However, applying such techniques can have negative consequences on execution time, debugging time (developers need to spend more time inspecting failing tests), and so forth. Moreover, as mentioned earlier, this approach can lead to creation of an excessive set of oracles. An alternative approach by Fraser and Zeller [56] uses mutation testing to derive a better subset of oracles that are effective at revealing defects. However, this may still lead to more failing tests that originate from the same code-changes, thus leading to wasted efforts on inspection and maintenance – see Section 4.2.3 and Chapter 6.

2.2.7 Maintaining Regression Test Suites

As a program evolves, regression tests need to evolve with it. Forming the majority of test changes are *new* tests which are added to test new functionality, test changed parts and bug fixes, or to increase coverage. Close to a third of test changes involve modifications of the tests, part of which involve repairing existing failing tests (e.g., when the expected behaviour of the system under test has changed), or other reasons such as improving the quality and effectiveness of existing tests. Tests can also be deleted as they become obsolete. Therefore, significant effort is required to maintain test suites along with the programs [130]. Maintaining automatically generated tests is even more time consuming than maintaining those written manually [162].

To help with maintenance efforts involving test repairs, work has been done in the literature to automate this process. Daniel et al. [37] present a technique to suggest how a failing test should be repaired, accompanied by a tool which allows developers to achieve this with a single click. Their approach involves first, instrumenting the program under test and then locating where the test needs to be repaired. Then using a strategy that is most adequate for the particular

change — which is based on different models of unit test failures – the repair is suggested based on the new behaviour of the program. Mirzaaghaei et al. [117] propose an alternative approach, such that existing failing tests are repaired by adapting them to the new behaviour of the program. In their approach, existing tests are evolved based on information available in the test, such that new passing tests are created to replace the failing ones. While these approaches – and other work on test repair – show promising result, their effect on test suite maintenance in the best case scenario can be minimal. As Pinto et al. [130] report, in practice, only 22% of test modifications involve repairing failing tests. Moreover, this does not reduce the time a developer has to spend inspecting the failing tests. Chapter 6 investigates whether automated test generation techniques can help with reducing these maintenance costs.

2.2.8 Summary

In this section, we looked at the problem of regression testing in software testing. In particular, we described the increasing cost of regression testing over time and some proposed approaches that can tackle the problem. We also looked at alternative techniques for generating and augmenting test suites, either using existing set of tests or by generating them from scratch. Furthermore, we discussed some of the challenges faced by the existing techniques.

Although much work has been done in this area to aid developers, a majority of the approaches either rely on existing manual testing effort, or suffer scalability issues. Moreover, while many techniques have been proposed to generate tests automatically such that they can be used for regression testing, whether the generated test suites are effective at finding real bugs remains an open question. Therefore, to better understand the state-of-the-art in automated test generation to detect real faults, we investigate this question in the next chapter.

Chapter 3

Evaluating Automated Unit Test Generation Tools Using Real Faults

The content of this chapter is based on work undertaken during this PhD by the author, which has been published elsewhere [156] © 2015 IEEE.

3.1 Introduction

In the previous chapters we discussed the benefits of automated unit test generation techniques and reviewed a number of different efforts towards this end in the literature. However, while a majority of papers proposing these works include a set of experiments evaluating the accompanied techniques, it is unclear how they compare with each other, and where the state-of-the-art techniques stand in terms of automatically generating tests that can detect *real* faults.

Moreover, automated unit test generation techniques are often evaluated against code-coverage criteria – where even achieving high levels of code coverage does not imply that the test will be effective at fault finding, for instance automatically generated unit tests can reach even higher coverage levels than those written by developers, yet they are less effective than manual tests [54] – or using mutation score as a proxy – where finding seeded faults does not

necessitate that they can detect faults that developers make. Therefore, a large-scale comparison of the state-of-the-art tools on their effectiveness at detecting real faults and a framework to do so is lacking from the literature. Additionally, access to such a framework is essential to evaluating our own proposed technique and approach which are presented later in Chapter 4 and Chapter 6. This chapter provides such a framework.

In this chapter, we empirically evaluate automatic unit test generation using the Defects4J dataset, which contains 357 real faults from open source projects [89]. We applied three state-of-the-art unit test generation tools for Java, Randoop [124], EvoSuite [51], and AgitarOne [167], on the Defects4J dataset, and investigated whether the resulting test suites can find the faults. Specifically, we aim to answer a) How do automated unit test generators perform at finding faults? and b) How do automated unit test generators need to be improved to find more faults? The answer to these two questions enables us to derive insights to support the research and development of automated test generation tools, and addresses some of the previously discussed shortcomings in the literature.

In particular, the contributions of this chapter are as follows:

- A framework for evaluating the effectiveness of tests generated using automated techniques at detecting changes
- A large-scale experiment, applying three state-of-the-art automated unit test generators for Java to the 357 faults in the Defects4J dataset.
- A detailed analysis of how well the generated suites performed at detecting the faults in the dataset.
- The presentation of a series of insights gained from the study as to how the test generators could be improved to support better fault discovery in the future.

The rest of this chapter is structured as follows: First in Section 3.2 we describe our methodology and the framework we implemented to conduct this study. We discuss the subject programs, the automated test generation tools used to test these programs, our strategy to filter and evaluate the generated

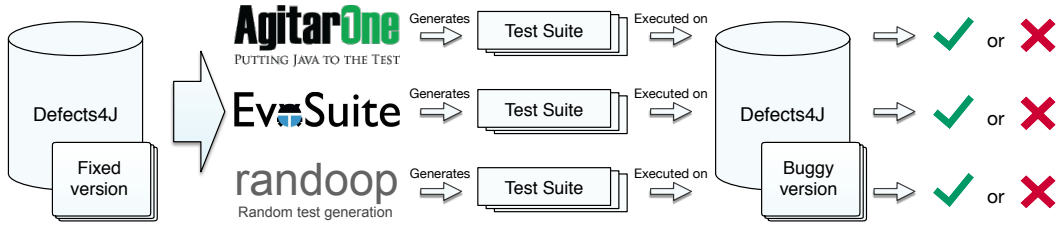


Fig. 3.1 Overview of the experimental setup. For each fault, the Defects4J dataset provides a *buggy* (i.e., faulty) and a *fixed* version. Test suites are generated with all tools on the fixed version, and executed on the buggy version. © 2015 IEEE

tests, and the threats to validity inherent to this study. Next, we detail our results and answer to our research questions respectively in sections 3.3 and 3.4. In Section 3.5 we compare our work to related work in the area, and finally conclude this chapter in Section 3.6.

3.2 Methodology

In order to answer our research questions, we designed an experiment according to the high-level methodology shown in Figure 3.1. We considered Defects4J, a database of known, real faults, where each fault is represented by a buggy and a fixed program version. We applied three automated test generation tools (AgitarOne, EvoSuite, and Randoop) on the fixed version, and executed each generated test suite on the buggy version to determine whether it detects the fault – that is, whether it fails on the buggy version. This means that we are considering a regression testing scenario, where a developer would apply test generation to guard against future faults. We then analysed in detail which faults were detected and how, and which faults were not detected, and why. This section describes the experimental setup in detail.

3.2.1 Subject Programs

The Defects4J [89] dataset consists of 357 real faults from five open source projects: *JFreeChart* (26 faults), *Google Closure compiler* (133 faults), *Apache Commons Lang* (65 faults), *Apache Commons Math* (106 faults), and *Joda Time* (27 faults). Defects4J makes analysing real faults easy: For each fault, it

provides 1) commands to access the buggy and the fixed program version, which differ by a minimized change that represents the isolated bug fix, 2) a developer-written test suite containing at least one test case that can expose the fault, and 3) a list of classes relevant to the fault – that is, all classes modified by the bug fix. As per the regression testing scenario (i.e., guarding against future faults in modified code), we used this list of relevant classes for test generation.

3.2.2 Automated Unit Test Generation Tools

All software projects in DEFECTS4J are written in Java, therefore we also considered test generation tools for Java. NightHawk [9], JCrasher [35], Carfast [128], T3 [132], and RANDOOP [124] are instances of random unit test generation tools. We chose RANDOOP as it is the most stable and popular representative of these random testing tools.

TestFul [18], eToc [171], and EVOSUITE [51] are instances of search-based unit test generation tools, which use meta-heuristic search techniques to optimize test suites with respect to code coverage criteria. We chose EVOSUITE as it subsumes the other tools in terms of functionalities, and is the only actively maintained tool out of the three.

Dynamic Symbolic Execution (DSE) testing tools for Java such as DSC [84], Symbolic PathFinder [129], and jCute [151] require test drivers, whereas our scenario of unit test generation assumes that the tests generated include these test drivers – as typical unit tests do (i.e., in terms of method call sequences). While EVOSUITE has an experimental DSE extension [58], it is not yet enabled by default, and therefore was not used in our experiments. Note that for C#, tools such as Seeker [168] or Symstra [188] can generate method call sequences, but the only publicly available tool – Pex [169] — has only a limited ability to do so. However, our experimental setup based on Defects4J requires tools that apply to Java.

Finally, there are commercial unit test generation tools for Java, of which naturally less is known about implementation details. We tried Analytix CodePro [64], Parasoft JTest [126], and AGITARONE [167]. CodePro achieved low coverage and is no longer officially supported. Although we were able to generate test suites with JTest, we were unable to execute the tests with our analysis

framework (even with help from Parasoft's support team) because the tests depend on JTest's proprietary stub generation utility. As AgitarOne exhibited fewer problems with test dependencies compared to JTest, we chose AgitarOne for our experiments.

Randoop [124] implements feedback-directed random test generation for object-oriented programs. This means that it iteratively extends sequences of method calls with randomly selected candidates until the generated sequence raises an undeclared exception or violates a general code contract. Randoop also executes its generated sequences and creates assertions that capture the behaviour of the class under test. However, Randoop cannot target a specific class under test since it uses a bottom-up approach that requires that all dependencies of a method call (i.e., arguments) have been instantiated in a prior iteration. Randoop therefore requires, as input, a list of all classes it should explore during test generation. For each class under test we provided a list containing the class under test and all its dependencies as determined by Defects4J[1]. We applied Randoop for three minutes on each run, using a unique random seed since Randoop is deterministic by default (i.e., the random seed is always 0). For all other settings, we applied the defaults, with the exception of enabling null values as method arguments with a probability of 0.1; this improves Randoop's effectiveness in our experience [90].

EvoSuite [51] applies a genetic algorithm in order to evolve a set of test cases that maximizes code coverage. It starts with a population of test suites of random test cases, and then iteratively applies search operators such as selection, mutation, and crossover to evolve them. The evolution is guided by a fitness function based on a coverage criterion, which is branch coverage by default. Once the search ends, the test suite with the highest code coverage is minimized with respect to the coverage criterion and regression test assertions are added [56]. EvoSuite then checks for each test whether it is syntactically valid by compiling it, and executes it to check whether it is stable (i.e., passing). Any failing assertions at this point are commented out by EvoSuite. Since EvoSuite can target a specific class under test, we generated, for each fault, a test suite targeting only classes relevant to that fault, as reported by Defects4J.

[1]Defects4J dynamically determines dependencies by monitoring the class loader during the execution of the developer-written tests.

We used EvoSuite with its default options, except for two settings: 1) We set the stopping criterion for the search to three minutes per class and 2) we deactivated the mutation-based filtering of assertions such that all possible regression assertions are included, because the filtering can be computationally costly and caused some timeouts.

AgitarOne [167] is a commercial test generation tool for Java developed by Agitar Technologies, which is advertised as being able to achieve 80% code coverage or better on any class. The tool consists of a client and a server application. According to AgitarOne's support, the test generation is "fairly" deterministic, and a user should *not* need to generate multiple test suites for a given program version[2]. The client is an Eclipse plugin that connects to the server[3] to request the generation of test cases, and the server takes a variable amount of time to do so, depending on the class under test. As we did not succeed in automating AgitarOne, we imported the fixed project version of each Defects4J bug into Eclipse and manually invoked AgitarOne's Eclipse plugin to generate test suites. AgitarOne can target a specific class under test, and we therefore generated a test suite for each class relevant to the fault. If AgitarOne failed to generate tests for a class, we re-attempted the request five times. Nevertheless, the tool was not able to generate a test suite for 34 (9.5%) faults (Randoop and EvoSuite were not able to generate any test suites for 2 and 1 (<1%) faults). AgitarOne makes heavy use of a proprietary mocking system that requires AgitarOne's own test runner. As we could not integrate this test runner in Defects4J's infrastructure, which uses Apache Ant's JUnit test runner, we collected the same experimental data as for the other tools using AgitarOne's dedicated test runner.

3.2.3 Experiment Procedure

We applied the following procedure.

[2]Personal communication, August 2015

[3]The test generation server was deployed on a Linux computer with 64GB of RAM and 32 CPU cores @2.1GHz

3.2.3.1 Test Generation

RANDOOP and EVOSUITE are both randomized tools, capable of producing different results on each invocation. To account for this randomness, we generated 10 test suites for each tool and fault on the fixed version as provided by Defects4J. As AGITARONE required manual effort to generate and analyse a test suite (e.g., manually selecting the classes under test and starting the tool in Eclipse), we only generated 1 test suite for each fault with AGITARONE.

3.2.3.2 Flaky Tests

In order to determine whether a test detects a fault, we require it to pass on the fixed version and fail on the buggy version. However, tools may generate *flaky* (unstable) tests, which may also fail on the fixed version. For example, a test case that contains an assertion that refers to the system time will only pass during generation, and will fail when re-executed later. We applied the following automated process to remove flaky tests: First, all non-compiling test classes were removed. Then, each compilable test suite was executed on the fixed version five times. If any of these executions revealed flaky tests, then these tests were removed, and the test suite re-compiled and re-executed. This process was repeated until all remaining tests passed five times in a row.

Technically, a test may reveal its flakiness only at a later stage (e.g., if a test depends on the current date, it might fail only after the current day is over). By that time, the source code of the class under test might have been changed, and one would need to spend time to understand whether a failing test has found a fault or it is just flaky. Although possible, we did not encounter any such cases in our experiments.

3.2.3.3 False Positives

Even if a test is not flaky, it might still fail on the buggy version for reasons that are unrelated to the actual fault – that is, it is a *false positive*. Flaky tests can technically also be false positives, but false positives mainly happen when test generation tools break the object-oriented principle of encapsulation, for

example by calling private methods directly through reflection, or by capturing outdated behaviour of dependency classes with mocks.

We identified false positives as follows: For each test that failed on a buggy version, we compared the failure message and stack trace produced by the failing tests of the developer-written test suite included in Defects4J with that of the generated test. If a test failed with the same exception or a similar assertion, we considered it a true positive. If the exception or assertion differed, we manually validated whether the failure was caused by the fault or whether it is a false positive.

We found false positives for Randoop and AgitarOne but not for EvoSuite. Note that Randoop suffered false positives only for Closure, and only with tests that executed a dependency class rather than any of the relevant classes. For AgitarOne we identified two common types of false positives: test failures due to mocking and accessing (missing) private class members. Consequently, we automatically classified an AgitarOne test as false positive if it only failed because of mocking or missing private class members (see Section 3.4.4). While we could have prevented this problem to a certain extent by changing the faults provided by Defects4J (i.e., inlining all changes or maintaining unused code), we argue that this would not reflect common practice.

3.2.3.4 Fault Detection

To determine fault detection we executed the test suites against the buggy version of each bug as provided by Defects4J. For Randoop and EvoSuite, we executed the test suites using Defects4J's JUnit test runner; for AgitarOne we used its proprietary JUnit test runner. For each executed test, we collected information on whether it passed or failed, and if it failed we logged the reason (i.e., the failure message and stack trace).

3.2.3.5 Coverage Analysis

In order to study how code coverage relates to fault detection, we measured statement coverage on each class relevant to the fault. Furthermore, given the set of program statements modified by the bug fix (i.e., the difference between

Table 3.1 Overall outcome of the test generation and execution process. For each *project* and *tool*, the table shows the percentage of *compilable* test classes in all test suites, the average number of generated *tests* in them, the percentage of how many of these tests are *flaky*, the percentage of failing non-flaky tests that were *false positives*, and the average code *coverage* ratio for all non-flaky test suites on classes relevant to the bug. It also shows the *max* and *average* number of bugs per project that each tool detected (excluding false positives), and details how the bugs were detected (i.e., a failing *assertion*, an unhandled *exception*, or a *timeout*). Note that for EvoSuite and Randoop, a bug might have been detected by only a subset of the 10 generated test suites. © 2015 IEEE

Project	Tool	Compilable	Tests	Flaky	False Pos.	Coverage	Max Bugs	Avg. Bugs	Assertion	Exception	Timeout
Chart	AgitarOne	100.0%	131.2	0.2%	30.6%	84.7%	17	17.0	10.0	11.0	0.0
	EvoSuite	100.0%	45.9	3.5%	0.0%	68.1%	18	9.7	5.4	5.2	0.3
	Randoop	100.0%	4874.9	36.8%	0.0%	54.8%	18	14.1	7.5	9.1	0.0
	Manual	100.0%	230.6	0.0%	0.0%	70.5%	26	26.0	17.0	12.0	0.0
Closure	AgitarOne	100.0%	199.4	0.4%	79.3%	79.1%	25	25.0	16.0	10.0	0.0
	EvoSuite	100.0%	34.9	1.7%	0.0%	34.5%	27	11.8	10.5	1.4	0.0
	Randoop	98.4%	5518.4	19.8%	15.8%	9.8%	9	2.2	0.5	1.7	0.0
	Manual	100.0%	3511.1	0.0%	0.0%	90.9%	133	133.0	103.0	42.0	0.0
Lang	AgitarOne	100.0%	127.7	1.0%	23.5%	50.9%	22	22.0	10.0	14.0	0.0
	EvoSuite	79.5%	48.6	5.4%	0.0%	55.4%	18	9.2	5.5	3.3	0.9
	Randoop	68.3%	11450.7	5.7%	0.0%	50.7%	10	7.0	1.7	6.3	0.0
	Manual	100.0%	169.2	0.0%	0.0%	91.4%	65	65.0	31.0	36.0	0.0
Math	AgitarOne	100.0%	105.8	0.1%	8.9%	83.5%	53	53.0	34.0	25.0	0.0
	EvoSuite	99.8%	29.7	0.2%	0.0%	77.9%	66	42.9	26.1	17.7	0.3
	Randoop	97.8%	7371.4	15.6%	0.0%	43.4%	41	26.0	17.8	10.8	0.0
	Manual	100.0%	167.8	0.0%	0.0%	91.1%	106	106.0	76.0	31.0	0.0
Time	AgitarOne	100.0%	187.2	3.3%	30.9%	86.7%	13	13.0	10.0	8.0	0.0
	EvoSuite	100.0%	58.0	2.8%	0.0%	86.7%	16	8.5	4.9	4.0	0.0
	Randoop	81.1%	2807.1	25.3%	0.0%	43.0%	15	4.5	3.8	1.1	0.0
	Manual	100.0%	2532.7	0.0%	0.0%	91.8%	27	27.0	13.0	17.0	0.0

the buggy and the fixed version), we measured *bug coverage* – that is, whether a fault was 1) fully covered (all modified statements covered), 2) partially covered (some modified statements covered), or 3) not covered.

For Randoop and EvoSuite, we used Cobertura[4] to measure code coverage. As AgitarOne's proprietary coverage tracking mechanism conflicts with Cobertura, we relied on the proprietary coverage files generated by AgitarOne. To that end, we extended Crap4j[5] [150] and extracted the code coverage ratio from the coverage files produced by AgitarOne's test runner for each relevant class. To determine whether a fault was fully, partially, or not covered, we manually inspected the visual code coverage indicators of the modified statements using AgitarOne's Eclipse plugin.

[4]http://cobertura.github.io/cobertura/, accessed October 2016

[5]Java project quality assessment tool, originally written by the developers at Agitar Technologies

3.2.4 Threats to Validity

In this study, we used bugs taken from only five Java open source projects, which may not generalise to all programming languages and different program characteristics, and thus constitutes a threat to *external validity*. Our study considered three state-of-the-art test generation tools, of which one is commercially available and actively used by developers. However, our study does not include tools based on Dynamic Symbolic Execution or other specialised techniques, and such tools may be more effective at some challenges we identified than the tools used in our study (e.g., complex conditions).

A potential threat to *internal validity* is that not all tests that detected a fault were manually investigated to ensure the validity of the bug detection result. However, we mitigated this threat by using several sanity checks, such as comparing failure reasons of generated and developer-written test suites. Furthermore, we manually inspected a large number of test suites, in particular the ones that exhibited an unexpected failure reason. AgitarOne may produce different test suites when invoked on the same class several times, depending on external factors such as available resources. Thus, there is a potential threat as we only generated a single test suite for each bug with AgitarOne. However, we experimentally validated the claim of Agitar's support of the tool being fairly deterministic by sampling 10 faults, and found that the tool is generally consistent in whether it detects a bug. Moreover, AgitarOne spent significantly more than three minutes on some classes, and allowing Randoop and EvoSuite more time for test generation may produce better results. However, based on our experience, a search budget of three minutes is sufficient for the search in EvoSuite to converge in most cases, such that more time would not further change the tests. For Randoop we observed that code coverage saturated already within <1 min, and the test suites exhibit a very high degree of redundancy. Therefore, we do not expect our choice of test generation time to affect effectiveness.

A potential threat to *construct validity* is the use of all bugs in the Defects4J dataset, as Defects4J does not distinguish the type or severity of faults. Furthermore, each bug is represented by a minimized diff between the buggy and a later fixed version (to reduce the number of false positive detections; for example, if a method unrelated to the bug exists in one version but not the

other, the same method is added to the other version such that tests generated for this method would not fail because the method simply doesn't exists, which is an error unrelated to the actual bug), rather than the actual code change that introduced the bug. Given that none of the tools actually use the diff to produce the test suites, the minimized diffs do not affect the effectiveness of any of the tools. Nevertheless, although the bugs are real bugs, not all may be representative for regression faults. Therefore, our study might underestimate the effectiveness of automated test generation tools for regression testing if some types of the undetected faults are unlikely to be inadvertently introduced by a developer in the future. Likewise, our study might overestimate the effectiveness if some types of the detected faults are unlikely to be inadvertently introduced in the future. Furthermore, our study relies on the versions of code committed to a public repository and does not include regressions that a developer identified before committing the changes to the repository. This may underestimate the effectiveness of test generation tools if these uncommitted changes are easier to detect. This threat is mitigated somewhat as the bugs are taken from all stages of the project history, including earlier stages of development.

3.3 Do Automated Unit Test Generation Tools Find Real Bugs?

3.3.1 How Many Usable Tests Are Generated?

The left-hand side of Table 3.1 reports the outcome of the test generation. Unlike AgitarOne, both EvoSuite and Randoop generated test classes that did not compile. Non-compilable tests are generated often due to bugs internal to the tools (e.g., when a tool has an internal representation of the test case, but when this representation is transformed to a JUnit test suite, the produced code is not valid Java code). Unsurprisingly, Randoop generated the largest number of tests for all projects as it does not target a specific class under test. In contrast, EvoSuite and AgitarOne generate tests specifically for the selected classes under test, resulting in substantially fewer tests. For reference, we also include the number of tests in the developer-written test suites (manual). Note that these numbers refer to all relevant tests, as reported by Defects4J: A test is

considered relevant if it directly or indirectly covers any of the classes relevant to the fault.

AgitarOne generated the lowest ratio of flaky tests overall, with a maximum of 3.3% for Time. As Time makes heavy use of the system time, this is not surprising. EvoSuite suffered between 0.2%–5.4% flaky tests, but interestingly fewer for Time – presumably due to its built-in test isolation and check for flaky tests. Unlike AgitarOne and EvoSuite, Randoop does not isolate tests from the environment, and as a result suffered between 5.7%–36.8% flaky tests. We mainly observed false positives for AgitarOne, in particular for Closure, due to the use of mocking and reflection to increase code coverage.

Out of the three test generation tools, AgitarOne generally achieved the highest code coverage ratio, except for Lang. EvoSuite and Randoop struggled to achieve code coverage on Closure – most likely because of the large number of private methods. In comparison to the developer-written test suites, the test generation tools achieved lower code coverage overall, except for Chart. Yet, all tools achieved higher code coverage than developer-written test suites for some classes.

EvoSuite *and* Randoop *generated 3.4% and 8.3% non-compilable test suites on average. Moreover, on average, 21% of* Randoop*'s tests were flaky, and 46% of* AgitarOne*'s failing tests were false positives.*

3.3.2 How Many Bugs Are Found?

Overall, the generated test suites found 199 out of the 357 bugs (55.7%). On the face of it, finding more than half of the bugs sounds like an encouraging result. However, consider Table 3.3, which gives a visual overview of the bug-finding results of the analyzed tools on the complete set of bugs: Found bugs are denoted with filled boxes, there is one row for each bug, and one column for each execution of a test generation tool. Clearly, the filled boxes are sparse in this table as only 19.9% of all executions detected a bug.

Considering tools individually, EvoSuite, AgitarOne, and Randoop found 145, 130, and 93 bugs, respectively. That is, the number of bugs found by each tool is comparable – around one third of all bugs, which is already

substantially less than the overall number of bugs found (199). However, as the sparsity of black boxes in Table 3.3 shows — even for bugs that were found – tools like Randoop and EvoSuite use randomized algorithms, so the properties of the generated tests differ for each run. Consequently, a bug may be found in one run, but not in the next. If we say a bug is "likely to be found" if it was found in more than half of the executions of the tool, the number of bugs found for Randoop and EvoSuite changes to 54 and 83, respectively. If we consider a bug as found only if *all* executions of a tool detected the bug, then these numbers decrease to a sobering 28 and 38.

Regarding the effectiveness of the tools per project, there are some distinct differences: The part of Table 3.3 for the Chart project is quite densely populated, whereas the Closure project seems to be generally more problematic for test generation tools. Table 3.1 summarizes the visual presentation of Table 3.3 in numbers, and confirms this intuition: For the Chart project, Randoop and AgitarOne found more than half of the bugs on average (only EvoSuite struggled and only detected 9.7 out of 26 bugs on average). For the Closure project, even AgitarOne discovered only 25 out of 133 bugs, and Randoop found just 2.2 bugs on average. The bug detection results for the Lang and Time project look similarly grim. The Math project seems to be slightly better suited for test generation, with AgitarOne finding half of the bugs, and EvoSuite coming close to this result.

The picture painted overall is that if one wants to find all bugs, one should not rely solely on an automated unit test generation tool. This suggests that plenty remains to be done to improve automated test generation tools. Nevertheless, the fact that 199 bugs were found fully automatically does showcase the potential of such tools for widely used practices such as regression testing. The results also show that none of the test generation approaches is strictly superior to the other two.

Automated test generation tools found 55.7% of the bugs we considered, but no tool alone found more than 40.6%.

Table 3.2 The percentage of detected bugs, categorized by whether the bug in question was found by the developer-written test suite with either an assertion, an exception, or both. © 2015 IEEE

Project	Tool	Assertions	Exceptions	Both
Chart	AgitarOne	64.3%	55.6%	100.0%
	EvoSuite	57.1%	77.8%	100.0%
	Randoop	57.1%	77.8%	100.0%
Closure	AgitarOne	18.7%	16.7%	25.0%
	EvoSuite	17.6%	30.0%	16.7%
	Randoop	3.3%	16.7%	8.3%
Lang	AgitarOne	31.0%	35.3%	50.0%
	EvoSuite	20.7%	29.4%	100.0%
	Randoop	6.9%	23.5%	0.0%
Math	AgitarOne	42.7%	70.0%	0.0%
	EvoSuite	56.0%	80.0%	0.0%
	Randoop	34.7%	50.0%	0.0%
Time	AgitarOne	30.0%	71.4%	0.0%
	EvoSuite	80.0%	42.9%	66.7%
	Randoop	40.0%	64.3%	66.7%

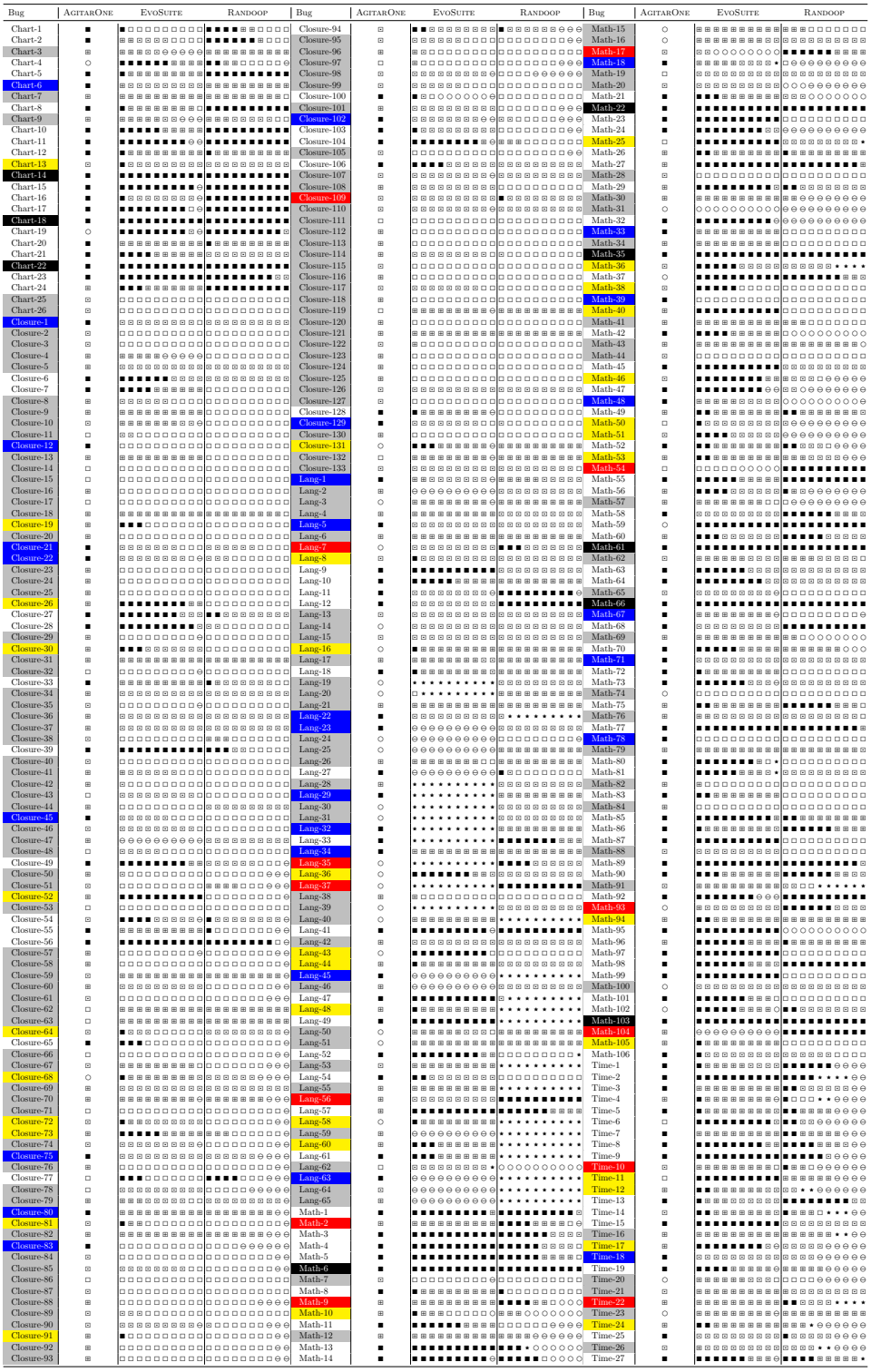
3.3.3 How Are the Bugs Found?

There are two important aspects to unit test generation, which are typically treated separately: Generating the code that constitutes the setup and exercises the target method, and generating the assertions that capture the expected behaviour (which in the case of our experiment, they were generated based on the behaviour of the correct/fixed version). Bugs are then found by a unit test if any of the assertions fail, or if an unexpected exception is thrown. Some automated test generation tools explicitly target unexpected exceptions [35], while others integrate techniques for efficient generation of assertions (e.g., the work of Fraser and Zeller [56]).

Overall, the generated test suites detected more bugs with an assertion (146) than with an exception (109). Note that 56 bugs were detected with both an assertion and an exception, by different tests. EvoSuite is the only tool that detected 5 bugs via timeout; we treat this as a case of exception in the following discussion. The right-hand side of Table 3.1 details the results and shows how effective assertions and exceptions are for each project.

Assuming that a developer-written test that reveals a bug is indicative of whether an assertion or an exception is required for that bug, we can determine if the tools are better at detecting bugs requiring an assertion or an exception.

Table 3.3 Summary of bug-finding results for each bug, tool, and execution. For each tool execution, if a compilable test suite was generated, a bug may have been detected (■), fully covered but not detected (⊞), partially covered but not detected (⊡), or not covered at all (□). Otherwise, the test suite was not compilable (⋆), resulted in configuration errors in our evaluation framework (⊖), or it was not even generated (○). Bugs that were either never found, always found, or found by only one tool, are highlighted with different colours (no tool, all tools, only AgitarOne, only EvoSuite, or only Randoop)



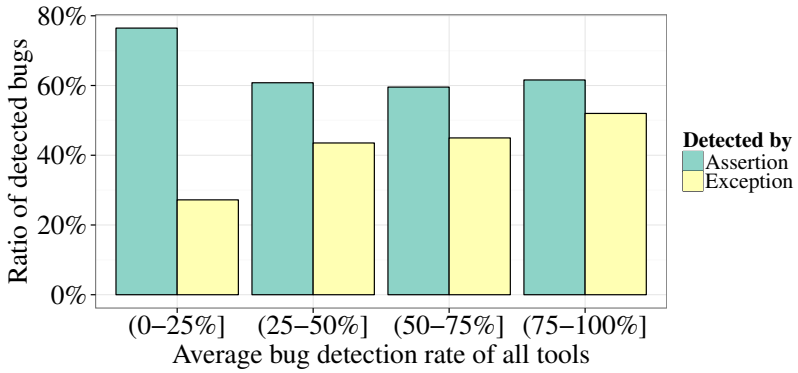


Fig. 3.2 Ratio of bugs that were detected by an assertion or an exception, grouped by the intervals of the average bug detection rate of all tools. The total number of bugs is 199 and 46, 77, 46, and 30, respectively for each interval. Note that a bug may be detected by both an assertion and an exception. © 2015 IEEE

Table 3.2 shows how many of the bugs (detected by the developer-written test suites with an assertion or an exception) were detected by the generated test suites. In the majority of cases, clearly more of the bugs that trigger an exception were found than of those that require an assertion. The main exception is EvoSuite for the Time project, where 80% of the bugs requiring an assertion were found, compared to only 42.9% of the bugs requiring an exception.

Figure 3.2 shows the ratio of bugs detected by assertion or exception for different intervals of the average bug detection rate of all tools. Note that the average bug detection rate is computed as the mean across tools rather than the mean across all test suites because the numbers of generated test suites differ for the tools. The plot suggests that hard to find bugs (i.e., bugs with a low average detection rate) are more often detected by an assertion than by an exception. Furthermore, bugs that are easier to find are more often detected by an exception than bugs that are harder to find.

More bugs were detected with a test assertion than with an exception (146 vs. 109), but the detection ratio is lower for bugs requiring assertions (37.34% vs. 49.4% avg. per tool).

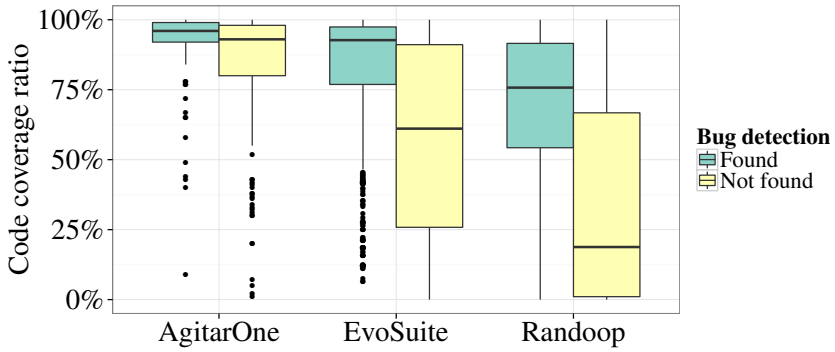


Fig. 3.3 Code coverage ratios for generated test suites that found a bug and generated test suites that did not. The differences are significant for all tools (Mann-Whitney U test, $p < 0.001$). © 2015 IEEE

Table 3.4 Bug coverage of test suites that did not detect the bug. For each project and tool, *Total* gives the number of bugs that were not always detected by all generated test suites. © 2015 IEEE

Project	Tool	Total	Not	Partially	Full
Chart	AgitarOne	7	28.6%	28.6%	42.9%
	EvoSuite	22	36.0%	18.6%	45.4%
	Randoop	15	29.8%	30.0%	40.2%
Closure	AgitarOne	47	44.7%	31.9%	23.4%
	EvoSuite	130	50.0%	30.7%	19.3%
	Randoop	133	73.2%	17.6%	9.2%
Lang	AgitarOne	30	66.7%	0.0%	33.3%
	EvoSuite	61	42.0%	23.7%	34.4%
	Randoop	61	37.2%	34.1%	28.7%
Math	AgitarOne	46	28.3%	21.7%	50.0%
	EvoSuite	83	23.4%	27.2%	49.3%
	Randoop	93	54.8%	25.4%	19.8%
Time	AgitarOne	11	27.3%	36.4%	36.4%
	EvoSuite	23	5.8%	27.4%	66.8%
	Randoop	27	52.1%	25.3%	22.6%

3.3.4 Are Bugs That Are Covered Usually Found?

Figure 3.3 compares the code coverage ratios of generated test suites that detected a bug and generated test suites that did not, clearly showing that code coverage matters when it comes to detecting faults. This is also confirmed by a strong correlation between code coverage and bug detection (Pearson correlation of 0.40 on average per tool).

However, a high code coverage ratio does not necessarily indicate that the bug was covered. Table 3.4 therefore reports the bug coverage (as described in Section 3.2.3.5) for all test suites that did *not* detect a bug. In general, fewer

than 50% of the undetected bugs are fully covered by the test suites. A bug that is fully covered but not detected is indicative of a problem in generating an assertion since a test suite very likely detects a fully covered bug, even without an assertion, if that bug raises an exception. A notable exception in terms of fully covered bugs is given by EvoSuite for the Time project, where 66.8% of the undetected bugs are fully covered. We surmise that in these cases EvoSuite removed assertions while checking for unstable tests. Although full bug coverage is not sufficient to trigger the bug (as discussed in Section 2.1.1.2), the tools struggled to achieve full bug coverage for the majority of bugs.

Of all test suites that did not reveal a fault, 46.8% did not fully cover it, and 26.6% did not even cover it partially.

3.4 How Can the Tools Be Improved?

Table 3.3 shows that the majority of the generated test suites did not detect the corresponding fault. The faults that were always detected by all generated test suites[6] are simple faults, such as a `NullPointerException` caused by a missing input validation, or easily executable and observable changes, such as Math-22:

```
1 public boolean isSupportLowerBoundInclusive() {
2 -    return true;
3 +    return false;
4  }
```

Note that in this and all following code snippets, '+' indicates an added line and '−' a removed line, in the fixed version.

However, most bugs are not this trivial. To gain insights on how to increase the fault detection rate of test generation tools, we investigated the challenges that prevent fault detection. To this end, we first looked at the 12 faults that no tool managed to cover (not even partially) and the 4 undetected faults that were fully covered. We then looked at the 76 faults that only one tool managed to detect – 28 by AgitarOne, 35 by EvoSuite, and 13 by Randoop; these

[6] Chart-{14, 18, 22}, and Math-{6, 22, 35, 61, 66, 103}

reveal strengths of a particular tool that others are missing. Finally, we studied the root causes for flaky and false-positive tests. The remainder of this section presents our findings.

3.4.1 Improving Coverage

A test has to cover a fault to detect it, and Figure 3.3 shows that test suites that detect a fault achieve significantly higher code coverage. We identified four challenges that inhibit test generation tools from achieving such high code coverage.

3.4.1.1 Creation of Complex Objects

Out of the 12 faults[7] that were not covered by any of the generated test suites, the majority (9) of them are from the project Closure. For these 9 faults, reaching the buggy code requires the creation of a complex data structure (e.g., a control flow graph). This is a highly complex and challenging task for an automated test generation tool since it requires a certain sequence of method calls prior to exercising the target method. This task is also demanding for human testers, as evident in Closure's developer-written tests that detect the fault: The tests create a complex string (i.e., program text) and re-use the compiler infrastructure to create an appropriate data structure. For an automated test generation tool, however, generating such a complex input string is as challenging as initializing a complex data structure. Consider the following fault (Closure-14):

```
for (Node finallyNode : cfa.finallyMap.get(parent)) {
-     cfa.createEdge(fromNode, Branch.UNCOND, finallyNode);
+     cfa.createEdge(fromNode, Branch.ON_EX, finallyNode);
}
```

Covering this fault requires constructing a control flow graph in advance, with an edge connecting two consecutive `finally` blocks. The developer-written test achieves this by constructing a control flow graph object from a complex input string:

[7] Closure-{14, 15, 17, 32, 53, 66, 71, 86, 111}, Lang-24, and Math-31

```
String src = "X:while(1){try{while(2){try{var a;break X;}" + "
    finally{}}}finally{}}";
ControlFlowGraph<Node> cfg = createCfg(src);
assertCrossEdge(cfg,Token.BLOCK,Token.BLOCK,Branch.ON_EX);
```

In contrast, no coverage guidance exists for generating such complex strings within an automated test generation tool.

Some bugs requiring the creation of complex objects were found; for example, for Closure-80, an AgitarOne test created a mocked instance of a particular `Node`, and for Closure-109, a Randoop test managed to create a valid control flow graph from a complex input string, reusing a string constant from a dependent class. Randoop also succeeded in detecting several faults[8] by indirectly testing the class under test through dependencies. In these cases, Randoop overcame the object creation problem by exploiting existing logic in classes that are clients of the class under test.

Some viable solutions (which are not implemented in the used tools) exist to address the generally acknowledged problem of complex objects [185]. For example, seeding objects observed at runtime [85], mining of common usage patterns of objects [55] to guide object creation, or carving of complex object states from system tests [41]. However, in the absence of example information the problem is unsolved.

3.4.1.2 String Optimization

Complex strings not only occur in the developer-written test suites for Closure, but are also a recurring pattern in the generated test suites for faults that are only detected by one tool. AgitarOne detected the string-related fault in Closure-155, and EvoSuite detected several faults[9] that require a specific input string. For example, consider Lang-16, a fault whose detection requires the satisfaction of the following condition:

```
if (str.startsWith("0x") || str.startsWith("-0x")
+       || str.startsWith("0X") || str.startsWith("-0X")) {
     return createInteger(str);
}
```

[8] Math-{2, 9, 54}, Time-{10, 22}, and Lang-56
[9] Lang-{16, 36, 44, 44, 58, 60}, Time-24, and Closure-73

EvoSuite generates strings using its genetic algorithm and seeded values [49], taken from string constants and runtime observations. The following test detects the fault above:

```
public void test085() throws Throwable {
    String string0 = "-0XeD";
    int int0 = NumberUtils.createNumber(string0);
    assertEquals((-237), int0);
}
```

Search-based tools are capable in principle of generating string inputs [4], but doing so with a search algorithm can take very long. Symbolic approaches using string solvers [59] or dedicated solvers for regular expressions [173] are generally restricted to fixed length strings. If an input grammar is known, then this can be used to generate test data more efficiently [22]. The results of web queries can also serve as useful test data [112]. Nevertheless, our experiments showed that state-of-the-art tools still struggle with string optimization.

3.4.1.3 Complex Conditions

Lang-24, a fault that no tool detected, is an example for a complex condition that needs to be satisfied to detect the fault:

```
if (chars[i] == 'l' || chars[i] == 'L') {
-    return foundDigit && !hasExp;
+    return foundDigit && !hasExp && !hasDecPoint;
}
```

Detecting this fault is challenging for two reasons: First, a randomly initialised character array (`chars`) is unlikely to satisfy the outer condition. Second, search-based tools like EvoSuite suffer from boolean flags such as `foundDigit`, `hasExp`, and `hasDecPoint`, which provide no guidance to the search. This problem of boolean flags is well known, and testability transformation [74] is generally accepted as solution. Note that DSE would not suffer from this problem [61, 58].

Lang-48 a fault that only EvoSuite detected, exemplifies a problem related to complex conditions involving subtyping:

```
public EqualsBuilder append(Object lhs, Object rhs) {
    ...
    Class lhsClass = lhs.getClass();
    if (!lhsClass.isArray()) {
-         isEquals = lhs.equals(rhs)
+         if (lhs instanceof java.math.BigDecimal) { ... }
+         else { isEquals = lhs.equals(rhs) }
    } ...
}
```

In principle, when creating an input value for a parameter of type `Object`, any class can be used. EvoSuite addresses this challenge by explicitly using classes that are used in casts or type comparisons, and therefore generates a test case that passes an object of type `BigDecimal` as argument to the `append` method.

3.4.1.4 Private Methods/Fields

Many of the faults in the Closure project, in particular those not detected by any tool, exist in private methods. This presents an additional challenge for an automated test generation tool, which usually tests using only the public interface of a class under test. For instance, in Closure-1, a simple change is introduced in a private method:

```
private void removeUnreferencedFunctionArgs(Scope fnScope) {
+   if (!removeGlobals) {
+     return;
+   }
Node function = fnScope.getRootNode();
...
```

It is, however, difficult to (indirectly) test this method due to the complex class hierarchy and data structures of this project.

AgitarOne tries to sidestep this problem by accessing private fields and methods. While it may not generally be desirable to explicitly call private methods, it enables AgitarOne to cover methods that are hard to reach through the public API of the class. Covering the `removeUnreferencedFunctionArgs`

method, AgitarOne triggers a `NullPointerException` by setting the value of the private field `removeGlobals` to `false`, and passing in `null` as `fnScope`:

```
RemoveUnusedVars removeUnusedVars = (RemoveUnusedVars)
    Mockingbird.getProxyObject(RemoveUnusedVars.class, true);
setPrivateField(removeUnusedVars, "removeGlobals", Boolean.FALSE);
Mockingbird.enterTestMode(RemoveUnusedVars.class);
callPrivateMethod("com.google.javascript.jscomp.RemoveUnusedVars", "
    removeUnreferencedFunctionArgs", new Class[] {Scope.class},
    removeUnusedVars, new Object[] {null});
```

Although the test *does* find the bug, it is unclear whether such a change could be triggered without modifying the state through private fields and methods. Similarly, AgitarOne detected Closure-{45, 83, 102} and Math-{18, 33, 78} by asserting the value of private fields using reflection. Whether or not accessing private methods and fields is a good approach is debatable but, as shown, it has the potential to reveal faults. However, it can cause false positives, as will be discussed in Section 3.4.4. This problem can only be overcome by improving test generation tools to achieve coverage of private methods fully through the public API.

3.4.2 Improving Propagation and Detection

Even if covered, a fault might not propagate or, if it does, the test oracle might not be able to detect the change in the outcome. Recall that a third of the undetected faults were fully covered and many more were partially covered (Table 3.4). This section details challenges in revealing those covered faults.

3.4.2.1 Propagation

Across all projects and test suites, we found five faults (Closure-{31, 70, 121} and Math-{12, 30}) that were always fully covered but never detected.

Unlike Closure-{31, 70, 121} and Math-30, Math-12 represents a unique case not directly related to propagation, where the fault is simply a forgotten implementation of the `Serializable` interface. This fault is trivially covered, but in order to detect it, a test would need to serialise an object of that class using an `ObjectOutputStream`.

Math-30 is an integer overflow error, which means that not only does the code need to be covered, but it also has to be executed with values that lead to an overflow. Closure-31, 70, and 121 have a potential influence on private members of the faulty class. However, these private members are complex objects, and a change to them can only be observed by involving the faulty object in further complex interactions, rather than simply writing an assertion on a return value of a public method. To some extent, this is the result of focusing on simple structural criteria such as branch coverage, rather than aiming to exercise more complex intra-class data flow dependencies [80].

3.4.2.2 Assertion Generation

Considering that more faults are detected with an exception, but the majority of faults require an assertion (Table 3.2), a key challenge for test generation tools is generating adequate assertions (i.e., test oracles).

Assertions are typically generated based on observations of the public API [186, 56, 124] during execution. However, our experiments revealed some particular cases where only AgitarOne was able to generate the appropriate assertions: For Chart-6, Closure-{12, 21, 22, 129}, and Math-48, AgitarOne detected the fault by asserting on the object state in the catch clause. In contrast, EvoSuite and Randoop only verify that an expected exception is thrown. For example, Closure-129 is a fault for which both the buggy and the fixed version throw a `NullPointerException`, but differ in *where* it is thrown. AgitarOne detected this change as follows:

```
1 try {
2     prepareAnnotations.visit(t, n, parent);
3     fail("Expected NullPointerException to be thrown");
4 } catch (NullPointerException ex) {
5     ...
6     assertThrownBy(PrepareAst.PrepareAnnotations.class, ex);
7 }
```

3.4.3 Flaky Tests

It is important for regression tests to be deterministic and to produce the same outcome in consecutive runs. Recall that we automatically removed 15.2% of flaky tests to achieve this goal. However, removing such a large portion of tests also means losing any additional coverage gained by such tests.

3.4.3.1 Environment Dependencies

Flaky tests are frequently caused by environmental dependencies of the software under test, such as the current time/date of the system. This problem is particularly frequent for Time, but also occurs in the other projects. For example, detecting the fault in Lang-8 requires a call to the method `format` of the `FastDatePrinter` class, which takes a `Calendar` as input – by default, a `Calendar` will be initialised to the current time.

EvoSuite addresses this problem by using a mocked version of the concrete implementations [14]. This means that if the program accesses the current time on the system, then EvoSuite provides a mocked time, so that any assertion that depends on the time value will deterministically pass or fail when executed at a later time. The following gives an example for a test, generated with EvoSuite, which uses a mocked version of the concrete class `GregorianCalendar`, `MockGregorianCalendar`:

```
1 String string0 = "Z,~jsZ/7'{p!wd";
2 int int0 = 0;
3 SimpleTimeZone simpleTimeZone0 = new SimpleTimeZone(int0, string0);
4 Locale locale0 = Locale.GERMAN;
5 String string1 = "*z";
6 FastDatePrinter fastDatePrinter0 = new FastDatePrinter(string1,
    simpleTimeZone0, locale0);
7 MockGregorianCalendar mockGregorianCalendar0 = new
    MockGregorianCalendar(locale0);
8 String string2 = fastDatePrinter0.format((Calendar)
    mockGregorianCalendar0);
9 assertEquals("*GMT", string2);
```

In contrast, RANDOOP does not use mocking, and for the 10 runs on the same fault, it generated 84% flaky tests. AGITARONE also applies mocking, but was not able to detect this fault, which additionally requires a specific constraint to hold: the time zone represented by the `SimpleTimeZone` instance (line 3) must differ from the time zone used in the `Calendar` instance (line 7).

The faults Time-12, Time-14, and Lang-65 pose similar challenges to the test generation tools. Note that the problem of environment dependencies is not restricted to objects generated explicitly by the test generator, as other classes may refer to the system time or other external resources directly. EVOSUITE tries to overcome this problem using bytecode instrumentation, such that the environment dependencies can be controlled when directly accessed by the code under test.

3.4.3.2 Static State

A local dependency on the static state of the system under test can also result in flaky tests, such that any changes to the state with one test can affect the outcome of the remaining tests. EVOSUITE explicitly tracks changes to static variables, and resets the static state before test execution [14], and as a result was the only tool to detect Time-11, where class `ZoneInfoCompiler` uses a static variable `cVerbose`. The issue of static state has been raised in the context of test generation previously [35], and has recently also been verified in the context of manually written test suites [201, 102, 21].

3.4.4 False Positives

During our evaluation, in particular during the validation of the test results, we encountered a number of false positives. The RANDOOP tests contained a few false positives due to non-deterministic failures unrelated to the fault. In particular, these tests caused an `IllegalStateException` by manipulating the threading behaviour in Closure. While these tests are technically flaky tests, their likelihood of failing is very low, explaining why they never failed on the original version when checking for flaky tests. The majority of false positives observed are caused by AGITARONE's access of *private fields/methods/classes* through

Java reflection, which breaks object-oriented principles such as *encapsulation*, and AgitarOne's use of *aggressive mocking*,

3.4.4.1 Accessing Private Fields/Methods

To maximize coverage, AgitarOne uses Java reflection to access the private API of the class under test. However, developers may add, remove, or change private fields or methods to improve code quality or optimize the existing implementation. These changes do not affect any client of the class under test as its public API remains unchanged. Detecting regressions that are purely related to the private API therefore increases the likelihood of false positive test results.

Closure-3 is an example of a change of a private method, including its signature:

```
- private boolean canInline() {
+ private boolean canInline(final Scope scope) {
      ...
+     case Token.NAME:
+     Var var = scope.getOwnSlot(input.getString());
+     if (var != null && var.getParentNode().isCatch()) {
+        return true;
+     } ...
```

While AgitarOne generated a test that detects this change with a `NoSuchMethodException`, the test does not fail because of the root cause — that is, the test does not cover the `Token.NAME` case and would therefore pass if the method signature would remain unchanged. Overall, AgitarOne suffered 26 (12%) false positives caused by the use of reflection.

3.4.4.2 Aggressive Mocking

Another example for breaking encapsulation is AgitarOne's *aggressive mocking*, which monitors and asserts on the internal state (e.g., the order of method calls) of the class under test, rather than testing the class on what its public method returns, and on the side effects it has on its input parameters once its methods have completed to execute. How such input objects are manipulated is an

internal detail that is not part of the public interface specification. As such, it can change without modifying the semantics of the methods. Consider the following example:

```
1 public int sum(Foo foo){
2 -    return foo.getX() + foo.getY();
3 +    return foo.getY() + foo.getX();
4 }
```

If `getY` and `getX` are pure (i.e., no side effects), then that function can be refactored as shown, and the order in which the function `sum` calls the methods in `Foo` is irrelevant. However, an aggressive mocking strategy would check the order in which the mocks are used, and fail if a different one is encountered.

For example, consider Closure-5, where the developers added a check to handle the special case of a deleted property:

```
1 +   if (gramps.isDelProp()) {
2 +       return false;
3 +   }
```

A valid way to detect this bug would be by using an assertion on the return value. However, with AgitarOne's aggressive mocking, adding such a method call would lead to a failure without even evaluating `isDelProp`, as the method is unexpected and thus triggers a `TestException`, indicating an error originating in AgitarOne's mocking framework. Similarly, deleting the method call would trigger a `TestException` for any subsequent method call on the same object – in neither case would the actual return value be considered.

For 67 faults, AgitarOne generated a test that failed with a `TestException` and a failure message referring to an "unexpected method". Such a mocking error may be a true positive if there is a specification on the order of method calls when communicating with external classes/resources. However, manual investigation of the bug descriptions of these 67 faults suggests that none of them are related to such a specification. We therefore consider each test that fails due to a `TestException` as a false positive caused by mocking.

Applying this interpretation, 31% of all test suites generated by AgitarOne suffered from false positives due to the use of aggressive mocking (10 of these test suites additionally include false positives due to accessing private methods/fields

with reflection). However, note that this may be an over-approximation because a mocking exception could be caused by a state change induced by the bug.

3.5 Related Work

The most closely related work to ours is that of Xiao et al. [185], who identify two main problems when aiming to achieve high code coverage with generated tests, *external method calls* and *complex object creation*. The external method call problem is related to the environment dependencies issue (Section 3.4.3.1) that we saw in terms of relation to date and time. Just like EvoSuite uses mocking to overcome dependencies on time or files [14], this is also possible, for example, for database applications [165]. We also saw several instances of the complex object creation problem in the faults we analysed. Xiao et al. [185] propose a collaborative approach between the tools and the developers, such that the underlying coverage problems is reported back to the developer to provide further guidance to the tool. The aim of our analysis is to identify problems that prevent automated test generation tools from finding faults rather than just covering code, with the hope to improve these tools in the future to find more faults.

There have been studies of the effectiveness of various software defect detection techniques, e.g., [183, 146, 118]. Generally, these studies showed that different techniques are complementary and dependent on the underlying faults; we saw similar results in our study: The individual performance of each of the tools in our study lies beneath the potential of combining all the tools.

3.6 Conclusions

Automated unit test generation tools are typically evaluated in terms of the code coverage that they can achieve on open source software projects. This chapter contributes a systematic study of the fault detection potential of the generated test suites, using three state-of-the-art test generation tools and the Defects4J dataset. The results show that 1) The test generation tools find 55.7% of faults, but no tool alone finds more than 40.6% of faults. 2) Despite the tools

finding more than half the faults, only 19.9% of all the test suites generated as part of our experiments detected the fault – that is, we cannot be confident that tools will always detect the fault they once revealed 3) Achieving code coverage remains a problem: 16.2% of the faults were never even executed by the generated tests, and 26.6% of the test suites that did not reveal a fault did not even cover the fault partially; however, 4) 63.3% of the non-found faults *were* covered by automatically generated tests at least once, suggesting problems that go beyond code coverage (i.e., state-infection and propagation are not tackled by the tools). Another specific issue is that 15.2% of all tests were flaky.

In order to guide future research on automated unit test generation, we investigated the challenges that need to be addressed in order to improve fault detection, and our qualitative analysis of difficult to find faults reveals specific challenges that prevent tools from achieving the required code coverage and generation of test oracles. Some of these challenges however relate to the fact that none of the techniques have been designed from the ground up to detect faults. In fact, a large number of automated unit test generation techniques purely target code-coverage metrics (i.e., generate tests that can cover as much of the CUT as possible). In the following chapter we look at designing such a technique that is targeted for regression testing: given two versions of a program, generate tests that reveals the difference between the two programs, as we described earlier in Chapter 1, and use the framework presented in this chapter to evaluate it on real faults.

Chapter 4

Differential Testing Using a Search-Based Approach

This chapter extends work undertaken during this PhD by the author, which has been published elsewhere [155, 153].

4.1 Introduction

As discussed and evaluated in the earlier chapters, automated test generation tools can help developers by creating tests that can detect regression faults early. An example of such regression faults is when a developer makes a change to a program, while at the same time unintentionally breaks an original functionality of the program. If the developer in this scenario does not already have access to a test suite which can reveal the regression, he/she may assume the version to be correct and make it available to the users. Therefore, having access to tests which can validate the correct functionality of the program can be beneficial to the developers to detect such regressions early.

However, as we reported earlier in the previous chapters, current techniques face challenges such as: they do not target test generation specifically for regression testing, they aim to reach the fault but do not propagate it to the output (as evident in Chapter 3), they require existing tests, or they suffer scalability limitations. Moreover, API changes across different versions of a

program may lead to large number of false-positive failures and/or flaky tests. To address the previous shortcomings, in this chapter we propose and evaluate a technique we name EvoSuiteR for automatically generating regression test suites that a) aims to simultaneously reach and propagate changes between two versions of a program, and b) does not require existing tests. Our approach takes two versions of a class under test as input, and uses a search-based algorithm to optimise for multiple objectives at the same time. In this approach, we use Genetic Algorithms (GAs) as a search technique which relies on the biological theory of evolution by Darwin [81].

In short, similar to search-based algorithms reviewed previously in Section 2.1.5.3, the algorithm works by initially generating a population of random individuals (test suites). Throughout each stage of the search, individuals in the population are evaluated for different metrics by the fitness function, and the better performing individuals are selected for the next generation. In the course of the search, the individuals may also be mutated and crossed over to increase the diversity among the population. The search continues until either a solution is found or the search budget is depleted. After the search is finished, the tests are compiled again and are executed against both versions of the software. At this stage (or during the search), if the tests observe any different execution results, assertions validating the original behaviour of the program are added. These steps are discussed in detail over the following section. A thorough evaluation of the work then follows in Section 4.3. The contributions of this chapter are as follows:

- A search-based technique for automatically generating regression unit tests to reveal differences across two versions of a program
- A multi-objective fitness function for GA-based differential testing
- Empirical evaluation of the search-based differential testing technique on finding real faults

4.2 Search-based Regression Test Generation

4.2.1 Representation and Fitness Function

Since our approach requires automated test generation, EvoSuite$\mathbb{R}$ was developed on top of the state-of-the-art test generation platform EvoSuite. While the EvoSuite platform is able to generate test suites for a single class with the aim of achieving high structural coverage, it was not built for testing and running multiple versions of a piece of software simultaneously. This required making changes to the platform's internal structure to support additional representations of the system under test.

As mentioned earlier, EvoSuite$\mathbb{R}$ relies on a genetic algorithm (GA), where a population of individual chromosomes evolve over time. Initially, based on the structure of the system under test, a population of random *regression chromosomes* is generated. Conceptually, the representation of these chromosomes has not been changed to that of EvoSuite, except with respect to their ability to run on two versions of a class simultaneously, while keeping their respective execution results. When these chromosomes are evaluated against the fitness function, a score named *fitness value* is associated with them, which measures how far the candidate solution is from an optimal solution. In each generation, the fitness of all individuals in the population is calculated, which is then used to determine which individuals will find their way to the next generation. An individual with a better fitness value is more likely to be included in the future generation, or to be used in reproduction.

Similar to EvoSuite, the evolution happens using the search functions mutation and crossover (as described in Section 2.1.5 and [47]). When evolving regression chromosomes, as new generations of individuals are created, before being evaluated by the fitness function, they are executed on both versions of the software, and the execution traces are stored for each individual. These execution traces are then given to the fitness function to calculate their fitness value. Afterwards, the fitness value is attached to the individuals. To avoid re-executing the same chromosomes twice (to increase performance), we also cache these values internally for the given chromosomes. As such, in the event

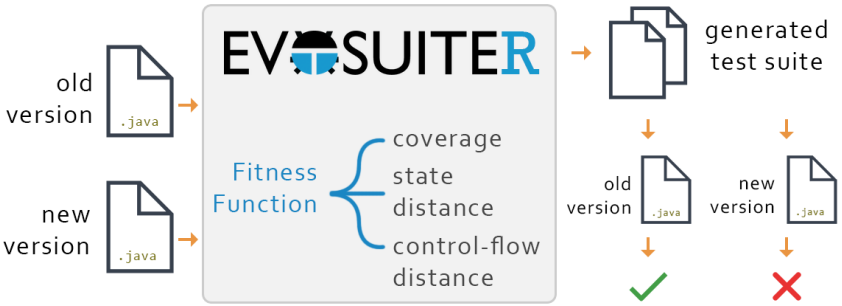


Fig. 4.1 Overview of the approach. Given two versions of a Java class, EvoSuiteR aims to generate a test suite which passes on one version of the class, and fails on the other.

of an individual remaining unchanged, EvoSuiteR will re-use the fitness value without having to re-calculate the fitness.

The fitness function aims to guide the search towards revealing differences between the two versions of the program. The problem of reaching potential regression faults in the program, and propagating the state changes to the output comprises of many different sub-problems. As mentioned in Section 2.1, according to the PIE model, to detect a fault it is required for the fault to be first executed, and execution alone does not necessarily result in propagating of the fault to an observable output. Given that the objective of our approach is to execute the regression faults and propagating any differences in the behaviour to an observable difference, we propose a multi-objective fitness function, which simultaneously guides individuals towards multiple goals. Our multi-objective fitness function allows us to optimise towards maximising structural coverage, while at the same time maximising the differences in the state and behaviour between the two versions. An overview of the approach is shown in Figure 4.1.

The metrics used in the fitness function are presented in the following sections.

4.2.1.1 Structural Coverage

Before being able to propagate the state resulted from executing regression faults to an observable difference in the output, it is important to reach and execute the faulty code, so as to solve the reachability aspect of regression testing. Particularly, in Chapter 3 we saw that reachability is not a solved issue for test generation tools, especially considering the fact that the tools only aim to cover

one version of the program. Therefore, our proposed structural coverage metric for EvoSuiteℝ aims to cover the structure of both versions of the program.

Considering that any part of the class might have been affected by the change, or may be necessary in order to propagate the infected state to the output, we aim to cover the whole structure of the class under test, and not only the changed area of code. To guide the search towards achieving high code coverage, we measure and use the level of coverage achieved as our first fitness objective. Therefore, the higher the level of coverage is, the fitter is the individual as a result.

Structural coverage in EvoSuiteℝ is measured by running the generated test suite on both the old and new versions of the software, and measures the level of branch coverage achieved. The values are then added together to create the final sum. The level of coverage is increased during the search by maximizing the value of this measurement.

Let C be the class under test, with two versions of `versionA` and `versionB`, the coverage parameter can be achieved using the function below. Each individual coverage value is taken using the measurements explained in Section 2.1.5.3, which also relies on the distance of the uncovered branches towards being covered.

$$\text{Coverage}(C) = \text{coverage}(versionA) + \text{coverage}(versionB)$$

4.2.1.2 State Difference

It is expected that while executing a regression fault, if the two versions of the software are reaching different states (either internally inside the objects or externally as observable return values), increasing such state difference between the two versions of a software would raise the chance of the fault to propagate to an observable output (i.e., the public API of the class) – we also refer to this metric as *state distance*. Therefore, the next measurement used in calculating the fitness is the observable difference between the behaviour of the two versions of the program. The level of state difference between the two versions of the software is increased by maximising the value of this measurement. The intuition of this measurement is similar to previous work, where difference between return values were calculated [56].

To measure this, at a high level, the two versions of the program are instrumented and the traces of execution are logged. Later, during the execution of each chromosome (which is a test suite) on each version of the program, pairs of execution traces for the two versions of the program are compared. State difference is measured by comparing the state of the test suite objects between the pairs of execution traces. To compare the objects, first all public and private objects are recursively inspected using Java Reflection and their values are recorded. Then, the numerical object distance [33] is calculated.

To compute a numerical value for the comparisons, for each test case, for each different type of object (i.e., based on the Class of the object) the differences are compared, and the highest measured difference is stored. To compare the differences, the resulting state after executing each test statement is stored, and then the states are respectively (respective to the test statement) compared across the two versions of the programs. Afterwards, the sum of all the highest differences per Classname are added together (sum) for each test case. And finally the mean of the values is calculated. The reason we use the mean distance value between the test-cases is that taking a sum would incentivise an increase in the number of test cases, and taking the maximum would result in the sub-optimal test cases being ignored, therefore resulting in a lower diversity among the solutions. Moreover, the reason for collecting and comparing distances after each statement execution – as opposed to at the end of the test case – is to avoid cases of fault-masking (see Section 2.1.1.2). A formal definition of this measurement is provided below.

To achieve the final state difference metric, first, the maximized distance between all the objects of the same class type is taken (denoted as `Cmax`), and added together for the class under test (denoted as `C`); and afterwards divided by the number of tests (denoted as `numTests`). Secondly, considering that our final fitness function is minimizing, and we are aiming to maximize the state difference, we use the inverse of the state difference. To keep a similar weighting compared to coverage and other metrics, we multiply the inverse of distance by the maximum value the coverage metric can achieve. In the following formulas, `O` and `R` respectively refer to the original (i.e., pre-change) and regression (i.e., post-change) versions of the program under test.

$$\text{distance}(C) = \frac{\sum_{i=1}^{m}(\texttt{Cmax}_{j=1}^{n}(|\texttt{objO} - \texttt{objR}|))}{\texttt{numTests}}$$

$$\text{maxCoverage}(C) = \texttt{maxCoverageO} + \texttt{maxCoverageR}$$

$$\text{stateDifference}(C) = \frac{1}{(1 + \text{distance}(C))) * (\text{maxCoverage}(C)}$$

State Difference in Action

To illustrate the effect of state difference (as measured in Section 4.2.1.2), this section demonstrates how state difference works using a non-trivial example. In this example, simply reaching and executing the changed code (which in our scenario contains a regression fault) modifies the state of the program internally, however, it does not result in immediate propagation of the internal change to an observable difference in the output between the two versions of the program. Instead, a certain method call has to be called a number of times before the different states of the program can propagate to the output.

The class *CreditCard*, as shown in Figure 4.2, considers a scenario in which a credit card company allows a customer to withdraw as long as their minimum repayment reaches a certain amount. The minimum payment is calculated as a fixed amount that increases based on the number of withdrawals. Over time, the credit card company decides to double the minimum repayment rate, however the developers forget to respectively raise the withdrawal limit (as in the `canWithdraw()` method). As a result, the change made by the developers creates a regression in the original functionality of the program, such that when the `withdraw()` method is called between 21 and 40 times consecutively, the program behaves differently compared to the old version of the program.

The challenge with detecting such a regression fault is that the fault does not occur until the `withdraw()` method is called for more than 20 times. The state-difference measurement tries to address such problems by trying to increase the chance of the internal state difference propagating to the output, by maximizing the state difference between the two versions of the program. Table 4.1 illustrates what happens internally during the execution of the *withdraw()* method, and

```
1  public class CreditCard {
2
3     private int minRepayment = 0;
4     private int balance = 0;
5
6     public boolean withdraw(int amount) {
7        if(canWithdraw() == false)
8           return false;
9
10       minRepayment+= 5  * 2 ;
11       balance += amount;
12       return true;
13    }
14
15    public boolean canWithdraw() {
16       return (minRepayment <= 200);
17    }
18 }
```

Fig. 4.2 Credit Card example class. A change is made to the class to increase the amount of minimum repayment after each withdraw. The highlighted area of code indicates the change that is applied to the class. In the *CreditCard* class, the highlighted change will not immediately affect the program behaviour. However, after 21 calls to `withdraw()`, `canWithdraw()` would return `false`, resulting in a difference in the behaviour between the two programs.

how the state-difference measurement can guide the search towards propagating the fault.

In order to show how our technique captures such state differences from test suites generated during the search, we present below an investigation of it in the context of the earlier example class `CreditCard`.

Tracking the state of test objects: Our state difference implementation starts by looking at the generated test cases in the population. Consider Figure 4.3 which shows two sample test cases generated for the `CreditCard` class. In the context of these test cases, there are three objects in the scope of the test cases which we keep track of:

- Variable `c`: An object of type `CreditCard`, containing two private fields: `minRepayment` and `balance`.

Table 4.1 The variable `minRepayment` is changed internally every time the method `withdraw()` is called. However, the bug is not found only until after 21 consecutive withdrawals.

`withdraw()` called for	Old Version	New Version	Bug Found?
1 time	`minRepayment` is 5	`minRepayment` is 10	✘
2 times	`minRepayment` is 10	`minRepayment` is 20	✘
3 times	`minRepayment` is 15	`minRepayment` is 30	✘
	⋮		
21 times	`minRepayment` is 105	`minRepayment` is 210	✔

- Variable `canWithdraw`: A primitive boolean value.
- Output result of the `withdraw()` method – which is discarded and not allocated to a any variables. A primitive boolean value, that we also keep track of and internally associate with a variable.

Recording and comparing the values: During the search, to evaluate the fitness of the individual chromosomes (i.e., test suites) in the population, EvoSuiteR executes the test suites on both versions of the software and stores the values of all objects in the scope of each test (as illustrated earlier) for both versions of the software. The values are then compared using the formula presented earlier in this section to produce the final fitness value.

In the context of our examples, all values of type `boolean` and `CreditCard` in the scope are compared, and the highest difference for each type is stored. As mentioned earlier, complex objects such as `CreditCard` are compared using their numerical object distance (which in turn uses the numerical distance of the internal fields within the objects), and for each object type, the object distance is normalised between 0 and 1, with 1 indicating the highest difference possible and 0 indicating the objects to be identical. For example, for a pair of boolean

```
1   @Test
2   public void Test1(){
3       CreditCard c = new CreditCard();
4       c.withdraw(150);
5       c.withdraw(350);
6       c.withdraw(50);
7       c.withdraw(100);
8       boolean canWithdraw = c.canWithdraw();
9   }
10
11  @Test
12  public void Test2(){
13      CreditCard c = new CreditCard();
14      c.withdraw(50);
15      c.withdraw(100);
16      boolean canWithdraw = c.canWithdraw();
17  }
```

Fig. 4.3 Example test suite for the `CreditCard` class, containing two test cases, created internally in EvoSuite to be used for measuring fitness values.

primitive values (`true`, `false`), the distance is 1, and for the tuple (`true`, `true`) the distance is 0.

Although the test cases in our example are both unable to detect the fault, the object `c` which internally has the field `minRepayment` will have different values across the two versions during each execution of the `withdraw()` method as demonstrated in Table 4.1. Therefore, to calculate the state-difference between the two test cases after executing them on both versions of the program, initially, the `c` object is compared. That is, the maximum observed distance value of `minRepayment` will be summed with the maximum observed distance value of `balance`, normalised into one value. Then, the normalised distance value of `c` is added to the normalised distance value of the other class variables `canWithdraw`. Finally, the result is divided by *two*, which is the number of test cases in the test suite. Therefore, according to the equation presented earlier, the distance is calculated as follows:

$$\text{distance}(C) = \frac{|20-40| + |10-20|}{2} = 15$$

4.2.1.3 Control-flow Distance

In order to further increase the chance of state infection, we identified another measurement to better guide the search. Generally during the execution, if a test case results in a diversion at a particular branch between different versions of a program, it raises the chance of the two programs to reach different states due to the change in the control flow. A *diversion* means that for a given branching condition, in one version of a program the `true` branch is executed (i.e., the condition evaluates to true), and in the other version of the program, the `false` branch is executed (i.e., the condition evaluates to false). Since each difference in the control flow during execution may contribute towards propagation, we aim to incentivise any diversion in the control flow at all branches. Thus, we implemented a *control-flow distance* measure, which works at branch-level, and measures the distance of the branch to diverge across the two versions. In this measurement, the higher the distance value, the further the branches are from diverging. As a result, minimizing this measurement brings the branches closer to the point of diversion.

Control-flow distance is composed of two separate values: a) The comparison of the branch distance values across the two version of the program, and b) the actual branch distance values. The first value is computed by measuring the level of difference for each branch between the two versions of the program. To obtain this value, for each branch, the distance of the branch to becoming `true` or `false` is first calculated and then compared across the two versions of the program (after executing a test case). The minimum difference for each branch across the whole test suite is selected, normalised and then value of all branches are summed together. The second value of control-flow distance is the minimum raw distance of each branch to true/false across the test suite, which is calculated for each branch, summed together and then normalised.

The first control-flow distance value aims to differentiate the branch distance values across the two versions of the program. Maximising this difference increases the chance of branch diversion across the two versions of the program. By minimizing the branch distance values themselves (the second value), the branches will move closer to the point of diversion. As a result, the combination of the two measurements will increase the difference of the branch distance across

the two versions while at the same time bringing the branches to the point of diversion. By minimizing the control flow distance, we will be maximizing the difference between the two branches, while at the same time minimizing the current distance to `true` / `false` at the branch. Therefore, if branches have any difference at the point of divergence, the control flow would diverge between the two versions of the program.

Let `TrueDist0` and `TrueDistR` respectively be the distance of branch to `true` (i.e., distance of the branch to becoming `true`) for the original (pre-change) and regression (post-change) class under test. And let `FalseDist0` and `FalseDistR` be their `false` distance counterparts. ω calculates the current value of the branch (denoted by `B`), and takes the mean between the two versions. The control flow distance for the class under test `C`, is then calculated by summing the minimum inverse of branch distance (in order to maximize the branch distance) added to the current branch value. Since the main fitness function is minimizing, by maximizing the control flow distance, we aim to maximize the difference of branch distance between the two versions, while aiming to lower the branch distance value. A diversion is expected if there is a difference between two versions, while the actual branch distance values approach zero.

$$\text{distance}(C, B) = |\texttt{TrueDist0} - \texttt{TrueDistR}| + |\texttt{FalseDist0} - \texttt{FalseDistR}|$$

$$\omega(C, B) = \frac{\texttt{TrueDistR} + \texttt{FalseDistR} + \texttt{TrueDist0} + \texttt{FalseDist0}}{2}$$

$$\text{cfgDistance}(C) = \sum_{i=1}^{m}(min_{j=1}^{m}(2 \times (1 - \text{distance}(c_j, b_i)) + \omega(c_j, b_i)))$$

Limitations of Measuring Control-Flow Distance

This metric however suffers several limitations. First, the branch distance value works best on conditions relying on numerical comparisons, and so boolean conditions such as `if(var==true)` or `if(foo(var))` provide no guidance towards causing a diversion between the two version of the program (more on this in Chapter 5). Secondly, since we compute this measurement for all branches,

classes with large number of branches can impose a high overhead on our technique. Thirdly, if the control-flow-graph (CFG) of the class under test has been modified across the two versions of the program (e.g., one version of the two programs contains more or different branches than the other), there may not be a one-to-one matching of the branches. Although such one-to-one mapping of branches across the two versions of the program can be approximated heuristically using techniques such as JDiff [10], several challenges remain: a) automatically inferring a mapping between two different versions of a program may be impossible in some cases, b) our internal evaluation of existing techniques such as JDiff revealed significant inaccuracies with the mapping; an incorrect mapping can exacerbate the problem by misguiding the search.

The latter limitation is in particular the most challenging. While much research has been done on creating matchings of program elements across two versions of a program, all existing work come with certain limitations [92]. These limitations are mainly due to the fact that a change to a program may ultimately turn it into an arbitrarily different program, which cannot be mapped heuristically to the previous version. Due to these challenges, some researchers even resorted to manual mapping of program elements when an incorrect mapping would negatively affect the outcome of the technique (e.g., [125]). In the case of our approach, although an incorrect mapping can result in misguiding the search when measuring control-flow distance, it does not mean that use of control-flow distance measurement is not practical. That is, the measurement can be applied when there are no changes in the branching structure across the two programs.

4.2.1.4 Exception Distance

Changes made to software may also alter the exceptional behaviour of the program. Therefore, different exceptions being thrown across the two versions of the program indicates a difference in the behaviour of the software, which can be due to underlying differences between the two versions. By increasing the number of different exceptions thrown, we expect to increase the likelihood of the change being propagated to the output.

This measurement is achieved by measuring and comparing the number of thrown exceptions during the execution of both versions of the software. Since the

main fitness function is minimizing, while we are aiming to maximize exception difference, the inverse of the value is added to the final fitness value.

$$\text{excptDis}(C) = 1/(1 + (|exceptionsThrownO - exceptionsThrownR|))$$

Given that different exceptional behaviour of a program immediately indicates an observable difference in the behaviour of the two versions of the program, we consider this measurement a *minor* metric in our fitness function. Therefore, since the objective of this work is to guide the search towards propagating changes to the output, and exception distance does not provide such guidance and only ensures that solutions with different exceptional behaviour have a better fitness value, we do not evaluate this metric as a separate measurement and include it during all experiments involving the *GA*.

4.2.1.5 Secondary Objective

During the search process, the length of the test suite and test cases will continue to increase. However, it is deemed to be undesirable to keep a longer test suite if it results in the same fitness value as a shorter test suite. Moreover, longer test suites increase the test execution time, in addition to being less convenient for the developers to read. Therefore, during the search, shorter test suites are preferred over longer solutions with identical fitness values.

4.2.1.6 Summary of Fitness Function

The fitness function overall combines the following measurements: Structural Coverage, State Difference, Control-Flow Distance[1], and Exception Distance. Moreover, between two individuals with the same fitness value, the one with the smaller test suite size is preferred. The overall minimizing fitness function is provided below:

$$\texttt{Fitness}(C) = \text{Coverage}(C) + \text{stateDifference}(C) + \text{cfgDistance}(C) + \text{excptDis}(C)$$

[1]If the branching structure between the versions of programs are identical

4.2.2 Generating Assertions

After the search finishes, EvoSuiteR selects the test suite with the best fitness value from the population, and analyses the test suite on both versions. At this stage, the test statements will be executed on both versions of the class under test, and if the test can reveal a difference in the output between the two versions, EvoSuiteR adds assertions to the test based on the original version of the program. This means that the generated test suite passed on the original version of the program, while one or more test cases may fail on the modified version of the program. This indicates a difference in the behaviour to the developer, and they can use this test to investigate whether or not the failure is due to a regression [56].

The assertion generation is based on observed return values from the test statements. To process the observed values and generate different types of assertions for the program, several different assertion generators observe the outputs during the execution, and process them due to their specific requirements as below (notice that we named the following categories based the type of assertion that more closely reflect JUnit assertion types):

- **Primitive assertions**: Method calls' return values of primitive type in Java (e.g., integers, doubles, floats, etc.) are compared against each other, and in case of any difference, an assertion for the original value is added to the test suite.
- **Same assertions**: Objects and values that are *already* in the test cases are compared against each other in the same test case, and if they are exactly the same in one version, but differ in the other version of the class under test, an assertion is be added.
- **Null assertions**: All object variables in the test suite are checked against null values. If a variable is null in one version, but it is not for the other version, a null assertion is added.
- **Inspector assertions**: While for primitive values it is easy to compare their equality, for objects a simple object comparison would only compare their reference pointer values. Inspector assertions compare the resulting values of executing all publicly available methods for the object variable, in addition to comparing the values of the objects' publicly available fields.

If method executions' return values or fields' values differ across the two versions, an assertion is added.

- **Comparison assertions**: If the comparison value of two variables differ from each other between the two versions of a class, compare assertions are added. For instance, if variable a is bigger than b, in one version, but it is smaller in the other version of a class, the assertion will be added. For non-primitive objects, if the objects have available comparators, the *compareTo()* value is considered.
- **Exception assertions**: If an exception is thrown in the original version that is not thrown any more in the modified version, an assertion is generated such that it expects the original exception to be thrown. In case of the reverse of this scenario happening – that is, an exception is thrown in the modified version which is not thrown in the original version – a comment is added to the test suite indicating that the test will fail on the modified version due to the exception, while it passes on the original version since no exceptions are thrown.

While this step is essential to the regression test generation process, it is possible that during the search a candidate solution can already detect an observable difference in the output between the two versions of the program. Thus, it may not be necessary to wait until after the search to check whether a candidate solution has been identified. As a result, on EvoSuiteℝ, the assertion generation step is performed after every generation of the *GA*, and the search stops as soon as a candidate solution is found that reveals a behavioural difference between the two versions of the program.

4.2.3 Isolation of Changes

A main challenge towards the adoption of any automated test generation technique by the industry is the overhead of time and effort required by developers to utilise the software testing technique. That is, even if there exists a tool that can successfully generate tests which reveal all behavioural differences, if inspection and maintenance of these tests is time consuming and so increasing the backlog, it is less likely to be adopted by industry. To address this concern, we apply a

post-processing step to isolate the changes and report a unique set of actionable tests in the generated test suite.

Our solution works such that once the search is finished and a test a suite is generated, we minimise the generated test suite for the sole purpose of differential testing. To achieve this, we take the following steps: first, we execute each test case of the generated test suite and remove all tests which do not satisfy our objective of differential testing – that is, the test should pass on one version and fail on the other one. Second, we collect the unique set of identified changes based on a set of multiple variables such as the type of failure, the type of assertion (if failure is due to an assertion failure), the method-call and/or the class-type of the variable involved. After this, all duplicate assertions are removed. If any of the resulting tests no longer reveal a difference, they are removed as well. Third, we remove all test statements that do not affect the outcome of the test. This is done by repeatedly removing each statement of a test case and validating whether the test still reveals a behavioural difference. Statements that do not contribute to the outcome of the test are then removed. Finally, we remove any remaining test cases that are no longer failing after applying the third step.

This approach significantly reduces the length of the test suite (e.g., over 90% reduction in test suite length) while at the same time removing duplicate failures. Although it is possible that some of the remaining failures can still be due to the same underlying changes in the code, we believe that this can be beneficial to the developers since they can see different types of side-effects of their changes.

4.3 Evaluation of the Search Objectives

To evaluate the approach presented in Section 4.2, initially, we evaluated the effectiveness of the approach on an example program with a complex issue, such as Figure 4.3. In our initial study, the multi-objective fitness function of EvoSuite$\mathbb{R}$ was found to be significantly more effective compared to using branch coverage as the only guidance metric; while the currently available state-of-the-art tool for regression test generation completely failed to identify the fault [155].

As a next step, we conducted a large empirical study to measure the effectiveness of the approach and individual measurements on real-world programs. In this section, first we present our research objectives (i.e., research questions), then we outline the test generation techniques we want to evaluate and the subjects used for the experiments. In Section 4.3 we describe the applied methodology to answer our research questions, and finally in Section 4.3.8 we present the results.

4.3.1 Research Questions

In order to further evaluate our approach, we aim to answer the following questions:

- ***RQ4.1.*** Which measurement is the most effective for propagating changes?
- ***RQ4.2.*** How effective is EvoSuiteR at detecting real faults?
- ***RQ4.3.*** How does EvoSuiteR compare with the state-of-the-art test generation techniques?

RQ4.1 aims to find the best performing measurement in guiding the search towards a test suite that is able to propagate the regression faults. In a nutshell, it aims to answer whether the multi-objective fitness function is indeed more effective than using the individual metrics such as branch coverage, state difference, or control-flow-distance. To answer this question, we look at the effectiveness of individual metrics in comparison to the combined fitness function.

While EvoSuiteR proposes a systematic approach to generating test cases that can propagate software changes, it comes at a significant processing cost. RQ4.2 aims to find out whether in a controlled experiment in which only the search algorithm differs, using our proposed GA can outperform a random search algorithm, given the same search budget. Therefore, to answer this question, we built a random test generation tool on top EvoSuiteR. In this technique, instead of using a genetic algorithm to search for the optimal solution, it constantly generates a random test case, runs it on both versions of the software, and compares their execution traces for differences. A similar approach is used by the current state of the art regression test generation tool BERT, however in

Table 4.2 Numbers of bugs in each project in Defects4J, along with the number of bugs used in this study.
*All experiments excluding the experiments involving control-flow-distance.

Project	**Bugs**		
	Total	**All Experiments***	**Control-flow-distance**
Apache Commons Lang	65	65	5
Apache Commons Math	106	96	25
Google Closure Compiler	133	120	7
JFreeChart	26	24	9
Joda-Time	27	22	4
Total	357	327	50

their original implementation the test generation is performed by Randoop rather than EvoSuite [123].

With RQ4.3 we aim to understand how well the technique compares to the state-of-the-art test generation techniques. To answer this, we compare the effectiveness of our technique with 3 popular test generation tools EvoSuite, AgitarOne, and Randoop.

4.3.2 Subject Programs

In order to answer our research questions and to evaluate our technique's effectiveness in detecting real regression faults in practice, we needed a set of subjects for which we would have access to the followings: a) Two versions of the program for each subject are available, b) There exists a change between the two versions, c) The change is not semantically equivalent, d) The change represents a real fault, and e) Preferably, state-of-the-art techniques have already been evaluated on those subjects. The Defects4J repository of real faults [89] contains 357 real faults taken from 5 open source repositories, and satisfies the requirements above. Particularly, in Chapter 3 we evaluated the effectiveness of three state-of-the-art test generation techniques on the same set of subjects.

Given that EvoSuite$\mathbb{R}$ detects changes at class-level, to ensure that a change exists between each pair of classes, we excluded subjects where more than one class was modified, resulting in 327 subjects. This was done to simplify and standardise our evaluation infrastructure setup across all subjects. Given that

only a small subset (30) of the DEFECTS4J repository subjects were excluded, we do not believe that this would influence the conclusion of any of our reported results. Moreover, as mentioned in Section 4.2.1.3, the Control-flow-distance measurement can only be used on pairs of versions where there is a one to one matching between all branches of the program. Therefore, to evaluate techniques including this measurement, we selected a subset of the faults that contain a one-to-one matching between the branches, resulting in 50 subjects. The matching was performed based on two conditions: 1) the two versions of the program contain the same number of branches, and 2) the matched branches between the two versions are of the same bytecode comparison family (as discussed in Section 5.2). Table 4.2 details the subjects selected for this study.

4.3.3 Evaluated Techniques

To find out the effectiveness of the measurements proposed in Section 4.2.1, we conducted experiment with the following algorithms, to which from hereon we will refer to as the text in bold:

1. GA-**Comb**: Combination of GA-Coverage and GA-State
2. GA-**Coverage**: Coverage on both versions
3. GA-**State**: State difference
4. GA-**CFD**: Control-flow distance
5. GA-**ALL**: All three measurements combined
6. **Random**: Random testing

As we mentioned in Section 4.2.1.3 and Section 4.3.2, CFD can only be used on subjects where there is a one-to-one matching between the control-flow branches of the two versions of the program. As such, the configurations GA-ALL and GA-CFD can be applied to subjects that satisfy this requirement. GA-Comb however does not have this limitation.

To implement the random test generation technique, we extended EVOSUITE$\mathbb{R}$ such that instead of using a genetic algorithm, test generation is performed

randomly. That is, as long as the search budget has not been exhausted, new tests are generated randomly and evaluated on both versions. The search stops as soon as a difference in the behaviour is detected.

4.3.4 Experiment Procedure

In this section we present our applied procedure to answer each research question:

4.3.4.1 RQ4.1: Comparison of fitness functions

In order to compare the different measurements used in our multi-objective fitness function, and to ensure that our combined fitness function is the most effective, we applied EvoSuiteR using different fitness functions and compared their effectiveness at fault finding. In particular, we applied and compared all 5 different versions of our fitness function as reported in Section 4.3.3.

To apply each technique for the purpose of test generation, we used pairs of subjects taken from the Defects4J repository, where each pair consists of a *buggy* and *fixed* version. We then applied EvoSuiteR on the pair such that it generates tests which pass on the fixed version and fail on the buggy one. Although once EvoSuiteR is finished generating tests it can report on whether or not it has identified a difference between the two versions, to increase our confidence in the data and to lower threats to the internal validity of our technique, we followed methodology from Chapter 3 and applied the same rigorous procedure to validate the generated tests (e.g., by removing false positives and flaky tests), and executed the tests using JUnit outsde the EvoSuiteR framework.

We consider a fault to be *found* or *detected* if a test suite successfully passes on the fixed version of the program, while one or more test cases from the test suite fail on the buggy version of the program. To detect whether a generated test suite has detected a fault, we executed each generated JUnit test suite on each respective pair of programs as follows:

- **Execution on *Fixed* version:** Since either of the generated test suite or the system under test may have non-deterministic behaviour, it is possible that some test cases can have flaky behaviour; that is, when a test case is

executed on the same version multiple times the outcome of the test may differ. Therefore, to ensure that the generated tests are not failing due to flaky behaviour, we followed methodology in previous work [156] and executed each test suite on the fixed version (i.e., the version the test suite was generated on) repeatedly until all tests within the generated test suite pass for at least 5 consecutive runs. We removed any failing tests during this process.

- **Execution on *Buggy* version:** After executing the test suite on the fixed version, we executed the resulting test suite on the buggy version, and recorded the following outcomes: a) The number of tests failing due to assertion failures, b) The number of failing tests due to errors caused by unexpected exceptions, c) In case of any failures, we recorded the execution report containing the reason for the failure of each test case, in addition to the respective stack trace. In a real software development scenario, these three outcomes are available to developers as well.

We then compared the different measurements in terms of the number of bugs they can detect. However, not all measurements were directly comparable. In particular, as we mentioned earlier, the techniques involving Control-flow distance require a one-to-one matching between the control-flow branches across the two versions of the program. As such, we conducted two separate set of experiments using two different sets of faults as reported in Section 4.3.2: 1) we compared the effectiveness of the GA-Comb against GA-Coverage and GA-State, 2) we compared the effectiveness of GA-All against GA-CFD and GA-Comb. The first set of experiments enables us to compare the combined fitness function against its individual components. The second set of experiments enables us to compare the combined fitness function with CFD, and answers whether or not using CFD is beneficial to the combined fitness function.

4.3.4.2 RQ4.2: Effectiveness of the technique

To answer our second research question, we compared the effectiveness of EvoSuite$\mathbb{R}$ using the combined fitness function (GA-Comb) against a baseline of random testing. We used the same methodology as in RQ4.1 for test-generation and execution.

4.3.4.3 RQ4.3: Comparison with the state-of-the-art

We answer this research question by comparing our approach EvoSuiteR against the evaluation framework we presented in Chapter 3 and [156]. Specifically, we compare the effectiveness of EvoSuiteR using the combined fitness function against the three state-of-the-art test generation techniques AgitarOne, EvoSuite and Randoop. For this comparison, we use the dataset from RQ4.1 for EvoSuiteR, and the dataset from our evaluation framework [156] (Chapter 3) for the other techniques.

4.3.5 Experiment setup

Testing environment: The experiments were conducted on the High Performance Computing Cluster named *iceberg* at the University of Sheffield. Each node on the cluster has a Sandy-bridge generation Intel Xeon processing, and we allocated each experiment 3GB of real and 6GB of virtual memory.

Search budget: Given that in this study we are evaluating an evolutionary algorithm, and considering the large overhead of our technique, we used a budget of 10 minutes for all experiments.

Number of repetitions: Considering that our approach is based on randomized algorithms, to lower the effect of randomness and also to lower the influence of the noise on the cluster, we repeated all experiments for 30 times.

4.3.6 Data Collection

During our preliminary evaluations of our approach, in order to record and analyse the search process, and to monitor the progress and impact of each of the measurements, we kept track of several measurements during the execution, as presented in Table 4.3. These measurements are calculated after each generation of the GA for the best individual in the population, and stored in a unique *CSV* file. After the search is completed, the relevant CSV file and generated test suite are stored separately for each individual run of EvoSuiteR.

Table 4.3 List of tracked measures during the search

Generation Age	Fitness Value	Test Count	Test Size
Exception Difference Value	Total Number of Exceptions	State Difference Value	Branch Difference Value
Coverage Value	Coverage Percentage on Old Version	Coverage Percentage on New Version	Number of Executed Statements
Number of Assertions	Time Since Start (ms)	Duration of Test Execution	Duration of Assertions
Duration of Coverage Calculation	Duration of State Difference Calculation	Duration of branch difference calculation	Duration of Object Collection

To further assist the analysis of the data, we developed a web-based platform [152] to help us aggregate and compare the large amount of data collected. The system takes the data collected during the execution as CSV files and stores all the data in a database for analysis. Using the web interface, we could then analyse and compare the performance and effectiveness of EvoSuite$\mathbb{R}$ on any of the chosen configurations. Users can view graphs of how each of the measurements evolve during the search, enabling them to better evaluate the approach. The generated test suites are also available for inspection, along with embedded assertions demonstrating the observed change.

The comparison tools on the system enable the user to quickly compare individual subjects across different experiments and presents the results using colour coding for easier inspection. Due to the scalable design of the system, it allowed us to quickly aggregate, visualise, and analyse over 200 million generations. Therefore, we believe such visualisations can be helpful during the development of search-based techniques in general.

4.3.7 Threats to Validity

External Validity: In this study, we studied the effectiveness of the tools on only five open-source subjects taken from open source repositories. As such, our results may not generalise to other types of programs. Moreover, we only used single-class changes (i.e., only a single class was modified between the two version of the program) in our evaluation. While this was done deliberately since our technique works at class-level, the effectiveness of the technique may differ when multiple classes are modified, and especially if finding the bug requires testing multiple classes at the same time.

For measuring the effectiveness of control-flow-distance, we only used a smaller subset of 50 subjects compared to the 327 subjects when comparing other techniques. While this was done such that our technique would be able to match control-flow branches across the two versions of the program, this can pose a threat to the level in which this data generalises.

Internal Validity: We implemented our proposed technique on top of EvoSuite, and as such, the effectiveness of the technique may be affected by the limitations of EvoSuite itself, such as the type of test statements it can generate, the performance bottlenecks of the code instrumentation and execution, etc., and as a result may not generalise to other tools. Moreover, bugs in our implementation, our experiment setup, or in EvoSuite may contribute towards our technique reporting false-positives (i.e., reporting a subject as detected while it was not), or on the other hand failing to report on subjects that were not detected, thus underestimating the effectiveness of the tool. In order to mitigate false positives, we followed the methodology in [156] for preventing flaky tests and removing false positives by manual inspection. However, due to the large number of tests, manually inspecting all tests was not feasible and only a sample of the generated tests were inspected.

Construct Validity: A threat to the validity of our comparison against the state-of-the-art tools (RQ4.3) is the difference in the search budget allocated to the techniques. That is, while we used a search-budget of 10 minutes for the EvoSuite$\mathbb{R}$-Comb technique, a smaller budget of 3-minute was used for

EvoSuite and Randoop– but not for AgitarOne– in our earlier study [156]. Therefore, it is possible that the effectiveness of the two state-of-the-art techniques may differ, given more time. This decision was made in consideration of a number of different factors: a) EvoSuite$\mathbb{R}$-Comb is currently a prototype implementation of the aforementioned algorithm, and inefficiencies in the current implementation of the approach reduce the speed of the technique. This is while the other techniques used in the comparison are relatively mature, and their implementations have been optimised over the past several years, b) this evaluation is only a preliminary comparison against the state-of-the-art techniques, to better understand where EvoSuite$\mathbb{R}$ stands – or can potentially stand by optimising the implementation – compared to other techniques, c) to avoid repetition of previous evaluations – which in turn involved significant manual and computational effort such as investigation and removal of flaky and false-positive tests –, we used data from RQ4.1 and Chapter 3, d) this difference in the allocated time only affected 2 of the 3 techniques, for which we do not believe that the allocation of an extra search budget would increase their effectiveness.

For EvoSuite we observed only a minimal difference when a larger search budget was used – this is confirmed in Chapter 5 [157]. Moreover, since EvoSuite aims to cover all branches with the smallest test suite, therefore, when coverage cannot be further improved, a smaller test suite with the same level of global coverage is preferred by the technique. Therefore, test suites generated by EvoSuite shrink in size when given a larger search budget, if the tool is unable to cover further goals. However, our previous experiments and recent work [60] have shown that these smaller test suites are less diverse and therefore, may be less effective at fault finding. While this problem does not exist for Randoop, we have observed that test suites generated by Randoop already saturate the code coverage in $<$ 1 minute, and the newly generated tests are heavily redundant [156]. Therefore, we believe that our conclusions should not be affected by this choice.

The type of bugs used in this study also pose a threat to the accuracy of this report on the effectiveness of the techniques. In particular, the bugs were taken from actively-maintained open source projects where developers have access to high quality and high coverage test suites. Moreover, the bugs were taken from publicly available commits taken from these repositories, and it is likely

that a number of regression faults were caught during the development process by the developers. Therefore, it is likely that our report underestimates the effectiveness of the tools. Another threat related the subjects used in this study is the nature of the bugs themselves. While we look at a regression that would occur by moving from a fixed version of the program to a buggy version, it is possible that a number of such cases do not represent real regression faults that may occur in practice.

4.3.8 Results

RQ4.1: Which measurement is the most effective for propagating changes? Although in our preliminary evaluation of the technique [155] we found the combined fitness measurement to be the most effective on toy examples, this research question investigates whether the individual measurements used for guiding the search are effective in practice on real faults. Moreover, it aims to answer whether the combined fitness function is the most effective overall, for the purpose of detecting changes – which can be regression faults. As such, we evaluated the effectiveness of our approach using each individual metric for the fitness function, in addition to the combined metric. Since different set of faults were investigated for techniques with or without control-flow distance, we first present the results on all subjects, and thereafter present the results for control-flow distance using subjects with equally matching branches.

Overall, out of 327, GA-Comb managed to find 162 bugs at least once, GA-Cov found 138, and GA-State found 152. While more bugs were found by the combined fitness function, not all bugs found by the individual measurements were also found by GA-Comb. For instance, 6 and 10 bugs were found by GA-Cov and GA-State respectively that were not found by GA-Comb. Moreover, out of 30 repetitions, the techniques had different success rates at detecting the same faults. In Figure 4.4 we compare the success rate of the combined technique against when individual metrics are used alone at detecting the faults. Moreover, in the same figure we present the number of subjects for which one technique had a significantly higher success rate at detecting the fault than the other. To compare significance, we used Fisher's Exact test [46, 13], and report on a subject to have a significantly higher success rate if the p-value of the test is

Fig. 4.4 Comparing the effectiveness of EvoSuiteR using the combined fitness function (Comb), against EvoSuiteR with only branch-coverage measurement, and EvoSuiteR with only state-distance measurement.

less than 0.05. Observe that the combined metric achieved a significantly higher success in more subjects than the other two measurements. It is noteworthy to point out the little difference observed between GA-Comb and GA-State. This indicates that while the addition of GA-Cov is beneficial, it provides only a small improvement to GA-State.

The combined fitness function is more effective than the individual components such as "Coverage" and "State-distance". However, on a majority of subjects, state-distance alone is equally effective.

Our second set of experiments evaluate the effectiveness of the Control-flow distance measurement when applied alone, or when applied in combination to all measurements. As such, we present our evaluation by first comparing GA-CFD against GA-All, and then comparing GA-All against GA-Comb (i.e., without CFD). Out of 50 bugs, both GA-All and GA-CFD found 33 bugs at least once. Moreover, GA-CFD did not detect any faults that GA-Comb was unable to detect. Figure 4.5 summarizes the comparison of the success rate of the techniques at detecting the fault. Notice that while CFD alone can be similarly as effective as GA-All, the combined measurement GA-All was not better than GA-Comb. In fact, for one subject, the technique had a lower success rate, likely due to the additional overhead of CFD calculations.

While using CFD alone is an effective way to propagate faults, it does not provide any additional improvements to GA-Comb.

In summary, in this research question we presented a comparison of the different measurements proposed in this chapter for the purpose of differential test suite generation using a search-based approach. Our results show that of the three proposed measurements each individual metric is effective way to guide the

Fig. 4.5 Comparing the effectiveness of EvoSuiteR using the combined fitness function (ALL), against EvoSuiteR with CFD alone, and EvoSuiteR with all measurements excluding CFD (Comb).
Legend: "Significantly higher" is the number of classes for which a technique generated a test that detected the change in a significantly higher number of runs than the other; "Higher" refers to the number of classes where a technique generated a test detecting the change in a (non-significant) higher number of runs; "Equivalent" is where the change was detected by both techniques for the same number of runs.

search towards generating differential tests that can detect potential regression faults. However, in our experiment we found control-flow distance to not provide any further guidance compared to the combination of coverage and state-distance. Moreover, we found state-distance alone to be surprisingly effective, even when no guidance exists towards reaching the fault. Therefore, our results indicate that maximising state-infections can be an effective way to propagate changes.

RQ4.1: *The combination of "Coverage" and "State-distance" is the most effective, compared to using individual measurements. Although Control-flow distance was effective alone, it did not provide any extra improvement towards guiding the search.*

RQ4.2: How effective is EvoSuiteR at detecting real faults?

To answer this research question we compare EvoSuiteR using the combined GA-Comb fitness function against random search. Out of 327, random testing found 160 bugs at least once while GA-Comb found 162. However, only 143 of these bugs were found by both techniques, and each technique finds different sets of bugs. Although GA-Comb finds a slightly higher number of faults at least once, random search has a larger number of subjects for which it achieves a significantly higher success rate at detecting the fault. Figure 4.6 summarises the statistical significance of the comparison of the success rate of the two techniques.

Observe that except for 33 subjects, random search is equally or more effective than our combined GA approach. This outcome can be particularly surprising given the high success rate of the random approach, and especially considering

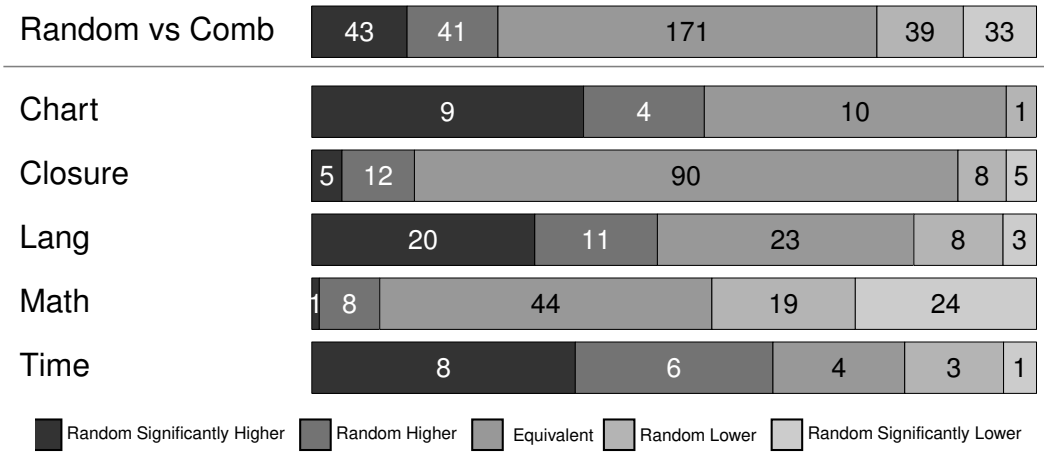


Fig. 4.6 Comparing EvoSuiteℝ using random search, against EvoSuiteℝ using *GA*-Comb (combined measurements).
Legend: "Significantly higher" is the number of classes for which a technique generated a test that detected the change in a significantly higher number of runs than the other; "Higher" refers to the number of classes where a technique generated a test detecting the change in a (non-significant) higher number of runs; "Equivalent" is where the change was detected by both techniques for the same number of runs.

the high popularity of evolutionary techniques in test generation techniques. Moreover, each of the techniques was more effective on a different set of projects. In particular, as can be seen in Figure 4.6, for projects `Chart`, `Lang` and `Time` random search appears to outperform GA, while on `Math` – short for Apache Commons Mathematics Library – the GA was more effective. The results indicate that there exists an underlying characteristic difference across the projects, that makes one technique better suited to detect the change than the other.

To ensure that the combined fitness function is still more effective than random search compared to the individual metrics used, Figure 4.7 summarizes the comparison of the effectiveness of individual metrics of the fitness function alone against random testing. This further confirms our observation in RQ4.1 that the combined fitness function is the most effective. Nevertheless, the successful outcome of random search presents interesting results that raise new questions – see Chapter 5

RQ4.2: *Differential testing using a random heuristic was significantly more effective at fault finding than differential testing using our combined GA in 43 subjects, while being significantly worse in only 33.*

Fig. 4.7 Comparing the effectiveness of EvoSuiteR using random search, against EvoSuiteR with only branch-coverage measurement, and EvoSuiteR with only state-distance measurement.
Legend: "Significantly higher" is the number of classes for which a technique generated a test that detected the change in a significantly higher number of runs than the other; "Higher" refers to the number of classes where a technique generated a test detecting the change in a (non-significant) higher number of runs; "Equivalent" is where the change was detected by both techniques for the same number of runs.

RQ4.3: How does EvoSuiteR compare with the state-of-the-art test generation techniques?

To answer our third research question we compare our approach against the state of the art test generation tools, using the evaluation framework we developed earlier in Chapter 3. In Figure 4.8 we summarize the comparison of the success rate of EvoSuiteR using GA-Comb against the three state of the art tools AgitarOne, EvoSuite, and Randoop. Notice that compared to EvoSuite and Randoop, EvoSuiteR was significantly more effective in a larger number of subjects.

Given our findings in Chapter 3, we were not surprised by AgitarOne's performance. AgitarOne takes advantage of several different techniques in order to generate high coverage and sensitive tests. These techniques include: 1) aggressive mocking: AgitarOne generates mock objects which test the internal coupling and dependencies between the methods and objects inside the program, 2) existing code/test: AgitarOne takes advantage of any existing tests manually written by the developers, or from existing code written in other classes in order to instantiate hard-to-create objects. These techniques however do not come without caveats, for instance, aggressive mocking can lead to a large number of failures due to false positive tests, or a lower success rate if the developers do not have access to an existing test suite.

In our earlier study, AgitarOne found 130 bugs at least once, EvoSuite found 145, and Randoop found 93. In this study, EvoSuiteR using the GA-Comb fitness function found 162 bugs at least once. While our technique finds

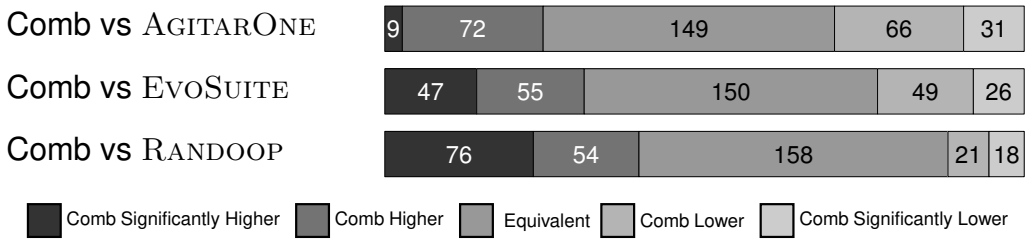


Fig. 4.8 Comparing the effectiveness of EvoSuiteR using the combined fitness function (Comb), against three state of the art test generation techniques: AgitarOne, EvoSuite, and Randoop.
Legend: "Significantly higher" is the number of classes for which a technique generated a test that detected the change in a significantly higher number of runs than the other; "Higher" refers to the number of classes where a technique generated a test detecting the change in a (non-significant) higher number of runs; "Equivalent" is where the change was detected by both techniques for the same number of runs.

more faults than the others, we should note that the techniques find different set of faults, and each technique alone finds fault that are not detected by others.

RQ4.3: EvoSuiteR *found more faults than the three state-of-the-art techniques. However, while it was more effective than* EvoSuite *and* Randoop *at finding the same faults, for 31 subjects, it was less successful than* AgitarOne.

4.4 Summary

In this section we presented a search-based approach for the purpose of differential test suite generation, and presented a large empirical study evaluating its effectiveness in practice – which was done according to the framework we presented earlier in Chapter 3. Our large-scale evaluation consisted of running different configurations of our technique on 327 real bugs taken from open source repositories, resulting in more than 50,000 test suites with a serial execution time of close to 9,000 hours.

Overall, the evaluation results show that tests generated using EvoSuiteR can be effective at detecting real faults, and that a combined measurement (i.e., our proposed multi-objective *GA*) approach is the most effective configuration, compared to any of the individual measurements. However, while the control-flow distance objective was effective at guiding the search when applied alone, it did not provide any additional benefit towards finding more faults. The similar

results of EvoSuiteR using the combined measurement as opposed to using state difference alone shows that the success of the approach relies heavily on maximising the state difference of the program.

In contrast to our expectations, for a considerable portion of the faults, EvoSuiteR using GA-Comb did not achieve a significantly better result than random search. Our conjecture for this outcome is that a) random-search does not have the fitness function overhead of the GA, and therefore can evaluate a larger portion of the search space given the same time budget, b) small differences in the underlying characteristics of the two versions of the program under test results in a relatively flat search landscape for the evolutionary technique (GA), and so, a higher diversity in the population is necessary to better explore this landscape (something in which random search can always outperform a systematic approach); we investigate this problem further in Chapter 5, to understand whether this problem is specific to generating regression tests and finding changes (e.g., changes represent needles in a haystack), or a wider problem that applies to search-based test generation techniques in general. Nevertheless, it is worth noting that our technique was able to detect faults that random search failed to – EvoSuiteR using GA-Comb in fact found *more* faults than any other approach we evaluated so far. Moreover, for a number of faults, our GA managed to achieve significantly better results compared to random testing.

When compared to three state-of-the-art test generation techniques, we found our technique to perform equally or better for a majority of subjects. The state-of-the-art techniques aim to generate tests for individual classes, rather than a pair of classes, which can be beneficial to revealing behavioural differences. However, while for a small number of subjects AgitarOne achieved a significantly higher success rate, we believe our generated test suites to be superior due to the size, length and complexity of the generated test suites. Our findings also suggest that using differential testing (e.g., using EvoSuiteR) can be a complementary approach for regression testing alongside existing practices such as coverage-driven testing (e.g., using EvoSuite), which we will investigate further in Chapter 6.

We have made our GA-based differential test generation tool EvoSuiteR open source and publicly available at `http://www.evosuite.org/evosuiter`. Appendix A provides the details on how to use and configure the tool.

Chapter 5

Comparing Random and Genetic Algorithm Search for Generating High-Coverage Test Suites

The content of this chapter is based on work undertaken during this PhD by the author, which has been published elsewhere [157].

5.1 Introduction

In the previous chapter, we proposed a search-based approach specifically targeted at the problem of regression testing. Briefly, our multi-objective algorithm aimed to reach and propagate changes between two versions of a program at the same time. While our evaluation found the technique to be more effective than current state-of-the-art techniques, a surprising observation was that when compared to a similar technique with the algorithm replaced with random search – while the rest of the tooling remaining identical – for a large majority of subjects, random-search was as effective. Although on one hand we may conjecture that this is a problem specific to regression testing – that is, the search landscape for regression problems is mostly flat (i.e., needles in the haystack) —, on the other hand our results indicated that this may be a bigger problem in general for automated test generation using search-based algorithms on object-oriented

programs. As a result, it is essential to know which of the two is the case. For instance, if the latter is the case, then improvements to the GA and tooling may be beneficial to both techniques. Our results may also help us to predict which algorithm to use for which set of programs.

As we reported in Chapter 2, many different techniques and algorithms for different types of software testing problems have been proposed. One particular application area where search-based techniques have been successfully applied is the unit testing of object-oriented programs, where test cases are sequences of object constructor and method calls. There are various tools available for languages such as Java and .NET, ranging from tools based on random search such as Randoop [124], JCrasher [35], JTExpert [147], NightHawk [9], T3 [132], or Yeti-Test [121], to tools based on evolutionary search such as EvoSuite [48], eToc [170] or Testful [18]. Systematic approaches such as JWalk [160] also exist, which perform exhaustive search with the objective of visiting new states.

Although the tools based on evolutionary search techniques are commonly thought to be superior, it is unclear whether this is actually the case in practice. It could be that differences in performance across tools may be accounted for by the differences in the programming language that they target, or in the way they have been engineered, as opposed to any specific benefits of the particular search algorithm that they apply. In order to shed more light on these questions, in this chapter we report on experiments to contrast the use of a Genetic Algorithm (GA) against random search. The GA algorithms optimize unit test suites for code coverage while the random search algorithm optimizes code coverage by adding random tests to a test suite.

To allow for a fair comparison, we use the GA and a common version of random test generation implemented in the same tool – EvoSuite, which generates branch covering test suites for Java classes. We run our experiments on a random sample of 1,000 classes from the SF110 corpus of open source projects [52] and evaluate the techniques in terms of the achieved code coverage. Specifically, we make the following contributions in this chapter:

- We report on the effectiveness of using a GA compared to an algorithm based on random search for the purpose of generating test suites with high branch coverage for object-oriented programs.

- We investigate the influence of certain types of branches within the classes under test on the performance of each technique.
- In order to better understand the effect of the search budget on the performance of each technique, we study the effectiveness of the GA and random search over time with an extended search budget.

In the following sections, first in Section 5.2 we introduce the concept of "branch types" in Java bytecode and discuss the influence it can have on the search algorithms. Next, in Section 5.3 we present the setup used in our experiment and the research questions this work is aiming to address. Thereafter we present the result of our experiments followed by a discussion of the results. Finally we discuss some related work done in this area, followed by a conclusion of this work.

5.2 Types of Branches in Java Bytecode

In this chapter, we study the application of evolutionary and random search to automatic test suite generation, as implemented in the EvoSuite tool. EvoSuite (as previously discussed in Section 2.1.5.3) aims to generate unit test suites that cover as many branches of a Java class as possible, while also executing all methods that are devoid of branches, referred to as "branchless" methods. Given that covering branchless methods can be trivial, this section looks at how the branches are represented and how they can be used for guidance.

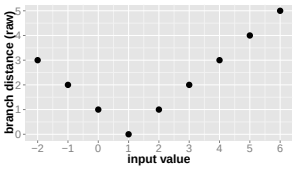
Considering that this study uses EvoSuite (a Java test generation tool) and evaluates the technique on Java programs, in this chapter we look at the representation of branches in the Java programming language. However, this is not limited to the Java language and set-aside a small set of differences, it should generalise to other object-oriented languages (e.g., .NET languages compile to Common Intermediate Language (CIL)). Java programs are compiled to bytecode for execution on a Java Virtual Machine (JVM), and it is at the level of the bytecode at which EvoSuite works – branch distances are computed by instrumenting and monitoring bytecode instructions. Different types of bytecode instruction can therefore give rise to different types of fitness landscape that may or may not be useful in guiding the search.

```
void m(int a) {
  if (a == 1) {
    // uncovered branch
  }
}
```

(i) Source code

```
void m(int);
  0: iload_1
  1: iconst_1
  2: if_icmpne 7
     [uncovered branch]
  7: return
```

(ii) Bytecode



(iii) Raw branch distance

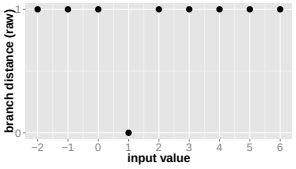
(a) Int-Int Branch

```
void m(int a) {
  boolean x = false;
  if (a == 1)
    x = true;
  if (x) {
    // uncovered branch
  }
}
```

(i) Source code

```
void m(int);
   0: iconst_0
   1: istore_2
   ...
   9: iload_2
  10: ifeq 15
      [uncovered branch]
  15: return
```

(ii) Bytecode



(iii) Raw branch distance

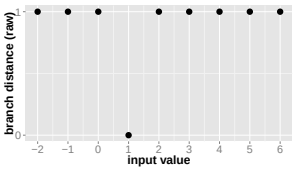
(b) Int-Zero Branch

```
void m(int a) {
  Object x = null;
  if (a == 1)
    x = this;
  if (this == x) {
    // uncovered branch
  }
}
```

(i) Source code

```
void m(int);
   0: aconst_null
   1: astore_2
   ...
   9: aload_0
  10: aload_2
  11: if_acmpne 16
      [uncovered branch]
  16: return
```

(ii) Bytecode



(iii) Raw branch distance

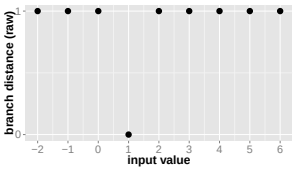
(c) Ref-Ref Branch

```
void m(int a) {
  Object x = null;
  if (a != 1)
    x = new Object();
  if (x == null) {
    // uncovered branch
  }
}
```

(i) Source code

```
void m(int);
   0: aconst_null
   1: astore_2
   ...
  15: aload_2
  16: ifnonnull 21
      [uncovered branch]
  21: return
```

(ii) Bytecode



(iii) Raw branch distance

(d) Ref-Null Branch

Fig. 5.1 Examples of different branch types (denoted "uncovered branch") and their effect on the respective fitness landscape for the GA through raw (unnormalised) branch distance values. We show both the original Java source and the compiled bytecode, as processed by EvoSuite. Note that the target true/false evaluation of the branches is inverted by the Java compiler. The first column gives an example of a "gradient" branch, providing true guidance to the search. Conversely, the remaining examples do not provide good guidance, with the majority of inputs to the method resulting in the same distance value, and consequently a fitness landscape that is flat other than for the value required to execute the branch of concern.

Given that the fitness function is one of the key differences between the evolutionary and random search, and that a major component of the fitness function is the calculation of distances for the branches in the class under test, we now classify the four types of branches that occur in the bytecode of Java programs, and discuss the level of guidance they can potentially afford the evolutionary search in EvoSuite.

This is important because it has been long known that not all branch predicates give "good" guidance, the archetypal example being that involving the boolean flag [73, 16]. Boolean conditions in branch predicates can only ever evaluate to true or false, offering one of only two distance values. Since one of these values corresponds to execution of the branch, no guidance is given to the search. Nevertheless, several branch predicates (e.g., integer comparisons, double comparisons, float comparisons, etc.) do indeed provide guidance, and result in a smooth "gradient" in the fitness landscape that a guided search can use to easily find test inputs.

5.2.1 "Integer-Integer" Branches

"Integer-Integer" branches involve the comparison of two integer values. The range of values possible for this comparison can potentially create a gradient for the search. Figure 5.1a shows an example of such a comparison, in which a method receives an integer parameter "`a`", and has a conditional statement on the parameter ("`a == 1`") (Figure 5.1a-i). The bytecode (Figure 5.1a-ii) shows this is compiled to a "`if_icmpne`" instruction, which compares the last two integers pushed to the stack, performing a jump to some other instruction in the bytecode if those two integers are not equal. Figure 5.1a-iii shows how the distance value decreases as the chosen input value gets closer to the value that would execute the uncovered branch.

Of course, "Integer-Integer" branches may not always result in a gradient: it depends on the underlying program. One example of this is where two boolean values are compared, since boolean values are represented as the integer values 0 and 1 in Java bytecode. Therefore, source code comparisons involving two boolean values are compiled to an integer comparison involving the usage of the

`if_icmpne` instruction. However, and as already discussed, boolean conditions do not induce any useful landscape gradient.

Furthermore, EvoSuite's special handling of `switch` statements falls into the "Integer-Integer" category. Java `switch` statements are compiled to either a `tableswitch` or `lookupswitch` bytecode instruction. These instructions pop the top of the stack to look up a "jump" target instruction in a map data structure, for which the keys are the values originally used in each `case` of the `switch`. For ease of fitness computation, EvoSuite simply instruments the bytecode by adding an explicit `if_icmpeq` for each case before the original `tableswitch` or `lookupswitch` instruction, comparing the top of the stack to each `case` value.

5.2.2 "Integer-Zero" Branches

"Integer-Zero" branches involve the comparison of an integer value with zero. One type of "Integer-Zero" branch occurs when boolean predicates are evaluated[1], for example as shown by Figure 5.1b. Here the branch involves the evaluation of the boolean value `x` (Figure 5.1b-i). The corresponding bytecode evaluates `x`, pushing the result (an integer, `0` or `1`) to the stack. The `ifeq` bytecode instruction then pops this value, performing a jump if it is zero. Such a condition can only be either true or false, and as such can only have one of two distance values, which, as shown by Figure 5.1b-iii, are not useful to guiding the GA to covering the branch. The "right" input must therefore be discovered purely by chance.

A further type of "Integer-Zero" branch occurs as result of comparisons involving values of `float`, `double` and `long` primitive Java types. Figure 5.2 shows an example of a double comparison. The original source (Figure 5.2a) performs the comparison in the branch predicate. This is decomposed into a sequence of bytecode instructions shown by Figure 5.2b. The comparison is performed by the `dcmpl` in relation to the top two double values pushed to the stack. The `dcmpl` instruction pushes an integer to the stack: `-1` if the first value is greater than the second, `1` if the first is less than the second, else `0` if they are equal. The `ifne` then performs a jump if the top of the stack is not `0`.

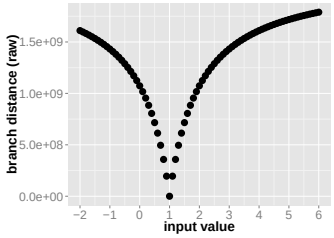
[1]Note that boolean predicate evaluations in branches differ in bytecode from comparing two boolean values – the latter type of branch falls into the "Integer-Integer" category.

```
void m(double a) {
  if (a == 1.0) {
    // uncovered branch
  }
}
```

(a) Source code

```
void m(double);
   0: dload_1
   1: dconst_1
   2: dcmpl
   3: ifne 10
   [uncovered branch]
  10: return
```

(b) Bytecode



(c) Raw branch distance

Fig. 5.2 An example of handling a `double` comparison, showing the source code (a) and the bytecode (b). Although these branches fall into the "Int-Zero" category, EvoSuite instruments the bytecode so that distances are recovered, resulting in a gradient landscape (c).

Since the original numerical comparison in the source code is transformed to a boolean comparison in the bytecode, a significant amount of useful distance information is "lost" in the compilation process that would have been useful in guiding the search. EvoSuite therefore instruments the bytecode so that distance information can be recovered. The branch distance plot for the example, shown by Figure 5.2, therefore restores a gradient that can be used to optimize input values towards execution of the uncovered branch.

5.2.3 "Reference-Reference" branches

"Reference-Reference" branches are where two object references are compared for equality. Since references are not ordinal types, no meaningful distance metric can be applied, and the situation is similar to boolean flags – either the references are the same or they are not. Figure 5.1c shows an example of this. The original source code conditional is "`if (this == x)`" (Figure 5.1c-i), which Java compiles to the bytecode instructions 9–11 in Figure 5.1c-ii. Instructions 9 and 10 push the references onto the stack. Instruction 11 is the branching point in the bytecode, with "`if_acmpne`" popping the top two stack references and performing a jump if they are not equal. The resulting plot of branch distances (Figure 5.1c-iii) shows the resulting plateau, providing no guidance to the required input that makes the references equal and executes the uncovered branch.

5.2.4 "Reference-Null" branches

"Reference-Null" branches are similar to "Reference-Reference" branches, except one side of the comparison is `null`. Again, no meaningful distance metric can be applied. Figure 5.1d shows an example. The source code compares `x` with `null`. In the bytecode, `x` is pushed onto the stack by instruction 15. Instruction 16 is the branching point, where the `ifnonnull` instruction performing the jump if the element popped off the top of the stack is not `null`.

5.2.5 Summary

We have summarized and classified the different types of branches that can occur in Java bytecode. Some of these instructions will potentially give rise to a "gradient" in the fitness landscape, while others will not. We now study the prevalence of these types of branches in real-world code, whether they potentially involve a gradient, and their potential impact on the relative performance of random search and fitness-guided *GA* search.

5.3 Experimental Setup

We designed an empirical study to test the relative effectiveness of test case generation using random, *GA* search, with the aim of answering the following research questions:

RQ5.1: Is the use of an evolutionary algorithm, such as GA, more effective at generating unit tests than random search?

RQ5.2: How do the results of the comparison depend on the types of branches found in the code under test?

RQ5.3: How do the results of the comparison depend on the time allowed for the search?

Table 5.1 Statistics for the sample of 980 classes.

	Min	**Avg**	**Max**	**Sum**	**SD**
Total Branches	0	26.87	1,016	26,336	79.3
Branchless Methods	0	7.18	155	7,041	11.5
Total Goals	1	34.06	1,026	33,377	84.1

5.3.1 Subjects

In order to compare and contrast the relative effectiveness and performance of random and evolutionary search, we needed a large set of classes from real-world projects – as opposed to a large set of bugs which we used in the previous chapters. As such, we selected a sample of classes from the SF110 corpus of open source projects [52]. The SF110 corpus is made up of 110 open source projects from the SourceForge open source repository (http://sourceforge.net), where 10 of the projects were the most popular by download at the time at which the corpus was constructed (June 2014) and the remaining 100 projects selected at random. Due to the large variation in the number of classes available in each project, we stratified our random sampling over the 110 projects, such that our sample involved at least one class from each of the 110 projects in the corpus, and comprised 1000 classes in total. However, 20 classes were removed from the sample for reasons such as not having any testable methods (e.g., they consisted purely of enumerated types, or did not have any public methods) or test suites could not be generated for some other reason that would allow us to sensibly compare the techniques (e.g., the class contained a bug or other issue that meant it could not be loaded independently without causing an exception).

The final number of classes in the study therefore totalled 980, comprising small classes with just a single coverage goal to larger classes with over 1,000 coverage goals, as shown by Table 5.1. In this table, *Branchless Methods* indicates the number of methods without conditional statements that can be covered by simply calling the method concerned.

5.3.2 Collation of Branch Type Statistics

So that we could answer RQ5.2, we collated a series of statistics on the types of branches in the bytecode of each class.

First, we simply collected the numbers of branches that fall into each of the categories detailed in Section 5.2 (i.e., "Integer-Integer" etc.) by statically analysing the bytecode of each class in turn.

Secondly, we attempted to classify each branch as either *potentially* having a gradient distance landscape ("Gradient Branches"), or, a plateau landscape ("Plateau Branches"). We programmed EvoSuite so that during test suite generation it would monitor the distance value of the predicate leading to the branch. If in any of the executions of a search algorithm in the experiments, a value other than 0 or 1 is observed, we assume a wider range of distance values is available for fitness computation and label the branch as a "Gradient Branch". Otherwise the search is labelled as a "Plateau Branch". Clearly, this analysis is only indicative (but helps in understanding our results, as we will show in the answer to RQ5.2). This is because a range of values does not necessarily imply a gradient that will be useful for guiding the search. Nor does only finding the distances 0 and 1 for a branch mean that there are not further distance values that could be encountered. For instance, given a branch predicate $x > 5$, if for the whole duration of the search only the values $x \in \{4, 5, 6\}$ are used, then this will result in the distance values of 1, 0, and 1 respectively; and the branch will be incorrectly classified as a plateau branch. However, it is quite unlikely that the branch would only be attempted with these values over the course of several executions of a search algorithm.

5.3.3 Experimental Procedure

We applied EvoSuite to conduct our experiments, with implementations of the genetic algorithm (*GA*), and the two random search algorithms (*Random+* and *Pure Random*) as described in Section 2.1.5.3.

We use all four algorithms in RQ5.1 only. *Pure Random* features only in RQ5.1 in order to analyse for possible effects with *Random+* due to its seeding mechanism.

For RQ5.1 we applied each technique with a search time of two minutes (which has been shown to be a suitable stopping condition in previous work [52]). To answer RQ5.2, we investigated the influence of the type of conditional predicates on the outcome of each technique. To do so, we used the statistics on branch types, collected as we described in the last section. To better understand the influence of the search budget over the outcome of the techniques for RQ5.3, we executed EvoSuite using the *GA* and *Random+* configurations with an increased search time of ten minutes and measured the level of coverage at one minute intervals.

We conducted the University of Sheffield's HPC Cluster (http://www.shef.ac.uk/wrgrid/iceberg). Each node has a Sandy-bridge Intel Xeon processor with 3GB real and 6GB virtual memory. We used EvoSuite's default configuration and ran it under Oracle's JDK 7u55.

5.3.4 Threats to Validity

Threats to the *internal validity* of our study include its usage of only one test generation tool (EvoSuite). While this was deliberate to facilitate a more controlled, fair comparison, it is plausible that specific implementation choices made in EvoSuite may limit the extent to which our results generalise (an associated external threat). The size of the test suites, for example, may influence the comparison; whereas *Random+* has no constraint in the test suite size, *GA* evolves test suites with limited size (100 test cases by default) which imposes boundaries in the search space.

The initial population of individuals and their evolution depend on the values of several parameters for *GA*. Results might thus be affected by the specific parameter values that we used in the experiments. This threat is relatively relevant for *GA*. In fact, *GA* has been used for long time in EvoSuite and all the parameters are well optimized to address test case generation.

Another threat to internal validity stems from the branch-classification analysis described in Section 5.3.2, which can mis-categorize branches in certain cases. We acknowledge the results of this analysis may only be approximate, but while testing the experimental setup we validated that the analysis categorized all branches correctly. Furthermore, chance can affect the results of randomized

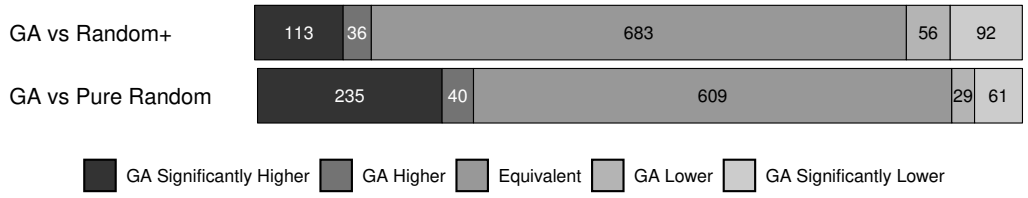


Fig. 5.3 Comparing *GA* performance with *Pure Random* and *Random+* over the 980 SourceForge classes.
("GA Significantly Higher" is the number of classes for which *GA* obtained significantly higher coverage than *Random+* over the 100 runs of the experiment; "GA Higher" – the number of class where a higher average coverage was obtained (but not significantly); "Equivalent", the number of classes where the average coverage level was the same, etc.)

search algorithms. To mitigate this threat, we repeated all experiments 100 times.

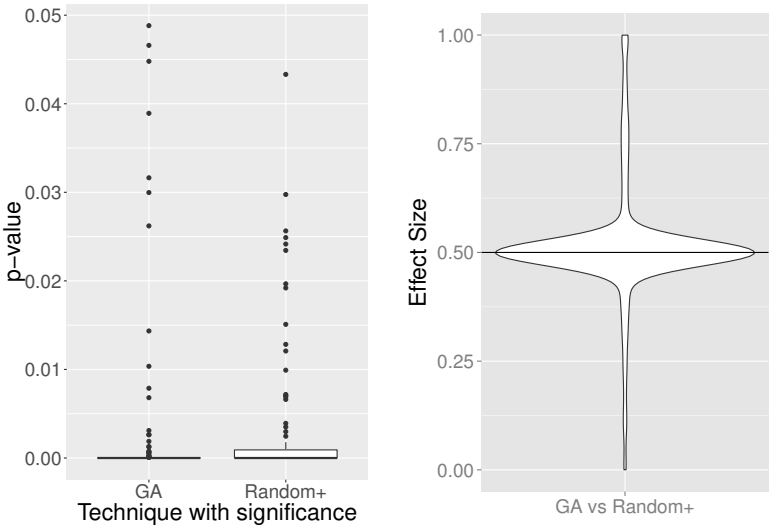
Threats to *external validity* affect the generalisation of our results. While we used a randomly selected sample of Java classes as subjects, our results may not generalise beyond the SourceForge project repository or to other programming languages/paradigms. Furthermore, we also used branch coverage as a proxy measure of the quality of the resulting test suites: results may vary for other test suite properties (e.g., size, length or fault detection ability).

5.4 Random or Genetic Algorithm Search for Test Suite Generation?

5.4.1 RQ5.1: Coverage Effectiveness.

On average over the 100 repetitions of the experiments, *GA* attains 67.59% branch coverage, *Random+* 67.12%, while *Pure Random* obtains 65.22%. Figure 5.3 summarizes the number of classes for which *GA* achieved a significantly higher or lower level of coverage than *Pure Random* and *Random+* over the 100 repetitions of the experiments. (We computed significance using the Mann-Whitney U test at a level of $\alpha = 0.05$.)

When comparing *GA* to *Random+*, there are 113 classes for which the *GA* achieves significantly higher coverage, and 92 classes on which *Random+* attains significantly higher coverage than *GA*. Figure 5.4a plots the p-values for the



(a) Box Plot **(b) Violin Plot**

Fig. 5.4 Comparing the performance of *GA* with *Random+*. (a) Box plot of p-values for classes where a significantly higher level of coverage was achieved with either the *GA* or *Random+*. (b) Violin plot of the effect sizes obtained using Vargha-Delaney's $\hat{A}_{12}$ statistic, here computing the proportion of the 100 repetitions for which the *GA* scores a higher level of coverage than *Random+* for each class; thereby reflecting their relative effectiveness.

significant cases for the *GA* and *Random+* comparison showing that the majority of cases are highly significant (particularly in the *GA* case) and thus unlikely to represent type-I errors.

We observe further similarities in the coverage achieved by *GA* against *Random+* with Figure 5.4b, which shows effect sizes computed with Vargha-Delaney's $\hat{A}_{12}$ statistic [172]. Here, the effect size estimates the probability that a run of *GA* achieves higher coverage than *Random+*. A value of $\hat{A}_{12} = 0.5$ indicates that both search strategies perform equally, $\hat{A}_{12} = 1$ indicates that all runs of *GA* will achieve higher coverage than *Random+*, and vice versa for $\hat{A}_{12} = 0$. The overall average effect size amounts to 0.51, which indicates that *GA* is only very marginally more effective.

For 683 classes *GA* and *Random+* achieve identical coverage. To a large extent, this can likely be attributed to the simplicity of these classes: *GA* achieves 100% coverage on 385 classes, and *Random+* on 398 classes. Classes with lower but identical coverage are likely classes where the possible coverage

is maximized, but less than 100% because of problems that EvoSuite cannot overcome regardless of search algorithm (e.g., due to environmental factors such as classes depending on databases or web services that were not available during the experiments).

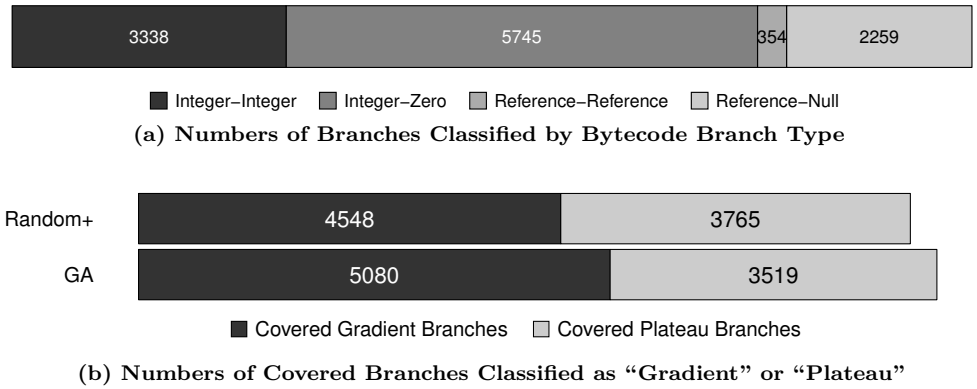
The comparison between *GA* and *Pure Random* shows larger differences, with 235 classes where *GA* achieves significantly higher coverage. This indicates that optimizations such as constant and dynamic seeding, which are used in *Random+*, are effective and help covering non-trivial classes.

RQ5.1. Our experiments showed no significant difference between *GA* and *Random+* in 79% of the classes.

RQ5.2: Influence of Branch Types. Although the comparison between *GA* and *Random+* showed 683 classes with no difference in coverage, there were also 205 classes with significant differences. RQ5.2 aims to shed light on these differences by studying the influence of different types of branches in a class on the effectiveness of the search algorithms.

Figure 5.5a shows the distribution of different branch types as taken from the bytecode of the classes. In total, there are 11,696 branches in the 980 classes. "Reference-Reference" branches are rare: this is not surprising as in most cases in Java a comparison is performed using the `equals` method on the objects, rather than comparing references. "Reference-Null" comparisons are more common accounting for approximately one quarter of the branches. Almost half of the branches (5,745) are "Integer-Zero" branches, from which only 303 involve `double`, `float` or `long` comparisons. Only these 303 branches, along with the 3,338 "Integer-Integer" branches have the potential to provide gradients.

Effectiveness on Gradient Branches. Intuitively, one would expect that the evolutionary algorithms should achieve higher coverage on gradient branches, as the branch distance values will influence the search operators and guide the search towards covering additional branches. Figure 5.6a compares *GA* against *Random+* in terms of the coverage achieved when only considering gradient branches; that is, the coverage is only calculated for classes that have at least one gradient branch, and the coverage values exclude non-gradient branches. There are 99 classes where *GA* achieves significantly higher coverage of the



(a) Numbers of Branches Classified by Bytecode Branch Type

(b) Numbers of Covered Branches Classified as "Gradient" or "Plateau"

Fig. 5.5 Numbers of different branch types in the classes under test.

gradient branches, with only 20 classes where the coverage is significantly lower. Figure 5.5b shows that overall the *GA* covered 5,080 gradient branches, whereas *Random+* covered only 4,548. This confirms that the *GA* benefits from the branch distances provided by the gradient branches.

The 20 cases where *Random+* has significantly higher coverage than *GA* can be explained by their large number of branches (73 total goals and 23 gradient branches on average): The fitness function that guides the *GA* considers all branches at the same time; this means that a test suite that is close to covering many gradient branches may have a better fitness value than a test suite that fully covers fewer branches. In these cases, the *GA* would simply require more time to eventually fully cover all these branches.

Effectiveness on Plateau Branches. Figure 5.6b compares *GA* against *Random+* when only considering the coverage of plateau branches. There are 109 classes in which the *GA* has significantly lower coverage compared to *Random+*, and 92 classes with significantly higher coverage. Figure 5.5b shows that overall the *GA* covered 3,519 plateau branches respectively, whereas *Random+* covered 3,765; that is, even though the *GA* covered more branches overall, they covered fewer plateau branches. Since the branch distance for these branches only has two values there is no guidance that the *GA* could exploit – a plateau branch is either covered or it is not covered. A possible conjecture is a loss of diversity of the evolutionary search algorithms compared to the random search: While *Random+* continuously creates independent new objects and call sequences, *GA* spends more time exploring the neighbourhood if existing individuals. In addition, the *GA* in EvoSuite prefers smaller test suites (when two test suites have the same

fitness value, they are ranked by size) and thus further exacerbating the removal of random "noise", focusing the search operators on the exploitation of achieved coverage and mutating existing objects.

Effectiveness on Branchless Methods. Branchless methods represent a special case similar to plateau branches, and intuitively they are simple to cover – they just require test cases to call the method, without any need to search for specific parameter values. Figure 5.6c compares *GA* against *Random+* with respect to the coverage of methods. Although *GA* achieves significantly higher coverage than *Random+* in 24 cases, there are 63 classes where the *GA* results in lower coverage, which is similar in proportions to the plateau branches. It is maybe surprising that there can be a difference in so simple coverage goals in the first place. Our conjecture is that this is because *Random+* has a higher probability of inserting new method calls: *GA* only mutates a test suite with a certain probability, and then each test in turn is only mutated with a certain probability, and finally insertion of new statements again does not always happen. In contrast, *Random+* generates tests by repeatedly adding new statements. Again it would only be a matter of time for evolutionary search to fully cover all branchless methods, although possibly more time than for *Random+*. Interestingly, classes on which the *GA* achieved more than 90% coverage have a median proportion of 100% branchless methods out of all coverage goals, providing further evidence that many classes in practice are trivial.

RQ5.2. Our experiments show that *GA* achieves higher coverage of gradient branches compared to *Random+*, but lower coverage of plateau branches, which constitute the majority of branches.

5.4.2 RQ5.3: Effects of the Time Allowed For the Search.

The results so far have shown that *GA* and *Random+* perform similarly for the majority of classes after two minutes of search, with some differences in performance on plateau and gradient branches. This raises the question whether the results are influenced by the allocated search budget – given more time, do the results change?

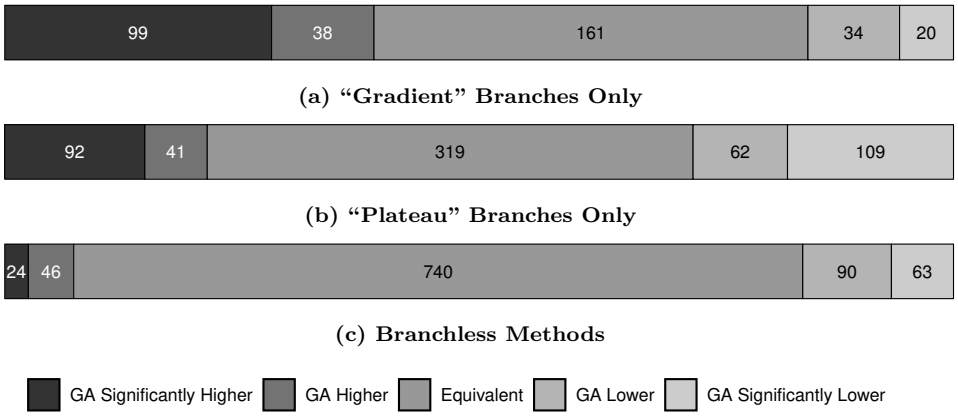


Fig. 5.6 Comparing *GA* performance with *Random+* for different types of branch and with branchless methods.
("GA Significantly Higher" is the number of classes for which *GA* obtained significantly higher coverage than *Random+* over the 100 runs of the experiment; "GA Higher" – the number of class where a higher average coverage was obtained (but not significantly); "Equivalent", the number of classes where the average coverage level was the same, etc.)

To analyse the impact of the search budget, we repeated the experiments with *GA* and *Random+* using an increased search budget of 10 minutes, and measured the coverage values at one minute intervals. Figure 5.7 compares the average coverage per class for each interval: There is a slight increase of coverage for both *GA* and *Random+* over time, and after 10 minutes *GA* achieves an overall average of 69.81% branch coverage, while *Random+* achieves 68.95%.

Given more time, *GA* will catch up on branchless methods and plateau branches covered compared to *Random+*. Figure 5.8 compares *GA* with *Random+* after 10 minutes, and shows that the *GA* has significantly lower coverage on only 83 classes after 10 minutes, compared to 92 after two minutes (Note that the number of classes with coverage data after 10 minutes is only 974, as there were 6 additional classes for which EvoSuite did not produce any data after 10 minutes). The *GA* will also continue to optimize gradient branches; however, the dynamic seeding used in EvoSuite will also help *Random+* in many cases to cover gradient branches. Figure 5.8 shows that there are 131 classes where *GA* has higher coverage after 10 minutes, compared to 113 after two minutes. For 760 classes the coverage is identical, which is likely because the maximum achievable level of coverage has been reached by both algorithms.

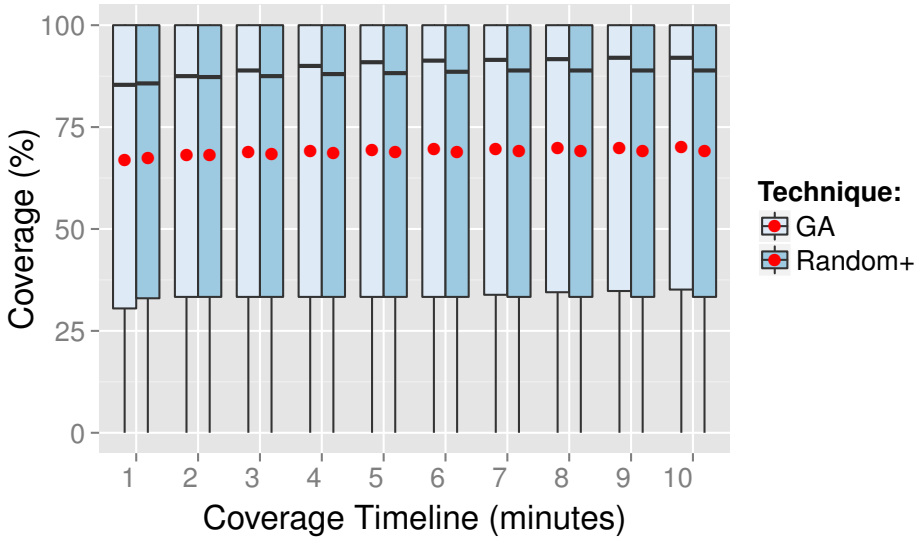


Fig. 5.7 Branch coverage comparison between *GA* vs. *Random+* over 10 minutes with one minute intervals. Dots represent mean averages.

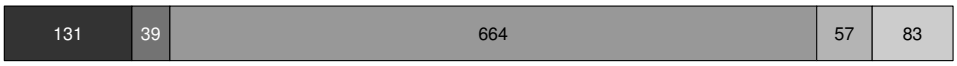


Fig. 5.8 Comparing branch coverage performance of *GA* against *Random+* using a search budget of 10 minutes. (Legend is as for Figure 5.6.)

RQ5.3. The coverage increase is higher for *GA* than for *Random+* over time, suggesting that the disadvantage on plateau branches is overcome, although the coverage increase is small in absolute terms.

5.5 The Impact of Branch Types

Our results indicate that, while the techniques have a similar outcome for the majority of classes, there are differences that influence the effectiveness. However, the fact that *Random+* can outperform the *GA* in a number of subjects raises the question of why this happens in practice. In this section we look at some of the factors in the search algorithms that may be the cause of these surprising results.

The analysis of RQ5.2 also suggests that the search operators of the *GA* have an effect on the diversity: The *GA* has a lower probability of generating

new tests, and may thus be slower at covering plateau branches or branchless methods (compare Figure 5.6, part b and c).

A further influencing factor is that a test suite produced by the *GA* may not cover all branches that were covered throughout the search. This is because the fitness function aims to maximize coverage: For instance, given a test suite T_1 that covers goals $\{A, B\}$, and another test suite T_2 that covers goals $\{B, C, D\}$, assuming T_2 has a better fitness value it will be selected as the best solution. As a result, although goal A was covered by T_1, it remains uncovered in the resulting test suite. In contrast, *Random+* generates a new test case on each iteration, and if the new test covers any new goals, it is added to the test suite. This suggests that creating an archive of solutions that cover new coverage goals would be important for the *GA*.

The large number of plateau branches could potentially be reduced by introducing testability transformations [74]; although EvoSuite implements certain transformations (e.g., on floating point numbers or string comparisons) it does not apply a transformation of boolean flags.

5.6 Related Work

There have been several papers that have compared GAs with random search in the procedural domain (e.g. [76], [177]). This work has found guided search to always outperform random. In general, procedural code tends to consist of larger functions than methods in object oriented code, and each function tends to involve more parameters. While random search typically covers a large percentage of the branches involved, the GA covers significantly more.

Sharma et al. [158] showed on 13 examples that random testing of object oriented container classes achieves the same coverage as shape abstraction, a systematic technique specific for container classes. The results of our experiments suggest that in practice, many object oriented classes are, similarly to container classes, simple in nature and thus well suited for random testing.

Earlier experiments with EvoSuite on the SF100 corpus [52] showed that a large number of classes are either trivially covered, or uncoverable without providing the test generator with additional features (e.g. to handle environmental

inputs such as web services or databases). This finding is in line with our results; however, a comparison with RANDOOP [124] in the same study suggested a large improvement of *GA* over random testing. The results of our experiments suggest that this improvement is largely due to the engineering of the tool rather than the search algorithm; for example, RANDOOP does not use seeding.

Eler et al. [44] analysed the SF100 corpus from the point of view of test data generation using dynamic symbolic execution. They also reported the large number of reference comparisons and the challenges of handling those in a constraint solver. They further reported the relatively low number of branches involving integer comparisons, which result in constraints that DSE is typically strong at handling. These findings are in-line with our general observations from SF110, despite the fact that we only used a sample of 1,000 classes in this study.

5.7 Conclusion and Future Work

In this chapter, we presented an empirical study comparing the effectiveness of evolutionary and random search-based algorithms for generating branch coverage test suites for real-world Java classes. One might expect algorithms such as a *GA* to vastly outperform random search for this task, but surprisingly we observed that all algorithms behaved similarly on the majority of classes, in particular when applying optimizations such as seeding of constant values, which applies to random search just as well as to evolutionary search in the domain of test generation. Although evolutionary search algorithms can exploit the guidance provided by certain types of branches, in practice there are many more branches that provide no such guidance, and on some classes with many such branches *GA* resulted in lower coverage than random search – even when a large search budget was used.

These results are in-line with our observations in Chapter 4. For instance, in Section 4.3.8 we saw that on the Apache Commons Math – a mathematics library with a large proportion of numerical comparisons – our GA-based approach was significantly more effective. Therefore, as we conjectured in the previous chapter, the high success rate of random-search is not specific to the problem of regression testing, but further generalises to other search-based testing techniques

such as evolutionary algorithms on object-oriented programs. Therefore, it is likely that further improvements on the GA will also result in an improvement in fault detection ability, for our search-based approach in Chapter 4.

Our findings suggest several specific areas for future work in order to improve the effectiveness of evolutionary search algorithms for the task of unit test generation:

- Our experiments with EvoSuite's *GA* used a basic implementation of the search algorithm. However, there are various attempts to extend this *GA* to a memetic algorithm, such as by applying dynamic symbolic execution as a type of local search [58], or using specifically designed local search operators [53]. While these local search operators would mainly benefit the search on gradient branches, the overall effects in comparison to a random search would need to be studied in detail.

- The high number of plateau branches suggests that testability transformation [74] could be used to convert some of these branches to gradient branches. While initial experiments on EvoSuite [100] showed the potential of this approach, a significant engineering effort remains to be done before the effects can be studied at large scale.

- The analysis of RQ5.2 suggests that the search operators of the *GA* have an effect on the diversity: While random search constantly generates new tests, these evolutionary search algorithms spend more time exploring the neighbourhood of existing tests through mutation, which may lead to less diversity, and negative effects on covering plateau branches or branchless methods (cf. Figures 5.6b and 5.6c).

- Future work could also explore the possibility of adapting the search to the specific fitness landscape of the problem at hand, and controlling search parameters such as the mutation rate. For instance, if a class appears to have mainly plateau branches, then the mutation rate could be increased.

From a practical point, however, if the objective is simply to quickly achieve a decent level of branch coverage on object-oriented classes, then our results suggest that using random search with seeding may be sufficient.

Chapter 6

Disposable Testing: Detecting Changes during Software Evolution without Keeping and Maintaining Generated Test Suite

The content of this chapter is based on work undertaken during this PhD by the author, parts of which have been published elsewhere [154].

6.1 Introduction

In Chapter 1 we set out with the idea of assisting developers with their regression testing efforts by automatically generating change-revealing tests on demand. We also evaluated this idea in Chapter 4. In this chapter, we look at how one can apply such techniques in practice, and whether what we envisioned in Chapter 1 is actually feasible.

As we discussed in the previous chapters, a standard approach to applying automated test generation techniques in practice is to take a version of a class as input, generate a set of tests that capture its behaviour with assertions, and

then to repeat the execution of these tests after a change has been made. If a test passes on one version but fails after a change, then it exposes a difference in behaviour. An example of such a generate-and-maintain approach to testing is *Coverage-driven testing*, where tests are generated with the aim of covering the code of the class. Resulting tests need to be maintained alongside the evolving program, just like manually written tests, but maintaining tests can be tedious and challenging, in particular for the often lengthy and non-familiar automatically generated tests.

We propose *disposable testing* as an alternative approach to eliminate the need for maintaining tests: Completely new tests are generated every time the program under test is changed. Developers are only shown those generated tests that actually reveal a behavioural difference caused by the program change, and decide whether this difference is intended or not. After this, the generated tests are thrown away.

However, throwing away tests may counter the objective of maximizing coverage, as automated test generation benefits from being applied incrementally over time [29]. This raises the question of whether *Differential testing* [108, 45] is better suited to implement disposable testing: With differential testing, a test generator receives two program versions, before and after a change, as input, and derives tests that demonstrate behavioural differences. We implemented a differential testing approach in the EvoSuite [48] framework, and conducted a set of experiments to compare it with EvoSuite's coverage-driven testing approach — with coverage-driven testing as an example of a traditional generate-and-maintain approach and differential testing as an example of disposable testing. We evaluated the approaches with the Defects4J [89] dataset of real faults, based on the framework we presented in Chapter 3.

The concept of disposable testing may at first seem counter-intuitive to developers, who usually like to keep as many as possible tests, resulting in huge and ever growing test suites. Indeed the idea of disposable testing raises several questions: When throwing away all tests rather than maintaining them, do we run the risk of missing some bugs? To answer this question, we compare coverage-driven testing and differential testing in terms of how well they perform at identifying bug-inducing changes.

RQ6.1: How do coverage-driven testing and differential testing perform at detecting changes?

For disposable testing to be a viable approach, a key requirement is that the generated tests would not be effective at finding future faults, and as such, can be thrown away. However, considering that code that changes often is more likely to contain faults [66], would tests that reveal faults not be particularly important to keep around and serve as valuable regression tests? To answer this question, we evaluate whether keeping tests generated by coverage-driven testing and differential testing is helpful in identifying subsequent changes using mutants as well as developer changes taken from the commit history of the programs under test.

RQ6.2: How do test suites generated by coverage-driven testing and differential testing perform at revealing subsequent changes?

Although no human effort is needed for test maintenance in disposable testing (i.e., generated tests are discarded rather than maintained), effort is required to inspect the generated tests before they are discarded. That is, developers need to inspect whether the test has detected a regression fault or an intended change. The question is, therefore, will the effort not spent on test maintenance merely be eclipsed by the additional effort that must be spent inspecting the generated tests? To answer this question, we compare the number of tests that need to be manually inspected during maintenance with coverage-driven testing and during disposable testing with differential testing.

RQ6.3: How does the maintenance effort of tests generated by DT for disposable testing compare to those evolved using coverage-driven testing?

The contributions of this chapter are as follows:

- We propose *disposable testing* as an alternative to the traditional generate-and-maintain approach.
- We evaluate whether differential testing can be used to implement disposable testing. This is the first large-scale evaluation of the effectiveness of differential testing, using the real faults taken from the Defects4J repository.

- We empirically and qualitatively compare coverage-driven testing and differential testing in detail in terms of fault finding ability and effectiveness of the generated tests during regression testing.
- We study the implications on human inspection effort when applying coverage-driven testing and differential testing.

The rest of this chapter is structured as follows: first we detail the methodology used for the study. Thereafter, the answer to the three research question we reviewed earlier follows. Finally, in Section 6.6 we conclude this chapter.

6.2 Methodology

In order to answer our three research questions on disposable testing, we conducted an elaborate empirical study, comparing coverage-driven testing, a standard approach of generate-and-maintain test generation, with differential testing, which we conjecture is most suitable for disposable testing. In this section, we describe the details of the experimental setup and methodology.

6.2.1 Test Generation Techniques

As a representative test generation tool for coverage-driven testing approaches, we used EvoSuite [48], which is a state-of-the-art test generation tool that uses search algorithms to generate coverage-optimized unit test suites, and is effective at fault-finding[144].

To be able to compare this to differential testing, we extended EvoSuite as follows: We added a second Java class loader which gives EvoSuite access to two different versions of the same class. Tests are then simultaneously executed on both versions of the same class. In line with previous work on behavioural regression testing [123], we then used random search [157] to find behavioural differences. The search operators in EvoSuite were modified slightly to make use of these two versions: First, only the intersection of the public interfaces of the two versions of the class are used for testing, such that any generated test would be syntactically valid on both classes. That is, if a method was added

Table 6.1 The total number of bugs in each DEFECTS4J project and the number applicable to and used in each research question

Project	Bugs		
	Total	**RQ6.1/6.2-M**	**RQ6.2-D/6.3**
Apache Commons Lang	65	65	31
Apache Commons Math	106	96	37
Google Closure Compiler	133	120	93
JFreeChart	26	24	3
Joda-Time	27	22	19
Total	357	327	183

or removed as part of the change, it was not included in the test generation, as it would be trivial to show a difference between the two versions of the class. Second, EVOSUITE's use of constant seeding, which helps with the tool's efficiency [49, 137], was modified to include values from both versions of the Class Under Test (CUT). The search objective was defined by comparing the test executions on the two versions of the class; any differences in return values, publicly visible state changes (e.g., by querying object states through inspector methods), or differences in the exception behaviour, are considered as revealing a difference. Any such difference is captured in a regression assertion, and at the end the search returns all tests found throughout the search that successfully show a difference between the two versions.

To more closely match the behaviour of EVOSUITE, tests returned are minimized with respect to the regression assertions. That is, a test is minimal if removing any of its statements would lead to the change no longer being revealed by the test (i.e., the test passes on one version but fails on the other). Given that differential testing relies on random test generation and several failing tests may be generated for the same behavioural change, similar to existing techniques [35] we grouped/filtered failures. However, our implementation is more basic and only limited to grouping assertions that check the output of method calls.

6.2.2 Subject Programs

To be able to experimentally answer the research questions, we required a set of program pairs with known, semantically non-equivalent changes. In particular,

to increase the practicality of our results, we were interested in changes that represent real faults. Therefore, we chose the DEFECTS4J [89] dataset, which is a collection of 357 real bugs taken from five open source projects, as detailed in Table 6.1.

6.2.3 Experiment Procedure

To answer RQ6.1 (How do coverage-driven testing and differential testing perform at detecting changes?), we needed to ensure that all modified classes contained a detectable change. Therefore, we first selected those bugs out of the DEFECTS4J dataset in which only one class was modified. This resulted in a total of 327 bugs (i.e., changes) to analyse for this research question overall. We then applied test generation to each pair of (buggy, fixed) classes as provided by DEFECTS4J. The fixed version in DEFECTS4J is the version following a bug fix (i.e., in the version history, the fixed version is committed after the buggy version). Therefore, there may be code or branches present in one version but not in the other version. While this would not be problematic for differential testing given that it takes both versions of the class as input, coverage-driven testing only aims to generate tests covering one version of the program under test. To address this, we evaluated coverage-driven testing in two different ways, for each bug in DEFECTS4J considering a buggy version of a class A and the fixed version B:

1. "A to B" (denoted by $CT^{\rightarrow}$). Tests were generated on A, and the tests were executed on B, to see if the change could be detected. This is test generation based on the actual temporal order of versions of the classes underlying the DEFECTS4J bugs.

2. "B to A" (denoted by $CT^{\leftarrow}$). Tests were generated on B and executed on A, to see if the change could be detected. This is a standard regression testing scenario, where the version on which tests are generated represents the correct behaviour, and the tests are checked on the "buggy" version to see if that bug is detected.

For RQ6.2 (How do test suites generated by coverage-driven testing and differential testing perform at revealing subsequent changes?), we conducted

two experiments: **(RQ6.2-M)** aims to answer this using *mutations* applied to the same area of code that was *previously modified*, and **(RQ6.2-D)** using subsequent changes made by developers.

To answer RQ6.2-M, in order to understand the effectiveness of the tests generated by the techniques at detecting other changes in the area modified between the pair of classes, we looked at mutations that can be applied to the modified lines of code. To achieve this, we used the Major Mutation Framework [88] (v1.2.1) to generate all mutants on all classes used in RQ6.1. We then selected all mutants that were applied on the lines changed between the two versions of the classes and discarded mutants related to lines of code that were not modified between the two versions of the program. Thereafter, we selected all test suites generated in RQ6.1 which were effective at revealing the original change (i.e., between the buggy and fixed versions of the program), and applied the test suites on the mutants to assess their effectiveness at killing the mutants.

To answer RQ6.2-D, we considered a scenario in which test generation is applied throughout software evolution, such that the test suite is augmented with additional tests after every change made by the developers. To select suitable classes, we referred to the open source projects themselves, and from their online version control repositories, we extracted all versions of the programs prior to the buggy version. Thereafter, we selected subjects which had at least 10 commits changing the faulty class (i.e., the class containing the fault between the buggy and fixed versions). This resulted in a smaller number of 183 bugs to analyse for this particular research question. Our decision to choose 10 commits was mainly to balance out the number of subjects and commits, that is, a large number of commits such as 20 would result in a considerably smaller number of subjects (112). The resulting test suites were also used to answer RQ6.3. Test generation was then applied for each technique as follows:

a) For coverage-driven testing, the objective is to increase coverage over time, such that when the program is modified, new tests are generated to cover the new changes. Campos et al. [29] extended EvoSuite to support this: After each commit to a version control system, EvoSuite is invoked on a subset of classes (e.g., classes that have been modified, or classes without full coverage) using previously generated test suites as the starting point of the generation.

Using this, we applied continuous test generation using EvoSuite to generate and augment test suites up to, but not including, the commit prior to the bug fix.

b) Since differential testing has not previously been used in the context of software evolution, we adapted the coverage-based strategy: After each change (i.e., commit to the version control system) we applied the differential testing extension of EvoSuite to each pair of classes before and after the change. Any resulting tests by definition reveal a change between two versions of the program (i.e., the tests pass on one version and fail on the other one). We repeated this on all pairs of commits up to, but not including, the fixed version – otherwise the change may have been revealed by executing differential testing on the last pair (buggy, fixed) as in RQ6.1 – and collected all generated tests along the way in a common test suite.

During the test generation process, as we evolved the test suites after each commit, we executed the test suite generated so far on the new commit, and discarded any failing tests. This is necessary as manually inspecting every failing test would make the experiment infeasible; this results in a worst case scenario where all changes are intended and the tests are always "wrong" and need to be fixed or deleted. However, this is a reasonable scenario as in our experimental setting the deleted tests are usually replaced with new tests improving coverage, or revealing the behavioural difference. Finally, we evaluated the effectiveness of the resulting test suites at detecting the difference between the buggy and fixed version.

To answer RQ6.3 (How does the maintenance effort of tests generated by DT for disposable testing compare to those evolved using coverage-driven testing?), we again assumed a scenario where all changes are intended (i.e., all failing tests need to be fixed), and to eliminate the threat of over-estimating the maintenance cost we assumed that the developers pick the easiest choice for maintaining the test: removing any failing test. This makes the number of failing tests the lower bound of the number of tests developers need to inspect; in practice, tests that are repaired may lead to additional failures later, requiring additional inspection effort. We then applied disposable testing using differential testing during software evolution using the subjects from RQ6.2-D, and compared

the number of failing tests by differential testing with the number of tests thrown away by coverage-driven testing during test suite evolution (RQ6.2-D).

Test Execution and Flaky Tests: Because both test generation techniques are based on randomized algorithms, we repeated all experiments 30 times, and allocated two minutes for each run. Given that tests generated by coverage-driven testing may propagate the change to the output but not report it with an oracle, we followed previous methodology [156] and enabled coverage-driven testing to generate all assertions. We conducted the experiments on University of Sheffield's HPC Cluster[1]. Each node has a Sandy-bridge Intel Xeon processor, and each experiment was allocated 3GB real and 6GB virtual memory. We used EvoSuite's default configuration and ran it under Oracle's JDK 8u71.

Occasionally, some tests generated by automated test-generation tools either do not compile, or their passing/failing behaviour is "flaky" (unstable). The flaky outcome of the test can be due to reasons such as environment dependencies (e.g., system time). To ensure that any reported failures are not due to flaky tests, we followed the methodology used in previous experiments [156], such that after removing non-compiling tests, we repeatedly executed all test cases on the version they were generated on until every test case passed at least 5 times consecutively. Any failing tests during the repeated executions were removed.

6.2.4 Experiment Analysis

After the execution of each test case on the second version of the program, we collected the number of failing tests for each generated test suite. We assume a test suite has detected the change if any test exists that passes on one version and fails on the other version of the program.

We recorded the number of runs in which a technique detected a change in each of the 30 repetitions. We then performed statistical tests to determine whether one technique was significantly more successful than another by using Fisher's Exact test [46, 94]. We say a technique successfully detects a change in a significantly higher number of runs if the p-value of the test is less than 0.05.

[1] http://www.shef.ac.uk/wrgrid/iceberg, accessed August 2016.

For the mutation analysis, to calculate the significance of the difference between the mutation score achieved by the techniques, we used the Mann Whitney U test with $\alpha = 0.05$.

To better understand the relationship between the coverage of the modified code across the two versions by the test suites and the effectiveness of the technique at detecting the change, for both RQ6.1 and RQ6.2-D, we executed all generated test suites on the two versions of each subject, and measured whether or not the changed lines were covered by the test suites. We also measured the level of branch coverage achieved by the tests on the CUT using Cobertura[2] on all generated tests, including the intermediate tests generated during test suite evolution in RQ6.2-D.

6.2.5 Threats to Validity

There are several threats to validity that are naturally inherent to a study such as this, which we detail as follows.

External Validity: Our subjects were selected from only five open source projects, and as such our results may not generalise to other projects. Only single-class changes were considered, and the techniques may perform differently when several classes are changed or a large number of changes are introduced. For our RQ6.3, only 10 commits were considered, and it is possible that with a larger number of commits, one technique may generate more robust regression test suites than the other. Another threat to the validity of the continuous experiment is that differential testing only generates tests when it can detect a difference, while there can be many commits that either do not change the code (e.g., documentation, styling changes), or that the changed code is a simple refactoring which is semantically equivalent. In comparison, executing coverage-driven testing on the same semantically equivalent code is likely to result in a higher-coverage test suite. To lower this threat we analysed the commit history on the CUT, and removed commits that contained only documentation or white-space changes, and selected only commits that modified the code. However, it is still possible that some commits in the history are semantically equivalent changes.

[2]http://cobertura.github.io/cobertura, accessed August 2016.

The different algorithms used by the techniques to generate the tests (i.e., coverage-driven testing using a genetic algorithm while differential testing using random test generation) also poses a potential threat. To mitigate this, we looked at and report on the coverage of the code that was modified between the two versions of the program, which can indicate whether or not the lack of effectiveness is due to the lack of coverage (especially for the case of coverage-driven testing). The different approaches taken by the techniques for generating oracles (i.e., assertions) is also a threat, especially for RQ6.2 and RQ6.3. Specifically, considering that other strategies [56] may reduce the effectiveness of coverage-driven testing, we are using oracle generation strategy intended for regression testing [187], as also used in other test generation tools such as RANDOOP [124] and also in previous studies [156].

Internal Validity: One threat to the internal validity of this study is the use of only one tool—EVOSUITE. While this decision was deliberate to have a more controlled comparison, the results may not generalise to other coverage-driven test generation tools. Furthermore, due to the large number of tests (close to $150,000$), not all tests and their execution results could be manually inspected to avoid false positives. To mitigate this, we followed the approach in [156] and compared the failure reason of the generated test suites with the test suites written by the developers of the projects, and in cases they differed, we manually inspected the tests. It is also possible that some test failures were false positives due to bugs in our automated setup or our extension of EVOSUITE, which we tried to mitigate by testing the software in addition to manual and automated sanity checks. Inaccuracies or bugs within the tools used during this study (e.g., Cobertura, EVOSUITE, Major) may also pose threats to the validity of this study. However, we believe that given that the tools have been standardized in the research community, and also considering the large number of subjects along with the 30 repetitions mitigates this threat to some extent.

Construct Validity A threat to the construct validity is that we only used bugs from DEFECTS4J, which consists of two versions: a version containing a bug, and another where the bug is fixed, and these pairs of versions may not represent real regression faults. Moreover, given that the bugs were collected from open source repositories, the bugs may not represent the regression faults that the developers may have fixed before publishing their changes to the repositories.

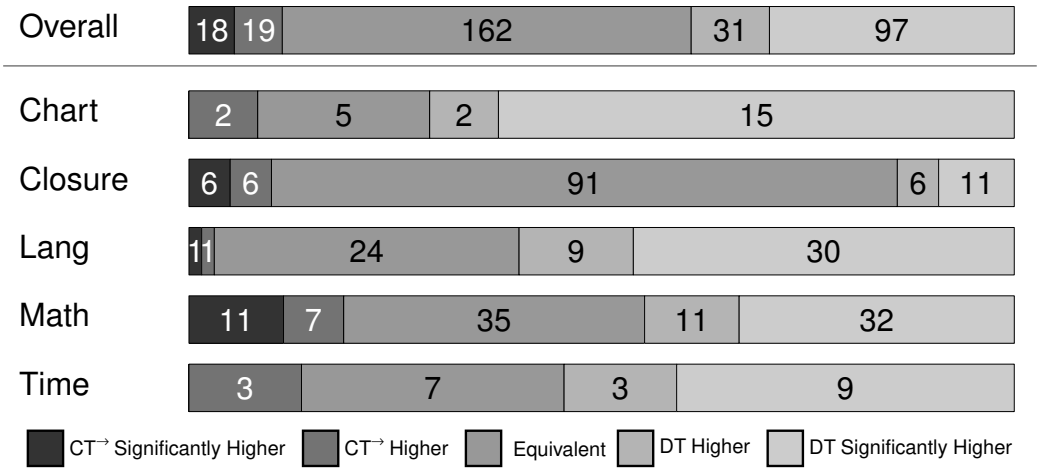


Fig. 6.1 Comparing $CT^{\rightarrow}$ and differential testing in regard to their effectiveness at detecting the change between the buggy and fixed versions of classes in Defects4J.

Legend: "Significantly higher" is the number of classes for which a technique generated a test that detected the change in a significantly higher number of runs than the other; "Higher" refers to the number of classes where a technique generated a test detecting the change in a (non-significant) higher number of runs; "Equivalent" is where the change was detected by both techniques for the same number of runs.

Specifically, the developers of the projects in Defects4J have access to large regression test suites that may have helped them to avoid trivial regression faults. Therefore, the effectiveness of the techniques may be underestimated.

6.3 Answers to RQ6.1 (Detection of Changes To Classes)

The first research question considers the performance of coverage-driven testing and differential testing in isolation; that is, when given a pair of program versions. Evaluated on the 327 bugs from Defects4J that only change an individual class, differential testing managed to detect 152 bugs at least once (46%), while coverage-driven testing based on the pre-change version (buggy version, $CT^{\rightarrow}$) found 124 (38%) bugs at least once in all repetitions of the experiment.

Although the numbers of bugs found by the techniques at least once seem close, the techniques differ in how consistent they are at finding the bugs. Figure 6.1 summarizes the number of subjects for which one technique detected the change in a significantly higher number of experiment repetitions than the other. Both

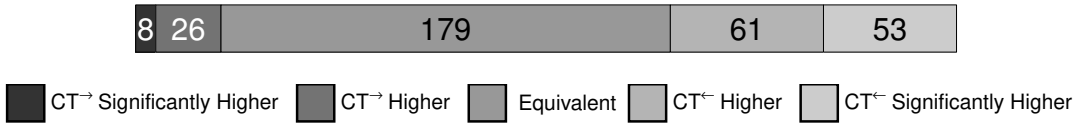


Fig. 6.2 Comparing $CT^{\rightarrow}$ and $CT^{\leftarrow}$ in regard to their effectiveness at detecting the change between the buggy and fixed versions of classes in DEFECTS4J.
(Please refer to Figure 6.1 for an explanation of the legend.)

techniques performed equally well on a large fraction of the bugs: 212 out of 327. However, out of these, 155 were not found by either of the techniques, while only 7 were always found by both. On the one hand, for the subjects that the techniques achieved statistically different outcomes, coverage-driven testing was significantly more successful than differential testing for 18 (6%) of the bugs. On the other hand, differential testing achieved a significantly higher number of runs in which the change was detected for 97 (30%) of the bugs. This indicates that, although neither of the techniques subsumes the other, for a sizeable portion of the bugs, differential testing is more successful.

RQ6.1: While both techniques managed to detect complementary sets of bugs, differential testing was more effective at detecting 30% *of them.*

Seeing that differential testing was more effective than coverage-driven testing on close to a third of subjects leads us to more questions: What is enabling differential testing to be more effective? Is it the types of the changes in DEFECTS4J that are influencing the outcome of the coverage-driven testing? Are the changes that are not detected by coverage-driven testing covered? Even if coverage-driven testing can cover the changes, why is it not as effective as differential testing in detecting them? We investigate the answer to these questions in the following sections.

6.3.1 The Influence of Testing Pairs of Classes

The nature of bugs in DEFECTS4J is such that the pre-change version is the buggy version, and the post-change version is the fixed version. It might be the case that coverage-driven testing is influenced by this (e.g., intuitively, the fixes may more often add code such as input validation checks). To see whether this is

the case and how this influences coverage-driven testing, we look at the data on coverage-driven testing based on the fixed version of the DEFECTS4J bugs. That is, tests are generated on the fixed version, and a change is detected if a test fails on the buggy version. Figure 6.2 shows the results of this analysis: coverage-driven testing based on the changed version (fixed version) was significantly more effective than coverage-driven testing based on the pre-change version on 53 subjects. A possible conjecture for this result is that, in DEFECTS4J, the fixed versions more often contain additional branching structure that leads to more tests when targeting branch coverage.

To validate this conjecture we studied some of the bugs where the result differs for coverage-driven testing depending on the "direction" of the change (A to B, or B to A). For example, consider `Math-3`[3], which was found by coverage-driven testing based on the fixed version ($CT^{\leftarrow}$), but never found when generating tests for the buggy version. In this subject, two arrays each with *one* element need to be passed to the method in order to trigger the change:

```
public static double linearCombination(final double[] a, final
    double[] b)
        ...
+       if (len == 1) {
+          // Revert to scalar multiplication.
+          return a[0] * b[0];
+       }
```

Given that coverage-driven testing based on the pre-change version has no incentive to generate this specific scenario, it would be unlikely for it to use such inputs for the method, whereas coverage-driven testing based on the changed version is guided towards producing these inputs.

Although the results suggest that on the whole coverage-driven testing applied to the post-changed version may be more effective, there are also 12 bugs in DEFECTS4J that were never found by test suites generated from the changed version, but found when generating tests based on the pre-change version. There were also 8 cases where coverage-driven testing on the pre-change version was better than using the post-change version. For instance, on `Math-100` from

[3]In DEFECTS4J, the notion "`Math-3`" indicates the bug with the ID 3 from the Apache Commons Math project.

the same project as the earlier example, coverage-driven testing based on the pre-change version always finds the fault, but not in the opposite direction. In this subject, no new branches are added or removed, however, a different method `getUnboundParameters()` is called on the input argument:

```
public double[][] getCovariances(EstimationProblem problem)
    ...
-     final int cols = problem.getAllParameters().length;
+     final int cols = problem.getUnboundParameters().length;
```

The key difference between the two methods is that for the test generation tool it is more difficult to generate a `problem` object which does not result in a `NullPointerException` for the new method call. As a result, coverage-driven testing based on the changed version always faces this exception and generates the shortest test that results in the exception being thrown. Conversely, coverage-driven testing based on the pre-change version constructs an object which does not result in an exception within `getAllParameters()`, and thus gains a higher coverage by proceeding further down the `getCovariances` method, which propagates to a different outcome. Therefore, coverage-driven testing based on the pre-change version is more effective in this particular case.

The higher effectiveness of coverage-driven testing when applied on the post-change version also changes the comparison between coverage-driven testing and differential testing: As summarized in Figure 6.3, differential testing was significantly more successful on 65 subjects, while being significantly worse on 23 subjects – compared to 97 and 18 respectively when applying coverage-driven testing in the other direction, as shown earlier. It is difficult to conclude whether the difference between pre-change and post-change result for coverage-driven testing generalises; this might be an artifact of how Defects4J is organized, and it may be that in practice not all changes performed on software may be similar to the bug fixes represented by Defects4J.

The strength of differential testing comes from looking at both versions, since coverage-driven testing is affected by which of the two versions it is applied to.

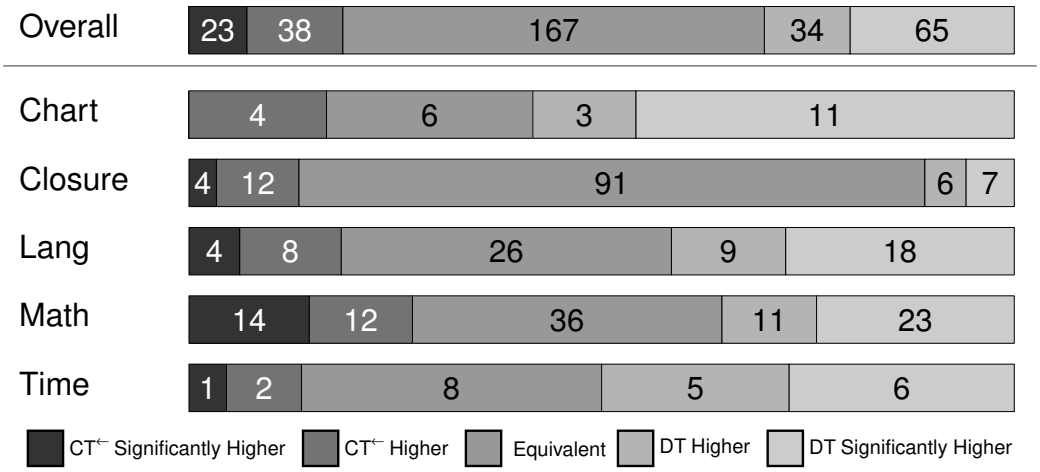


Fig. 6.3 Comparing $CT^{\leftarrow}$ and differential testing in regard to their effectiveness at detecting the change between the buggy and fixed versions of classes in DEFECTS4J.
(Please refer to Figure 6.1 for an explanation of the legend.)

6.3.2 Influence of Optimization for Coverage on Test Suite Generation

Analysis of the branch coverage ratio of the test suites generated by the techniques shows that, on average, coverage-driven testing achieved a coverage ratio of 53%, more than an order of magnitude higher than the 4% attained by differential testing. However, notice that despite the higher coverage achieved by coverage-driven testing, it achieved a lower success rate than differential testing. Given that the overall coverage ratio may be irrelevant to the coverage of the changed area of code between the two version of the program, we investigated whether in our case the higher level of coverage achieved overall translates to covering the changed area of code as well.

To answer this, we first selected the subjects where differential testing detected the change at least once. Then for these subjects, we looked at all test suites generated by coverage-driven testing which failed to detect the change, and observed whether these tests covered the change area of the code. That is, whether the lines modified between the two version of the program were covered. To our surprise, 67% of these test suites which did not find the fault, *fully* covered the code containing the defect, and another 25% covered it *partially*. Looking broadly at all subjects, as shown in Table 6.2, except for Closure, more than

Table 6.2 Coverage of the changed area of code, based on the outcome of the test suite at detecting the change. (e.g. For all test suites that detected the change, whether they fully or partially covered the changed lines between the two versions.)

		Bug Detected		Bug Not Detected		
Project	Config.	Full	Partial	Full	Partial	Not
Chart	$CT^{\rightarrow}$	80.9%	19.1%	69.0%	8.0%	23.1%
	$CT^{\leftarrow}$	83.0%	17.0%	66.1%	15.6%	18.2%
	differential	67.2%	32.8%	6.2%	0.0%	93.8%
Closure	$CT^{\rightarrow}$	66.9%	27.6%	24.2%	13.8%	62.0%
	$CT^{\leftarrow}$	66.8%	17.8%	24.6%	11.5%	63.9%
	differential	68.4%	31.6%	11.3%	0.9%	87.8%
Lang	$CT^{\rightarrow}$	76.6%	20.2%	53.7%	32.1%	14.2%
	$CT^{\leftarrow}$	76.5%	21.2%	59.0%	28.1%	12.9%
	differential	58.1%	41.9%	12.2%	0.1%	87.7%
Math	$CT^{\rightarrow}$	90.3%	9.7%	54.4%	21.4%	24.2%
	$CT^{\leftarrow}$	76.6%	23.4%	54.2%	20.2%	25.5%
	differential	62.3%	37.7%	3.6%	2.1%	94.4%
Time	$CT^{\rightarrow}$	90.6%	9.4%	62.7%	30.1%	7.2%
	$CT^{\leftarrow}$	81.1%	18.9%	66.6%	26.6%	6.8%
	differential	54.3%	45.7%	7.1%	0.0%	92.9%

50% of test suites generated by coverage-driven testing that failed to detect the change, fully covered it. This indicates that for the majority of the test suites generated by coverage-driven testing, the changed code was executed, yet the change did not propagate to the output. Therefore, the lower effectiveness of coverage-driven testing is not due to uncovered changes. In fact, for subjects detected by differential testing, 67% of test suites generated by coverage-driven testing fully covered the change, yet they failed to detect it.

To better understand the underlying reason behind the lower effectiveness of coverage-driven testing despite it fully covering a majority of undetected changes, we studied these changes in more detail. For example, `Math-30` is a bug where differential testing was always successful at detecting the change, while coverage-driven testing applied in either direction failed. In this subject, a simple change is made in the method responsible for calculating p-values, where the type of the `n1n2prod` variable is changed from `int` to `double` to fix an integer overflow bug when large data sets are used:

```
1 private double calculateAsymptoticPValue(final double Umin,
2   final int n1, final int n2) ... {
3 -     final int n1n2prod = n1 * n2;
4 +     final double n1n2prod = n1 * n2;
```

Although coverage-driven testing fully covered this method, it did not have any incentive to create tests for the method with both large and small input values. Conversely, differential testing detected a difference in the output of the two versions of the program when a large input value was provided.

As a further example, consider `Chart-6`, where coverage-driven testing applied in either direction is significantly less successful than differential testing. In this subject, the `equals` method is modified to work based on a `ShapeList`. While in the pre-change version the `equals` method of the superclass was used, in the changed version the objects in the `ShapeList` are expected to be instances of `Shape`, as evident on line 5:

```
1  public boolean equals(Object obj) { ...
2  +   ShapeList that = (ShapeList) obj;
3  +   int listSize = size();
4  +   for (int i = 0; i < listSize; i++) {
5  +       if (!ShapeUtilities.equal((Shape) get(i), (Shape) that.get(i)
       )) {
6  +           return false;
7  +       }
8  +   }
9  +   return true;
10 -   return super.equals(obj);
11 }
```

Differential testing finds the fault by setting a `ShapeList` object as an element of the list, which triggers a `ClassCastException` as evident in the test case below. Although the changes for this bug add new branches, coverage-driven testing never detects the fault, since providing a `ShapeList` on line 6 would trigger the `ClassCastException` which results in covering neither the true or false branch of the condition, thus resulting in a lower coverage. This could be countered by using stronger or multiple coverage criteria [135]; however, as our experiments

targeted only branch coverage, the coverage-driven testing had no incentive to retain a fault triggering test.

```
@Test public void test0() {
    ShapeList shapeList0 = new ShapeList();
    ShapeList shapeList1 = (ShapeList)shapeList0.clone();
    shapeList0.set(0, shapeList1);
    // The following line throws java.lang.ClassCastException:
    //  java.lang.String cannot be cast to java.awt.Shape
    shapeList0.equals(shapeList1);
}
```

It is interesting to note that, as shown in Figure 6.1 and Figure 6.3, differential testing always performed statistically better or equal compared to coverage-driven testing on bugs in project Chart, whereas there are no notable differences for the other projects. A likely explanation is that achieving branch coverage in this project is relatively simple, but branch coverage tests do not capture the faults in this project well in general, as in the presented examples.

Optimizing for coverage can lead to omission of change-revealing tests.

6.4 Answers to RQ6.2 (Detection of Subsequent Changes)

We have seen that both coverage-driven testing and differential testing can be used to detect differences (changes) between a pair of classes. The question now is, would retaining these tests build an effective regression test suite? In a real software development scenario, in which developers apply automated test generation techniques over time to generate and augment test suites as their system under test evolves, are these generated tests sensitive enough to detect newly introduced changes of the same class? We answer this by considering two different sets of changes: a) future changes to the same area of code (i.e., of an earlier change), b) future changes made to the same class by the developers.

Table 6.3 Overall outcome of test generation and execution. For each project and technique the table shows the average number of generated tests, the percentage of which that were flaky, and the time taken on average to generate tests including post-processing steps such as minimization and oracle generation. The table also shows the maximum number of bugs detected by the techniques for each project, and the average number of detected bugs over the 30 repetitions.

Project	Config.	Tests	Flaky	Time	Max Bugs	Avg. Bugs
Chart	$CT^{\rightarrow}$	30.3	0.0%	170s	15	4.2
	$CT^{\leftarrow}$	29.9	0.0%	170s	17	8.0
	differential	0.6	0.0%	98s	18	14.4
Closure	$CT^{\rightarrow}$	14.5	0.4%	170s	21	6.7
	$CT^{\leftarrow}$	14.4	0.4%	171s	26	7.3
	differential	0.2	0.0%	140s	19	9.9
Lang	$CT^{\rightarrow}$	66.9	0.0%	147s	31	9.8
	$CT^{\leftarrow}$	66.3	0.0%	148s	42	21.1
	differential	0.5	0.0%	80s	42	30.0
Math	$CT^{\rightarrow}$	25.3	0.2%	140s	44	22.7
	$CT^{\leftarrow}$	25.5	0.2%	140s	59	31.8
	differential	0.5	0.0%	76s	57	39.2
Time	$CT^{\rightarrow}$	37.9	0.0%	159s	13	3.4
	$CT^{\leftarrow}$	38.7	0.0%	158s	12	6.6
	differential	0.5	0.0%	97s	16	9.0

6.4.1 RQ6.2-M: Detection of Mutants

To simulate subsequent fault-introducing changes that may occur in the future, we considered all mutants of the changed lines between the two versions of each bug in Defects4J. We then applied all test suites generated in RQ6.1 which were successful at detecting the change between the two versions of the class, on the generated mutants. Considering that changes between two versions of a program usually involve lines of code that are added or removed in either version, we report on the mutation results based on the two versions A and B, where mutations are applied on the lines of code changed in one version with respect to the other. In the context of Defects4J, A and B respectively refer to *buggy* and *fixed* versions.

Table 6.4 shows the overall outcome of the techniques at detecting the mutants. In this table, we report on two different mutation score values: the number of mutants killed by the techniques out of all generated mutants (of the changed lines), and the normalised mutation score, that is, the number of mutants killed out of all mutants (of the changed lines) that the test suite *covered*. Notice that both the overall and normalised mutation scores achieved

by coverage-driven testing are consistently higher than differential testing on either version for *all* projects.

Comparing the statistical significance of the difference between the effectiveness of the techniques, for 82 subjects, coverage-driven testing achieved a significantly higher mutation score than differential testing, while being significantly lower in 5. This outcome can be surprising, especially considering the higher success rate of differential testing in RQ6.1. The results suggest that tests generated by differential testing are over-specified to the specific change they were generated on, and may not be effective enough at detecting future changes to the same area.

To support our conjecture, we further investigated some of the mutants. In Section 6.3.2 we showed `Chart-6` as an example of a case where differential testing was more effective than coverage-driven testing. However, when applying the generated tests by the techniques on the mutants of the subject, coverage-driven testing was significantly more effective at detecting the following mutant:

```
1 -   return super.equals(obj);
2 +   return false;
```

As it can be seen from the generated test case shown in Section 6.3.2, differential testing does not attempt to generate tests that cover both scenarios in which the `equals` method returns `true` and `false`, neither does it validate the output of the method, as it has no incentive to do so (i.e., such validations do not relate to change differential testing was aiming to reveal). As a result, while the generated test may be effective at triggering the `ClassCastException`, the test is not equipped to detect other changes to the CUT.

Considering the different nature of the tests generated by the two techniques, we were also interested to understand how these generated tests compared with manually written tests by the developers. In Table 6.4, we also report on the mutation data for the developer-written tests. Looking at the results of the tests written by developers after fixing the bug (i.e., version B), it is clear that across all the projects the manually written tests have both a higher coverage and higher mutation score than both techniques. This indicates that in contrast to differential testing, test sets written by developers are less specific to individual changes. Our general observation is that in terms of effectiveness the manually

Table 6.4 Mutation outcome of the fault revealing test suites generated by each technique per project. A and B indicate the version of the program on which mutation was performed. *Subjects* shows the number of subjects for which all configurations had at least one fault-revealing test suite, *Mutants* shows the number of mutants generated for each version, *Mutants Covered* shows the average ratio of the mutants that the test suites were able to cover, *Mutation Score* shows the average number of mutants killed with respect to the generated mutants, and *Normalised M.S.* shows the mutation score normalised with respect to the number of mutants covered by the test suites.

		Subjects		Mutants		Mutants Covered		Mutation Score		Normalised M.S.	
Project	Config.	version A	version B	version A	version B	version A	version B	version A	version B	version A	version B
Chart	Coverage	9	14	51	180	88.9%	85.5%	64.0%	59.6%	72.0%	68.6%
	Differential	9	14	51	180	70.4%	64.8%	27.0%	43.9%	34.8%	60.7%
	Manual	9	14	51	180	58.7%	87.5%	44.4%	66.5%	66.7%	76.9%
Closure	Coverage	10	14	51	91	75.7%	75.7%	39.1%	51.5%	43.4%	68.9%
	Differential	10	14	51	91	54.9%	66.4%	18.3%	30.9%	30.6%	50.6%
	Manual	10	14	51	91	88.0%	89.8%	50.4%	76.4%	56.0%	86.0%
Lang	Coverage	19	37	190	707	90.9%	86.2%	66.6%	58.5%	73.7%	67.8%
	Differential	19	37	190	707	71.2%	69.2%	43.2%	31.5%	59.9%	46.1%
	Manual	19	37	190	707	82.2%	88.0%	56.0%	68.1%	65.8%	76.4%
Math	Coverage	25	47	303	1016	99.1%	94.9%	66.6%	65.4%	67.4%	68.5%
	Differential	25	47	303	1016	71.3%	77.7%	35.8%	40.1%	50.2%	52.0%
	Manual	25	47	303	1016	66.2%	96.6%	48.8%	76.8%	72.6%	79.2%
Time	Coverage	8	11	106	322	90.8%	87.0%	44.6%	60.6%	46.3%	67.0%
	Differential	8	11	106	322	60.7%	66.8%	22.0%	34.9%	33.0%	44.7%
	Manual	8	11	106	322	91.7%	91.4%	51.7%	74.6%	51.7%	80.4%

written tests by the developers differ from tests generated by either techniques, however, the underlying reason behind this remains an open question for further research.

RQ6.2-M: Test suites generated by differential testing overfit to the change they are generated on, and are less effective at finding subsequent changes.

6.4.2 RQ6.2-D: Detection of Developer Changes to Classes with Continuous Augmentation

To answer RQ6.2 in terms of real changes, we applied coverage-driven testing and differential testing once per commit over 9 consecutive commits on the same class, and executed the resulting test suite on the 10th commit – which in the case of DEFECTS4J is the change that represents the bugfix – to understand whether the generated tests can be effective at detecting this change.

Overall, out of 183 subjects, coverage-driven testing found 64 (35%) changes at least once, while differential testing found 24 (13%). Although there is a large

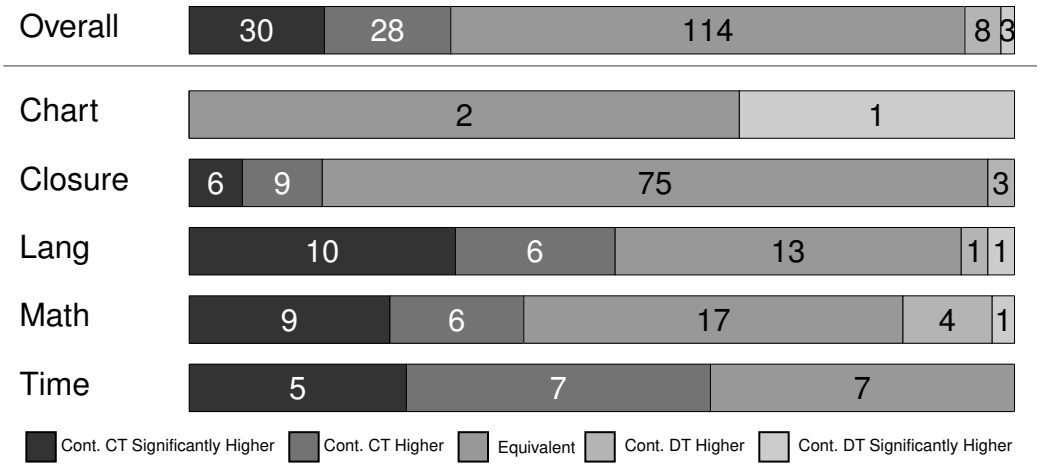


Fig. 6.4 Comparing coverage-driven testing and differential testing tests evolved over 9 commits in regard to their effectiveness at detecting the change between the buggy and fixed versions of classes in DEFECTS4J.
(Please refer to Figure 6.1 for an explanation of the legend.)

overlap between the changes found, 7 out of 24 changes found by differential testing were never found by coverage-driven testing. Conversely, of the 64 bugs found by coverage-driven testing, 47 were never found by differential testing. This provides further evidence towards the fundamental difference between the test suites generated by the two techniques.

Figure 6.4 shows the comparison of differential testing and coverage-driven testing in more detail. Observe that differential testing detected changes on a significantly higher number of runs compared to coverage-driven testing on 3 subjects, while it was significantly less effective on 30 subjects.

RQ6.2-D: Test suites augmented using differential testing were significantly less effective than coverage-driven testing at detecting developer changes.

Given that overall test suites evolved by differential testing were less effective than coverage-driven testing and found fewer changes, in the following two sections we further investigate the underlying reasons.

6.4.2.1 The Influence of the History of Changes

The effectiveness of differential testing is dependent on the history of changes leading up to the one under consideration: If the previous edits are unrelated to this change, then differential testing is unlikely to have generated relevant tests in the past. For example, Figure 6.4 shows that there are several cases in the Time and Chart projects where differential testing is significantly worse at detecting the change than coverage-driven testing. A manual analysis of five bugs in `Time` which were not detected at all by differential testing (`Time-4`, `Time-7`, `Time-11`, `Time-23`, `Time-25`) reveals that in none of these five cases the lines of code changed in the bug fix were edited in any of the past 9 commits. On the other hand, Figure 6.4 suggests that in projects Lang and Math there are subjects for which the effectiveness of differential testing is higher than that of coverage-driven testing. A closer look at these bugs shows that, as expected, in these cases there are previous commits in the version history that modified the same lines as in the bug fix.

For example, consider `Math-48`, for which differential testing is more effective than coverage-driven testing at detecting the change. In this subject, developers have added a condition to throw an exception when the algorithm is stuck instead of waiting for the algorithm to exceed the limit of maximum iterations:

```
case REGULA_FALSI:
+      // Detect early that algorithm is stuck, instead of waiting
+      // for the maximum number of iterations to be exceeded.
+      if (x == x1) {
+        throw new ConvergenceException();
+      }
break;
```

The reason differential testing finds the change is due to an earlier commit in the class that modified the code within the same `case` block. One commit earlier, the developers had emptied the code in the `case` block; two commits earlier, they had the same `x == x1` conditional block, however with a different set of instructions inside which did not throw a `ConvergenceException`. For coverage-driven testing, generating tests for the `REGULA_FALSI` case block often

resulted in reaching the resource limit related to the program and provided no additional coverage gain to keep the test.

However, cases like `Math-48` were rare in the set of subjects we examined, and for a majority of subjects, either the changed area was not modified in the past, or differential testing failed to generate tests for the earlier changes. For instance, `Closure-46` is a change that is always found by coverage-driven testing and never found by differential testing. In this subject, the overridden `getLeastSupertype` method is removed to fix a bug:

```
1 - @Override
2 - public JSType getLeastSupertype(JSType that) {
3 -     if (!that.isRecordType()) {
4 -         return super.getLeastSupertype(that);
5 -     }
6 -     ...
7 - }
```

While this appears like a simple case for coverage-driven testing to detect, for differential testing it is necessary to first have changes in the history that modified the method, and second, for it to have detected the earlier change. Although one commit earlier, the same method was modified after line 5, differential testing was not able to detect that change. As a result, it failed to produce a test to detect the change applied in the later commit.

In our experiments, cases for which differential testing could benefit from the changes in history were rare.

6.4.2.2 The Influence of Coverage on Detecting Changes Made in the Future

To better understand the underlying reason behind the low detection rate of differential testing (13%) compared to coverage-driven testing (35%), we looked at the level of coverage obtained by the techniques, as the test suites evolved along with the program.

Figure 6.5 shows the number of branches covered by each technique over time. A clear difference is apparent between the level of coverage achieved by the

techniques. In particular, after 9 commits, coverage-driven testing on average obtained 53% branch coverage whereas differential testing only achieved 9%. This outcome can be surprising, considering that differential testing achieved 6 times less coverage than coverage-driven testing while performing statistically better or equal in 84% of the subjects. The results suggests that for the subjects where differential testing was less effective than coverage-driven testing, it may relate to the lack of full coverage of the changed area of code.

To further investigate the influence of coverage on detecting future changes, we looked at the coverage of the change applied in the last commit (i.e., the 10th commit). Out of 183 subjects, for only 28 bugs differential testing *fully* covered the change at least once, 5 of which it managed to always cover. This indicates that the low detection rate of differential testing can indeed be explained by the lack of coverage.

To better demonstrate the reason behind this lack of coverage, consider the change in `Lang-7` as shown below. In this subject, the `createNumber` method is modified by removing a redundant sanity check that is addressed later in the same method (after line 4, omitted for brevity). Instead, when the input `str` satisfies the condition in line 2, a `NumberFormatException` is thrown later in the code.

```
1 public static Number createNumber(String str) ...
2 -       if (str.startsWith("--")) {
3 -           return null;
4 -       }
```

However the modified area of code was not changed earlier in the history, as such, differential testing did not have any incentive to generate a test to cover the condition.

Augmented over 9 commits, coverage-driven testing achieved a coverage of 53% *(on average),* 6 *times higher than the coverage obtained by differential testing.*

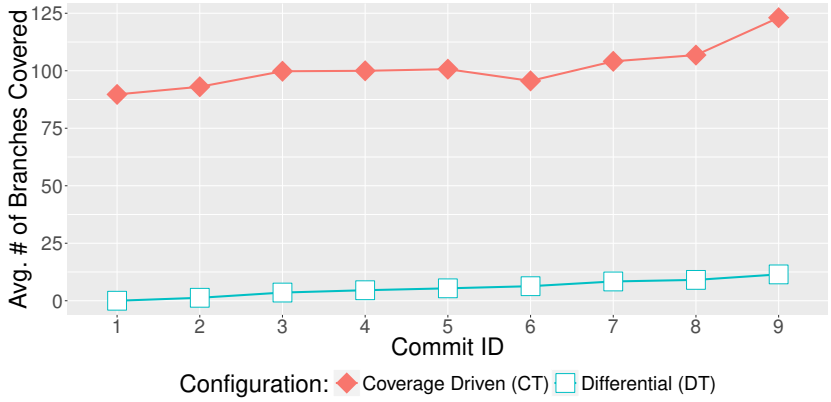


Fig. 6.5 The average number of covered branches at each commit when evolving test suites using coverage-driven testing and differential testing.

6.5 Answers to RQ6.3 (Comparison of the Maintenance Overhead)

Evolving a regression test suite alongside the software involves effort to inspect and maintain all failing test cases over time. When applying disposable testing, this effort does not exist, but there still is effort to inspect tests that demonstrate behavioural differences. Although we cannot directly compare the human effort involved in maintaining or inspecting tests without running a study involving human participants, we can compare the effort in terms of the number of tests that needs to be inspected in both approaches.

To make this comparison, we assume a scenario where every inspected test in coverage-driven testing reveals intended behaviour and the maintenance action consists of deleting this test. Although in practice tests may be fixed and retained, this scenario gives us a *lower bound* on the maintenance effort of coverage-driven testing. If tests would be fixed rather than deleted, then test suites would grow bigger over time, thus increasing the potential maintenance effort.

For disposable testing, we assume a scenario where developers apply differential testing for the whole duration of the search budget and have to inspect all behavioural differences. The number of failing tests in this case should provide an *upper bound* for the maintenance effort of disposable testing, as our minimization technique is only basic, and developers may in practice stop inspecting further tests once they have identified an error in the program that needs to be fixed.

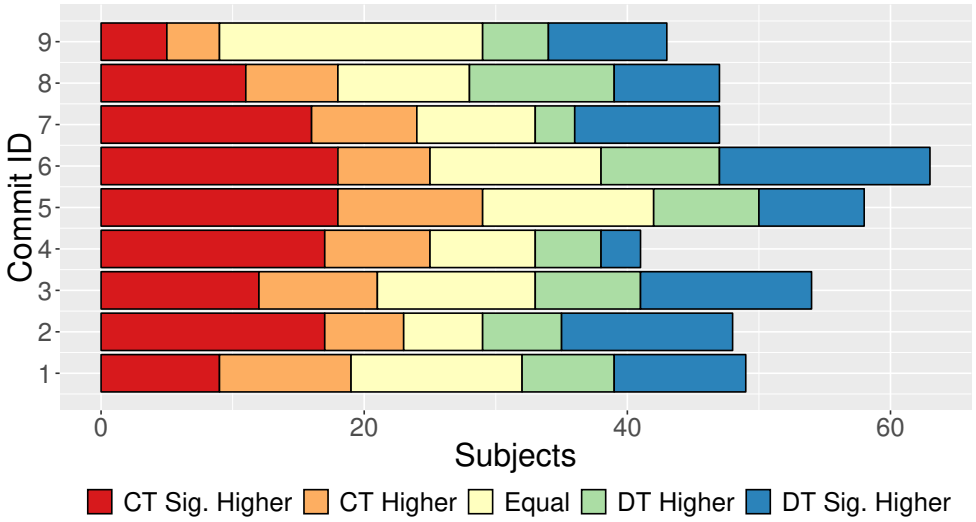


Fig. 6.6 Comparing coverage-driven testing (maintained) and differential testing (disposable) with regards to the number of tests developers need to inspect, for the changes found by both techniques, over each commit.

Legend: "Sig. higher" is the number of classes for which a technique generated a significantly higher number of failing tests than the other (i.e., developers need to inspect more tests); "Higher" refers to the number of classes where a technique generated a higher number of failing tests on average than the other, but not significantly so; "Equivalent" is where test suites generated by both techniques had the same number of failures.

Comparing this against the lower-bound of the maintenance effort of coverage-driven testing tests provides us a conservative indication on the extent of effort that disposable testing can potentially reduce. To achieve this, we compare the number of failing tests from differential testing applied consecutively to the pairs of commits as in RQ6.2-D (without keeping the generated tests), against the number of tests thrown away during test suite evolution using coverage-driven testing in RQ6.2-D.

Figure 6.6 compares the number of subjects where one technique had significantly higher number of failing test cases (i.e., test cases that need to be inspected) compared to the other – significance calculated using the Mann Whitney U test. Our results show that for the 450 changes detected by both techniques across 9 commits (from 158 unique subjects), for 123 subjects, a significantly higher number of failing tests need to be inspected for coverage-driven testing, while for 91 subjects, a significantly higher number of tests need to be inspected for differential testing. The number of tests to inspect ranged from 1 to 141, and we did not observe a pattern in the number of failing tests across

the commit history we investigated. For instance, while after the first commit on average 5.3 differential testing tests were failing when a change was found, for coverage-driven testing 3.8 tests were failing; these numbers for the 7th commit were respectively 2.9 and 7.6.

Our analysis shows that for the majority (80%) of changes, differential testing either reduces the number of tests to inspect, or does not result in an increase. Nevertheless, we manually investigated the subjects for which developers would need to inspect a larger number of tests. We found that for all investigated test suites, the large number of failing test cases was due to our naïve filtering mechanism failing to group duplicate failures into one test case. In particular, our implementation does not support grouping tests by exception errors, or expected exceptions. Therefore, we believe that improving test suite minimization can reduce the number of tests to inspect across all subjects.

RQ6.3: Disposable testing eliminates the maintenance effort, and does not increase inspection effort for most changes.

6.6 Conclusions

As developers develop and evolve their programs, they write unit tests to ensure that changes in the future will not break intended functionality. Automated test generation techniques can support developers by producing tests to enhance or replace their own testing efforts. However, these generated tests still need to be maintained along with the program. In this chapter we proposed *disposable testing* as an alternative approach, where a completely new test suite is generated each time a program is changed, but these tests are thrown away after a developer has inspected those tests that reveal a behavioural change.

We used differential testing as a means to implement disposable testing, and compared its effectiveness against a traditional generate-and-maintain approach such as coverage-driven testing on a large set of real faults taken from the Defects4J repository. We also evaluated the benefit of keeping these tests in the test suite, and assessed whether they can be effective at finding changes in the future.

Our results show that generating tests using differential testing is more effective than using coverage-driven testing. However, tests generated with differential testing are less effective at detecting future changes as they are over-fitted to the specific change they originally targeted – that is, these tests are actually disposable. We also showed that in addition to removing the effort required to maintain the tests, using differential testing does not result in an increase in the number of tests that need to be inspected by the developers compared to the maintenance effort for a coverage-oriented test suite. For all these reasons, disposable testing is a viable alternative for automated regression testing.

Although our experiments have suggested clear benefits to using disposable testing, research remains to be done on developing techniques better suited to generate such tests. For instance: a) in this chapter we apply differential testing using random test generation – which was based on our findings in Chapter 4 and Chapter 5 –, while more systematic techniques may be more effective, b) although the generated test suites shown to the developers can be thrown away, the input values and knowledge gained by generating the previous test suites can be kept and seeded to the test suite generator to increase effectiveness and performance, c) readability of the generated tests can be improved to lower the overhead of manual inspection.

Developers may wish to use the insights provided in this chapter to better decide which automated techniques to use, and what to do with the resulting tests. In practice, this may not be an either-or choice: Because test generation takes time, regression test suites may be important to provide quick feedback. Thus, an optimal solution may combine traditional and disposable testing in order to benefit from the best from both.

We have made our extension of EvoSuiteR open source and publicly available. See Appendix A for details on how to run the tool.

Chapter 7

Conclusions and Future Work

7.1 Summary of Contributions and Achievements

In this thesis, we investigated various aspects related to the idea of automatically testing programs as they evolve. In particular, in Chapter 1 we set out with the idea of automatically and effectively generating change-revealing tests after a change has been made. To this end, we explored and investigated a number of research questions during the course of this work, which we summarise below.

- Effectiveness on Real Faults: Are automated test-generation tools effective at finding real faults?
- Limitations of Existing Techniques: How do existing state-of-the-art test generation techniques find real faults? What are the limitations and shortcomings?
- Search-Based Propagation of Regression Faults: Given that generating change-revealing tests involves tackling multiple problems at the same time, can we use a multi-objective search-based approach to generate these tests? Is it effective at finding real faults? Do the individual objectives guide the search? How does it compare with state-of-the-art techniques?

- Search Algorithm: Do we really need a GA to help us generate our desired tests? What does the search landscape look like? In which cases using a GA can help us as opposed to a naïve random search approach?
- Search Budget: Is the result obtained earlier heavily influenced by the search budget?
- Usability in Practice: Can automatically generated change-revealing tests be effective in practice, as opposed to common coverage-driven approaches?
- Maintenance: What shall we do afterwards with the generated tests? Does it really help if we keep and maintain them over time?

In the following sections, we summarise our answers to the questions above, resulting in the contributions of this thesis.

7.1.1 Effectiveness of Tools on Detecting Real Faults

In Chapter 2 we showed that while many different techniques have been proposed in the literature for automated test generation, a large-scale evaluation and comparison of these techniques on real faults is not available. Such an evaluation framework is however essential to understanding where we stand in terms of effectively revealing real faults, if we use these techniques in practice. Moreover, to make progress on the existing state, insight into the shortcomings of the current approaches is necessary to focus research in the right directions. Finally, a clear evaluation of future techniques would not be possible without involving an in-depth comparison with the state-of-the-art techniques.

Therefore, in Chapter 3, we evaluated the effectiveness of three state-of-the-art automated test generation tools, on a set of 357 real bugs taken from open source repositories. To achieve this, we developed a framework consisting of our methodology to generate, filter (removing flaky and non-compiling tests), and execute tests. We also report on how the resulting outcome should be evaluated, based on the types of failures, and to detect the set of false-positives that need to be excluded — which can be challenging given the large scale of the study.

Our results showed that while tools overall detected more than half of the faults, several big challenges remain: 1) the techniques do not reliably find the

faults, 2) problems such as generating false-positive and flaky tests hinder the potential of the techniques to be used by developers, 3) while tools can often achieve high levels of code-coverage, this still needs improving, 4) a majority of undetected faults were covered by the tests, suggesting that state infection and propagation also remain as major challenges. Using the evaluation framework and insight gained from it, we describe how we proceeded to propose our search-based approach in the next section.

7.1.2 Search-Based Differential Testing

Seeing the shortcomings of existing approaches in Chapter 2 and Chapter 3, we saw opportunities to improve on the state-of-the-art techniques. First, seeing the low success rate of the techniques at finding the fault, and considering the fact that a large number of generated tests failed to cover the changed code, we conjectured these to be due to not targeting both versions of the program (before and after the change is made). As such, we considered our technique to use two versions of the program for test generation. Second, seeing the high number of false-positives and non-compiling tests, and in contrast to differential-testing techniques such as BERT [123], we considered our technique to generate tests that compile and execute correctly on both versions (e.g., if a method has been removed, it is likely intended, and non-compiling failures or mocking failures are false positives). Third, and most importantly, we have seen both from literature and our evaluation that fault propagation is a significant challenge, and simply reaching the code is not enough. To address this, we proposed a multi-objective search-based approach named EvoSuiteℝ, which simultaneously aims to cover the program, as well as increasing the chance for the fault to propagate to the output — this was done by increasing the differences in the state and control-flow across the two programs.

We evaluated the technique based on the evaluation framework we devised earlier on detecting real faults. Our results showed that our proposed multi-objective GA was similarly or more effective than the techniques we evaluated earlier. Moreover, the multi-objective fitness function was more effective than when individual objectives were set, such as only guiding for increased code-coverage. Moreover, the technique was more reliable than the tools we investigated earlier:

when a change was detected, more than half of all generated tests detected the fault in 30 repetitions. This is in contrast to 15% for RANDOOP and close to 20% for EVOSUITE. Therefore, our technique had a higher success rate, found more bugs and found them more frequently, and generated no false-positives or non-compiling classes. However, when we compared our GA-based approach against a naïve random search using the same tooling, we found an interesting result: for the majority of subjects, random search is as effective at detecting the bug. This leads to the question of whether this insight generalises to search-based test generation techniques in general. We investigate this in the following section.

7.1.3 Random or GA for Search-Based Test Generation

In Chapter 4 we observed that on a majority of subjects, using random search can be equally as effective as an advanced algorithm as GA. While in the literature evolutionary techniques have been shown to be superior, such evaluations often use different tooling for each technique. Moreover, the search landscape on real object-oriented programs is often ignored. For instance, theoretically, if all software branches provide guidance, then GAs can more effectively guide the search towards covering all branches. But is this the case for real programs? Lastly, can our findings be simply influenced by the search budget?

To answer these questions, in Chapter 5 we first provide an in-depth insight into the types of branches in object oriented programs, specifically, those based on the bytecode from the Java Virtual Machine (JVM). We show the type of branches that can provide guidance to the search (gradient branches), and those that do not provide such guidance (plateau branches). Then we conduct a large-scale evaluation, comparing GA and random search for the problem of coverage-driven test generation – as in EVOSUITE– on 1,000 classes, while at the same time we observed the type of covered and uncovered branches. We also investigate the effectiveness of the algorithms when given an extended budget.

Our results show that overall, for a majority of subjects, random search can be as effective as GA. Moreover, the search budget does not seem to affect this insight: extending the search budget resulted in only marginal improvements by the GA. We found the reason for this outcome to be due to the fitness landscape provided by the different type of branches. Our findings show that for the object

oriented programs we investigated, the majority of branches do not provide any guidance to the search. As such, evolutionary approaches suffer from the lack of guidance to the fitness function. We also find that random techniques are better suited to covering plateau branches — and in-fact cover more of these branches —, while as expected, GA covers more gradient branches. These findings provide practical insights for researchers and developers.

7.1.4 Maintaining Automatically Generated Tests

Maintaining unit-tests can be a tedious task as programs evolve, and this is exacerbated by automatically generated tests. If we aim to help developers with their regression testing efforts using automated techniques (as we aimed to achieve in Chapter 1), using the technique during software development should also be practical. The traditional approach towards generated tests is to keep and maintain them – the same way as manually written ones. However, is this also the case for automatically generated change-revealing tests? In Chapter 6, we propose an alternative, such that these tests are thrown away. Our approach named "disposable testing" involves generating tests once a change has been made — to reveal the behavioural difference between the two versions – and once developers have verified whether or not a regression fault has occurred – where in case of the former, they would use the test to fix the bug – the tests are thrown away.

To assess whether such a technique can be practical, we need a technique that can effectively generate change-revealing tests, otherwise, keeping the tests could lead to detection of more changes. We take *differential testing* as an example of such a technique, and based on our findings in Chapter 4 and Chapter 5 we use a random search algorithm to reveal changes. We compare this against the traditional approach of generating coverage-driven tests, and use the framework we devised in Chapter 3 to evaluate it. We also evaluate the number of tests one needs to inspect after a change has been made.

Our results show that disposable testing using differential testing can be more effective than coverage-driven testing at revealing changes. At the same time, keeping these tests does not help with finding more faults, and as such, there is no need to maintain these tests. Besides removing the maintenance

effort, the inspection effort is lowered, given that fewer failing test cases need to be inspected when a test suite detects a change. In sum, while disposable testing may appear as a controversial approach, it is both feasible and effective compared to the traditional keep and maintain approach. In the next section we look at future work that can build on top of the contributions of this thesis.

7.2 Future Work

The work in this thesis explored a number of different techniques and approaches to test programs as they evolve. Our investigations revealed a series of new interesting challenges in the field to be tackled in future work.

7.2.1 Human Study of Disposable Testing

Although many techniques have been presented and studied thus far (within this thesis, or in the literature), little attention has been given to the applicability of such techniques in practice. For example, can automated tools actually save development time while at least remaining equally effective? Given that many potential users of these tools need to adapt to the new practices, is it easy to incorporate such tools into existing development practices? Moreover, tools often do not find the faults deterministically (see Chapter 3), thus, what approach should developers take when using these tools? Answers to questions like these can have a positive impact on the adoption of current state-of-the-art research in industry, as such, research into practical usability of the techniques is essential.

Although work has been done on evaluating the use of automated test generation tools during the software development process (e.g., [136]), little work has been done on applying differential testing in practice. In particular, we proposed disposable testing in Chapter 6 as an alternative to the traditional techniques, but whether the approach can be applied in practice is a question that can be best answered with a human study.

7.2.2 Test Readability

In Chapter 3, Chapter 4 and Chapter 6 we showed that test generation tools can be effective at fault finding, and also that even applying disposable testing can help developers detect regression faults. However, inspecting and understanding automatically generated unit tests is a new challenge that does not exist when developers are using their own manually written tests. Therefore, making the tests more readable can improve the chance of which the techniques can be adapted by developers and industry. While this problem is an active area of study, and work is being done on improving the readability of tests (e.g., [36, 1, 55]) in this section we discuss future work more related to the problem of testing evolving programs (e.g., using differential testing).

7.2.2.1 Test Clustering

When a test suite reveals a change between two versions of a program, often more than one failing test is revealing the change. This is often the case when applying integration testing (e.g., one broken module causes all tests involving that module to fail), or when automatically generating tests (see Chapter 3 and Chapter 6). When developers starts to manually inspect these failing tests, they first need to identify the location of the bug – this practice is also referred to as *fault localisation* [181]. We believe that this technique can applied to improve the readability of automatically generated differential tests.

By finding the location of the fault, if the root cause of the failures can be automatically determined, we can reduce the number of failing tests (e.g., by clustering similar tests) as well as further minimising the tests. For instance, if four tests are failing due to the same underlying reason, then only one of them can be shown to the developer. Furthermore, the generated test can be further minimised by removing code unrelated to the root cause of the failure.

7.2.2.2 Utilising Test Patterns

Even if test readability techniques become ultimately successful such that automatically generated tests become as readable as manually written ones, if a technique such as disposable testing is applied, then inspecting these tests will

always involve the effort of re-learning and re-familiarising. One solution to tackle this is to increase the similarity of new tests with the previously generated tests. For instance, the structural pattern of the previously generated tests can be learned using machine learning techniques, and the newly generated tests can mimic or adapt to the same pattern. This can automatically make developers familiar with the newly generated tests.

7.2.3 Detecting Non-Functional Regressions Using Differential Testing

Although this thesis mainly focuses on testing programs' functional behaviour, non-functional regressions such as usability, security and performance can often be equally or more important in practice. Considering performance testing as an example, while much has been done on helping developers detect such regressions (e.g., applying performance benchmarks before Linux kernel releases to avoid performance regressions [31]), the techniques often assume that existing testing infrastructures are in place (e.g., [82]). Work has also been done on generating such tests automatically (e.g., [200], [67]). However, in the context of automated differential testing, revealing performance differences between a pair of programs remains an open problem. This has been investigated in the literature (e.g., Pradel et al. [131] look at differential performance testing for thread-safe classes), but whether this can be done in the context of disposable unit-testing remains an open problem.

7.2.4 Addressing State Infection in Differential Testing

As mentioned in Section 2.1, to reveal a fault, a test has to reach the fault, infect the state and then propagate it. The GA-based approach we presented in Chapter 4 aims to reach/execute changes (by maximizing the coverage), while at the same time aiming to propagate the infected state to the output (using the state-distance/control-flow-distance metrics). However, the state infection is left to chance. As such, simply the order in which statements are executed may not affect the fitness value (i.e., it may not increase the coverage value), but is important for detecting behavioural changes. For instance, in a test that

even fully covers the class under test, to infect the state, the order in which the statements are executed is important. Consider the class `Foo` below:

```
1  public class Foo{
2      private int value = 1;
3
4      public void setValue(int newValue){
5  -       value = newValue;
6  +       value = 2 * newValue;
7      }
8
9      public int getValue(){
10         return value;
11     }
12 }
```

Observe that the `setValue` method has been changed to set the private field `value` to twice the provided input. A simple test can detect the change:

```
1 public test0(){
2     Foo foo = new Foo();
3     f.setValue(1);
4     int value = f.getValue();
5     assertEquals(value, 1);
6 }
```

In the test case above, the assertion fails on the changed version – and thus detects the change, since `value` now equals to 2. However, if the lines 3 and 4 in the test case above are swapped, the coverage remains at 100%, yet the test is unable to identify any changes. One way to overcome this would be to derive symbolic conditions that can result in state infections. Currently, none of the measurements used in the fitness function of the GA reward the use of different data-flows in the test suites. As a result, for instance, if a branch is already covered by any test in the test suite, the search can get stuck in a local optimum, since any new solution that covers the same branch in a different way is discarded. Work has been done in the literature to increase the diversity in the population, where the researchers found this to improve the efficiency and

effectiveness of the search [91]. We also propose looking at metrics to increase the diversity of data-flows involved in covering the same code.

7.2.5 Hyper-heuristics Search and Adaptive Approaches

When using search algorithms to a solve a problem, we often need to specifically design the algorithm and tune it to make the technique effective at tackling the problem, however, despite this effort, when considering the wider range of problems in the search space, another algorithm simply outperforms our specifically design-and-tuned technique. Holistically, this is what the no-free-lunch theorem states [179]. In particular, we saw in Chapter 4 and Chapter 5 that a simple random testing technique can not only be equally effective on a majority of the subject programs (which were randomly selected from a large corpus of open-source programs, thus it is likely that they span across a wide range of the problem space), but also, in some instances it can outperform guided techniques.

A possible alternative is dynamically adapting the technique to the problem [72]. For instance, in Chapter 5 we saw that for certain type of programs, the fitness landscape is flat for the GA, and as such, guiding the search towards covering plateau branches can be difficult. A self-adaptive technique can automatically devise the best function and search parameters to overcome the problem. The methodologies and approaches to design such search heuristics that are more generally applicable, are also known as "Hyper-heuristics" [25]. In the field of software testing, hyper-heuristic search techniques have been applied with strong positive result (e.g., [86]). We believe that using such approaches can provide significant benefit to the problem of differential testing, and to help with designing more generic techniques for automatically generating tests that can reveal regression faults.

7.3 Final Remarks

Software programs are ever evolving, and ensuring that their intended functionality is unaffected by the new changes is a tedious and expensive task. As software engineers, we strive to reduce "tedious and expensive" tasks, and aim

to automate them. As such, this thesis explored the problem of automatically testing programs as they evolve. First, a number of existing state-of-the-art test generation tools were evaluated on real faults, to create a better picture of where automated techniques stand against real faults in practice. Then an alternative approach was presented and evaluated to generate change-revealing tests using a multi-objective search-based approach. Although the resulting technique was successful and more effective than existing tools, for a majority of subjects, a naïve random search approach was found to be equally effective. Further investigation of this claim suggested that it holds for the problem of test generation in general, when applied to object-oriented programs. Finally, based on the observations made so far, a test generation approach named "disposable testing" was proposed to use automated generation of change-revealing tests in practice. Overall, this thesis contributes a new framework for evaluating test generation tools, and provides an effective technique to generate differential unit tests, in addition to presenting a validated approach for using it in practice which requires less effort to be used alongside existing testing practices by the developers. Automated testing techniques have come a long way over the past decades, and the findings of this thesis show that these techniques are on the verge of enabling us to fully automate the testing process. Perhaps, in the future, writing and maintaining regression tests can be a thing in the past with techniques such as disposable testing, such that developers can be immediately made aware of the implications of their changes within their development environment, without having to write a single line of test code.

References

[1] Afshan, S., McMinn, P., and Stevenson, M. (2013). Evolving readable string test inputs using a visceral language model to reduce human oracle cost. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 352–361. IEEE.

[2] Agitar (2013). *JUnit Factory* URL: http://www.agitar.com/developers/junit_factory.html. Last visited on 24.04.2013.

[3] Alshahwan, N. and Harman, M. (2011). Automated web application testing using search based software engineering. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 3–12.

[4] Alshraideh, M. and Bottaci, L. (2006). Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification and Reliability (STVR)*, 16(3):175–203.

[5] Ammann, P. and Offutt, J. (2008). *Introduction to software testing*. Cambridge University Press.

[6] Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., Harman, M., Harrold, M. J., McMinn, P., et al. (2013). An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001.

[7] Andreessen, M. (2011). Why software is eating the world. *Wall Street Journal*, 20.

[8] Andrews, J. H., Briand, L. C., and Labiche, Y. (2005). Is mutation an appropriate tool for testing experiments? In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 402–411. IEEE.

[9] Andrews, J. H., Li, F. C., and Menzies, T. (2007). Nighthawk: A two-level genetic-random unit test data generator. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 144–153. ACM.

[10] Apiwattanapong, T., Orso, A., and Harrold, M. (2007). Jdiff: A differencing technique and tool for object-oriented programs. *Automated Software Engineering*, 14(1):3–36.

[11] Apiwattanapong, T., Santelices, R., Chittimalli, P. K., Orso, A., and Harrold, M. J. (2006). Matrix: Maintenance-oriented testing requirements identifier and examiner. In *Testing: Academic Industrial Conference - Practice And Research Techniques (TAIC PART'06)*, pages 137–146. IEEE.

[12] Arcuri, A. and Briand, L. (2011). Adaptive random testing: An illusion of effectiveness? In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 265–275. ACM.

[13] Arcuri, A. and Briand, L. (2014). A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Software Testing, Verification and Reliability (STVR)*, 24(3):219–250.

[14] Arcuri, A., Fraser, G., and Galeotti, J. P. (2014). Automated unit test generation for classes with environment dependencies. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 79–90. ACM.

[15] Arcuri, A., Iqbal, M. Z., and Briand, L. (2012). Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering (TSE)*, 38(2):258–277.

[16] Baresel, A. and Sthamer, H. (2003). Evolutionary testing of flag conditions. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 2442–2454. Springer.

[17] Baresel, A., Sthamer, H., and Schmidt, M. (2002). Fitness function design to improve evolutionary structural testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 1329–1336.

[18] Baresi, L., Lanzi, P. L., and Miraz, M. (2010). Testful: an evolutionary test approach for java. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 185–194. IEEE.

[19] Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., and Yoo, S. (2015). The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering (TSE)*, 41(5):507–525.

[20] Beizer, B. (2002). *Software testing techniques*. Dreamtech Press.

[21] Bell, J. and Kaiser, G. (2014). Unit test virtualization with VMVM. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 550–561. ACM.

[22] Beyene, M. and Andrews, J. H. (2012). Generating string test data for code coverage. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 270–279. IEEE.

[23] Böhme, M., Oliveira, B. C. d. S., and Roychoudhury, A. (2013). Partition-based regression verification. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 302–311. IEEE Press.

[24] Bühler, O. and Wegener, J. (2008). Evolutionary functional testing. *Computers & Operations Research*, 35(10):3144–3160.

[25] Burke, E. K., Gendreau, M., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., and Qu, R. (2013). Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724.

[26] Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C. S., Sen, K., Tillmann, N., and Visser, W. (2011). Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1066–1071. ACM.

[27] Cadar, C. and Palikareva, H. (2014). Shadow symbolic execution for better testing of evolving software. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 432–435. ACM.

[28] Cadar, C. and Sen, K. (2013). Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90.

[29] Campos, J., Arcuri, A., Fraser, G., and Abreu, R. (2014). Continuous test generation: enhancing continuous integration with automated test generation. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 55–66. ACM.

[30] Chen, H. Y., Tse, T., and Chen, T. Y. (2001). Taccle: a methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(1):56–109.

[31] Chen, T., Ananiev, L. I., and Tikhonov, A. V. (2007). Keeping kernel performance from regressions. In *Linux Symposium*, volume 1, pages 93–102.

[32] Chen, T. Y., Leung, H., and Mak, I. (2004). Adaptive random testing. In *Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making*, pages 320–329. Springer.

[33] Ciupa, I., Leitner, A., Oriol, M., and Meyer, B. (2006). Object distance and its application to adaptive random testing of object-oriented programs. In *Proceedings of Workshop on Random testing*, pages 55–63. ACM.

[34] Clark, D. and Hierons, R. M. (2012). Squeeziness: An information theoretic measure for avoiding fault masking. *Information Processing Letters*, 112(8):335–340.

[35] Csallner, C. and Smaragdakis, Y. (2004). JCrasher: an automatic robustness tester for java. *Software: Practice and Experience (SP&E)*, 34(11):1025–1050.

[36] Daka, E., Campos, J., Fraser, G., Dorn, J., and Weimer, W. (2015). Modeling readability to improve unit tests. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, pages 107–118. ACM.

[37] Daniel, B., Jagannath, V., Dig, D., and Marinov, D. (2009). Reassert: Suggesting repairs for broken unit tests. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 433–444. IEEE.

[38] De Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer.

[39] Di Penta, M., Bruno, M., Esposito, G., Mazza, V., and Canfora, G. (2007). Web services regression testing. In *Test and Analysis of web Services*, pages 205–234. Springer.

[40] Díaz, E., Tuya, J., and Blanco, R. (2003). Automated software testing using a metaheuristic technique based on tabu search. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 310–313. IEEE.

[41] Elbaum, S., Chin, H. N., Dwyer, M. B., and Dokulil, J. (2006). Carving differential unit test cases from system test cases. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, pages 253–264. ACM.

[42] Elbaum, S., Malishevsky, A. G., and Rothermel, G. (2000). Prioritizing test cases for regression testing. *SIGSOFT Software Engineering Notes*, 25(5).

[43] Elbaum, S., Rothermel, G., and Penix, J. (2014). Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, pages 235–245. ACM.

[44] Eler, M., Endo, A., and Durelli, V. (2014). Quantifying the characteristics of Java programs that may influence symbolic execution from a test data generation perspective. In *Proceedings of the International Conference on Computer Software and Applications Conference (COMPSAC)*, pages 181–190. IEEE.

[45] Evans, R. B. and Savoia, A. (2007). Differential testing: a new approach to change detection. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, pages 549–552. ACM.

[46] Fisher, R. A. (1922). On the Interpretation of χ^2 from Contingency Tables, and the Calculation of P. *Journal of the Royal Statistical Society*, 85(1):pp. 87–94.

[47] Fraser, G. and Arcuri, A. (2011a). Evolutionary generation of whole test suites. In *Proceedings of the International Conference on Quality Software (QSIC)*, pages 31–40. IEEE.

[48] Fraser, G. and Arcuri, A. (2011b). EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, pages 416–419. ACM.

[49] Fraser, G. and Arcuri, A. (2012). The seed is strong: Seeding strategies in search-based software testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 121–130. IEEE.

[50] Fraser, G. and Arcuri, A. (2013a). Handling test length bloat. *Software Testing, Verification and Reliability (STVR)*, 23(7):553–582.

[51] Fraser, G. and Arcuri, A. (2013b). Whole test suite generation. *IEEE Transactions on Software Engineering (TSE)*, 39(2):276–291.

[52] Fraser, G. and Arcuri, A. (2014). A large-scale evaluation of automated unit test generation using EvoSuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2).

[53] Fraser, G., Arcuri, A., and McMinn, P. (2014). A memetic algorithm for whole test suite generation. *Journal of Systems and Software*, 103(0).

[54] Fraser, G., Staats, M., McMinn, P., Arcuri, A., and Padberg, F. (2013). Does automated white-box test generation really help software testers? In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 291–301. ACM.

[55] Fraser, G. and Zeller, A. (2011). Exploiting common object usage in test case generation. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 80–89. IEEE.

[56] Fraser, G. and Zeller, A. (2012). Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering (TSE)*, 38(2):278–292.

[57] Freedman, R. S. (1991). Testability of software components. *IEEE Transactions on Software Engineering (TSE)*, 17(6):553–564.

[58] Galeotti, J. P., Fraser, G., and Arcuri, A. (2013). Improving search-based test suite generation with dynamic symbolic execution. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 360–369. IEEE.

[59] Ganesh, V., Kieżun, A., Artzi, S., Guo, P. J., Hooimeijer, P., and Ernst, M. (2011). Hampi: A string solver for testing, analysis and vulnerability detection. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 1–19. Springer.

[60] Gay, G. (2017). The fitness function for the job: Search-based generation of test suites that detect real faults. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. IEEE.

[61] Godefroid, P., Klarlund, N., and Sen, K. (2005). Dart: directed automated random testing. *ACM Sigplan Notices*, 40(6):213–223.

[62] Godefroid, P., Levin, M. Y., Molnar, D. A., et al. (2008). Automated whitebox fuzz testing. In *Proceedings of Network and Distributed Systems Security*.

[63] Godlin, B. and Strichman, O. (2009). Regression verification. In *Proceedings of the 46th Annual Design Automation Conference*, pages 466–471. ACM.

[64] Google (2014). *Analytix CodePro* URL: https://web.archive.org/web/20150906113523/https://developers.google.com/java-dev-tools/codepro/. Last visited on 26.08.2016.

[65] Graves, T. L., Harrold, M. J., Kim, J.-M., Porter, A., and Rothermel, G. (1998). An empirical study of regression test selection techniques. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 188–197. IEEE Computer Society.

[66] Graves, T. L., Karr, A. F., Marron, J. S., and Siy, H. (2000). Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering (TSE)*, 26(7):653–661.

[67] Grechanik, M., Fu, C., and Xie, Q. (2012). Automatically finding performance problems with feedback-directed learning software testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 156–166. IEEE.

[68] Grechanik, M., Xie, Q., and Fu, C. (2009). Maintaining and evolving gui-directed test scripts. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 408–418. IEEE.

[69] Groce, A., Holzmann, G., and Joshi, R. (2007). Randomized differential testing as a prelude to formal verification. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 621–631. IEEE.

[70] Harman, M. (2007). The current state and future of search based software engineering. In *2007 Future of Software Engineering*, pages 342–357. IEEE Computer Society.

[71] Harman, M. and Alshahwan, N. (2008). Automated session data repair for web application regression testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 298–307. IEEE.

[72] Harman, M., Burke, E., Clark, J. A., and Yao, X. (2012a). Dynamic adaptive search based software engineering. In *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–8. IEEE.

[73] Harman, M., Hu, L., Hierons, R., Baresel, A., and Sthamer, H. (2002). Improving evolutionary testing by flag removal. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*. MK Pub.

[74] Harman, M., Hu, L., Hierons, R., Wegener, J., Sthamer, H., Baresel, A., and Roper, M. (2004). Testability transformation. *IEEE Transactions on Software Engineering (TSE)*, 30(1).

[75] Harman, M. and Jones, B. F. (2001). Search-based software engineering. *Information and Software Technology*, 43(14):833–839.

[76] Harman, M. and McMinn, P. (2010). A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering (TSE)*, 36(2):226–247.

[77] Harman, M., McMinn, P., de Souza, J. T., and Yoo, S. (2012b). Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical Software Engineering and Verification*, pages 1–59. Springer.

[78] Harrold, M. and Souffa, M. (1988). An incremental approach to unit testing during maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 362–367.

[79] Harrold, M. J., Jones, J. A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S. A., and Gujarathi, A. (2001). Regression test selection for java software. In *ACM SIGPLAN Notices*, volume 36, pages 312–326. ACM.

[80] Harrold, M. J. and Rothermel, G. (1994). Performing data flow testing on classes. *ACM SIGSOFT Software Engineering Notes*, 19(5):154–163.

[81] Holland, J. H. (1975). *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press.

[82] Huang, P., Ma, X., Shen, D., and Zhou, Y. (2014). Performance regression testing target prioritization via performance risk analysis. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 60–71. ACM.

[83] Inozemtseva, L. and Holmes, R. (2014). Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 435–445. ACM.

[84] Islam, M. and Csallner, C. (2010). Dsc+Mock: A test case + mock class generator in support of coding against interfaces. In *Proceedings of the International Workshop on Dynamic Analysis (WODA)*, pages 26–31. ACM.

[85] Jaygarl, H., Kim, S., Xie, T., and Chang, C. K. (2010). OCAT: object capture-based automated testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 159–170. ACM.

[86] Jia, Y., Cohen, M. B., Harman, M., and Petke, J. (2015). Learning combinatorial interaction test generation strategies using hyperheuristic search. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 540–550. IEEE Press.

[87] Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering (TSE)*, 37(5):649–678.

[88] Just, R. (2014). The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 433–436. ACM.

[89] Just, R., Jalali, D., and Ernst, M. D. (2014a). Defects4J: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440. ACM.

[90] Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R., and Fraser, G. (2014b). Are mutants a valid substitute for real faults in software testing? In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, pages 654–665. ACM.

[91] Kifetew, F. M., Panichella, A., De Lucia, A., Oliveto, R., and Tonella, P. (2013). Orthogonal exploration of the search space in evolutionary test case generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 257–267. ACM.

[92] Kim, M. and Notkin, D. (2006). Program element matching for multi-version program analyses. In *Proceedings of the International Workshop on Mining Software Repositories (MSR)*, pages 58–64. ACM.

[93] King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394.

[94] Klein, J. P. and Moeschberger, M. L. (2003). *Survival Analysis Techniques for Censored and Truncated Data*. Springer Science & Business Media.

[95] Korel, B. (1990). Automated software test data generation. *IEEE Transactions on Software Engineering (TSE)*, 16(8):870–879.

[96] Korel, B., Tahat, L., and Vaysburg, B. (2002a). Model based regression test reduction using dependence analysis. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 214–223.

[97] Korel, B., Tahat, L. H., and Vaysburg, B. (2002b). Model based regression test reduction using dependence analysis. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 214–223. IEEE.

[98] Le Traon, Y., Jéron, T., Jézéquel, J.-M., and Morel, P. (2000). Efficient object-oriented integration and regression testing. *IEEE Transactions on Reliability*, 49(1):12–25.

[99] Leitner, A., Oriol, M., Zeller, A., Ciupa, I., and Meyer, B. (2007). Efficient unit test case minimization. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 417–420. ACM.

[100] Li, Y. and Fraser, G. (2011). Bytecode testability transformation. In *Symposium on Search-Based Software Engineering (SSBSE)*, pages 237–251. Springer.

[101] Li, Z., Harman, M., and Hierons, R. M. (2007). Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering (TSE)*, 33(4):225–237.

[102] Luo, Q., Hariri, F., Eloussi, L., and Marinov, D. (2014). An empirical analysis of flaky tests. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, pages 643–653. ACM.

[103] Ma, L., Artho, C., Zhang, C., Sato, H., Gmeiner, J., and Ramler, R. (2015). Grt: Program-analysis-guided random testing. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*.

[104] Marinescu, P., Hosek, P., and Cadar, C. (2014). Covrig: a framework for the analysis of code, test, and coverage evolution in real software. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 93–104. ACM.

[105] Marinescu, P. D. and Cadar, C. (2012). make test-zesti: A symbolic execution solution for improving regression testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 716–726. IEEE Press.

[106] Marinescu, P. D. and Cadar, C. (2013). Katch: high-coverage testing of software patches. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 235–245. ACM.

[107] Marré, M. and Bertolino, A. (2003). Using spanning sets for coverage testing. *IEEE Transactions on Software Engineering (TSE)*, 29:974–984.

[108] McKeeman, W. M. (1998). Differential testing for software. *Digital Technical Journal*, 10(1):100–107.

[109] McMinn, P. (2004). Search-based software test data generation: a survey. *Software Testing, Verification and Reliability (STVR)*, 14(2):105–156.

[110] McMinn, P. (2011). Search-based software testing: Past, present and future. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 153–163. IEEE.

[111] McMinn, P., Harman, M., Fraser, G., and Kapfhammer, G. M. (2016). Automated search for good coverage criteria: moving from code coverage to fault coverage through search-based software engineering. In *Proceedings of the 9th International Workshop on Search-Based Software Testing (SBST)*, pages 43–44. ACM.

[112] McMinn, P., Shahbaz, M., and Stevenson, M. (2012). Search-based test input generation for string data types using the results of web queries. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 141–150. IEEE.

[113] Memon, A., Nagarajan, A., and Xie, Q. (2005). Automating regression testing for evolving gui software. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(1):27–64.

[114] Memon, A. M. (2008). Automatically repairing event sequence-based gui test suites for regression testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(2):4.

[115] Memon, A. M. and Soffa, M. L. (2003). Regression testing of guis. *ACM SIGSOFT Software Engineering Notes*, 28(5):118–127.

[116] Mirzaaghaei, M., Pastore, F., and Pezze, M. (2010). Automatically repairing test cases for evolving method declarations. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 1–5. IEEE.

[117] Mirzaaghaei, M., Pastore, F., and Pezze, M. (2012). Supporting test suite evolution through test case adaptation. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 231–240. IEEE.

[118] Mouchawrab, S., Briand, L. C., Labiche, Y., and Di Penta, M. (2011). Assessing, comparing, and combining state machine-based testing and structural testing: a series of experiments. *IEEE Transactions on Software Engineering (TSE)*, 37(2):161–187.

[119] Myers, G. J., Sandler, C., and Badgett, T. (2011). *The art of software testing*. Wiley.

[120] Ng, S., Murnane, T., Reed, K., Grant, D., and Chen, T. (2004). A preliminary survey on software testing practices in australia. In *Proceedings of Software Engineering Conference, 2004 Australian*, pages 116–125. IEEE.

[121] Oriol, M. and Tassis, S. (2010). Testing. net code with yeti. In *Proceedings of IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 264–265. IEEE.

[122] Orso, A., Harrold, M. J., Rosenblum, D., Rothermel, G., Soffa, M. L., and Do, H. (2001). Using component metacontent to support the regression testing of component-based software. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 716–725. IEEE.

[123] Orso, A. and Xie, T. (2008). Bert: Behavioral regression testing. In *Proceedings of the International Workshop on Dynamic Analysis (WODA)*, pages 36–42. ACM.

[124] Pacheco, C. and Ernst, M. D. (2007). Randoop: feedback-directed random testing for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 815–816. ACM.

[125] Palikareva, H., Kuchta, T., and Cadar, C. (2016). Shadow of a doubt: testing for divergences between software versions. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1181–1192. ACM.

[126] Parasoft (2014). *Parasoft JTest* URL: http://www.parasoft.com/jtest. Last visited on 01.08.2014.

[127] Pargas, R. P., Harrold, M. J., and Peck, R. R. (1999). Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability (STVR)*, 9(4):263–282.

[128] Park, S., Hossain, B. M. M., Hussain, I., Csallner, C., Grechanik, M., Taneja, K., Fu, C., and Xie, Q. (2012). CarFast: achieving higher statement coverage faster. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, pages 35:1–35:11. ACM.

[129] Păsăreanu, C. S. and Rungta, N. (2010). Symbolic PathFinder: symbolic execution of Java bytecode. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 179–180. ACM.

[130] Pinto, L. S., Sinha, S., and Orso, A. (2012). Understanding myths and realities of test-suite evolution. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, page 33. ACM.

[131] Pradel, M., Huggler, M., and Gross, T. R. (2014). Performance regression testing of concurrent classes. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 13–25. ACM.

[132] Prasetya, I. W. B. (2014). T3, a combinator-based random testing tool for java: benchmarking. In *Future Internet Testing*, pages 101–110. Springer.

[133] Qi, D., Roychoudhury, A., and Liang, Z. (2010). Test generation to expose changes in evolving programs. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 397–406. ACM.

[134] Roest, D., Mesbah, A., and Van Deursen, A. (2010). Regression testing ajax applications: Coping with dynamism. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 127–136. IEEE.

[135] Rojas, J. M., Campos, J., Vivanti, M., Fraser, G., and Arcuri, A. (2015a). Combining multiple coverage criteria in search-based unit test generation. In *Search-Based Software Engineering*, pages 93–108. Springer.

[136] Rojas, J. M., Fraser, G., and Arcuri, A. (2015b). Automated unit test generation during software development: A controlled experiment and think-aloud observations. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 338–349. ACM.

[137] Rojas, J. M., Fraser, G., and Arcuri, A. (2016a). Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability (STVR)*.

[138] Rojas, J. M., Vivanti, M., Arcuri, A., and Fraser, G. (2016b). A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering*, pages 1–42.

[139] Rothermel, G. and Harrold, M. J. (1997). A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(2):173–210.

[140] Rothermel, G., Harrold, M. J., Ostrin, J., and Hong, C. (1998). An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 34–43. IEEE.

[141] Rothermel, G., Harrold, M. J., Von Ronne, J., and Hong, C. (2002). Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability (STVR)*, 12(4):219–249.

[142] Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. (1999). Test case prioritization: An empirical study. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 179–188. IEEE.

[143] Rothlauf, F. (2006). *Representations for Genetic and Evolutionary Algorithms*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[144] Rueda, U., Just, R., Galeotti, J. P., and Vos, T. E. (2016). Unit testing tool competition: round four. In *Proceedings of the 9th International Workshop on Search-Based Software Testing (SBST)*, pages 19–28. ACM.

[145] Runeson, P. (2006). A survey of unit testing practices. *IEEE software*, 23(4):22–29.

[146] Runeson, P., Andersson, C., Thelin, T., Andrews, A., and Berling, T. (2006). What do we know about defect detection methods? *Software, IEEE*, 23(3):82–90.

[147] Sakti, A., Pesant, G., and Guéhéneuc, Y.-G. (2015). Instance generator and problem representation to improve object oriented code coverage. *IEEE Transactions on Software Engineering (TSE)*, 41(3):294–313.

[148] Santelices, R., Chittimalli, P., Apiwattanapong, T., Orso, A., and Harrold, M. (2008). Test-suite augmentation for evolving software. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 218–227. IEEE.

[149] Santelices, R. and Harrold, M. (2011). Applying aggressive propagation-based strategies for testing changes. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 11–20.

[150] Savonia, A. and Evans, B. (2014). *Crap4J* URL: http://www.crap4j.org/. Last visited on 19.01.2015.

[151] Sen, K. and Agha, G. (2006). CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 419–423. Springer.

[152] Shamshiri, S. (2014). Evosuiter result processor. http://evosuiter.sina.sh.

[153] Shamshiri, S. (2015). Automated unit test generation for evolving software. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 1038–1041. ACM.

[154] Shamshiri, S., Campos, J., Fraser, G., and McMinn, P. (2016). Disposable testing: Avoiding maintenance of generated unit tests by throwing them away. In *Companion Proceedings of the International Conference on Software Engineering (ICSE-C)*. IEEE.

[155] Shamshiri, S., Fraser, G., McMinn, P., and Orso, A. (2013). Search-based propagation of regression faults in automated regression testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 396–399. IEEE.

[156] Shamshiri, S., Just, R., Rojas, J. M., Fraser, G., McMinn, P., and Arcuri, A. (2015a). Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 201–211. IEEE.

[157] Shamshiri, S., Rojas, J. M., Fraser, G., and McMinn, P. (2015b). Random or genetic algorithm search for object-oriented test suite generation? In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 1367–1374. ACM.

[158] Sharma, R., Gligoric, M., Arcuri, A., Fraser, G., and Marinov, D. (2011). Testing container classes: Random or systematic? In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer.

[159] Simons, A. J. H. (2006). A theory of regression testing for behaviourally compatible object types. *Software Testing, Verification and Reliability (STVR)*, 16(3):133–156.

[160] Simons, A. J. H. (2007). Jwalk: a tool for lazy, systematic testing of java classes by design introspection and user interaction. *Automated Software Engineering*, 14(4):369–418.

[161] Simons, A. J. H. and Thomson, C. D. (2008). Benchmarking effectiveness for object-oriented unit testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 375–379. IEEE.

[162] Skoglund, M. and Runeson, P. (2004). A case study on regression test suite maintenance in system evolution. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 438–442. IEEE.

[163] Taneja, K. and Xie, T. (2008). Diffgen: Automated regression unit-test generation. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 407–410. IEEE.

[164] Taneja, K., Xie, T., Tillmann, N., and de Halleux, J. (2011). express: guided path exploration for efficient regression test generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–11. ACM.

[165] Taneja, K., Zhang, Y., and Xie, T. (2010). Moda: Automated test generation for database applications via mock objects. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 289–292. ACM.

[166] Tassey, G. (2002). The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, pages 02–3.

[167] Technologies, A. (2014). *Agitar One* URL: http://www.agitar.com/solutions/products/agitarone.html. Last visited on 20.10.2016.

[168] Thummalapenta, S., Xie, T., Tillmann, N., De Halleux, J., and Su, Z. (2011). Synthesizing method sequences for high-coverage testing. *ACM SIGPLAN Notices*, 46(10):189–206.

[169] Tillmann, N. and De Halleux, J. (2008). Pex–white box test generation for .NET. In *Tests and Proofs*, pages 134–153. Springer.

[170] Tonella, P. (2004a). Evolutionary testing of classes. *ACM SIGSOFT Software Engineering Notes*, 29(4):119–128.

[171] Tonella, P. (2004b). Evolutionary testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 119–128. ACM.

[172] Vargha, A. and Delaney, H. D. (2000). A critique and improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong. *Educational and Behavioral Statistics*, 25(2).

[173] Veanes, M., De Halleux, P., and Tillmann, N. (2010). Rex: Symbolic regular expression explorer. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 498–507. IEEE.

[174] Voas, J. (1992). Pie: A dynamic failure-based technique. *IEEE Transactions on Software Engineering (TSE)*, 18(8):102–112.

[175] Voas, J. M. and Miller, K. W. (1993). Semantic metrics for software testability. *Journal of Systems and Software*, 20(3):207–216.

[176] Walcott, K. R., Soffa, M. L., Kapfhammer, G. M., and Roos, R. S. (2006). Timeaware test suite prioritization. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–12. ACM.

[177] Wegener, J., Baresel, A., and Sthamer, H. (2001). Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14).

[178] White, L. J. (1996). Regression testing of gui event interactions. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 350–358. IEEE.

[179] Wolpert, D. H. and Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82.

[180] Wong, W., Horgan, J., Mathur, A., and Pasquini, A. (1997). Test set size minimization and fault detection effectiveness: a case study in a space application. In *Proceedings of the International Conference on Computer Software and Applications Conference (COMPSAC)*, pages 522–528.

[181] Wong, W. E., Gao, R., Li, Y., Abreu, R., and Wotawa, F. (2016). A survey on software fault localization. *IEEE Transactions on Software Engineering (TSE)*, 42(8).

[182] Wong, W. E., Horgan, J. R., London, S., and Mathur, A. P. (1995). Effect of test set minimization on fault detection effectiveness. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 41–41. IEEE.

[183] Wood, M., Roper, M., Brooks, A., and Miller, J. (1997). Comparing and combining software defect detection techniques: a replicated empirical study. *ACM SIGSOFT Software Engineering Notes*, 22(6):262–277.

[184] Woodward, M. R. and Al-Khanjari, Z. A. (2000). Testability, fault size and the domain-to-range ratio: An eternal triangle. *ACM SIGSOFT Software Engineering Notes*, 25(5):168–172.

[185] Xiao, X., Xie, T., Tillmann, N., and De Halleux, J. (2011). Precise identification of problems for structural test generation. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 611–620. ACM.

[186] Xie, T. (2006). Augmenting automatically generated unit-test suites with regression oracle checking. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 380–403. Springer.

[187] Xie, T., Marinov, D., and Notkin, D. (2004). Rostra: A framework for detecting redundant object-oriented unit tests. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 196–205. IEEE Computer Society.

[188] Xie, T., Marinov, D., Schulte, W., and Notkin, D. (2005). Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 365–381. Springer.

[189] Xie, T., Taneja, K., Kale, S., and Marinov, D. (2007). Towards a framework for differential unit testing of object-oriented programs. In *Proceedings of the Second International Workshop on Automation of Software Test*, page 5. IEEE Computer Society.

[190] Xu, L., Dias, M., and Richardson, D. (2004). Generating regression tests via model checking. In *Proceedings of the International Conference on Computer Software and Applications Conference (COMPSAC)*, pages 336–341. IEEE.

[191] Xu, L., Xu, B., Chen, Z., Jiang, J., and Chen, H. (2003). Regression testing for web applications based on slicing. In *Proceedings of the International Conference on Computer Software and Applications Conference (COMPSAC)*, pages 652–656. IEEE.

[192] Xu, Z., Cohen, M. B., Motycka, W., and Rothermel, G. (2013). Continuous test suite augmentation in software product lines. In *Proceedings of the 17th International Software Product Line Conference*, pages 52–61. ACM.

[193] Xu, Z., Cohen, M. B., and Rothermel, G. (2010a). Factors affecting the use of genetic algorithms in test suite augmentation. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 1365–1372, New York, NY, USA. ACM.

[194] Xu, Z., Kim, Y., Kim, M., and Rothermel, G. (2011). A Hybrid Directed Test Suite Augmentation Technique. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 150–159.

[195] Xu, Z., Kim, Y., Kim, M., Rothermel, G., and Cohen, M. B. (2010b). Directed test suite augmentation: techniques and tradeoffs. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, pages 257–266. ACM.

[196] Xu, Z. and Rothermel, G. (2009). Directed test suite augmentation. In *Proceedings of Asia-Pacific Software Engineering Conference (APSEC)*, pages 406 –413.

[197] Yau, S. S. and Kishimoto, Z. (1987). A method for revalidating modified programs in the maintenance phase. In *Proceedings of the International Conference on Computer Software and Applications Conference (COMPSAC)*, volume 87, pages 272–277.

[198] Yoo, S. and Harman, M. (2012). Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability (STVR)*, 22(2):67–120.

[199] Yu, Y., Jones, J. A., and Harrold, M. J. (2008). An empirical study of the effects of test-suite reduction on fault localization. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 201–210. ACM.

[200] Zhang, P., Elbaum, S., and Dwyer, M. B. (2011). Automatic generation of load tests. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 43–52. IEEE Computer Society.

[201] Zhang, S., Jalali, D., Wuttke, J., Muşlu, K., Lam, W., Ernst, M. D., and Notkin, D. (2014). Empirically revisiting the test independence assumption. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 385–396. ACM.

[202] Zhu, H., Hall, P. A., and May, J. H. (1997). Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427.

Appendix A

Generating Differential Test Suites Using EvoSuiteR

A.1 Introduction

This appendix presents details of the EvoSuiteR tool as first described in Chapter 4, and provides information on how the tool can be used for the purpose of regression test suite generation in practice, or to replicate the data presented in this thesis. We made differential test generation using EvoSuite (a.k.a. EvoSuiteR) available as a built-in feature in EvoSuite, which is open source[1] and available on `http://www.evosuite.org`.

In this appendix, we first present the command-line version of the tool along with the available configuration options, and then present different use-cases of the tool along with example commands.

A.1.1 Requirements

In order to use EvoSuiteR, please make sure you have the latest version of Java Development Kit (JDK) 1.8 installed and available on your environment path. Additionally, the tool requires two versions of either the whole program or class(es) under test available, and compiled. EvoSuite operates at bytecode-

[1] Github Repository for EvoSuite: `https://github.com/EvoSuite/evosuite`

level (i.e., using compiled `.class` files), therefore, access to the source code is not required in order to generate unit tests.

A.2 Command-line and Configuration Options

At the time of this publication, EvoSuiteR is only available as a command-line tool. Therefore, we first look at the command-line arguments and the optional configuration options that are available to the users. Considering that EvoSuiteR is built on top of the EvoSuite test suite generation framework, it inherits a large number of different configuration options. For instance, users have the option to minimise a test suite after it has been generated. Although the minimisation process is performed differently for EvoSuiteR, it shares the same configuration property `minimize` with EvoSuite. For the purpose of brevity, in this section we present only the additional configuration options available on EvoSuiteR.

Before providing extra arguments to EvoSuiteR, it has to be executed under the regression testing mode. This can be enabled by passing `-regressionSuite` as the first parameter in the command line. In default settings, executing EvoSuiteR is similar to that of EvoSuite as below:

```
$ java -jar evosuite.jar -regressionSuite <target> [options]
```

In the command above, the EvoSuite's packaged jar is executed using Java, and the runtime argument `-regressionSuite` executes the framework under the regression testing mode – that is, the intention of the user is to generate a regression unit test suite. In the following two sections we look at how the target and options can be set.

Target: The *target* refers to the set of commands dedicated to setting the target program – or versions of the program in the case of EvoSuiteR – to be tested. In particular, EvoSuiteR can be executed on a particular pair of classes, or on all class-pairs across the two versions of the project.

Essentially, EvoSuiteR requires two sets of Java class paths[2] to be provided to the tool. The first one is `projectCP` which refers to the original class path of the program under test, and the second one is `regressioncp` which refers to the *alternate* class path. Given these two paths, EvoSuiteR will aims to generate tests that pass on the `projectCP` (e.g., the original version) and fail on the `regressioncp` (e.g., the changed version). Here is an example of how these class paths can be provided to the program on the command line:

```
$ java -jar evosuite.jar  -regressionSuite  -projectCP "original/
   build/classes" -Dregressioncp="modified/build/classes" <target>
    [options]
```

Notice that the `regressioncp` requires a property indicator `-D` as a prefix and an equal (`=`) sign to set the property value. This is due to the fact that `projectCP` is a shared property with EvoSuite. Also notice that we still have `<target>` in the command above. That is due the fact that the target program/class to be tested is still not specified. To test a class users can pass the `-class` property along with the full classname (e.g., `-class my.package.Foo`), and to test all class pairs across two versions of the program the `-target` property should be provided along with the name of the jar file containing the program under test (e.g., `-target my_program.jar`). More details of these two target properties will follow in this chapter.

Options: Custom configurations can also be applied on EvoSuiteR, such as setting parameters useful for the search, duration of the search, algorithm used for the search, and so forth. As mentioned earlier, EvoSuiteR is built upon and available as a native part of EvoSuite, and as such, it shares a majority of configurations available on EvoSuite. In this section we only discuss configurations and options specific to EvoSuiteR; the rest of the options can be found on EvoSuite's documentation.

All tool configurations can be set by providing `-D`*`property_name=value`*. The *property_name* refers to the name of the configuration property name. The most important configuration of EvoSuiteR is the search algorithm or the

[2]For more details on Java class path, please refer to the official documentation available at: `https://docs.oracle.com/javase/8/docs/technotes/tools/unix/classpath.html`. Please note that multiple classpath entries are separated by a semicolon `;` on Windows-based operating systems, and by a colon `:` on unix-based operating systems.

Fitness Function	Description
ALL_MEASURES	The combined EvoSuiteR GA*
STATE_DIFFERENCE	State Difference
BRANCH_DISTANCE	Control-Flow Distance
COVERAGE	Coverage – on both versions
COVERAGE_OLD	Coverage – on the original version
COVERAGE_NEW	Coverage – on the new/modified version
RANDOM	[default] By default, EvoSuiteR uses a random search for regression test suite generation. This option disables the GA mode.

Table A.1 Configuration options for EvoSuiteR's Genetic Algorithm.
*As mentioned in Chapter 4, by default, only Coverage and State-difference are included.

fitness function of the GA used for generating the differential tests. The property for this configuration is `regression_fitness`, and Table A.1 lists the possible values this property can have. By default, EvoSuiteR uses a random search algorithm for generating the tests. GA can be selected by providing the fitness function, e.g., `-Dregression_fitness=ALL_MEASURES`.

Other notable options available on EvoSuiteR are listed in Table A.2. All the listed options in the table are optional, boolean (can only have a value of true or false), and disabled (false) by default. Although some of these options will be discussed later in this appendix, it is worth noting that in order to use branch-distance measurement during the search, both of the configuration options `regression_branch_distance` and `regression_skip_different_cfg` need to be enabled. The former enables the calculation of branch-distance (affecting the first and third fitness functions in Table A.1), and the latter ensures that classes with different control-flow graphs are skipped – otherwise, the fitness function may produce incorrect values.

A.3 Differential Test Suite Generation

In this section we demonstrate how to generate a differential test suite for a Java class `Thesis`, under the package `shef.phd` – thus making the fully qualified name of the class: `shef.phd.Thesis`. For the sake of this example, we assume that we have access to the compiled `.class` file of two versions this class, stored respectively on `a/build/classes` and `b/build/classes` directories. To

Configuration	Description
regression_branch_distance	Enable/Disable control-flow distance calculation
regression_skip_different_cfg	Skip running EvoSuiteR on pairs of classes when the control-flow graph of the program is different between the two versions of the program
regression_skip_similar	Skip running EvoSuiteR on pairs of classes where the code inside the methods are unchanged between the two versions of the program
regression_statistics	Generate EvoSuiteR statistics
serialize_regression_test_suite	Serialise the generated test suite

Table A.2 Optional feature flags on EvoSuiteR. Each flag is a boolean parameter (TRUE|FALSE) and is by default disabled (FALSE).

generate a differential test suite to reveal the change between the two classes, the following command can be used:

```
$ java -jar evosuite.jar  -regressionSuite  -projectCP "a/build/
    classes" -Dregressioncp="b/build/classes" -class shef.phd.Thesis
```

This will generate a test suite for the given class and the test suite can be found under new directories that will be created by the tool. At the time of this writing, the default search budget is 60 seconds, and the default search algorithm for test generation is random test. The budget can be modified using the `-Dsearch_budget` property – more details on this can be found on EvoSuite's documentation – and the algorithm can be set using the property mentioned in the previous section.

A.4 Generating Test Suites for Whole Projects

While so far we have discussed and shown differential test suite generation for single classes, large Java projects may contain hundreds of thousands of classes, and executing EvoSuiteR individually for each class after a change has been made may be tedious. As a result, the tool has a functionality to generate test suites for all class pairs in the program. To achieve this, the user only needs access to two `.jar` files containing the two versions of the project. EvoSuiteR will then automatically generate tests for each pair of classes across the two `jar` files.

Assuming we have a `thesis.jar` in directory `a` and a new version of this program in directory `b`, and the two programs have different set of dependencies – which for the sake of brevity, we only include one dependency, but as many dependencies can be put on the classpath – the following command can be used to generate a test suite for all pairs of classes across the two versions:

```
$ java -jar evosuite.jar  -regressionSuite  -projectCP "a/thesis.jar:
   a/dependency.jar" -Dregressioncp="b/thesis.jar:b/dependency.jar"
   -target thesis.jar
```

When generating differential tests across two versions of a program, it is likely that not all classes have been modified across the two versions. EvoSuiteR has a functionality to only generate tests for pairs of classes where the underlying code has been changed. To enable this, you can add the property `-Dregression_skip_similar=true`.

A.5 Visualisation of the Search Outcome

To aid with analysing the outcome of the techniques, and help with the development of new search-based algorithms targeted at finding regression faults, we developed and make available a separate web-based tool. By enabling EvoSuiteR's individual statistics reporting using `-Dregression_statistics=true`, a separate statistic file is generated, which can then be uploaded to the tool for analysis and visualisation. Using this tool, users can observe the effectiveness of the technique at revealing changes on their specific problem over time, as well as observing the improvement of individual *GA* components over time. The service is currently available at: `http://evosuiter.sina.sh/`.

A.6 Summary

In this appendix we presented details of the EvoSuiteR tool and described how the tool can be interacted with. Particularly, we presented how the tool can be used to generate differential tests using different algorithms, as detailed earlier in this thesis. We also showed how the tool can be used in different scenarios such as generating tests for single classes or whole projects. Given

the fact that EvoSuite and EvoSuiteR are work under progress and the fact that the usage guideline detailed in this thesis may grow outdated with the software, up-to-date documentation on the tool can be found on the project's wiki: `https://github.com/EvoSuite/evosuite/wiki`.