Search-based Unit Test Generation for Evolving Software

The University Of Sheffield.

Search-based Unit Test Generation for Evolving Software

José Carlos Medeiros de Campos

Scientific supervision by
Dr. Gordon Fraser, The University of Sheffield, UK
Dr. Rui Abreu, Instituto Superior Técnico, Lisboa, PT

In partial fulfillment of requirements for the degree of
Doctor of Philosophy

The University of Sheffield
Faculty of Engineering
Department of Computer Science

November 2017

This thesis contains original work undertaken at The University of Sheffield, between 2013 and 2017. This work was funded by the Department of Computer Science of The University of Sheffield as a doctoral scholarship.

"Search-based Unit Test Generation for Evolving Software"
Copyright © 2017 by José Carlos Medeiros de Campos

Dedicated to my lovely family and friends.

ABSTRACT

Search-based software testing has been successfully applied to generate unit test cases for object-oriented software. Typically, in search-based test generation approaches, evolutionary search algorithms are guided by code coverage criteria such as branch coverage to generate tests for individual coverage objectives.

Although it has been shown that this approach can be effective, there remain fundamental open questions. In particular, which criteria should test generation use in order to produce the best test suites? Which evolutionary algorithms are more effective at generating test cases with high coverage? How to scale up search-based unit test generation to software projects consisting of large numbers of components, evolving and changing frequently over time? As a result, the applicability of search-based test generation techniques in practice is still fundamentally limited.

In order to answer these fundamental questions, we investigate the following improvements to search-based testing. First, we propose the simultaneous optimisation of several coverage criteria at the same time using an evolutionary algorithm, rather than optimising for individual criteria. We then perform an empirical evaluation of different evolutionary algorithms to understand the influence of each one on the test optimisation problem. We then extend a coverage-based test generation with a non-functional criterion to increase the likelihood of detecting faults as well as helping developers to identify the locations of the faults. Finally, we propose several strategies and tools to efficiently apply search-based test generation techniques in large and evolving software projects.

Our results show that, overall, the optimisation of several coverage criteria is efficient, there is indeed an evolutionary algorithm that clearly works better for test generation problem than others, the extended coverage-based test generation is effective at revealing and localising faults, and our proposed strategies, specifically designed to test entire software projects in a continuous way, improve efficiency and lead to higher code coverage. Consequently, the techniques and toolset presented in this thesis — which provides support to all contributions here described — brings search-based software testing one step closer to practical usage, by equipping software engineers with the state of the art in automated test generation.

PUBLICATIONS

The material presented in this thesis have been published in peer review symposiums or conferences.

[T1] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. "Combining Multiple Coverage Criteria in Search-Based Unit Test Generation". *Search-Based Software Engineering: 7th International Symposium, SSBSE 2015, Bergamo, Italy, September 5-7, 2015, Proceedings*. Ed. by Márcio Barros and Yvan Labiche. **Best Paper with industry-relevant SBSE results**. Cham: Springer International Publishing, 2015, pp. 93–108. ISBN: 978-3-319-22183-0.

[T2] José Campos, Yan Ge, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. "An Empirical Evaluation of Evolutionary Algorithms for Test Suite Generation". *Proceedings of the 9th International Symposium Search-Based Software Engineering (SSBSE)*. Ed. by Tim Menzies and Justyna Petke. **Distinguished Paper Award**. Cham: Springer International Publishing, 2017, pp. 33–48. ISBN: 978-3-319-66299-2.

[T3] José Campos, Rui Abreu, Gordon Fraser, and Marcelo d'Amorim. "Entropy-based Test Generation for Improved Fault Localization". *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. ASE'13. Silicon Valley, CA, USA: IEEE Press, 2013, pp. 257–267. ISBN: 978-1-4799-0215-6.

[T4] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. "Continuous Test Generation: Enhancing Continuous Integration with Automated Test Generation". *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. Vasteras, Sweden: ACM, 2014, pp. 55–66. ISBN: 978-1-4503-3013-8.

[T5] José Campos, Gordon Fraser, Andrea Arcuri, and Rui Abreu. "Continuous Test Generation on Guava". *Search-Based Software Engineering: 7th International Symposium, SSBSE 2015, Bergamo, Italy, September 5-7, 2015, Proceedings*. Ed. by Márcio Barros and Yvan Labiche. Cham: Springer International Publishing, 2015, pp. 228–234. ISBN: 978-3-319-22183-0.

[T6] Andrea Arcuri, José Campos, and Gordon Fraser. "Unit Test Generation During Software Development: EvoSuite Plugins for Maven, IntelliJ and Jenkins". *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Apr. 2016, pp. 401–408.

[T7] Gordon Fraser, José Miguel Rojas, José Campos, and Andrea Arcuri. "EvoSuite at the SBST 2017 Tool Competition". *Proceedings of the 10th International Workshop on Search-Based Software Testing*. SBST '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 39–41. isbn: 978-1-5386-2789-1.

In addition to the above list of research papers published during the PhD programme of study, I have also published the following papers, the work of which does not feature in this thesis.

[O1] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. "Modeling Readability to Improve Unit Tests". *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. **ACM SIGSOFT Distinguished Paper Award**. Bergamo, Italy: ACM, 2015, pp. 107–118. isbn: 978-1-4503-3675-8.

[O2] Ermira Daka, José Campos, Jonathan Dorn, Gordon Fraser, and Westley Weimer. "Generating Readable Unit Tests for Guava". *Search-Based Software Engineering: 7th International Symposium, SSBSE 2015, Bergamo, Italy, September 5-7, 2015, Proceedings*. Ed. by Márcio Barros and Yvan Labiche. Cham: Springer International Publishing, 2015, pp. 235–241. isbn: 978-3-319-22183-0.

[O3] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. "Evaluating and Improving Fault Localization". *Proceedings of the 39th International Conference on Software Engineering*. ICSE '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 609–620. isbn: 978-1-5386-3868-2.

[O4] Sina Shamshiri, José Campos, Gordon Fraser, and Phil McMinn. "Disposable Testing: Avoiding Maintenance of Generated Unit Tests by Throwing Them Away". *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. May 2017, pp. 207–209.

ACKNOWLEDGEMENTS

This thesis would have not been possible without the support of several people. The following lines are dedicated to all of them. First of all, I wish to express my gratitude to my supervisors Gordon Fraser and Rui Abreu for their support, advice, and patience throughout my time in academia. Special thanks are due to my collaborators Andrea Arcuri, José Miguel Rojas, Mattia Vivanti, Marcelo d'Amorim, Marcelo Eler, and Yan Ge. I am also very thankful to Sina Shamshiri, Mathew Hall, David Paterson, Tom White, Nasser Albunian, Ermira Daka, and Abdullah Alsharif for their support, their friendship, and for the many enjoyable moments we spent together.

I will be eternally grateful to my family, in particular to my parents and my sister, for their unconditional love and for encouraging me all the way. I hope you can forgive me for not being there when you needed me, but I was pursuing the dream we all started to pave years ago. My fellows in Portugal, *"Os Sardas"*, thank you for the lovely time we had every time I visited my *home sweet home*. Finally, I would like to thank my life partner and my *bestie*, Sandra (aka *"Flor"*). You are the best friend and partner I could have ever asked for. Thank you so much for your love, endless support, and patience during all these years.

CONTENTS

1 INTRODUCTION 1
1.1 Motivational Example . . . 2
1.2 Problem Statement . . . 5
1.3 Contributions . . . 6
1.4 Thesis Outline . . . 7
1.5 Origin of the Chapters . . . 8
2 LITERATURE REVIEW 11
2.1 Software Testing . . . 11
2.1.1 Concepts & Definitions . . . 12
2.1.2 Evaluating the Effectiveness of Testing . . . 12
2.1.3 Automated Test Generation . . . 14
2.2 Random Testing . . . 15
2.2.1 Adaptive Random Testing . . . 16
2.2.2 Effectiveness of Random Testing . . . 18
2.3 Symbolic Execution for Software Testing . . . 19
2.3.1 Dynamic Symbolic Execution . . . 21
2.4 Search-Based Software Testing . . . 23
2.4.1 Representation . . . 25
2.4.2 Random Search . . . 26
2.4.3 Local Search Algorithms . . . 26
2.4.4 Global Search Algorithms . . . 28
2.4.5 Fitness Functions . . . 30
2.4.6 Seeding . . . 36
2.4.7 Enhancing Search-based Software Testing with Symbolic Execution . . . 36
2.5 Regression Testing . . . 37
2.5.1 Test Case Minimisation, Selection, and Prioritisation . . . 38
2.5.2 Test Suite Maintenance . . . 40
2.5.3 Test Suite Augmentation . . . 40
2.5.4 The Oracle Problem . . . 41
2.6 The EvoSuite Unit Test Generation Tool . . . 42
2.7 Summary . . . 43
3 COMBINING MULTIPLE COVERAGE CRITERIA IN SEARCH-BASED UNIT TEST GENERATION 47
3.1 Introduction . . . 47
3.2 Whole Test Suite Generation for Multiple Criteria . . . 50
3.2.1 Fitness Functions . . . 51
3.2.2 Combining Fitness Functions . . . 54
3.3 Experimental Evaluation . . . 54
3.3.1 Experimental Setup . . . 55
3.3.2 Results and Discussion . . . 57

3.4 Related Work . . . 62
3.5 Summary . . . 62
4 AN EMPIRICAL EVALUATION OF EVOLUTIONARY ALGORITHMS FOR TEST SUITE GENERATION 63
4.1 Introduction . . . 63
4.2 Empirical Study . . . 65
4.2.1 Experimental Setup . . . 65
4.2.2 Parameter Tuning . . . 67
4.3 Experimental Results . . . 70
4.3.1 RQ1 – Which evolutionary algorithm works best when using a test archive for partial solutions? . . . 70
4.3.2 RQ2 – How does evolutionary search compare to random search and random testing? . . . 72
4.3.3 RQ3 – How does evolution of whole test suites compare to many-objective optimisation of test cases? . . . 73
4.4 Related Work . . . 75
4.5 Summary . . . 76
5 ENTROPY: A NON-FUNCTIONAL CRITERION TO IMPROVE THE DIAGNOSTIC ABILITY OF AUTOMATICALLY GENERATED UNIT TESTS 77
5.1 Introduction . . . 77
5.2 Background . . . 78
5.2.1 Spectrum-Based Fault Localisation (SBFL) . . . 79
5.2.2 Candidate Generation . . . 80
5.2.3 Candidate Ranking . . . 81
5.3 Entropy as a Non-Functional Criterion for Automated Test Generation . . . 83
5.3.1 Estimating Entropy: Coverage Density Fitness Function . . . 85
5.3.2 Integrating Coverage Density in Evolutionary Algorithms . . . 87
5.4 Empirical Study . . . 88
5.4.1 Experimental Setup . . . 88
5.4.2 Coverage Density Tuning . . . 94
5.4.3 RQ1 – Can optimisation of entropy improve the fault detection ability of automatically generated tests? . . . 96
5.4.4 RQ2 – Can optimisation of entropy improve the fault localisation ability of automatically generated tests? . . . 97
5.4.5 RQ3 – Does optimisation of entropy affect the coverage achieved or the number of automatically generated tests? . . . 99
5.5 Related Work . . . 100

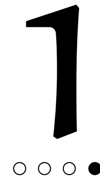
5.6 Summary 101
6 CONTINUOUS TEST GENERATION: ENHANCING CONTINUOUS INTEGRATION WITH AUTOMATED TEST GENERATION 103
6.1 Introduction 103
6.2 Testing Whole Projects 106
6.2.1 Simple Budget Allocation 107
6.2.2 Smart Budget Allocation 108
6.2.3 Seeding Strategies 110
6.3 Continuous Test Generation (CTG) 111
6.3.1 Budget Allocation with Historical Data 111
6.3.2 Seeding Previous Test Suites 112
6.4 Empirical Study 113
6.4.1 Experimental Setup 114
6.4.2 Testing Whole Projects 120
6.4.3 Continuous Test Generation 123
6.5 Related Work 128
6.6 Summary 129
7 UNIT TEST GENERATION DURING SOFTWARE DEVELOPMENT: EVOSUITE PLUGINS FOR MAVEN, INTELLIJ AND JENKINS 131
7.1 Introduction 131
7.2 Unit Test Generation in Build Automation 133
7.2.1 Integrating Generated Tests in Maven 133
7.2.2 Generating Tests with Maven 134
7.3 IDE Integration of Unit Test Generation 137
7.4 Continous Test Generation 139
7.4.1 Invoking EvoSuite in the Context of CTG 139
7.4.2 Accessing Generated Tests from Jenkins 140
7.5 Lessons Learnt 142
7.5.1 Lightweight Plugins 143
7.5.2 Compile Once, Test Everywhere 146
7.6 Summary 147
8 CONCLUSIONS & FUTURE WORK 149
8.1 Summary of Contributions 149
8.1.1 Optimisation of Multiple Coverage Criteria . . 149
8.1.2 Evolutionary Algorithms for Test Suite Generation 150
8.1.3 Diagnostic Ability of Automatically Generated Unit Tests 150
8.1.4 Continuous Test Generation 150
8.1.5 The EvoSuite Toolset 151
8.2 Future Work 151
8.2.1 Coverage Criteria 151
8.2.2 Hyper-heuristics Search Algorithms 152
8.2.3 Oracle Problem 152
8.2.4 Scheduling Classes for Testing 153
8.2.5 The EvoSuite Unit Test Generation Tool 154

BIBLIOGRAPHY 157

ACRONYMS

LOC	Lines of Code	82
MBD	Model-Based Diagnosis	80
MHS	Minimal Hitting Set	80
SBSE	Search-Based Software Engineering	23
SBST	Search-Based Software Testing	15
GA	Genetic Algorithm	28
WTS	Whole Test Suite	25
CUT	Class Under Test	51
CTG	Continuous Test Generation	105
SE	Symbolic Execution	19
DSE	Dynamic Symbolic Execution	15
RT	Random Testing	15
ART	Adaptive Random Testing	16
RRT	Restricted Random Testing	16
ART-B	Adaptive Random Testing by Bysection	17
ART-BR	Adaptive Random Testing by Bysection with Restriction	17
ART-RP	Adaptive Random Testing by Random Partitioning	17
D-ART	Distance-based Adaptive Random Testing	19
ARTOO	Adaptive Random Testing for Object-Oriented	18
SPD	Symbolic Program Decomposition	20
RWset	Read-Write set	20
SDSE	Shortest-Distance Symbolic Execution	21
CCBSE	Call-Chain-Backward Symbolic Execution	21
AVM	Alternative Variable Method	37
EAs	Evolutionary Algorithms	25
SBFL	Spectrum-Based Fault Localisation	79
HGS	Harrold-Gupta-Soffa	38
FEP	Fault Exposing Potential	39

INTRODUCTION



○ ○ ○ ●

"The majority of catastrophic software failures can easily be prevented by performing simple testing."

— Yuan et al., 2014 [1]

The idea of a having a mechanical machine executing a list of instructions, i.e., an *algorithm*, was first outlined by Ada Lovelance in the 19th century for the *Analytical Engine* initially invented by Charles Babbage. Around 100 years later, but before the invention of digital computers, Alan Turing [2] first proposed the theory of a computer program. However, a computer program as we currently know it — a collection of programmed instructions stored in the memory of a digital computer — was first written by Williams [3] in 1948 to calculate the highest factor of the integer 2^{18}. Before that, the very first electronic devices were rewired in order to be "reprogrammed". Ten years later, the word *software* was first introduced by John Tukey [4] in 1958 to describe a computer program.

Since then, *software* has helped humankind at achieving goals that would not have been possible without it. For instance, the software in the Lunar Module of Apollo 11 mission helped Neil Armstrong to land on the Moon; while, Curiosity, a car-sized robotic rover, has been exploring the planet Mars since 2012. Software also allows us to automate some of our daily activities, such as bank transactions, or to communicate with anyone anywhere. However, as the development of software still remains a manual activity, errors (that could lead to extremely dangerous situations for people) are involuntarily made. In 2017, Equifax, one of the largest credit reporting agencies in the US, reported that the records of 143 million users, i.e., names, social security numbers, credit card numbers, were stolen due to a vulnerable version of an external software they were using. Recently, the Health and Safety Executive ministry of the United Kingdom reported that, due to a software bug, thousands of medical scans such as X-Rays and ultrasounds might be incorrect and could have led to misdiagnoses, and therefore to a wrong type of treatment. Last year, the autopilot installed in a Tesla car may have caused the death of a human driver because "it was unable to recognise the white side of the tractor trailer, that had driven across the car's path, against a brightly lit sky".

How can we, as software developers, ensure the correctness of our own software?

Good practices on software engineering suggested a process named *software testing* to validate and verify the correctness of the software [5]. Validation aims to determine whether developers have built the correct software according to the user requirements — does the software do what it is supposed to do? On the other hand, verification aims to determine whether developers have built the software correctly — does the software correctly do what it has been specified? However, as famously stated by Dijkstra [6], "testing can be used to show the presence of bugs, but never to show their absence!". The reason behind his claim is that exhaustively testing a software is not feasible as the different number of inputs or configurations to execute a software could be extremely large or even infinite. Thus, to increase the confidence in the correctness of software (however not to prove it), testing aims to detect as many errors as possible (ideally, as soon as possible).

The increasing number of incidents due to software bugs over the years has raised software testing to become one of the most important aspects of the software development process. This however comes at a significant cost, making testing also one of the most expensive parts of the development lifecycle. It has been estimated that 50% of the total cost and time to develop software is fully dedicated to software testing [5]. Mostly, because (i) assessing whether a piece of software performs correctly could be extremely complex, and (ii) software testing is traditionally a manual process which is subject to incompleteness and further errors.

1.1 MOTIVATIONAL EXAMPLE

One of the most popular open-source Java libraries on GitHub is Google Guava[1]. It provides additional features to Java programs such as new collection types like multimap, APIs for concurrency, string processing, etc. In version 20, a new package called `graph` was added to Guava. The graph package provides graph-structured data which could be used to model, for example, airports and the routes between them. A common graph is composed by nodes, and edges. Each edge connects nodes to each other. By default, three types of graphs can be created: the common `Graph` (in which each edge is an anonymous connection between two nodes), `ValueGraph` (each edge is represented by a value), and `Network` (each edge is an unique object).

On August 23rd of 2017 an issue related to class `ValueGraph` of the graph package was reported [7] (Figure 1.1 illustrates the life cycle of the issue reported). The method reported as likely faulty was `edgeValueOrDefault` of `ConfigurableValueGraph` class (which is a sub class of `ValueGraph`). According to its documentation, "If there

1 At the time of writing this thesis, Google Guava project on GitHub `https://github.com/google/guava` had more than 20,000 stars.

is an edge connecting `nodeU` to `nodeV`, method `edgeValueOrDefault` returns the non-null value associated with that edge; otherwise, it returns a `defaultValue`.". For instance, assuming there are two nodes "A" and "B" connected by an edge ("A" to "B") with a value of 5, `edgeValueOrDefault(A, B, 10)` returns 5. On the other hand, `edgeValueOrDefault(B, A, 10)` returns the `defaultValue` (i.e., the third parameter, 10) because, however, there is one edge connecting both nodes, its direction is from "A" to "B" and not "B" to "A". When performing a code refactoring on the July 13th 2017 a bug was introduced. After the refactoring, `edgeValueOrDefault(B, A, 10)` started to return `null` rather than the value 10. However, Guava's developers only realised that 41 days later. On October 5th of 2017, 85 days after the bug was introduced and 44 days after being reported, the issue was fixed by the patch in Listing 1.1 [8]. It is worth noting that although the manually-written test cases fully exercise the code before and after the refactoring, the bug was not detected when introduced.

Listing 1.1: Fix for Guava issue #2924.

```
--- guava/src/com/google/common/graph/ConfigurableValueGraph.java
@@ 11d3683..a8f4ebc @@
 public V edgeValueOrDefault(N nodeU, N nodeV, @Nullable V
     defaultValue) {
   checkNotNull(nodeU);
   checkNotNull(nodeV);
   GraphConnections<N, V> connectionsU = nodeConnections.get(nodeU);
-  return connectionsU == null
-      ? defaultValue
-      : connectionsU.value(nodeV);
+  V value = (connectionsU == null) ? null : connectionsU.value(nodeV);
+  return value == null ? defaultValue : value;
 }
```

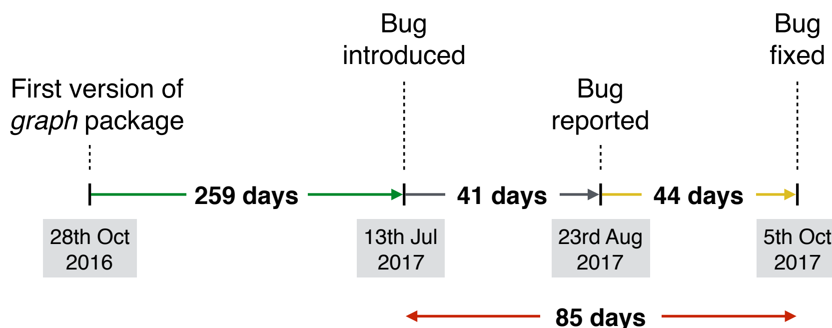


Figure 1.1: Life cycle of Guava issue #2924.

Could Guava's developers have fixed it earlier?

When the package `graph` was first introduced on Guava (on October 28th of 2016) it was accompanied with a set of test cases to verify the new functionalities and also to guard the code against potential future bugs. However, despite the fact that the accompanied

manually-written set of test cases fully exercise the package `graph` when first introduced, they did not exercise it with the right inputs, i.e., they were not able to detect the bug #2924.

Automating the creation of test cases offers several benefits over manually writing the test cases. It is computationally cheap to automatically generate test cases, and they are often more complete as they are generated in systematically way. Automatic test generation is a two step process: 1) generation of *test data*, i.e., inputs to exercise the software, and 2) generation of *test oracles* (also known as assertions) to verify whether the execution of the test data reveals any fault. Several techniques for *test data* generation have been proposed in the literature, including *random* testing, in which a software is executed with randomly generated inputs, *symbolic-execution* which explores control/data-paths of the software, and *search-based* testing in which efficient meta-heuristic search algorithms are used to generate test cases that resemble manually written tests (i.e., few short tests that exercise most of the code under test) are the most popular ones. The generation of *test oracles* is by far a much more challenging task, as without a formal specification of the software, it is not possible to automatically determine its correct behaviour. For this reason, automatic test generation is typically used in a *regression* scenario. That is, test data is generated to exercise the current version of the program, and test oracles are generated according to its current behaviour. These tests can then be used after performing modifications to the software, for example, to check whether a change lead to some undesired side-effects, i.e., to the introduction of a bug. Hence, would automatically generated tests have been able to detect Guava's issue?

EvoSuite [9], the state of the art tool on automatic test generation, is a search-based tool that uses a search-based algorithm to automatically generate test suites which aim at maximising code coverage of Java classes. If EvoSuite had been applied to Guava project right after the first version of class `ConfigurableValueGraph`, it would have automatically generated 39 test cases, one of which is reported in Listing 1.2.

Listing 1.2: Automatically generated test case that reveals Guava issue #2924.

```
@Test(timeout = 4000)
public void test34() throws Throwable {
  NetworkBuilder<Object, Object> network0 = NetworkBuilder.directed();
  ConfigurableMutableValueGraph<Presence, Object> graph0 = new
      ConfigurableMutableValueGraph<Presence, Object>(network0);
  Presence presence0 = Presence.EDGE_EXISTS;
  graph0.addNode(presence0);
  Object obj0 = new Object();
  Object obj1 = graph0.edgeValueOrDefault(presence0, presence0, obj0);
  assertSame(obj0, obj1);
}
```

This test case creates a graph with a single node and then evaluates the outcome of function `edgeValueOrDefault`. As by default there is an edge connecting any existing node to itself with a `null` value, the method `edgeValueOrDefault` returns `obj0` (as it is supposed to according to its documentation). However, when bug #2924 was introduced, this test case would have failed because, rather than returning `obj0`, method `edgeValueOrDefault` would have returned `null`. At this point, a developer would have had to inspect test case "`test34`" (which is only 7 lines long) in order to understand whether the test is revealing a bug or it is obsolete (e.g., if the specification of a requirement has changed). In this case, the test case would have not been considered obsolete and it would have indeed revealed the bug introduced. Therefore, the answer to the question *"Could Guava's developers have fixed it earlier?"* is yes. If they had used a tool such as EvoSuite to automatically generate test cases for the package `graph`, they would have been able to detect the bug right when it was introduced without having to manually write any test case.

1.2 PROBLEM STATEMENT

Test cases for object-oriented programs (e.g., Guava library described in the previous section) are programs themselves. Each test is a sequence of program invocations which create and manipulate objects to exercise and test the correctness of a particular behaviour of the program under test (e.g., test case in Listing 1.2). Despite the number of test generation techniques proposed in the literature, search-based testing has been the most successful technique at generating tests for object-oriented programs. However, there are a number of open problems that need to be addressed in order to improve the effectiveness and efficiency of search-based test generation (in particular, when applied on programs that are, typically, developed continuously). In summary, the main issues addressed by this thesis are:

– As software testing can only show the presence of bugs but not their absence, ideally, automated test generation techniques should explore different properties of the software under test as much as possible in order to find those bugs. However, automated test generation techniques in the literature are mostly guided by a single coverage criterion, i.e., branch coverage. Which other coverage criteria can be explored? How can a search-based algorithm efficiently optimise several coverage criteria simultaneously?

– Genetic algorithms are the most common search-algorithm used in search-based software testing. However, there is a large number of other search-based algorithms that are also suitable for automatic test generation. Which search-based algorithm works

best at generating unit tests for object-oriented software programs?

– Although a bug can only be detected by a test case if and only if it exercises the faulty code, only optimising test cases for code coverage might not be enough, as the faulty code needs to be covered with the *right* input in order to trigger the faulty behaviour. Which non-coverage criteria can be optimised to improve the ability of automatically generated test cases at detecting and findings faults? How can a search-based algorithm be extended to optimise coverage and non-coverage criteria at the same time?

– The current literature on automatic test generation makes the assumption that each component of a software program (e.g., a class in Java) is tested independently and in isolation. However, a software is usually composed by thousands of components, each depending on another and evolving over time. Considering a software program and its evolution as a whole, which components should be subjected to test generation? In which order are components tested? How much time can be allocated to test each component?

– Although several automated test generation tools have been proposed in the literature, there is still a lack of adoption from practitioners. Which development environments are worth to integrate automated test generation techniques? Which challenges are faced when integrating automated test generation techniques into developers' processes?

1.3 CONTRIBUTIONS

In this section we outline the five main contributions of this thesis. The first three contributions correspond to the application of search-based test generation on a single version of a software program, and the last two contributions are based on the integration of automatic test generation techniques in a continuous testing scenario.

Multiple Coverage Criteria. Our first contribution is mainly motivated by the fact that most literature on automated test generation is guided by a single coverage criterion, i.e., branch coverage. As the optimisation of a single criterion may not exhibit other properties of the software under test, we define five coverage criteria, and extend search-based test generation to optimise the combination of those five in addition to the ones commonly used in the literature (i.e., statement and branch coverage, and weak mutation).

Evolutionary Algorithms for Test Suite Generation. Despite the fact that a simple genetic algorithm can achieve good results on average, many other evolutionary algorithms are suitable for test generation. We perform an empirical evaluation of six evolutionary algorithms and two random approaches at optimising a single criterion (i.e., branch coverage) and at optimising several criteria (i.e., the criteria defined in the previous contribution).

Diagnostic Ability of Automatically Generated Unit Tests. As stated by our first contribution, other properties (and not just branch coverage, or not even just coverage criteria but also non-functional criteria) of the software program may be explored in order to exhibit properties developers would desire, e.g., ability to automatically find faults. We propose a non-functional criterion to improve the effectiveness of coverage-based unit tests at detecting and localising faults. We integrate it on the most effective evolutionary algorithm found by our second contribution.

Continuous Test Generation. A typical automated unit test generation technique targets one component of a software program (e.g., a class in Java) at a time. A class, however, is usually part of a software project consisting of many more classes which are subject to changes over time. We introduce Continuous Test Generation (CTG), which includes automated unit test generation during continuous integration. CTG offers several benefits over traditional test generation approaches: first, it answers the question of how much time to spend on each class; second, it helps to decide in which order to test them. Finally, it answers the question on which classes test generation should be applied.

The EvoSuite Toolset. In order to improve the integration of the EvoSuite test generation tool into the development process of software engineers, we present a set of new plugins for Maven, IntelliJ IDEA, and Jenkins; and we also report the challenges arisen when developing those plugins.

1.4 THESIS OUTLINE

This thesis is structured as follows. First, the state of the art on software testing and on automatic test generation is reviewed in Chapter 2. Then, the five contributions described in the previous section are presented in detail in five chapters. The research topic(s) that each chapter contributes to is shown in Figure 1.2.

The first three chapters focus on automatic test generation for a single version of a software under test. Chapter 3 proposes the optimisation of several coverage criteria for test generation. Chapter 4 eval-

uates which evolutionary algorithm works best for test generation. Chapter 5 presents a non-functional criterion which aims to improve the diagnostic ability of a coverage based test generation approach.

Chapters 6 and 7 are devoted to the application of automatic test generation on software that evolves over time. Chapter 6 introduces the concept of continuous test generation and describes several approaches to efficiently apply automatic test generation on every iteration of software that is, typically, developed continuously. Chapter 7 describes the development of three new plugins for the state of the art tool on automatic test generation, i.e., EvoSuite, which aim to reduce the gap between what tools are proposed by researchers and what is actually used by practitioners in industry. It also discusses lessons learnt when developing and evaluating these plugins in practice.

Finally, Chapter 8 presents our final conclusions and discusses potential directions for future work.

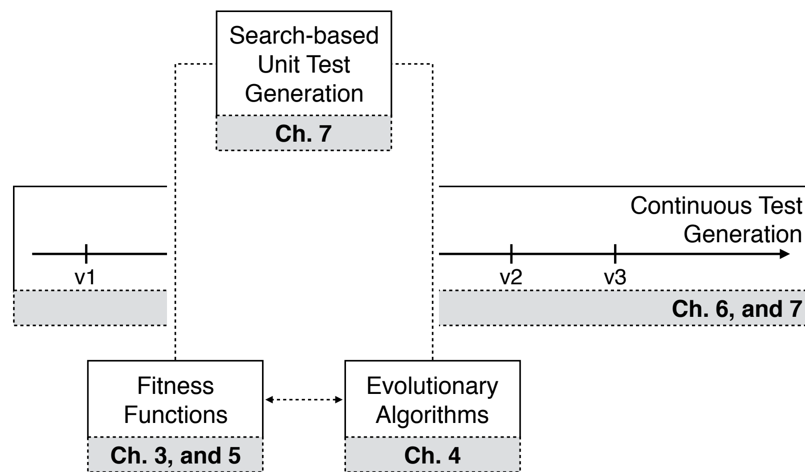


Figure 1.2: Thesis outline.

1.5 ORIGIN OF THE CHAPTERS

Besides Chapters 1, 2 and 8, each chapter of this thesis is based on at least one paper published in a peer review symposium or international conference. The following list summarises these publications per chapter.

Chapter 3 is based on a paper published in the proceedings of the 7th International Symposium on Search-Based Software Engineering (SSBSE), 2015 [10].

Chapter 4 is based on a paper published in the proceedings of the 9th International Symposium on Search-Based Software Engineering (SSBSE), 2017 [11].

Chapter 5 is based on material published in the proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2013 [12].

Chapter 6 is based on a paper published in the proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE), 2014 [13]; and it is also based on a paper published in the proceedings of the 7th International Symposium on Search-Based Software Engineering (SSBSE), 2015 [14].

Chapter 7 is based on a paper published in the proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST), 2016 [15]; and and it is also based on a paper published in the proceedings of the 10th International Workshop on Search-Based Software Testing (SBST), 2017 [16].

2

LITERATURE REVIEW

In this chapter we survey the most relevant concepts and works related to the contributions of this thesis. We first introduce software testing, then we review the state of the art techniques on automated test generation and automated regression testing.

2.1 Software Testing 11
2.2 Random Testing 15
2.3 Symbolic Execution for Software Testing 19
2.4 Search-Based Software Testing 23
2.5 Regression Testing 37
2.6 The EvoSuite Unit Test Generation Tool 42
2.7 Summary 43

2.1 SOFTWARE TESTING

> *"Software testing is the process of operating a system or component under specified conditions, observing and recording the results, and making an evaluation."*
>
> — IEEE Std. 610.12-1990 [17]

Despite the variety of different software development processes such as the waterfall model or agile, there is one activity that is shared by all of them, *software testing*. Software testing is conducted by developers to check the correctness and completeness of the code they wrote, and to guard it against future regression faults. However, as the software cannot be exhaustively tested in general, it can never show the absence of faults (as claimed by Dijkstra [6] in 1972). Thus, the main goal of software testing is to find as many faults in the software as possible.

Software testing has become such an important piece of software development process, that it is commonly estimated that half of the total cost/time to develop a software program is dedicated to testing & debugging [5]. The reason is that, although it is very common to use automated tools to execute test cases, such test cases are commonly hand-written which is a tedious and error prone task. Automating the creation of such test cases offers several benefits, however it also raises some issues that would have to be addressed in order to be useful to use those tests. On one hand, automation could reduce the cost/time of the testing process, and it could also create a much more complete

set of test cases (as they would be systematically generated). On the other hand, there are two main issues that need to be considered when generating test cases automatically: 1) *test data* (which inputs should be used to exercise the software under test?), and 2) *test oracle* (does the execution of the test reveal any fault?).

In the following sections, we describe the concepts and definitions in software testing, and how to measure the quality of a test suite. Then, we survey the state of the art techniques on automated test generation.

2.1.1 *Concepts & Definitions*

A *test suite* is a collection of *test cases* for a target software under test which comprises a set of methods or functions, each of which consists of a list of statements. Each statement can be a conditional statement (e.g., `if`), a method call or a *regular* statement. A conditional statement results in two branches depending on the evaluation of its predicate. A *test case* is an executable function which sets up a test scenario, calls some methods/functions in the software under test, and checks that the observed behaviour matches the expected one — typically by using *test oracles* also known as *assertions*. For simplicity, a test case can be regarded as a sequence of calls to methods of the software under test. Executing a test case yields an *execution trace*, i.e., a sequence of executed statements which can either end normally with a regular statement, or with an uncaught exception. If the execution of a test case does not match the expected behaviour it can indicate a *defect* in the software under test. A *defect* is a flaw, or imperfection in the software under test, such as an incorrect design, algorithm, or implementation. It is also known as a *fault* or *bug* and it is typically identified when a test case throws an *error* or a *failure*. An *error* is a discrepancy between the intended behaviour of a system and its actual behaviour inside the system boundary. A *failure* is an instance in time when a system displays behaviour that is contrary to its specification.

Throughout this thesis we may use the terms test suite and suite interchangeably, in which case we normally intend the former. Moreover, we may use the terms test and test case interchangeably, in which case we normally intend the latter. Furthermore, we may use the terms *fault* and *bug* to refer to a *defect*, and *failure* to refer to both cases of *errors* and *failures*.

2.1.2 *Evaluating the Effectiveness of Testing*

Due to the large number of parameters or configurations to test a software program, it is infeasible (in practice) to test all possible combinations. So, when should we stop testing? Which parameters or configurations are more adequate, e.g., more likely to reveal faults?

In order to answer these questions, several techniques to measure the quality of test cases have been proposed [18]. The two most common techniques are *coverage analysis* and *mutation analysis*.

2.1.2.1 *Coverage Analysis*

Coverage analysis is a deterministic technique which uses coverage criteria to evaluate whether there is at least one test case exercising each pre-defined coverage target. Each target is commonly represented by a single statement, branch, or condition in the software. For instance, the statement coverage criterion requires all statements in the software to be executed by at least one test. Branch coverage criterion requires all branches in the software to be satisfied at least once during testing. For example, the software under test in Listing 2.1 is fully covered at branch level if, one of the sub-conditions on line 2 is evaluated as true, and if both sub-conditions on line 2 are evaluated as false by at least one test case. However, for this particular software under test, just exercising one of the conditions to satisfy the branch may not be enough to cover the faulty code (i.e., `a < c`). Conditional coverage criterion, on the other hand, requires *all* sub-conditions in a conditional statement to be satisfied at least once during testing. For example, to fully satisfy conditional statement on line 2, both sides of each sub-condition have to be satisfied by at least one test case — a total of four test cases, assuming one test case per side of each sub-condition.

Listing 2.1: Motivational example for the evaluation of the effectiveness of software testing. On line 2 there is a fault, instead of a < c it should be read b < c.

```
1 public boolean foo(int a, int b, int c) {
2   if (a > b || a < c) { /* FAULT */
3     return true;
4   }
5   return false;
6 }
```

2.1.2.2 *Mutation Analysis*

The purpose of software testing is to find faults in the software. However, as the location of those faults is usually unknown (otherwise they would have been fixed), the effectiveness of test cases at detecting potential faults is, typically, measured on *artificial* faults. These faults (also known as *mutants*) are small syntactic variations created by applying mutation operators to the original code. Typically, mutation operators only replace relational operations, modify conditional statements, or delete statements [19]. For example, supposing the original program described in Listing 2.1, a single mutant could be created by changing `a > b` to `a < b` on line 2. A mutant is *killed* if

there is at least one test case that reveals the changed behaviour, i.e., its outcome is different when executed on the mutated and original program. Otherwise, the mutant is considered *alive*. The effectiveness of a test suite is measured by the ratio (also known as *mutation score*) of the number of mutants which are *killed* by the test suite / total number of mutants.

Although mutation analysis has been used in several testing scenarios such as test generation [20], regression testing [21, 22], fault localisation [23], etc., several limitations are still a barrier for mutation testing techniques being adopted in practice. For instance, the efficiency of mutation testing. Mutants can be automatically and systematically created [24, 25], however, each mutant requires the execution of all test cases. That is, given a large software program for which thousands of mutants can be created, executing all test cases against each mutant would be extremely expensive. A recent study conducted by Pearson et al. [23] on the effectiveness of coverage-based and mutation-based fault localisation techniques, reported that, experiments with mutation-based techniques took more than 100,000 CPU hours to complete. To alleviate this limitation several approaches have been proposed to reduce the number of mutants, for example, by identifying *redundant* mutants [26] — semantically equivalent variations of the original software. However, the identification of *redundant* mutants is an undecidable problem [27].

As mutants have been using as a proxy to real faults, one might ask whether test suites that are effective at detecting mutants are also effective at detecting real faults [28]. A recent study conducted by Just et al. [29] compared the effectiveness of manually written and automatically generated test cases at detecting real faults and mutants. They found that for the majority of faults there is a correlation between detecting real faults and mutants. For a large sample of those faults for which such correlation did not exist, stronger or new mutation operators are required.

2.1.3 *Automated Test Generation*

Software testing is still the most effective approach at ensuring a software program does what it was designed for. However, manually writing test cases is an error prone and time consuming task. To reduce this cost, researchers have devised approaches to automate the generation of test cases.

Some of the proposed approaches assume the existence of a formal model of the software (e.g., UML [30]), and many other popular approaches require only source code. To generate test cases from source code, the simplest approach is to do so *randomly* [31]. This approach can produce large numbers of tests in a short time, and the main intended usage is to exercise generic object contracts [32, 33] or code

contracts [34]. Approaches based on symbolic execution have been popularized by *Dynamic Symbolic Execution (DSE)* [35], which systematically explore paths from a given entry point of the program under test. Generating tests that resemble manually written tests (i.e., few short test cases with high coverage) is commonly done using *Search-Based Software Testing (SBST)*. When applying SBST for test generation, efficient meta-heuristic search algorithms such as genetic algorithms are used to evolve sequences of method calls with respect to a certain set of criteria, e.g., coverage [36].

2.2 RANDOM TESTING

The most naïve test generation technique is Random Testing (RT). In RT, the software under test is exercised with *randomly* [31] generated inputs from the whole input domain of the software, and its observed output. Due to its simplistic nature, RT can be applied in practice with litter overhead and it has been widely used to, for example, exercise generic object contracts [32, 33], code contracts [34], unexpected security problems [37], and to reveal failures in several software systems [35, 38]. However, there are some disagreements between researchers and practitioners on the coverage and effectiveness achieved by RT techniques on test generation [5, 39, 40]. The main point of criticism among researchers is the lack of a strategy to generate inputs, as RT techniques do not take into account any information about the software under test [5], i.e., in theory, every test input in the input domain has the same probability of being selected. For example, consider the code under test in Listing 2.2. The probability of the conditional statement `if (x == 10)` being satisfied is 1 in 2^{32} (assuming `x` is a 32-bits value), which illustrates the limitation of RT approaches.

Listing 2.2: Motivational random testing example adapted from Godefroid et al. [37].

```
public String returnTen(int x) {
  if (x == 10) {
    return "Six"; /* FAULT */
  } else {
    return "Other number";
  }
}
```

The technique proposed by Pacheco et al. [33] (and the accompanied tool named Randoop [41]) is slightly different from the pure random technique described above. Randoop is a feedback-oriented technique which explores the execution of tests as they are created to avoid generating invalid inputs. First, it generates a sequence of methods calls (each one selected at random), and methods arguments from previously created sequences. Then, it executes a sequence in order

to provide feedback to the test generator, e.g., to avoid generation of tests that lead to runtime exceptions or to generate assertions that could trigger future changes. It has been shown [33] that Randoop is able to generate tests that are able to detect previously-unknown errors (not found by pure random techniques) in widely used Java libraries. However, the large number of test cases generated by random testing techniques (including Randoop) may limit their adoption in practice. As executing, evaluating, and maintaining such tests can become impractical over time.

2.2.1 *Adaptive Random Testing*

Based on the assumption that inputs that could trigger a failure are localised on continuous regions [42–45] of the input domain, Adaptive Random Testing (ART) was first proposed by Chen et al. [46] as an enhanced alternative to RT. The idea behind ART is that if any previously generated input have not revealed the failure, new inputs should be widely spread across the input domain to increase the likelihood of covering likely faulty areas.

Several approaches have been proposed on ART. For instance, the approach proposed by Chen et al. [47] starts by generating a single random test input and adds it to a pool of test inputs. Then for each test input in the test pool, it generates a set of k (recommended value is $k = 10$) random test inputs. The k_i input with the highest euclidean distance to all previously selected test inputs (i.e., the ones in the test pool) is selected and added to the pool of test inputs. The main disadvantages of this approach are: 1) the size of the pool of test inputs could grow out of hands, and 2) the euclidean distance could be very expensive to calculate for a large pool of test input.

Chen et al. [48] described the input domain as an m-dimensional hyper-cube and generated inputs that are evenly-spaced as mathematical possible across the input domain. Due to the limited number of inputs their approach generates on each iteration [48], it can only be applied on problems with a finite number of dimensions [49]. Although previous studies [48, 50] showed that their approach could produce better test inputs than RT, there is little evidence that it could perform better than any other ART approach [51].

Restricted Random Testing (RRT) proposed by Chan et al. [52] is an ART approach that *excludes* areas of the input domain. RRT starts by randomly generating a test input from the entire input domain (for example, test input t_1 in the left side of Figure 2.1) and creating an exclusion region around t_1. Then, new test candidates are generated, for example, c_1 and c_2. However, as they are in an exclusion region, both are discarded. If a test candidate is successfully generated out of an exclusion region (e.g., c_3), it becomes a valid test input (e.g., t_2) and a new exclusion region around it is created. If an exclusion

region is to small, similar test inputs could be generated. On the other hand, if an exclusion region is too large, similar inputs would never be generated and the total number of inputs that could be explored would be limited. It is worth noting that outside of exclusion regions candidates are selected with the same probability.

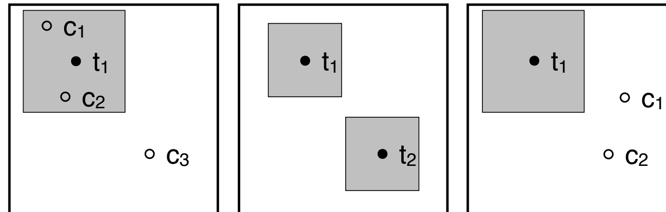


Figure 2.1: RRT example adapted from Liu et al. [53].

In order to verify whether a new candidate is inside/outside of an exclusion region, RRT approach measures the euclidean distance between the new candidate and all test inputs previously selected, which could be very time consuming for a large number of test inputs. To reduce this computational overhead, approaches such as *mirroring* [54] and *forgetting* [55] have been proposed. In the *mirroring* approach, ART is just applied to a sub-domain and then test inputs are mirrored to other sub-domains. In the *forgetting* approach, only a constant number of previously generated test inputs (and not all of them) are considered when evaluating new ones.

Similar to the idea of partitioning the input space, Chen et al. [56] proposed two other approaches: Adaptive Random Testing by Bysection (ART-B) and Adaptive Random Testing by Random Partitioning (ART-RP). As in a typical ART approach, ART-B and ART-RP first generate a random test input from the entire input domain (according to a uniform distribution). Then, the ART-B approach bisects the input domain into two (in the case of two-dimensional input domain) equal-sized partitions (see Figure 2.2). Any following test candidate can only be selected as a valid test input, if it is in the empty partition (which does not contain any previously generated test input) rather than from the whole input domain. On the other hand, the ART-RP approach partitions the input domain at the selected test input and the following test candidates can only be selected if they are from the empty partition (see Figure 2.3). The generation of test inputs stops when a termination condition is met, e.g., there are no more partitions to explore.

The issue shared by both ART-B and ART-RP approaches is that all test inputs from an empty partition have the same probability of being selected. Therefore, test inputs that are close to previously generated inputs could be selected, which could decrease the effectiveness of the tests [57]. To address this issue several approaches based on localisation of previously selected test cases have been proposed. For instance, Adaptive Random Testing by Bysection with Restric-

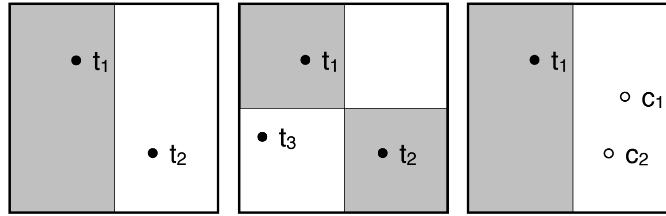


Figure 2.2: ART-B example adapted from Liu et al. [53].

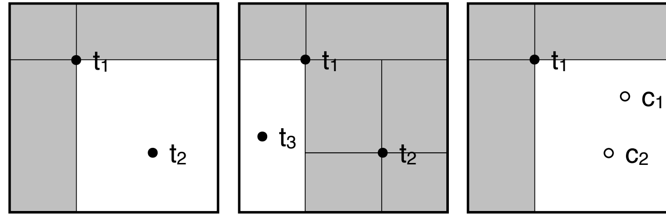


Figure 2.3: ART-RP example adapted from Liu et al. [53].

tion (ART-BR) [58] approach introduces a restriction which prevents test candidates that are near in the input domain to a previously selected partition or test input, of being selected (similar to the RRT approach proposed by Chan et al. [52]).

Opposite to previous approaches, Tappenden et al. [49] applied an evolutionary search algorithm to find a new test input that maximises the minimum distance from all previously generated test inputs. The evolution is guided by a distance-based fitness function, e.g., the euclidean distance.

Ciupa et al. [59] applied ART on object-oriented programs instead of numeric ones and proposed an approach named Adaptive Random Testing for Object-Oriented (ARTOO) software. In their approach the distance between objects is the combination of elementary distance (the distance value for primitive/reference types, e.g., levenshtein distance for string attributes), type distance (the path length to any similar ancestor, and the number of their attributes/methods), and field distance (the distance value to other objects). ARTOO uses the average distance between objects rather than the maximum minimum distance as typically used by ART techniques. Although it has been shown that ARTOO does not generate as many test inputs as RT, it requires more time to detect failures than RT [59]. Lin et al. [60] also proposed an approach to generate test inputs for object-oriented programs named ARTGen. ARTGen creates a pool of objects and boundary values of the input space, and then uses ART to select objects from the pool.

2.2.2 *Effectiveness of Random Testing*

Although random testing approaches have been studied in detail, there are different conclusions in the literature about the effective-

ness of it. Thayer et al. [39] argued that RT should be recommended as a fundamental step of the testing process, however Myers et al. [5] stated that RT is the poorest test input methodology.

Mak [61] compared RT and ART in terms of number of test inputs required to detect the first failure, and concluded that ART is able to detect the first failure with 30% (occasionally 50%) less test inputs. Although ART may be quicker or require less test inputs to detect the first failure than RT, ART requires more computational time and memory because the additional task of generating test inputs evenly spread across the input domain [49, 56, 62, 63]. An empirically study conducted by Mayer et al. [64] confirmed that although Distance-based Adaptive Random Testing (D-ART) and RRT are the most effective ART approaches, their runtime may become extremely long. More recently, Arcuri et al. [65] reported that although ART could perform better than RT, the chance of finding faults with ART is less than 1%.

In the next section we survey another testing technique which rather than generating random inputs to explore a program under test, it generates specific inputs to exercise a given path of the program.

2.3 SYMBOLIC EXECUTION FOR SOFTWARE TESTING

Symbolic Execution (SE) is a program analysis approach that executes a software program with symbolic values instead of concrete inputs (as the approaches discussed in Sections 2.2 and 2.4), and represents the values of program variables as symbolic expressions [66]. SE approaches proposed in the literature have been successful at finding subtle bugs in several NASA's projects [67], at testing newly-modified source code [68, 69], at automated debugging [70], and in many other areas [71, 72].

To better understand how SE works, consider the following snippet of code:

```
1 public void foo(int x) {
2   int y = x * 3;
3   if (y == 42) {
4     System.out.println("GOOD");
5   } else {
6     System.out.println("BAD");
7   }
8 }
```

In an execution with *concrete* inputs, `foo` would be called with a concrete value (e.g., 7). Then, `y` would get the result of multiplying 7 by 3, i.e., 21. As 21 is not equal to 42, the condition on line 3 would

be evaluated as `false`, and therefore the execution would print the word "BAD". In a symbolic execution, `foo` would be called with a symbolic value (e.g., β). The execution then proceed with the multiplication and assigns $\beta \times 3$ to `y`. Therefore, the condition to be evaluated on line 3 is no longer `if (y == 42)` but `if (β × 3 == 42)`. At this point in the execution, β could take any value. To solve the constraint $\beta \times 3 == 42$, i.e., to generate two values such that each one could satisfy each outcome of the expression (i.e., `true` and `false`), constraint solvers such as Z3 [73] are usually used. For this particular example, the value 14 would make the condition to be evaluated as `true`, and any other value would make the condition to be evaluated as `false`. Therefore, SE has explored all feasible paths of this toy example. However, the number of paths in a program can grow exponentially with respect to the size of the program — a problem known as path explosion — or with the presence of loops (where the number of possible iterations could make the number of paths infinite). Therefore, applying traditional SE approaches to real and large software programs can become impractical [71, 72]. Nevertheless, several approaches have been proposed to address this issue and we explore them in the following.

Boonstoppel et al. [74] proposed an approach called Read-Write set (RWset) which discards paths that will produce the same result (i.e., paths that cover the same basic blocks) as any previously explored path. RWset tracks all reads and writes of all variables in order to be able to detect that the suffix of a path, i.e., the remainder steps of a path can be determined and are equivalent to a suffix of a previously explored path. In such case, the execution is considered redundant and the path is pruned.

Majumdar et al. [75] argued that generating test cases for one independent variable at a time could eliminate the test combination of every single input, and therefore, reduce the number of paths needed to explore the program under test. To achieve this, their approach computes the control and data dependencies between variables. Paths with the same trace are considered redundant and pruned.

Santelices et al. [76] presented an approach named Symbolic Program Decomposition (SPD), in which symbolic execution is just performed in a group of paths (known as path families), i.e., paths that have the same control and data dependencies. Similarly, Qi et al. [77] proposed an approach to group program paths based on the program output. In their approach, paths are considered equivalent if the output is affected when: 1) statements of control dependencies are executed, and 2) statements of potential dependencies are not executed. The main difference between Santelices et al. [76] and Qi et al. [77] approach is the precision of each one. The approach proposed by Santelices et al. [76] uses a static analysis which over-approximates (i.e., precision is sacrificed) the exploration of path families. On the

other hand, the approach proposed by Qi et al. [77] uses a dynamic analysis which under-approximate the exploration of relevant slices.

Ma et al. [78] proposed an approach to automatically find a program execution (i.e., path) that is able to reach a particular target goal (e.g., statement). They proposed two strategies: Shortest-Distance Symbolic Execution (SDSE), and Call-Chain-Backward Symbolic Execution (CCBSE). SDSE can be described as a top-bottom strategy. It executes the program symbolically, and uses a distance metric to guide the symbolic exploration through the control-graph to a particular target. This means that during the symbolic execution, the path with the shortest distance to the target goal is always selected. On the other hand, CCBSE can be described as a bottom-top strategy. It starts at the target goal and goes backwards until it finds a feasible path from the start of the program.

2.3.1 *Dynamic Symbolic Execution*

Approaches based on symbolic execution heavily rely on the precision of the underlying constraint solver to generate concrete inputs. However, symbolic values related to native code, third-party libraries, or just too complex symbolic constraints may not be handled by the underlying constraint solver, which can lead to an incomplete execution of a path as the effect of that code would be completely ignored [79]. To address this issue, Dynamic Symbolic Execution (DSE) [35] (also referred as concolic execution [80]) has been proposed. In DSE, the program under test is simultaneously executed with concrete and symbolic values, and when symbolic values can not be handled by any underlying constraint solver (e.g., Z3 [73]), they are replaced with concrete values.

A popular variant to apply DSE is to manually write a parameterised unit test as an entry function, and then to explore the paths through a program by deriving values for the parameters of the parameterised test [81]. Or use randomly generated test inputs to explore as many paths as possible, and then apply DSE to cover the paths that were not covered by any randomly generated test input [82].

Godefroid et al. [37] presented an approach which applies SE on *fuzz testing* context. Their approach executes the program with an initial input and it creates the initial path constraint. Then, instead of just expanding one path constraint using depth-first search (to expand the first constraint) or breadth-first search (to expand the last constraint) as usually done by SE techniques, their approach attempts to expand all constraints at once. Hence, maximising the number of inputs generated in each symbolic execution.

Babić et al. [83] proposed an approach to automatically prioritise in which order paths should be explored. Their approach can be de-

scribed in three main steps. First, a static and dynamic analysis is performed by executing a set of existing tests to identify indirect jumps in binary files. Second, it creates a control-flow graph of the program under test and identifies possible vulnerabilities based on loop pattern heuristics and out-of-bound accesses. Third, it executes the program under test with symbolic values and generates concrete values that are able to trigger the vulnerabilities identified in the previously steps. The effectiveness of their approach highly depends on the initial set of tests.

Anand et al. [79] proposed a technique called type-dependence analysis which performs static analysis to identify areas of the program under test that could not be executed symbolically. A report of those problematic parts (accomplished with some context information) is then provided to the developer, so that he or she can improve the program under test by performing the suggested changes. Similarly, Anand et al. [84] proposed an approach called *heap cloning* which identifies the areas of the code that introduce imprecision in a symbolic execution, e.g., when executing native code. Their approach creates two heaps ("concrete heap" and "symbolic heap") for the same program, each with a copy of all program's objects. During the execution of the program, objects in the "concrete heap" are updated when native code is execute, objects in "symbolic heap" are updated when code of the program under test is executed. Thus, the side-effects of native code that introduced imprecision could be automatically identified by simply comparing both heaps.

To reduce the number of paths explored by a DSE techniques due to loops in the program under test, Godefroid et al. [85] proposed an approach to summarise a loop body during a symbolic execution. For instance, to cover statement `abort1` in Listing 2.3 (line 9), `x` has to be greater than zero and `c` equal to 50. To cover statement `abort2` (on line 16), `x` should be zero, and `c` equal to x_i (where x_i is the iterated value of `x`). Then, and based on this information, Godefroid et al. [85]'s approach summarises the variables that are modified within loop by a constant value, by creating the precondition (`x > 0`) and the postcondition $((x = 0) \wedge (c = x_i))$. Assuming the underlying constraint solver can handle these pre/postconditions, the number of paths to test would be reduced.

Listing 2.3: Motivational symbolic execution example adapted from Godefroid et al. [85].

```
1 public void foo(int x) {
2  int c = 0;
3  int p = 0;
4  while (true) {
5   if (x <= 0) {
6    break;
7   }
8   if (c == 50) {
```

```
9       abort1(); /* FAULT 1 */
10    }
11    c++;
12    p = p + c;
13    x--;
14  }
15  if (c == 30) {
16    abort2(); /* FAULT 2 */
17  }
18 }
```

Saxena et al. [86] also proposed an approach to reduce the overhead of loops on DSE techniques. In their approach, for each program loop an extra symbolic variable is used to count the number of times a loop is executed. Then, static analysis is performed to analyse the relations between that symbolic variable and the values of the variables of the program under test. The main difference between both works is that, the one proposed by Godefroid et al. [85] analyses loop's structure *on the fly* without using any other tools, as opposed to the approach proposed by Saxena et al. [86] which detects the loop's structure using static analysis.

Although, overall, SE/DSE approaches can effectively generate high-coverage tests, they may not scale to complex programs (i.e., programs with a large number of paths) or object-oriented programs (for which a sequence of statements is require to invoke and interact with the program, rather than just optimising input values to cover specific paths). In the following section we survey search-based software testing approaches which have been successfully applied to the test generation problem [87, 88].

2.4 SEARCH-BASED SOFTWARE TESTING

Although the term Search-Based Software Engineering (SBSE) was first introduced by Harman and Jones in 2001 [89] as the application of meta-heuristic search algorithms to address software engineering problems [90–93], the first application of optimisation techniques is commonly attributed to the work proposed by Miller et al. [94] in 1976. Miller et al. [94] used numerical optimisation techniques to generate floating point *test data* to cover paths of a software program. Since then, the application of meta-heuristic search algorithms to software testing — known as Search-Based Software Testing (SBST) [95] — has become the most successful and popular area of SBSE [88]. In SBST, test cases (or only test inputs) represent the search space of a meta-heuristic search algorithm and they are typically optimised for structural criteria [36, 87, 96–100]. However, other criteria such as functional and non-functional requirements [101, 102], mutation [20, 103], and exceptions [104–106] have been also explored [88].

Earlier works on structural testing aimed to automatically generate *test data* (i.e., numeric inputs) for procedural code [87, 88]. For instance, in the work proposed by Korel [97] the software under test is instrumented and first executed with random inputs. If, by chance, the chosen path (previous selected by a software tester) is fully covered, the test inputs are saved. Otherwise, branch distance is computed at the point where the execution diverges from the desired path, and a local search algorithm is applied to find alternative inputs that could satisfy that particular branch while at the same time preserving the coverage until that point. Xanthakis et al. [96] proposed a similar approach. In their work the selected path is first explored by a random search approach and all branch predicates are extracted. Then, a search algorithm is applied in order to satisfy *all* branch predicates at the same time.

To alleviate the effort of manually selecting program paths (which could be very time consuming, in particular for complex programs), Korel [107] proposed a goal-oriented technique. Instead of selecting a program path, the developer only has to select a target goal, for example, a statement. Then, their technique uses the program's control flow to filter out non-relevant branches (the ones that the search does not need to satisfy to reach the target goal). Finally, they apply a search algorithm to generate inputs that satisfy all relevant branches. Pargas et al. [99] proposed a control oriented technique which aims to cover specific structural points (e.g., a statement) by maximising the number of executed control dependent nodes (i.e., the nodes a structural point depends on). A test input that executes more control dependent nodes of a structural point, should be closer to reach it. However, no guidance is provided to the search on how close a test input is to cover a node.

Later works on structural testing have aimed to automatically generate *test cases* (i.e., sequences of method calls) for Object Oriented (OO) software [87]. Note that, due to the nature of OO paradigm (e.g., inheritance and polymorphism, object parameters may have to be in a particular state in order to be able to cover the target goal, etc), testing OO software could be more complex than testing procedural programs [108, 109]. In such scenario, a test case is no longer a simple set of values but rather a much more complex sequence of method calls and their respective parameters.

The first attempt to test OO software with meta-heuristic search algorithms was proposed by Tonella [36]. Their approach uses a genetic algorithm to generate targeted test cases for each individual target goal (e.g., branch). Custom evolutionary operators (i.e., crossover and mutation) are applied to evolve individuals. At the end, all generated test cases are combined into a single test suite. Although effective, their approach may not scale for software with a large number of target goals, as the number of test cases may grow with the increase

on the number of target goals. As opposed to their work, Wappler et al. [110] used standard evolutionary operators to evolve individuals, which could generate infeasible ones and therefore negatively influence the fitness function. To address that issue, a follow up work by Wappler et al. [111] proposed the use of genetic programming to enforce the generation of feasible individuals.

In contrast to the work proposed by Tonella [36], Fraser et al. [112, 113] proposed an approach named Whole Test Suite (WTS), which evolves test suites (i.e., the individuals of a population are sets of test cases, and each test case is a sequence of calls) targeting *all* testing goals at the same time. Thus, removing the need to select an order in which to target individual coverage goals. WTS has been shown to be more effective than iteratively generating individual test cases [113].

Although a common approach in SBST is to use genetic algorithms, numerous other algorithms (including random-search) have been proposed in the domain of nature-inspired algorithms, as no algorithm can be best on all domains [114]. In the following sections, we review several meta-heuristic search algorithms and enhancements proposed in the literature. Moreover, in Chapter 4 we perform an empirical comparison of the most adequate algorithms for the unit test generation problem.

2.4.1 *Representation*

Evolutionary Algorithms (EAs) are inspired by natural evolution, and have been successfully used to address many kinds of optimisation problems [92, 93]. In the context of EAs, a solution is encoded "genetically" as an individual ("chromosome"), and a set of individuals is called a population. For test suite generation, the individuals of a population are sets of test cases (test suites); each test case is a sequence of calls. The population is gradually optimised using genetic-inspired operations such as crossover, which merges genetic material from at least two individuals to yield new offspring, and mutation, which independently changes the elements of an individual with a low probability. Crossover on test suites is based on exchanging test cases [113]; mutation adds/modifies tests to suites, and adds/removes/changes statements within tests. While standard selection techniques are largely used, the variable size representation (number of statements in a test and number of test cases in a suite can vary) requires modification to avoid bloat [115]; this is typically achieved by ranking individuals with identical fitness based on their length, and then using rank selection.

2.4.2 *Random Search*

Random search is a baseline search strategy which does not use crossover, mutation, or selection, but a simple replacement strategy [116]. Random search consists of repeatedly sampling candidates from the search space; the previous candidate is replaced if the fitness of the new sampled individual is better. Random search can make use of a test archive [117] (which store tests for covered goals) by changing the sampling procedure, i.e., new tests may be created by mutating tests in the archive rather than randomly generating completely new tests. Random testing is a variant of random search in test generation which builds test suites incrementally. Test cases (rather than test suites) are sampled individually, and if a test improves coverage, it is retained in the test suite, otherwise it is discarded. It has been shown that in test generation, due to the flat fitness landscapes and often simple search problems, random search is often as effective as EAs, and sometimes even better [118].

2.4.3 *Local Search Algorithms*

2.4.3.1 *Hill Climbing*

Hill Climbing [119] is a local search algorithm which evaluates solutions according to a fitness function. It starts with a random solution and in an, e.g., 1-dimensional problem, evaluates two neighbours (one to the right and one to the left). The solution with the best score value, i.e., fitness value, replaces the current one. However, due to lack of search power, the Hill Climbing algorithm does not make any assumptions about the landscape (a plot of the fitness) of the problem as described in Figure 2.4. Therefore, it only performs movements in the landscape if the next individual is better than the current, which could lead to be trapped in a local optimum solution.

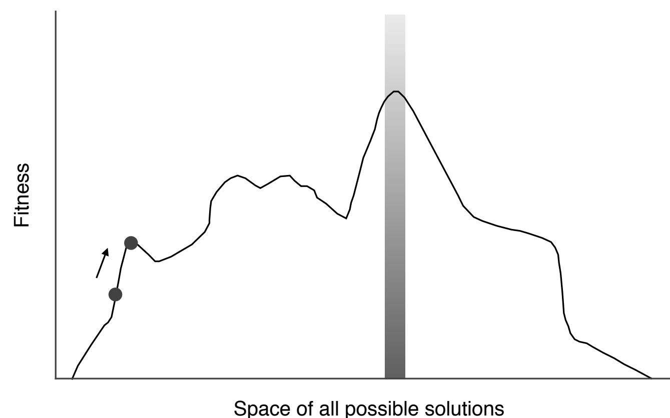


Figure 2.4: Hill Climbing landscape example taken from McMinn [95].

2.4.3.2 *Tabu-search*

Tabu-search [120, 121] is a local search algorithm which incorporates adaptive memory and responsive exploration. First, it generates a random solution and evaluates it. Then, it generates several other solutions and evaluated them. If any of the additional solutions is better than the current solution, the current one is replaced. Otherwise, new sets of solutions are generated until a new solution that is better than the current one is found. However, and like Hill Climbing, Tabu-search could also stop on a local optimum solution. For instance, if two solutions in the search are always considered as the best ones, then the search may spend all the time bouncing between the two indefinitely. To avoid that, Tabu-search keeps a list of all previous solutions and restrain their re-selection. Thus, increasing the likelihood of finding an optimum solution.

2.4.3.3 *Simulated Annealing*

Simulated Annealing [104, 122] is a meta-heuristic algorithm similar to Hill Climbing, however, movements through the search space are not so restricted. To explore a large portion of the search-space, it uses a control parameter called *temperature* as the probability of accepting worse solutions, i.e., solutions with a lower fitness value. It starts with a high *temperature* value, but as the search evolves, the temperature decreases until it reaches zero, in which the search would work similar to the Hill Climbing algorithm. As Hill Climbing and Tabu-search

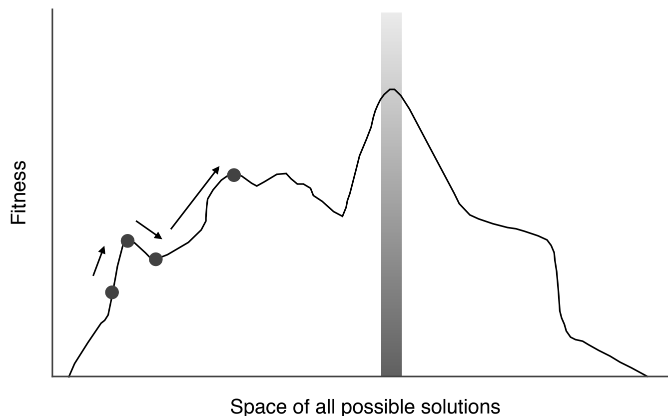


Figure 2.5: Simulated Annealing landscape example taken from McMinn [95].

algorithms, Simulated Annealing only considers one solution at time and it does not make any assumption about the landscape. If the *temperature* cools down to quickly, it might get stuck in a local optimum as the Hill Climbing algorithm.

2.4.4 *Global Search Algorithms*

2.4.4.1 *Genetic Algorithm (GA)*

The Genetic Algorithm (GA) [123] is one of the most widely-used EAs in many domains because it can be easily implemented and obtains good results on average. Algorithm 1 illustrates a Standard GA. It starts by creating an initial random population of size p_s (Line 1). Then, a pair of individuals is selected from the population using a strategy s_f, such as rank-based, elitism or tournament selection (Line 6). Next, both selected individuals are recombined using crossover c_f (e.g., single point, multiple-point) with a probability of c_p to produce two new offspring o_1, o_2 (Line 7). Afterwards, mutation is applied on both offspring (Lines 8–9), independently changing the genes with a probability of m_p, which usually is equal to $\frac{1}{n}$, where n is the number of genes in a chromosome. The two mutated offspring are then included in the next population (Line 10). At the end of each iteration the fitness value of all individuals is computed (Line 13).

Algorithm 1 Standard Genetic Algorithm

Input: Stopping condition C, Fitness function δ, Population size p_s, Selection function s_f, Crossover function c_f, Crossover probability c_p, Mutation function m_f, Mutation probability m_p
Output: Population of optimised individuals P

```
1:  P ← GenerateRandomPopulation(p_s)
2:  PerformFitnessEvaluation(δ, P)
3:  while ¬C do
4:      N_P ← {}
5:      while |N_P| < p_s do
6:          p_1, p_2 ← Selection(s_f, P)
7:          o_1, o_2 ← Crossover(c_f, c_p, p_1, p_2)
8:          Mutation(m_f, m_p, o_1)
9:          Mutation(m_f, m_p, o_2)
10:         N_P ← N_P ∪ {o_1, o_2}
11:     end while
12:     P ← N_P
13:     PerformFitnessEvaluation(δ, P)
14: end while
15: return P
```

2.4.4.2 *Monotonic GA*

Many variants of the Standard GA have been proposed to improve effectiveness. For example, a *monotonic* version of the Standard GA which, after mutating and evaluating each offspring, only includes either the best offspring or the best parent in the next population

(whereas the Standard GA includes both offspring in the next population regardless of their fitness value).

2.4.4.3 *Steady State GA*

Another variation of the Standard GA is a *Steady State* GA, which uses the same replacement strategy as the Monotonic GA, but instead of creating a new population of offspring, the offspring replace the parents from the current population immediately after the mutation phase.

2.4.4.4 *1 + (λ, λ) GA*

The $1 + (\lambda, \lambda)$ GA, introduced by Doerr et al. [124], starts by generating a random population of size 1. Then, mutation is used to create λ different mutated versions of the current individual. Mutation is applied with a high mutation probability, defined as $m_p = \frac{k}{n}$, where k is typically greater than one, which allows, on average, more than one gene to be mutated per chromosome. Then, uniform crossover is applied to the parent and best generated mutant to create λ offspring. While a high mutation probability is intended to support faster exploration of the search space, a uniform crossover between the best individual among the λ mutants and the parent was suggested to repair the defects caused by the aggressive mutation. Then all offspring are evaluated and the best one is selected. If the best offspring is better than the parent, the population of size one is replaced by the best offspring. $1 + (\lambda, \lambda)$ GA could be very expensive for large values of λ, as fitness has to be evaluated after mutation and after crossover.

2.4.4.5 $\mu + \lambda$ *Evolutionary Algorithm (EA)*

The $\mu + \lambda$ Evolutionary Algorithm (EA) is a mutation-based algorithm [125]. As its name suggests, the number of parents and offspring are restricted to μ and λ, respectively. Each gene is mutated independently with probability $\frac{1}{n}$. After mutation, the generated offspring are compared with each parent, aiming to preserve the so-far best individual including parents; that is, parents are replaced once a better offspring is found. Among the different (μ+λ) EA versions, two common settings are (1+λ) EA and (1+1) EA, where the population size is 1, and the number of offspring is also limited to 1 for the (1+1) EA.

2.4.4.6 *Many-Objective Sorting Algorithm (MOSA)*

Unlike the *whole test suite* generation proposed by Fraser et al. [113], the Many-Objective Sorting Algorithm (MOSA) [126] regards each coverage goal as an independent optimisation objective. MOSA is a variant of NSGA-II [127], and uses a preference sorting criterion to

reward the best tests for each non-covered target, regardless of their dominance relation with other tests in the population. MOSA also uses an archive to store the tests that cover new targets, which aiming to keep record on current best cases after each iteration.

Algorithm 2 illustrates how MOSA works. It starts with a random population of test cases. Then, and similar to typical EAs, the offspring are created by applying crossover and mutation (Line 6). Selection is based on the combined set of parents and offspring. This set is sorted (Line 9) based on a non-dominance relation and preference criterion. MOSA selects non-dominated individuals based on the resulting rank, starting from the lowest rank (F_0), until the population size is reached (Lines 11-14). If fewer than p_s individuals are selected, the individuals of the current rank (F_r) are sorted by crowding distance (Line 16-17), and the individuals with the largest distance are added. Finally, the archive that stores previously uncovered targets is updated in order to yield the final test suite (Line 18). In order to cope with the large numbers of goals resulting from the combination of multiple coverage criteria, the DynaMOSA [128] extension dynamically selects targets based on the dependencies between the uncovered targets and the newly covered targets. Both, MOSA and DynaMOSA, have been shown to result in higher coverage of some selected criteria than traditional GAs for WTS optimisation [126, 128].

2.4.5 *Fitness Functions*

In search-based test generation, the selection of individuals is guided by fitness functions (which measure how good a test case or test suite is with respect to the search optimisation objective), such that individuals with good fitness values are more likely to survive and be involved in reproduction. Fitness functions are usually based on metrics [129] such as structural coverage [87, 97, 100], functional and non-functional requirements [101, 102], or mutation [20, 103]; and provide additional search guidance leading to satisfaction of the goals. For example, just checking in the fitness function whether a coverage target is achieved would not give any guidance to help covering it.

Although structural coverage criteria are well established in order to evaluate existing test cases [18] (as we previously described in Section 2.1.2.1), they may be less suitable in order to guide test generation. As with any optimisation problem, an imprecise formulation of the optimisation goal could lead to unexpected results: for example, although it is generally desirable that a reasonable test suite covers all statements of a software under test, the reverse may not hold – not every test suite that executes all statements is reasonable.

In the following sections we describe the most simple structural coverage fitness function (i.e., line coverage) and the most common

Algorithm 2 Many-Objective Sorting Algorithm (MOSA)

Input: Stopping condition C, Fitness function δ, Population size p_s, Crossover function c_f, Crossover probability c_p, Mutation probability m_p
Output: Archive of optimised individuals A

1: $p \leftarrow 0$
2: $N_p \leftarrow$ GENERATERANDOMPOPULATION(p_s)
3: PERFORMFITNESSEVALUATION(δ, N_p)
4: $A \leftarrow \{\}$
5: **while** $\neg C$ **do**
6: $N_o \leftarrow$ GENERATEOFFSPRING(c_f, c_p, m_p, N_p)
7: $R_t \leftarrow N_p \cup N_o$
8: $r \leftarrow 0$
9: $F_r \leftarrow$ PREFERENCESORTING(R_t)
10: $N_{p+1} \leftarrow \{\}$
11: **while** $|N_{p+1}| + |F_r| \leqslant p_s$ **do**
12: CALCULATECROWDINGDISTANCE(F_r)
13: $N_{p+1} \leftarrow N_{p+1} \cup F_r$
14: $r \leftarrow r + 1$
15: **end while**
16: DISTANCECROWDINGSORT(F_r)
17: $N_{p+1} \leftarrow N_{p+1} \cup F_r$ with size $p_s - |N_{p+1}|$
18: UPDATEARCHIVE(A, N_{p+1})
19: $p \leftarrow p + 1$
20: **end while**
21: **return** A

structural coverage fitness function, *branch coverage* [87, 97, 100]. Note that, although structural coverage fitness functions are the most common used ones, there has been little innovation in the fitness functions for structural/path coverage over the past 25 years. In the following sections we also describe a fitness function which has been reported as effective at finding *faults* [130, 131], *weak mutation* [20]. Furthermore, in Chapter 3 we define a few more coverage-based fitness functions and propose a simple approach to combine all of them; and in Chapter 5 we present a non-functional functional criterion which can guide a search algorithm to produce test case that are effective at diagnosing a faulty software.

2.4.5.1 *Line Coverage*

A basic criterion in procedural code is statement coverage, which requires all statements to be executed. Modern test generation tools for Java (e.g., EvoSuite [9]) or C# (e.g., Pex [81]) often use the bytecode representation for test generation, and bytecode instructions may not directly map to source code statements. Therefore, a more common

alternative in coverage analysis tools, and the de-facto standard for most Java bytecode-based coverage tools, is to consider coverage of *lines* of code. Each statement in a software has a defined line, which represents the statement's location in the source code of the software. The source code of a software consists of non-comment lines, and lines that contain no code (e.g., whitespace or comments). A test suite satisfies the Line Coverage criterion only if it covers each non-comment source code line of the software under test with at least one of its tests. Line Coverage is very easy to visualise, interpret, and to implement in an analysis tool; all these reasons probably contribute to its popularity.

To cover each line of source code, each basic code block must to be reached. In traditional search-based testing, this reachability would be expressed by a combination of approach-level [100] and branch distance [87, 97] as illustrated in Figure 2.6. The approach-level [100] measures how far an individual execution and the target statement are in terms of the control dependencies (i.e., distance between point of diversion and target statement in control dependence graph). The branch distance estimates how far a predicate is from evaluating to a desired target outcome. For example, given the first predicate `a >= b` and an execution with values `a=5` and `b=3`, the branch distance to the predicate evaluating to true would be $|3-5| = 2$, whereas an execution with values `a=5` and `b=4` is closer to being true with a branch distance of $|4-5| = 1$. Branch distances can be calculated by applying a set of standard rules [87, 97].

In contrast to test case generation, the optimisation of test suites to execute all statement does not require the approach level, as all statements will be executed by the same test suite. Thus, it only needs to consider the branch distance of all branches that are control dependencies of any of the statements in the software under test. That is, for each conditional statement that is a control dependency for some other statement in the code, it is required that the branch of the statement leading to the dependent code is executed. Thus, the Line Coverage fitness value of a test suite can be calculated by executing all its tests, calculating for each executed statement the minimum branch distances $d_{min}(b, \mathit{Suite})$ among all observed executions to every branch b in the set of control dependent branches B_{CD}, i.e., the distances to all the branches which need to be executed in order to reach such a statement. The Line Coverage fitness function is thus defined as:

$$f_{LC}(\mathit{Suite}) = \nu(|\ \mathit{NCLs}\ | - |\ \mathit{CoveredLines}\ |) + \sum_{b \in B_{CD}} \nu(d_{min}(b, \mathit{Suite}))$$

where *NCLs* is the set of all non-comment lines of code in the software under test, *CoveredLines* is the total set of lines covered by the execution traces of every test in the suite, and $\nu(x)$ is a normalising function in $[0, 1]$ (e.g., $\nu(x) = x/(x+1)$) [132].

```
public void foo(int a, int b, int c, int d) {
  if (a >= b) {
    if (b >= c) {
      if (c >= d) {
        // target
```

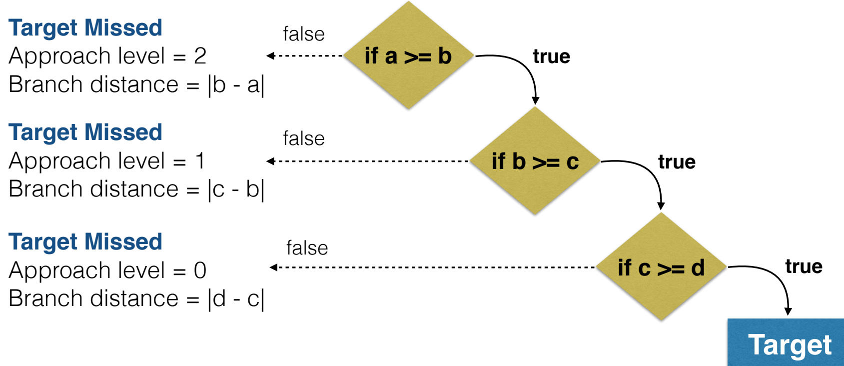


Figure 2.6: Example of how approach level and branch distances are calculated[1].

2.4.5.2 *Branch Coverage*

The concept of covering branches is also well understood in practice and implemented in popular tools, even though the practical definition of branch coverage may not always match the more theoretical definition of covering all edges of a program's control flow. Branch coverage is often interpreted as maximising the number of branches of conditional statements that are covered by a test suite. Hence, a test suite is said to satisfy the Branch Coverage criterion if and only if for every branch statement in the software under test, it contains at least one test whose execution evaluates the branch predicate to *true*, and at least one test whose execution evaluates the branch predicate to *false*.

The fitness function for the Branch Coverage criterion estimates how close a test suite is to covering all branches of the software. The fitness value of a test suite is measured by executing all its tests, keeping track of the branch distances $d(b, Suite)$ for each branch in the software under test. Then:

$$f_{BC}(Suite) = \sum_{b \in B} v(d(b, Suite))$$

1 Example based on the one presented by Phil McMinn at TAROT 2010, and by Gordon Fraser at TAROT 2014.

Here, $d(b, \mathit{Suite})$ for branch $b \in B$ (where B is the set of all branches in the software) on the test suite is defined as follows:

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

Note that a predicate must be executed at least twice, because the true and false evaluations of the predicate need to be cover; if the predicate were only executed once, then the search could theoretically oscillate between true and false.

Flag Problem

A very well known problem of search-based approaches on software testing is the "flag" problem. This problem occurs when a flag (boolean variable) is involved in branch predicates as described in Listing 2.4. In this case, the landscape of the search-space consists of two plateaus (one for each branch outcome). A plateau region occurs when the search is so flat, that the fitness value of a particular point is indistinguishable from the value returned by its neighbours. For example, in Listing 2.4 the fitness function value is zero if `d` is equal to zero, otherwise is always one (no matter what is the value of `d`). Harman et al. [133] proposed the use of program transformation to remove flag variables from branch predicates. In their work, flags are replaced from the branch predicates with the expression that could led to their determination, thus removing plateau regions created by flag variables.

Listing 2.4: Motivational flag example taken from McMinn [95].

```
boolean flag = (d == 0);
if (flag) {
  result = 0;
} else {
  result = n / d;
}
```

2.4.5.3 *Weak Mutation*

Test generation tools typically include values generated to satisfy constraints or conditions, rather than values developers may prefer; in particular, anecdotal evidence suggests developers like boundary cases [134]. Test generation can be forced to produce such values using weak mutation testing, which applies small code modifications to the software under test, and then checks if there exists a test that can

distinguish between the original and the *mutant*. In weak mutation, a mutant is considered to be covered (i.e., "killed") if the execution of a test on the mutant leads to a different state than the execution on the software under test, i.e., if it *infects* the state. A test suite hence satisfies the Weak Mutation criterion if and only if for each mutant for the software under test at least one of its tests reaches state infection [135].

The fitness function for the Weak Mutation criterion guides the search [136] using infection distances with respect to a set of mutation operators [103]. For instance, "replace variable" operator which replaces the value of any variable with the value of any other variable of the same type and in the same scope. For this mutation operator the infection distance is 0 if old and new values differ, 1 otherwise. Assuming a minimal infection distance function $d_{min}(\mu, Suite)$ exists (e.g., one of the infection distances proposed by Fraser et al. [103]):

$$d_w(\mu, Suite) = \begin{cases} 1 & \text{if mutant } \mu \text{ was not reached,} \\ \nu(d_{min}(\mu, Suite)) & \text{if mutant } \mu \text{ was reached.} \end{cases}$$

This results in the following fitness function for weak mutation testing:

$$f_{WM}(Suite) = \sum_{\mu \in \mathcal{M}_C} d_w(\mu, Suite)$$

where $\mathcal{M}_C$ is the set of all mutants generated for the software under test.

2.4.5.4 *Non-Functional Coverage Criteria*

Despite the fact that *branch coverage* is the most common structural criterion [93, 137], many other have been proposed [138–140]. For instance, to increase the likelihood of human developers integrating automatically generated tests in their software projects, Daka et al. [141, 142] proposed the use of a human-based model to automatically generate *readable* test cases. Afshan et al. [143] used natural language processing to improve inputs used in test cases. Their results shown that users are significantly quicker at understanding the natural language-based inputs, than inputs generated by a coverage approach. However, the question on how to integrate any of these two non-functional criteria with a functional criterion such as coverage still remains.

The integration of functional and non-functional properties in test generation is usually described as a multiple-objective problem. For example, Ferrer et al. [144] proposed the optimisation of coverage and the oracle cost using multiple-objective algorithms such as NSGA-II [127] and SPEA2 [145]. Harman et al. [146] proposed

the optimisation of branch coverage and memory consumption also using multiple-objective algorithms. However, it has been reported that the integration of a functional criterion such as coverage and non-functional criteria has a negative impact on the final coverage achieved [144, 146], and on the usefulness of automatically generated test cases due to implicit trade-offs between functional and non-functional criteria [147]. I.e., the most effective test case in terms of memory may not be the one with the highest coverage, and vice-versa.

Unlike these multiple-objective approaches, Fraser et al. [115] proposed the use of non-functional criteria (e.g., length of a test suite) as a secondary objective of the search. I.e., test suites are still optimised for coverage, but when selecting which test suites should form the new offspring, their approach prefers the shortest test suites to those with the same coverage level but are longer. As shorter test suites would require less memory and execution time, their approach improves the performance of the search. Instead of *length* as a secondary objective, Palomba et al. [147] proposed the optimisation of *cohesion* (which measures the textual similarity within a test case) and *coupling* (which measures the textual similarity between all tests within a test suite). Besides improving *cohesion* and *coupling*, their approach also increases the coverage achieved and produces shorter tests.

2.4.6 *Seeding*

One of the many parameters [148] that could influence the efficiency of evolutionary algorithms is the *initial population* (as we empirically evaluate in Chapter 4). Miraz et al. [149] proposed to initialise the initial population with the best individuals of a randomly generated population. A study conducted by Fraser et al. [150] concluded that in the earlier steps of the search, seeding strategies — which exploits previous related knowledge, e.g., the reuse of previously solutions to seed the initial population, can lead to an overall improvement of the final solutions. Rojas et al. [151] explored the seeding of 1) constants extracted from the source code, and 2) values identified at runtime during the execution of test cases. They found that seeding can significantly improve the performance of the search.

2.4.7 *Enhancing Search-based Software Testing with Symbolic Execution*

Generating test inputs for software programs with loop structures is not just a problem of SE approaches (as described in Section 2.3), search-based approaches are also affected by the same problem. In order to address the loop problem on search-based approaches, Baresel et al. [139] included the dependencies of a single loop iteration on

the evaluation of the fitness function. Tracey et al. [106] computed the branch distance of each loop iteration and used the minimum branch distance to compute the final fitness value.

Considering the main issues of DSE approaches (see Section 2.3.1), e.g., no support for generating sequences of method calls to initialise non primitive arguments [152], and the main issues of search-based approaches, e.g., no sufficient guidance [118]; several works have proposed the integration of SBST and DSE. Inkumsah et al. [153] proposed an approach that combines SBST and DSE to overcome the general weaknesses of both strategies and maximise code coverage. On their approach, the evolutionary testing tool eToc [36] is used to generate sequences of method calls and the DSE tool jCute [152] to generate methods arguments.

Lakhotia et al. [154] presented a combination of DSE and SBST using an Alternative Variable Method (AVM) [97], to overcome the imprecision of constraint solvers to compute floating point numbers used on DSE. AVM is an optimisation algorithm (like Hill Climbing) where values (solutions) are increased/decreased one-by-one by a delta value. If the search gets stuck, the algorithm restarts from a random input. Recently, Galeotti et al. [155] proposed a hybrid approach that combines the best of SBST and DSE, which depending on the search properties, DSE is adaptively used to satisfy coverage goals that are difficult for SBST.

2.5 REGRESSION TESTING

"Yesterday, My Program Worked.
Today, It Does Not. Why?"

— Zeller, 1999 [156]

Usually, a software program is not developed on a single iteration. Instead, it is developed over time and each version that is released is the sum of all previous iterations. In each iteration, developers add new functionalities to the software, remove deprecated or obsolete functionalities, and address bugs that have been reported. However, despite the fact that these changes aim to enhance the software, they may also introduce unintended side-effects. If an existing test suite is available — also know as regression test suite — it can be executed to ensure that all existing functionalities (i.e., the ones before changes are made) have not been affected by any new change. If any test case passes before the changes are made but fails after, it may indicate that a regression bug has been introduced or just that the failing test has become obsolete (e.g., the tested functionality has been modified or removed) and it has to be repaired to match the new requirements. This testing process is known as *regression testing* and it has been widely studied [157] and adopted by software engineers [158].

As the software evolves over time, the number of regression test cases tend to grow (as new test cases need to be created to ensure new functionalities would still work as expected after future changes). Therefore, executing all test cases every time a change is made on the software can be very expensive. In order to reduce this cost, three approaches have been proposed in the literature [157]: test case minimisation, selection, and prioritisation.

2.5.1 *Test Case Minimisation, Selection, and Prioritisation*

Test case minimisation [159] aims to identify and remove redundant test cases. Most of the approaches proposed on test minimisation are based on structural coverage [157]. For instance, Chvatal [160] proposed the use of a greedy algorithm to identify the minimum set of test cases that are required to cover most of the program. Although the algorithm is efficient, the minimum set of tests may still include some redundant ones. Suppose a program with three components (e.g., statements, c_1, c_2, and c_3) and a test suite with four test cases (t_1, t_2, t_3, and t_4). t_1 covers all components but c_3, t_2 covers c_1, t_3 covers c_2, and t_4 covers c_3. As t_1 is the test case that covers most of the program, it is selected first. Then, t_2, t_3, and t_4 are selected in order since they all cover the same number of components. Although t_1 is subsumed by the coverage of t_2, t_3, and t_4, it is not considered redundant. Harrold et al. [159] proposed an approach named Harrold-Gupta-Soffa (HGS) which starts by selecting components that are covered by k test cases (k starts with one). Then, from all those k test cases, it selects the one that covers more components (in case of a tie, it chooses one at random). Finally, it excludes the components covered by the selected test, and repeats these steps until all components have been covered, or if there is not any other test case left to select. A potential risk shared by all minimisation techniques is the fact that a test case that actually reveals a failure could end up being discarded. An empirical study conducted by Rothermel et al. [161] has shown that after minimising the set of tests, its effectiveness is reduced by 50% and sometimes even 100%, i.e., the minimum set of tests would only able to detect half of the failures and sometimes would not even able to detect any failure.

Test case selection [162] is very similar to test minimisation but rather than removing redundant tests, it selects a subset of test cases that need to be re-executed for a particular change. That is, test selection takes into account the change that has been made to the software and selects a set of test cases that are relevant to test it [157], i.e., which may help in revealing a regression fault. Rothermel et al. [163] proposed an approach that creates two control flow graphs of a given program: one before a change has been introduced, and one after changing the program. Then, it compares both control flow graphs to

identify each parts have been changed. Finally, it selects the test cases that execute the changed code. Although their approach is considered *safe*, i.e., it includes all test cases that are able to reveal a fault, if the changed code does not introduce any difference to the original control flow graph, their approach would not be as effective as suggested. Other approaches that explore other strategies such as symbolic execution [164], integer programming [165], and textual differences [166] have also been proposed. However, their high computational cost may limit their adoption in practice. A recent study has shown that selection techniques run less test cases than minimisation techniques, and they are more effective at revealing any fault related to the changed code [167].

Test case prioritisation [168] aims to reduce the cost of regression testing by scheduling test cases in a specific order such that any existing fault is reveal as soon as possible. That is, instead of directly reducing the number or the set of test cases that are executed as test minimisation and selection, test prioritisation aims to reveal a regression as soon as possible. Rothermel et al. [21] empirically evaluated several prioritisation techniques in addition to two random baseline techniques. Their study reported that a technique named Fault Exposing Potential (FEP) is the most effective of all techniques evaluated. FEP is based on the ability of test cases at "killing" mutated versions of the program under test. As we previously described, in mutation testing different versions of the same program are created by introducing small changes (e.g., changing > to <). These changes (usually referred as mutants) are "alive" if undetected by any test case, or "killed" once a test case detects the mutant. FEP repeatedly selects the test case that kills the highest number of mutants that have not already been killed by any other test case in the prioritised suite. One of the downsides of this approach is that it is notoriously expensive. In order to calculate which mutants are killed by which tests, each mutant must be run against the entire test suite in isolation. However, as the number of tests and mutants grow, the cost of performing such mutation analysis increases dramatically. Li et al. [169] compared the performance of different meta-heuristics techniques at prioritising test cases. In their study they considered a hill climbing algorithm, a genetic algorithm, and a greedy algorithm (and two variations of it). The study found that although a greedy algorithm can be more effective at prioritising test cases, it is not significantly more effective than a genetic algorithm. Other test prioritisation approaches include, but are not limited to scheduling test cases based on their execution history [170], prioritisation based on the execution cost of each test case [171], and model-based prioritisation techniques [172].

2.5.2 *Test Suite Maintenance*

As a software program evolves over time, test cases do also evolve [173]. Due to changes to the software, some test cases may become obsolete (e.g., test cases that cover a functionality that no longer exists) and should be removed, others may start to fail (e.g., requirements of a certain functionality have changed) and should be repaired, and new test cases are created to, for example, validate a new functionality, or increase coverage [174]. Analysing test cases that become obsolete or start failing after a change was performed to the software can be a very time consuming task, especially for a large test suite [175]. For instance, a developer would have to determine whether each failing test case is revealing a regression that needs to be addressed or if it only needs to be repaired due to a recent change to the software.

Daniel et al. [176] proposed the first automated technique to repair failing test cases. Their technique (named ReAssert) starts by instrumenting the test code and executing the failing test cases to identify the location of the failures. Then, it applies one of the many strategies proposed to repair each failure, e.g., replacing of literal values. This process is repeated until one of the following three conditions is met: (i) all failures have been repaired, (ii) there is not any suitable strategy to repair a particular failure, or (iii) the maximum number of repairs has been reached. The suggested repairs are based on the current behaviour of the software. Although in their study ReAssert was able to repair 45% of all failures, its effects on test suite maintenance can be minimal. Pinto et al. [174] reported that less than 10% of all test modifications involve repairing assertions only. Mirzaaghaei et al. [177] proposed a technique to repair test cases that are no longer valid (i.e., tests that do not compile) due to changes in method signatures. Their technique analyses the compilation errors of each test case to identify the broken method calls, and collects initialisation values during the execution of the test case on the previous version of the program. Then, it attempts to replace the broken method calls with valid ones (using suitable values for each parameter). Although their technique fixes method calls involved in the addition, deletion, or modification of parameters, the technique however does not support the addition of method calls, which is a common type of change required to repair broken test cases [174].

2.5.3 *Test Suite Augmentation*

According to a study conducted by Pinto et al. [174], after the software is changed, a large number of test cases need to be created to exercise the changed code, as the existing ones may not be able to exercise it. This activity of adding new test cases is referred to as *test suite*

augmentation and several techniques have been explored to automate it. For instance, some test suite augmentation techniques aim to restore code coverage in test suites after the software is changed by producing new tests for new behaviour [68], while other approaches explicitly try to exercise changed code to reveal differences introduced by the changes [178, 179].

Orso et al. [178] proposed a technique named BERT which aims to identify behavioural differences between two versions of a software program through dynamic analysis. BERT starts by creating a large set of automatically generated test cases that exercise the modified code. Then, it executes each test case on the previous version of the software and on the current version to identify any difference in the behaviour of each test. Finally, it analyses the identified differences and presents them to the user. Although, in their study, BERT managed to find regression faults between two versions of a software program, its effectiveness is limited to the automatically generated test cases. The approach proposed by Santelices et al. [179] performs symbolic execution on two different versions of the software program (one before the software is changed, and one right after it is modified), to help developers at augmenting any existing test suite with new test cases. That is, rather than automatically generating tests (as the technique proposed by Orso et al. [178]), this approach only provides guidelines on how to augment a test suite.

2.5.4 *The Oracle Problem*

A common assumption when studying automated test generation techniques is that these techniques are applied in a *regression testing* scenario: Tests are generated automatically with assertions on the current state of the software, and they are executed repeatedly throughout software evolution to check if software modifications lead to undesired side-effects, which are revealed by failing tests. An alternative, but much less studied, scenario is the use of automated test generation to find bugs in the *current version* of the program. The challenge with this scenario is that it requires an explicit *test oracle*: Unless there is some sort of specification (e.g., code contracts), someone has to manually decide for every single generated test whether it reveals undesired behaviour, and thus a bug [180]. The lack of oracles or automated techniques to generated them is known as *the oracle problem* [181].

In software testing of object-oriented programs, test oracles are represented as test assertions that check properties of objects created as part of the test. Providing such test oracles for a generated test can be a difficult task: Generated tests may not represent realistic scenarios and may not be as nicely readable as human written tests. Test generation tools also tend to produce large numbers of tests. Thus, it

may not be feasible for a human developer to annotate all generated tests for a program under test with a test oracle. Although several approaches have been proposed to address the oracle problem, it still remains an open problem [181].

In order to generate oracles that are easier to understand by a human developer as opposed to oracles randomly generated, McMinn et al. [182] proposed the extraction of knowledge from the source-code and documentation of a software program. Afshan et al. [143] proposed an approach based on natural language. The main intuition is that if a test case with a readable test oracle starts to fail due to future changes, a developer would be able to easily understand the testing goal and judge whether the test is correct (i.e., it reveals a a bug in the program) or it needs to be fixed. Harman et al. [183] proposed an approach to reduce the number of generated test cases to overcome the oracle cost problem without losing code coverage. Fraser et al. [20] proposed a mutation-based approach to select a subset of oracles per test case. Their approach generates all possible oracles for each test case, and then filters out oracles that are *weak at killing* mutated versions of the program under test. Pastore et al. [184] proposed the use of crowdsourcing to address the oracle problem. In their approach, users are asked whether an oracle (that reflects the current behaviour of the program) matches the behaviour described in the documentation. If not, a bug has been found.

2.6 THE EVOSUITE UNIT TEST GENERATION TOOL

In this thesis, we use the state of the art automatic test generation tool for Java programs, EvoSuite [9]. EvoSuite works at Java bytecode level (so it can also be used on third-party systems with no available source code), and it is fully automated: it does not require manually written test drivers or parameterised unit tests. For example, when EvoSuite is used from its Eclipse plugin, a user just needs to select a class, and tests are generated with a mouse-click. EvoSuite is a tool that was mature enough to win a recent competition on unit test generation tools [16, 185–188].

EvoSuite has been extended in several ways. For example, a hybrid approach has been proposed [155] to combine the best of SBST and DSE to generate unit test suites for individual Java classes. EvoSuite uses a genetic algorithm in which it evolves whole test suites, which has been shown to be more efficient at achieving code coverage than generating tests individually [103, 113]. Depending on the search properties, DSE is adaptively used to satisfy coverage goals that are difficult for SBST. To achieve even higher coverage, several optimisations [189] have been implemented on EvoSuite. For example, proper

handling of test length bloat [115], smart seeding strategies [150, 151, 190], support for Java code that uses Java Generics [191], support of Java Enterprise Edition features [192], and the creation of mock objects to cover target goals that cannot be easily covered without mocking [193].

Once unit tests with high code coverage are generated, EvoSuite applies various post-processing steps. First, it applies *minimisation* in order to optimise the size of the resulting test suite both in terms of total number of lines of code and in number of unit tests. Minimisation works as described in Algorithm 3. For each coverage goal defined by the selected criterion, a test that covers this goal is selected from the generated test suite. Then, on a copy of that test, all statements that do not contribute to satisfaction of the goal are successively removed. When minimising for multiple criteria (as required in Chapter 3), the order in which each criterion is evaluated may influence the resulting minimised test suite. In particular, if criterion C_1 subsumes criterion C_2, then minimising for criterion C_2 first and then for C_1 may lead to tests being added during minimisation for C_2, but made redundant later, by tests added during minimisation for C_1. EvoSuite handles this problem with a second minimisation pass where a final minimised test suite with no redundant tests is produced. Second, it adds test assertions that capture the current behaviour of the tested classes. To select the most effective assertions, EvoSuite uses mutation analysis [20]. EvoSuite can generate test suites covering different kinds of coverage criteria, such as line and branch coverage (described in Sections 2.4.5.1 and 2.4.5.2 respectively), weak and strong mutation testing [103] (described in Section 2.4.5.3), and it can also aim at triggering undeclared exceptions [130]. This latter feature made it possible to automatically find thousands of faults in several open source projects [130, 194] — in Chapter 3 we present and integrate a few other coverage criteria. EvoSuite can be integrated into a programmer's development environment with its Eclipse plugin, or it can be used on the command line — in Chapter 7 we augment the number of development environments supported by EvoSuite.

2.7 SUMMARY

In this chapter, we surveyed the literature that is most related to our main contributions or that is in line with the research topics studied in this thesis.

In particular, we revisited the problem of software testing, the testing concepts used in this thesis, and how developers usually estimate the effectiveness of manually-written test cases. Moreover, we reviewed the literature of three different approaches on testing: random testing, symbolic execution, and search-based software testing. Although each approach has advantages and disadvantages, search-

Algorithm 3 Test case minimisation algorithm

```
Input: Test Suite T, Coverage Goals G
Output: Minimised Test Suite M
1: coveredGoals ← { }
2: T′ ← [ ]
3: for all g ∈ G do
4:     if g ∈ coveredGoals then
5:         next // if 'g' has been covered, there is no need to search
   for yet another test case to cover it
6:     end if
7:     // Step 1 collect all test cases that cover goal 'g'
8:     testsThatCoverGoal ← [ ]
9:     for all t ∈ T do
10:        if isGoalCoveredByTest(g, t) then
11:            testsThatCoverGoal ← testsThatCoverGoal ∪ {t}
12:        end if
13:    end for
14:    SortTestsByAscFitnessValue(testsThatCoverGoal)
15:    // Step 2 minimise the best test case that covers goal 'g'
16:    if testsThatCoverGoal ≠ ∅ then
17:        t ← testsThatCoverGoal[0]
18:        for i = NumStatements(t) to 0 do
19:            copy ← Copy(t)
20:            if RemoveStatement(i, t) fails then
21:                t ← copy // deletion of statement 'i' has failed due
   to, e.g., other statement(s) that depend on that one
22:            else
23:                fa ← CalculateFitnessValue(g, t)
24:                fb ← GetFitnessValue(g, copy)
25:                if fa > fb then
26:                    t ← copy // shorter version is worse (assumes
   a fitness function has been minimised)
27:                end if
28:            end if
29:        end for
30:        coveredGoals ← coveredGoals ∪ GetCoveredGoals(t)
31:        T′ ← T′ ∪ {t}
32:    end if
33: end for
34: // Step 3 additional pass to remove redundant test cases
35: M ← RemoveRedundantTestCases(T′, G)
36: return M
```

based has been the most successful approach on automatic test generation. Furthermore, we reviewed approaches to reduce the effort of testing a software program that is, typically, developed in a continu-

Algorithm 4 Remove redundant test cases

Input: Test Suite T, Coverage Goals G
Output: Minimised Test Suite M

1: $covGoals \leftarrow \{\}$
2: $T' \leftarrow \{\}$
3: **for all** $t \in$ REVERSE(T) **do** // assumes subsuming test cases have been inserted in the back, therefore those are consider first
4:     **if** $|covGoals| = |G|$ **then**
5:         **stop** // as all goals have been covered, there is no need to consider any other test case
6:     **end if**
7:     **for all** $g \in G \wedge g \notin covGoals$ **do**
8:         **if** ISGOALCOVEREDBYTEST(g, t) **then**
9:             // test case 't' covers at least one goal that has not been covered by any other test case
10:             $covGoals \leftarrow covGoals \cup$ GETCOVEREDGOALS(t)
11:             $T' \leftarrow T' \cup \{t\}$
12:             **stop** // as 'covGoals' already contains all goals covered by 't', there is no need to search for another goal(s) covered by 't'
13:         **end if**
14:     **end for**
15: **end for**
16: **return** REVERSE(T')

ous way. In particular, we looked at techniques that efficiently execute existing test cases to verify the correctness of the software after it is modified, and techniques to maintain and augment any existing test suite. Finally, we provided an overview of the automatic test generation tool used in this thesis.

Although much work has been done in the automation of software testing, in particular on automatic generation of test cases using search-based techniques, the applicability of search-based test generation techniques in practice is still fundamentally limited. For instance, which criteria should test generation use in order to produce the best test suites? Which evolutionary algorithms are more effective at generating test cases with high coverage? How to scale up search-based unit test generation to software projects consisting of large numbers of components, evolving and changing frequently over time? In order to answer these fundamental questions, in the following chapters we will enhance search-based software testing with several criteria to improve the search guidance of a test generator, we will evaluate which evolutionary algorithm performs best, and we will investigate several strategies to automatically generate test cases for evolving software. We will also present a set of plugins for the EVOSUITE tool that will allow developers to automatically generate test cases from different development environments.

3

COMBINING MULTIPLE COVERAGE CRITERIA IN SEARCH-BASED UNIT TEST GENERATION

○○●●

ABSTRACT

Automated test generation techniques typically aim at maximising coverage of well-established structural criteria such as statement or branch coverage. In practice, generating tests only for one specific criterion may not be sufficient when testing object oriented classes, as standard structural coverage criteria do not fully capture the properties developers may desire of their unit test suites. For example, covering a large number of statements could be easily achieved by just calling the `main` method of a class; yet, a good unit test suite would consist of smaller unit tests invoking individual methods, and checking return values and states with test assertions. There are several different properties that test suites should exhibit, and a search-based test generator could easily be extended with additional fitness functions to capture these properties. However, does search-based testing scale to combinations of multiple criteria, and what is the effect on the size and coverage of the resulting test suites? To answer these questions, we extended the EvoSuite unit test generation tool to support combinations of multiple test criteria, defined and implemented several different criteria, and applied combinations of criteria to a sample of 650 open source Java classes. Our experiments suggest that optimising for several criteria at the same time is feasible without increasing computational costs: When combining nine different criteria, we observed an average decrease of only 0.4% for the constituent coverage criteria, while the test suites may grow up to 70%.

3.1 Introduction 47
3.2 Whole Test Suite Generation for Multiple Criteria . 50
3.3 Experimental Evaluation 54
3.4 Related Work 62
3.5 Summary 62

3.1 INTRODUCTION

To support developers in creating unit test suites for object-oriented classes, automated tools can produce small and effective sets of unit tests. Test generation is typically guided by structural coverage criteria; for example, the search-based unit test generation tool EvoSuite

```java
public class ArrayIntList extends RandomAccessIntList
      implements IntList, Serializable {
  public int set(int index, int element) {
    checkRange(index);
    incrModCount();
    int oldval = _data[index];
    _data[index] = element;
    return oldval;
  }
}
```

(a) Source code excerpt.

```java
@Test
public void test9() throws Throwable {
  ArrayIntList arrayIntList0 = new ArrayIntList();
  // Undeclared exception!
  try {
     int int0 = arrayIntList0.set(200, 200);
     fail("Expecting IndexOutOfBoundsException");
  } catch(IndexOutOfBoundsException e) {
    // Should be at least 0 and less than 0, found 200
  }
}
```

(b) Test case generated by EvoSuite.

Figure 3.1: This example shows how EvoSuite covers method `set` of the class `ArrayIntList`: the method is called, but statement coverage is not achieved.

by default generates test suites optimised for branch coverage [9], and these tests can achieve higher code coverage than manually written ones [195]. However, although manual testers often *check* the coverage of their unit tests, they are usually not *guided* by it in creating their test suites. In contrast, automated tools are only guided by code coverage, and do not take into account *how* this coverage is achieved. As a result, automatically generated unit tests are fundamentally different to manually written ones, and may not satisfy the expectations of developers, regardless of coverage benefit [196].

For example, consider the excerpt of class `ArrayIntList` from the Apache Commons Primitives project in Figure 3.1a. Applying EvoSuite results in a test suite including the test case in Figure 3.1b: The test calls `set`, but with parameters that do not pass the input validation by `checkRange`, and so an exception is thrown. Nevertheless, EvoSuite believes `set` is covered with this test, and adds no further tests, thus not even satisfying statement coverage in the method. The reason is that EvoSuite follows common practice in bytecode-based coverage analysis, and only checks if branching statements evaluated to true and false [197].

```
public class Complex {
  public Complex log() {
    if (isNaN) {
      return NaN;
    }
    return createComplex(FastMath.log(abs()),
                         FastMath.atan2(imaginary, real));
  }

  public Complex pow(double x) {
    return this.log().multiply(x).exp();
  }
  ...
}
```

(a) Source code excerpt.

```
@Test
public void test1() throws Throwable {
  Complex complex0 = new Complex(Double.NaN);
  Complex complex1 = complex0.pow(Double.NaN);
  assertEquals(Double.NaN, complex1.getArgument(),0.01D);
}

@Test
public void test2() throws Throwable {
  Complex complex0 = Complex.ZERO;
  Complex complex1 = complex0.pow(complex0);
  assertFalse(complex1.isInfinite());
  assertTrue(complex1.isNaN());
}
```

(b) Test cases generated by EvoSuite.

Figure 3.2: This example shows how EvoSuite covers method `log`, even though there is no test that directly calls the method.

To fully cover the `set` method, one would also need to aim at covering all instructions. However, when optimising test suites to cover branches *and* instructions, automated techniques may find undesired ways to satisfy the target criteria. For example, consider the excerpt of class `Complex` from the Apache Commons Math project shown in Figure 3.2a: EvoSuite succeeds to cover method `log`, but because `log` is called by `pow`, in the end often only tests calling `pow` (see Figure 3.2b) are retained, which makes it hard to check the behaviour of `log` independently (e.g., with test assertions on the return value of `log`), or to debug problems caused by faults in `log`. Thus, a good test suite needs to exhibit properties beyond those captured by individual structural coverage criteria.

In this chapter, we define different criteria and their fitness functions to guide search-based test suite generation, and investigate the

effects of combining these during test generation. In particular, we investigate the effects on (i) the size of resulting test sets, and (ii) on the effectiveness of the test generators used at optimising multiple criteria. To investigate these effects, we performed a set of experiments on a sample of 650 open source classes. In summary, the contributions of this chapter are as follows:

- Identification of additional criteria to guide test suite generation.
- Implementation of these criteria as fitness functions for search-based test suite optimisation.
- An empirical study of the effects of a multiple criteria optimisation on effectiveness, convergence, and test suite size.

Our experiments suggest that optimising for several criteria at the same time is feasible without increasing computational costs, or sacrificing coverage of the constituent criteria. The increase in size depends on the combined criteria; for example, optimising for line and branch coverage instead of just line coverage increases test suites by only 10% in size, while optimising for nine different criteria leads to an increase of 70% in size. The effects of the combination of criteria on the coverage of the constituent criteria are minor; for criteria with fine-grained fitness functions the overall coverage may be reduced slightly (0.4% in our experiments), while criteria with coarse fitness functions (e.g., method coverage) may benefit from the combination with other criteria.

The chapter is structured as follows. Section 3.2 formally defines six fitness functions to guide test suite generation and presents a simple strategy to combine them. Section 3.3 presents our experimental setup, the research questions this chapter is aiming to address, and discusses the results of our experiments. Thereafter, Section 3.4 discusses the most relevant related work and Section 3.5 summarises the chapter.

3.2 WHOLE TEST SUITE GENERATION FOR MULTIPLE CRITERIA

In principle, the combination of multiple criteria is independent of the underlying test generation approach. For example, dynamic symbolic execution can generate test suites for any coverage criteria as by-product of the path exploration [198]. However, our initial usage scenario lies in unit testing for object oriented classes, an area where search-based approaches have been shown to perform well. In search-based testing, the test generation problem is cast as a search problem, such that efficient meta-heuristic search algorithms can be applied to

create tests. In the context of whole test suite generation [113], which refers to the generation of test suites rather than individual test cases, the search algorithm starts with a population of random test suites, and then evolves these using standard evolutionary operators [113]. The evolution is guided by a *fitness function* that estimates how close a candidate solution is to the optimal solution; i.e., 100% coverage in coverage-driven test generation.

3.2.1 *Fitness Functions*

In search-based test suite generation, a fitness function measures how good a test suite is with respect to the search optimisation objective, which is usually defined according to a test coverage criterion. Importantly, a fitness function usually also provides additional search guidance leading to satisfaction of the goals. For example, just checking in the fitness function whether a coverage target is achieved would not give any guidance to help covering it.

3.2.1.1 *Method Coverage*

Method Coverage is the most basic criterion for classes and requires that all methods in the Class Under Test (CUT) are executed by a test suite at least once, either via a direct call from a unit test or via indirect calls.

3.2.1.2 *Top-level Method Coverage*

For regression test suites it is important that each public method is also invoked directly (see Figure 3.2). Top-level Method Coverage requires that all methods are covered by a test suite such that a call to the method appears as a statement in a test case.

3.2.1.3 *No-exception Top-level Method Coverage*

In practice, classes often consist of many short methods with simple control flow. Often, a generated test suite achieves high levels of coverage by calling these simple methods in an invalid state or with invalid parameters (see Figure 3.1). To avoid this, No-exception Top-level Method Coverage requires that all methods are covered by a test suite via direct invocations from the tests and considering only normal-terminating executions (i.e., no exception).

The fitness functions for Method Coverage, Top-level Method Coverage and No-exception Top-level Method Coverage are *discrete* and thus have no possible guidance. Fitness values are simply calculated by counting the methods that have been covered by a test suite. Let

TotalMethods be the set of all public methods in the CUT and *CoveredMethods* be the set of methods covered by the test suite, then:

$$f_{crit}(Suite) = |\ TotalMethods\ | - |\ CoveredMethods_{crit}\ |$$

3.2.1.4 *Direct Branch Coverage*

When a test case covers a branch in a public method indirectly, i.e., without directly invoking the method that contains the branch, it is more difficult to understand how the test relates to the branch it covers (see Figure 3.2). Anecdotal evidence, from previous work with EvoSuite, also indicates that developers dislike tests that cover branches indirectly, because they are harder to understand and to extend with assertions [195]. Direct Branch Coverage requires each branch in a public method of the CUT to be covered by a direct call from a unit test, but makes no restriction on branches in private methods. The fitness function is the same as the Branch Coverage fitness function, but only methods directly invoked by the test cases are considered for the fitness and coverage computation of branches in public methods.

3.2.1.5 *Output Coverage*

Class `ArrayIntList` from Figure 3.1 has a method `size` that simply returns the value of a member variable capturing the size of the internal array; class `Complex` from Figure 3.2 has methods `isNaN` or `isInfinite` returning boolean member values. Such methods are known as *observers* or *inspectors*, and method, line, or branch coverage are all identical for such methods. Developers in this case sometimes write unit tests to cover not only in the input values of methods, but also in the output (return) values they produce; indeed output diversity can help improve the fault detection capability [199].

To account for output uniqueness and diversity, the following function maps method return types to abstract values that serve as output coverage goals:

$$output(Type) = \begin{cases} \{true, false\} & \text{if } Type \equiv Boolean \\ \{-, 0, +\} & \text{if } Type \equiv \mathit{Number} \\ \{alphabetical, digit, *\} & \text{if } Type \equiv Char \\ \{null, \neq null\} & otherwise \end{cases}$$

A unit test suite satisfies the Output Coverage criterion only if for each public method M in the CUT and for each $V_{abst} \in output(\mathit{type}(M))$, there is at least one unit test whose execution contains a call to method M for which the concrete return value is characterised by the abstract value V_{abst}.

The fitness function for the Output Coverage criterion is then defined as:

$$f_{OC}(Suite) = \sum_{g \in G} \nu(d_o(g, Suite))$$

where G is the total set of output goals for the CUT and $d_o(g, Suite)$ is an output distance function that takes as input a goal $g = \langle M, V_{abst} \rangle$:

$$d_o(g, Suite) = \begin{cases} 0 & \text{if } g \text{ is covered by at least one test,} \\ \nu(d_{num}(g, Suite)) & \text{if } type(M) \equiv Number \text{ and } g \text{ is not covered,} \\ 1 & \text{otherwise.} \end{cases}$$

In the case of methods declaring numeric return types, the search algorithm is guided with normalised numeric distances (d_{num}). For example, if a call to a method m with integer return type is observed in an execution trace and its return value is 5 (positive integer), the goal $\langle m, + \rangle$ has been covered, and the distances 5 and 6 are computed for goals $\langle m, 0 \rangle$ and $\langle m, - \rangle$, respectively.

3.2.1.6 *Exception Coverage*

One of the most interesting aspects of test suites that is not captured by standard coverage criteria is the occurrence of run-time errors, also known as *exceptions*. If exceptions are directly thrown in the CUTs with a `throw` statement, those will be retained in the final test suites if for example we optimise for line coverage. However, this might not be the case if exceptions are unintended (e.g., a null-pointer exception when calling a method on a null instance) or if thrown in the body of external methods called by the CUT. Unfortunately, it is not possible to know ahead of time the total number of feasible undeclared exceptions (e.g., null-pointer exceptions), in particular as the CUT could use custom exceptions that extend the ones in the Java API.

As a coverage criterion, we consider all possible exceptions in each method of the CUT. However, in contrast to the other criteria, it cannot be defined with a percentage (e.g., we cannot say a test suite covers 42% of the possible exceptions). We rather use the sum of all unique exceptions found per CUT method as a metric to maximise. The fitness function for Exception Coverage is thus also discrete, and is calculated in terms of the number of exceptions N_E, explicit and implicit, that have been raised in the execution of all the tests in the suite:

$$f_{EC}(Suite) = \frac{1}{1 + N_E}$$

Tracey et al. [105] were the first to present a search-based approach able to optimise test cases towards raising exceptions in order to exercise structural elements of the exception handler. Experiments on

seven simple programs reported that a search-based approach could 1) generate test cases that are able to raise almost all the exception conditions in each program, and 2) fully cover all branches of the exception handling code.

3.2.2 *Combining Fitness Functions*

All criteria considered in this chapter are non-conflicting: we can always add new tests to an existing suite to increase the coverage of a criterion without decreasing the coverage of the others. However, with limited time it may be necessary to balance the criteria, e.g., by prioritising weaker ones to avoid over-fitting for just some of the criteria involved. Thus, multi-objective optimisation algorithms based on Pareto dominance are less suitable than a linear combination of the different objectives, and we can define a combined fitness function for a set of n non-conflicting individual fitness functions $f_1 \dots f_n$ as: $f_{comp} = \sum_{i=1}^{n} w_i \times f_i$, where $w_1 \dots w_n$ are weights assigned to each individual function which allow for prioritisation of the fitness functions involved in the composition. Given enough time, a combined fitness search is expected to have the same result for each involved non-conflicting fitness function as if they were optimised for individually.

For some of the fitness functions defined above, a natural partial order exists. For instance, Method Coverage subsumes Top-level Method Coverage. The intuition is that we first want to cover all methods, independently of whether they are invoked directly from a test case statement or not. In turn, Top-level Method Coverage subsumes No-Exception Top-level Method Coverage, that is, covering all methods with direct calls from test cases is more general than covering all methods with direct calls from test cases which do not raise any exception. However, there is no natural order between other functions like for instance Output Coverage and Weak Mutation. In this chapter, we arbitrarily assign $w_i = 1$ for all i and leave the question of what are optimal w_i values for future work.

3.3 EXPERIMENTAL EVALUATION

In order to better understand the effects of combining multiple coverage criteria, we empirically aim to answer the following research questions:

RQ1: What are the effects of adding a second coverage criterion on test suite size and coverage?

RQ2: How does combining of multiple coverage criteria influence the test suite size?

RQ3: Does combining multiple coverage criteria lead to worse performance of the constituent criteria?

RQ4: How does coverage vary with increasing search budget?

3.3.1 *Experimental Setup*

To answer our research questions, we performed two different studies. The first one try to clarify if the number of test cases generated is influenced by combining more than one test criterion. The second aims to identify if a multiple criteria approach influences the global coverage of the test suite.

3.3.1.1 *Unit test generation tool*

We used EvoSuite [9], which already provides support for several criteria, in particular: Branch Coverage (Section 2.4.5.2), and Weak Mutation (Section 2.4.5.3). For this study, we implemented all the criteria described in Section 3.2.1 in the EvoSuite [9] tool. See Section 2.6 for more information about EvoSuite.

3.3.1.2 *Subject Selection*

We used the SF110 corpus [200] of Java classes for our experimental evaluation. SF110 consists of more than 20,000 classes in 110 projects; running experiments on all classes would require an infeasibly large amount of resources. Hence, we decided to select a stratified random sample of 650 classes. That is, we constructed the sample iteratively such that in each iteration we first selected a project at random, and then from that project we selected a class and added it to the sample. As a result, the sample contains classes from all 110 projects, totalling 63,191 lines of code.

3.3.1.3 *Experiment Procedure*

For each selected class, we ran EvoSuite with ten different configurations: 1) Combination of all fitness functions defined in Section 3.2.1, Branch Coverage, and Weak Mutation defined in Sections 2.4.5.2 and 2.4.5.3 respectively. 2) Only Line Coverage (baseline). 3-10) For each fitness function f defined in Section 3.2.1 (except Line Coverage) and also Branch Coverage, and Weak Mutation, a fitness function combining f and Line Coverage. Combining the other criteria with Line Coverage instead of using each of them in isolation allows a more objective evaluation, since not all the fitness functions for these other criteria can provide guidance to the search on their own. Each configuration was run using two different search budgets for the search: a small search budget of 2 minutes (it has been shown [200] that 2 minutes is sufficient for the search in EvoSuite to converge on

average), and a larger search budget of 10 minutes to study the effect of the search budget on the coverage achieved. Test suite minimisation was enabled, so that all gathered statistics refer to the final test suites EvoSuite normally produces(please refer to Section 2.6 for an explanation of how EvoSuite's minimisation works). To take the randomness of the genetic algorithm into account, we repeated the two minutes experiments 40 times, and the 10 minute experiments five times. All experiments were executed on the University of Sheffield Iceberg HPC Cluster [201].

3.3.1.4 *Experiment Analysis*

We used coverage as the main measurement of effectiveness, for all the test criteria under study. Furthermore, we also analysed the size of the resulting test suites; as the number of unit tests could be misleading, we analysed the size of a test suite in terms of its total number of statements. Statistical analysis follows the guidelines discussed by Arcuri et al. [202]: We used the Vargha-Delaney $\hat{A}_{ab}$ [203] to evaluate if a particular configuration a used on experiments performed better than another configuration b. E.g, an $\hat{A}_{ab}$ value of 0.5 means equal performance between configurations; when $\hat{A}_{ab}$ is less than 0.5, the first configuration (a) is worse; and when $\hat{A}_{ab}$ is more than 0.5, the second configuration (b) is worse. Furthermore, we used Wilcoxon-Mann-Whitney statistical symmetry test to assess the performance of different experiments. As a configuration could be better than another (i.e., $\hat{A}_{ab} > 0.5$) on some classes, but worse on other classes (i.e., $\hat{A}_{ab} < 0.5$), the statistical test checks if effect sizes (one per class) are symmetric around 0.5. I.e., it checks if there are as many classes in which a configuration gets better results as there are classes in which it gets worse results. As suggested by Fraser et al. [103], the Wilcoxon-Mann-Whitney symmetry test should only be used on a statistical sample of subjects, as it is the SF110 corpus [200] we used in our experiments. Finally, we also report the standard deviation σ and confidence intervals of averaged values using bootstrapping at 95% significance level.

3.3.1.5 *Threats to Validity*

To counter internal validity, we have carefully tested our framework, and we repeated each experiment several times and followed rigorous statistical procedures in the analysis. To cope with possible threats to external validity, the SF110 corpus was employed as case study, which is a collection of 100 Java projects randomly selected from SourceForge and the top 10 most popular projects [200]. We used only EvoSuite for experiments and did not compare with other tools; however, at least in terms of the generated tests EvoSuite is similar to other unit test generation tools. Threats to construct validity might re-

Table 3.1: Coverage results for each configuration, average of all runs for all CUTs. Size is measured in number of statements in the final minimised test suites.

Criteria	Lines	Branches	D. Branches	Methods	Top Methods	M. No Exc.	Exceptions	Mutation	Output	Size
ALL	0.78	0.75	0.75	0.87	0.90	0.88	1.35	0.75	0.64	38.01
Lines	0.78	0.73	0.22	0.81	0.74	0.71	0.45	0.69	0.27	22.25
L. & Branches	0.78	0.77	0.24	0.81	0.74	0.72	0.47	0.70	0.27	24.92
L. & D. Branches	0.78	0.76	0.76	0.87	0.85	0.82	0.48	0.70	0.27	26.73
L. & Methods	0.79	0.73	0.22	0.87	0.80	0.77	0.46	0.70	0.27	22.33
L. & Top Methods	0.78	0.73	0.22	0.87	0.89	0.86	0.48	0.70	0.27	24.89
L. & M. No Exc.	0.78	0.73	0.23	0.87	0.89	0.88	0.40	0.69	0.27	25.26
L. & Exceptions	0.78	0.72	0.22	0.81	0.78	0.70	1.93	0.70	0.27	28.00
L. & Mutation	0.79	0.75	0.23	0.81	0.75	0.72	0.50	0.76	0.27	27.45
L. & Output	0.77	0.71	0.21	0.80	0.77	0.75	0.36	0.69	0.64	23.98
Standard deviation (σ)										
ALL	0.33	0.35	0.34	0.30	0.27	0.28	1.24	0.35	0.32	46.37
Lines	0.33	0.35	0.27	0.34	0.34	0.34	0.61	0.35	0.44	30.79
L. & Branches	0.33	0.35	0.30	0.34	0.34	0.34	0.62	0.35	0.44	35.25
L. & D. Branches	0.33	0.35	0.34	0.30	0.30	0.31	0.63	0.35	0.44	36.86
L. & Methods	0.33	0.35	0.27	0.30	0.31	0.32	0.62	0.35	0.44	31.47
L. & Top Methods	0.33	0.35	0.27	0.30	0.27	0.29	0.62	0.35	0.44	34.42
L. & M. No Exc.	0.33	0.35	0.27	0.30	0.28	0.29	0.54	0.35	0.44	34.96
L. & Exceptions	0.33	0.35	0.27	0.34	0.32	0.34	1.90	0.35	0.44	37.02
L. & Mutation	0.33	0.35	0.29	0.33	0.33	0.34	0.66	0.35	0.44	38.22
L. & Output	0.33	0.35	0.27	0.34	0.34	0.34	0.46	0.35	0.32	28.30
Confidence Intervals (CI) at 95% significance level										
ALL	0.75,0.80	0.72,0.78	0.72,0.78	0.85,0.90	0.88,0.92	0.86,0.91	0.61,0.74	0.72,0.78	0.62,0.67	34.59,41.81
Lines	0.76,0.81	0.70,0.75	0.20,0.24	0.79,0.84	0.71,0.76	0.69,0.74	0.19,0.26	0.66,0.72	0.23,0.30	20.04,24.78
L. & Branches	0.76,0.81	0.74,0.79	0.22,0.27	0.79,0.84	0.72,0.77	0.69,0.74	0.20,0.27	0.68,0.73	0.23,0.30	22.31,27.68
L. & D. Branches	0.76,0.81	0.73,0.79	0.73,0.79	0.85,0.89	0.83,0.87	0.80,0.85	0.20,0.27	0.67,0.73	0.23,0.30	24.13,30.24
L. & Methods	0.76,0.81	0.70,0.75	0.20,0.24	0.85,0.89	0.77,0.82	0.74,0.79	0.19,0.26	0.67,0.72	0.23,0.30	20.23,24.95
L. & Top Methods	0.76,0.81	0.70,0.75	0.20,0.25	0.85,0.89	0.87,0.92	0.84,0.89	0.21,0.27	0.67,0.72	0.23,0.30	22.40,27.69
L. & M. No Exc.	0.76,0.81	0.70,0.75	0.21,0.25	0.85,0.90	0.87,0.91	0.86,0.90	0.17,0.23	0.67,0.72	0.23,0.30	22.51,28.16
L. & Exceptions	0.76,0.81	0.70,0.75	0.20,0.24	0.79,0.84	0.76,0.81	0.68,0.73	0.85,1.07	0.67,0.72	0.23,0.30	25.47,31.28
L. & Mutation	0.76,0.81	0.72,0.78	0.21,0.26	0.79,0.84	0.72,0.77	0.70,0.75	0.21,0.29	0.73,0.78	0.23,0.30	24.71,30.69
L. & Output	0.74,0.79	0.68,0.74	0.19,0.24	0.78,0.83	0.74,0.80	0.72,0.78	0.15,0.20	0.66,0.71	0.61,0.66	21.99,26.26

sult from our focus on coverage; for example, this does not take into account how difficult it will be to manually evaluate the test cases for writing assert statements (i.e., checking the correctness of the outputs). Although this is beyond the scope of this thesis, we have been investigating this problem [141].

3.3.2 *Results and Discussion*

In this section we present and discuss the results of each research question.

3.3.2.1 *RQ1: What are the effects of adding a second coverage criterion on test suite size and coverage?*

Table 3.1 shows the results of the experiments when using a two minutes timeout for the search. Considering line coverage as baseline, adding a further coverage criterion does not increase test suite size by a large amount. For example, adding branch coverage only increases average test suite size from 22.25 statements to 24.92 (a relative $\frac{24.92-22.25}{22.25} = 12\%$ increase). The largest increase is for the Exception Coverage testing criterion, which adds a further 28.00 - 22.25 = 5.75 statements on average to the test suites.

Regarding coverage of all criteria combined, a basic criterion like line coverage can achieve reasonable results. For instance, when line coverage is explicitly combined with branch coverage, the number of covered branches only increases 4% (from 73% to 77%). For other criteria, improvements are higher. For example, we obtain a 88 - 71 = 17% coverage improvement of No-exception Top-level Method Coverage, although with the need of 25.26 - 22.25 = 3.01 more statements. Of particular interest is the case of output coverage, where any combination of criteria (but output) achieves the same output coverage (27%), and the explicit optimisation of output achieves a higher coverage of 64% (37% increase). It is fair to assume that a method of a class under test could be fully covered at line level by a single test case and only exercise an output goal. In such case, we could say that the output coverage achieved was a simple side effect of optimising for line coverage. However, when output diversity is explicit targeted (i.e., "All" and "Line & Output" configurations), the search would try to satisfy the line criterion and exercise all possible output goals at the same time. Assuming a test case exercises one single feature (as the test cases generated by EvoSuite), exercising different output goals explicit require more test cases: 'Line" configuration generates a test suite with 22.25 statements on average, and "Line & Output" configuration generates slightly larger test suites with 23.98 statements on average. The Weak Mutation criterion reports a higher mutation score than reported by previous studies [20, 103]. Our conjecture is that the high mutation score achieved in our study is due to the different set of classes used. The Direct Branch Coverage criterion shows the largest increase (76 - 22 = 54%), which confirms that in the traditional approach code is often covered through indirect calls; this increase comes at the cost of 26.73 - 22.25 = 4.48 statements on average.

> *RQ1: In our experiments, adding a second criterion increased test suites size by 14%, and coverage by 20% over line coverage test suites.*

3.3.2.2 *RQ2: How does combining of multiple coverage criteria influence the test suite size?*

When combining all criteria together, test suite sizes increase substantially, from 22.25 to 38.01 statements. However, we argue that the resulting test suites could still be manageable for developers: 1) their size is still less than twice the size of the average baseline test suite; and 2) the increase of 15.76 (38.01 - 22.25) statements on average is also less than the sum of the increases observed for each criterion in isolation (25.56). This shows that the criteria are related and lead to coincidental coverage, where tests covering one particular goal may lead to coverage of other goals. Nevertheless, a controlled experiment with real developers to access whether the size of the resulting test suites is or is not manageable needs to be addressed in future work.

Table 3.2: For each criterion, we compare the "All" configuration for that criterion with the configuration for that criterion and line coverage. Averaged effect sizes are reported with p-values of the statistical tests of symmetry around $\mu = 0.5$, σ and CI report the standard deviation and confidence intervals using bootstrapping at 95% significance level of the effect size, respectively.

Criterion	All	Just Line & Criterion	Avg. $\hat{A}_{12}$	σ	CI	p-value
Line	0.78	0.78	0.47	0.10	[0.46,0.48]	$\leqslant 0.001$
Branch	0.75	0.77	0.47	0.11	[0.46,0.47]	$\leqslant 0.001$
Direct Branch	0.75	0.76	0.47	0.10	[0.47,0.48]	$\leqslant 0.001$
Exception	1.35	1.93	0.43	0.15	[0.42,0.44]	$\leqslant 0.001$
Method	0.87	0.87	0.50	0.04	[0.49,0.50]	0.015
Top Method	0.90	0.89	0.50	0.05	[0.50,0.51]	0.025
Method No Exc.	0.88	0.88	0.51	0.08	[0.51,0.52]	$\leqslant 0.001$
Mutation	0.75	0.76	0.46	0.10	[0.46,0.47]	$\leqslant 0.001$
Output	0.64	0.64	0.51	0.09	[0.51,0.52]	$\leqslant 0.001$

RQ2: In our experiments, combining all nine criteria increased test suites size by 70%.

3.3.2.3 *RQ3: Does combining multiple coverage criteria lead to worse performance of the constituent criteria?*

When combining different criteria together, the test generation becomes more complicated. Given the same amount of time, it could even happen that for some criteria we would get lower coverage compared to just targeting those criteria in isolation. For example, the class `Auswahlfeld` in the SF110 project `nutzenportfolio` consists of 29 methods, each consisting of only a single line. There are only 15 mutants, and when optimising for line coverage and weak mutation all mutations are easily covered within two minutes. However, when using all criteria, then the number of additional test goals based on the many methods (many of which return primitive types) means that on average after two minutes of test generation only seven mutations are covered.

On the other hand, it is conceivable that coverage criteria can "help each other", in the sense that they might smoothen the search landscape. For example, the `NewPassEventAction` class from the `jhandballmoves` project in SF110 has two complex methods with nested branches, and the `if` statements have complex expressions with up to four conditions. When optimising method calls without exceptions, after two minutes the constructor is the only method covered without exceptions, as the search problem is of the type of needle-in-the-haystack problem. However, if optimising for all criteria, then branch coverage helps reaching test cases where both methods are called without triggering any exception.

Table 3.2 shows the comparison of the "All" configuration on each criterion with the configuration that optimises line coverage and each particular criterion. For each class, we calculated the Vargha-Delaney $\hat{A}_{ab}$ effect size [202]. For each configuration comparison, we calculated the average $\hat{A}_{ab}$ and ran a Wilcoxon-Mann-Whitney symmetry test on $\mu = 0.5$, to see if a configuration leads to better or worse results on a statistically higher number of classes.

There is strong statistical difference in all the comparisons except Method Coverage and Top-level Method Coverage, which seem to consist of methods that are either trivially covered by all criteria, or never covered. For No-exception Top-level Method Coverage and Output Coverage there is a small increase in coverage; this is likely because these criteria provide little guidance and benefit from the combination with criteria with better guidance. For Exception Coverage targeting all criteria decreases the average number of exceptions substantially from 1.93 to 1.35, which may be caused by the search focusing more on valid executions related to branches and mutants, whereas without that the search becomes more random. For all other criteria there is a decrease in coverage, although very small ($\leqslant 2\%$).

RQ3: Combining multiple criteria leads to a 0.4% coverage decrease on average; criteria with coarse fitness functions can benefit more from the combination than criteria with finer grained guidance.

3.3.2.4 *RQ4: How does coverage vary with increasing search budget?*

Figure 3.3 compares the performance of the "All" configuration with the ones of Line Coverage combined with each further criterion. Performance is measured with different coverage criteria in each subplot based on the type of comparison. For example, Branch Coverage is used as performance metric when "All" is compared with "Line & Branch" configuration, whereas Method Coverage is used as performance metric when "All" is compared with "Line & Method". Performance is reported through time, from one minute to ten. The vertical y axes are scaled between the minimum and maximum value each metric obtained.

Given enough time, one could expect that the performance of "All" in each metric would become maximised and equal to just generating data for that criterion alone. Figure 3.3 shows that for the majority of criteria the performance of the "All" configuration remains slightly below the more focused search, and for Exception Coverage the more focused search even improves over time. For Output Coverage both configurations seem to converge around ten minutes and for Method Coverage the "All" configuration even takes a small lead. Overall, these results suggest that 10 minutes might not be a sufficient time interval to see convergence for all criteria – in fact, further computation, i.e., an even larger search budget than 10 minutes, may allow

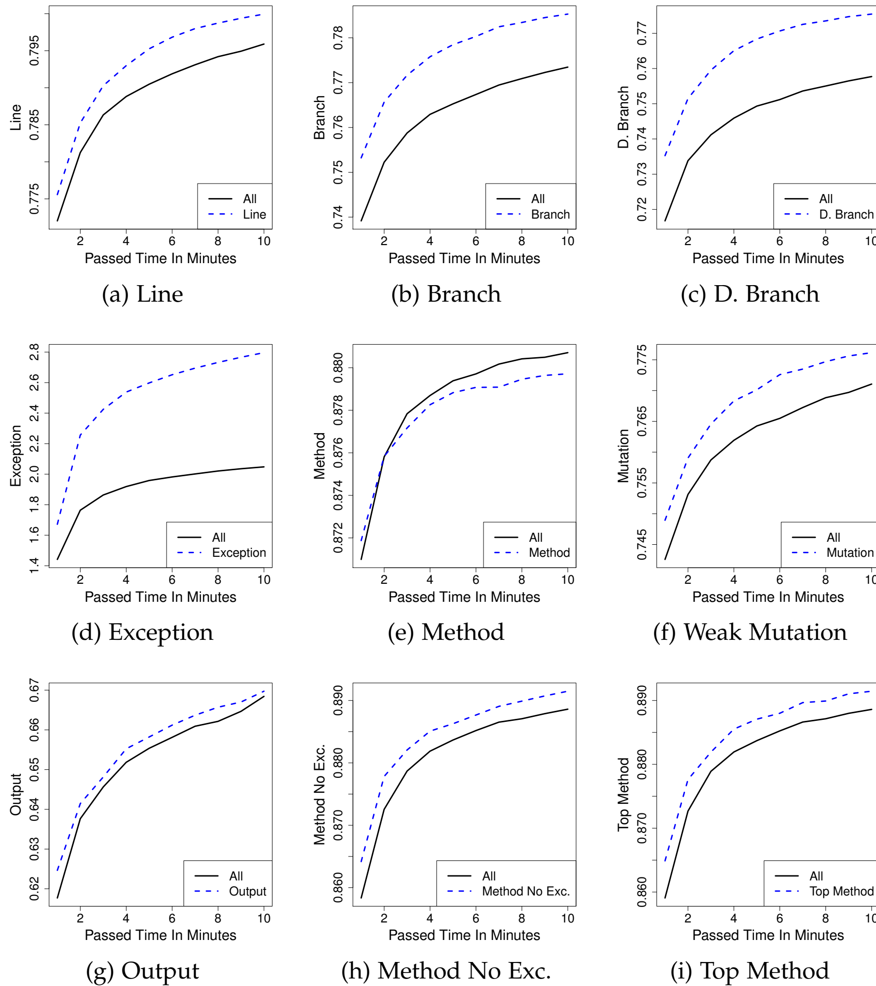


Figure 3.3: Time analysis, per minute, for each criterion for the "All" configuration compared with just optimising Line Coverage together with each of those criteria, one at a time.

the underlying evolutionary algorithm to perform even better, as the coverage achieved has not flattened out. There might also be side-effects between the combination of criteria in the "All" configuration that generate fitness plateaus in the search landscape. Another possible conjecture is that, because the search in EvoSuite minimises size as a secondary objective, over time the amount of exploration in the search space will be reduced, making it more difficult to hit additional targets that are not closely related to what is already covered. This could in principle be overcome by keeping an archive of already covered goals and matching tests, and letting the fitness function focus on uncovered goals.

RQ4: The influence of combining criteria is not limited to early phases of the search but persists over longer time, and the combination does not catch up with focused search within ten minutes.

3.4 RELATED WORK

As we discussed in Section 2.1.2, coverage criteria are well established to estimate the quality of test sets [18], and combinations of criteria have been considered in the context of regression testing [204]. For example, using multiple criteria can improve the fault detection ability after minimisation [205], and Yoo et al. [206, 207] combined coverage criteria with non-functional aspects such as execution time during minimisation. Non-functional aspects have also been considered during test generation; for example, Harman et al. [146] generated tests optimised for branch coverage and memory consumption. In contrast to this approach, we combine different non-conflicting functional criteria, and thus do not require specialised multi-objective optimisation algorithms.

3.5 SUMMARY

Although structural coverage criteria are well established in order to evaluate existing test cases [18], they may be less suitable in order to guide test generation. As with any optimisation problem, an imprecise formulation of the optimisation goal will lead to unexpected results: For example, although it is generally desirable that a reasonable test suite covers all statements of a Class Under Test (CUT), the reverse may not hold — not every test suite that executes all statements is reasonable. Additionally to this, developers do not only write test cases that maximise the number of, e.g., covered branches, they also try to write tests that cover other aspects of the program under test, e.g., exceptions. Hence, the desirable properties of a test suite are indeed multi-faceted.

In this chapter, we have tried to identify standard criteria used in practice as well as functional aspects that are not captured by standard structural coverage criteria, but are still common practice in object oriented unit testing. We have implemented a search-based approach to generate test suites optimised for combinations of these criteria. Experiments with a sample of open source Java classes have shown that such a combination does neither mean that the test suite sizes become unreasonable large, nor that the test generation performance suffers. In fact some aspects can even benefit from the combination, for example when search guidance in the case of search-based test generation is only coarse.

Equipped with several coverage criteria and a simple approach to optimise all of them, in the following chapter we perform an empirical study to identify which evolutionary algorithm is more effective at optimising (i) a single criterion (i.e., branch coverage), and (ii) the combination of all criteria evaluated in this chapter.

4

AN EMPIRICAL EVALUATION OF EVOLUTIONARY ALGORITHMS FOR TEST SUITE GENERATION

ABSTRACT

Evolutionary algorithms have been shown to be effective at generating unit test suites optimised for code coverage. While many aspects of these algorithms have been evaluated in detail (e.g., the length of generated tests), the influence of the specific algorithms has to date seen less attention in the literature. As it is theoretically impossible to design an algorithm that is best on all possible problems, a common approach in software engineering problems is to first try a Genetic Algorithm, and only afterwards try to refine it or compare it with other algorithms to see if any of them is better suited to address the specific problem. This is particularly important in test generation, since recent work suggests that random search may in practice be equally effective, whereas the reformulation as a many-objective problem seems to be more effective. To shed light on the influence of the search algorithms, we empirically evaluate seven evolutionary algorithms and two random approaches on a selection of non-trivial open source classes. Our study shows that the use of a test archive makes evolutionary algorithms clearly better than random testing, and it confirms that the many-objective search is the most effective.

4.1 Introduction 63
4.2 Empirical Study 65
4.3 Experimental Results 70
4.4 Related Work 75
4.5 Summary 76

4.1 INTRODUCTION

Search-based testing has been successfully applied to generating unit test suites optimised for code coverage on object-oriented classes. A popular approach is to use evolutionary algorithms where the individuals of the search population are whole test suites, and the optimisation goal is to find a test suite that achieves maximum code coverage [113]. Tools like EvoSuite [9] have been shown to be effective in achieving code coverage on different types of software [200],

and there is evidence that developers can benefit from using such tools [134, 196].

Since the original introduction of whole test suite generation [113], many different techniques have been introduced to improve performance even further and to get a better understanding of the current limitations. For example, the insufficient guidance provided by basic coverage based fitness functions has been shown to cause evolutionary algorithms to often be equally effective as random search [118]. As we presented in Chapter 3, optimisation no longer focuses on individual coverage criteria, but combinations of criteria [131, 134]. To cope with the resulting larger number of coverage goals, whole test suite optimisation has been re-formulated as a many-objective optimisation problem [126], and evolutionary search can be supported with *archives* [117] that keep track of useful solutions encountered throughout the search. In the context of these developments, one aspect of whole test suite generation remains largely unexplored: What is the influence of different evolutionary algorithms applied to evolve test suites?

In this chapter, we aim to shed light on the influence of the different evolutionary algorithms in whole test suite generation, to find out whether the choice of algorithm is important, and which one should be used. As we previously discussed in Section 2.4, although it is impossible to comprehensively cover all existing algorithms, in this chapter we evaluate common variants of evolutionary algorithms for test suite optimisation such as Standard GA, Monotonic GA, Steady State GA, 1+(λ, λ) GA, $\mu + \lambda$ EA, MOSA, DynaMOSA, Random Search and Random Testing. By using a large set of complex Java classes as case study, and the EvoSuite [9] search-based test generation tool, we specifically investigate:

RQ1: Which evolutionary algorithm works best when using a test archive for partial solutions?

RQ2: How does evolutionary search compare to random search and random testing?

RQ3: How does evolution of whole test suites compare to many-objective optimisation of test cases?

We investigate each of these questions in the light of individual and multiple coverage criteria as optimisation objectives, and we study the influence of the search budget. Our results show that in most cases a simple μ+λ EA is better than other, more complex algorithms. In most cases, the variants of EAs and GAs are also clearly better than random search and random testing, when a test archive is used. Finally, we confirm that many-objective search achieves higher branch coverage, even in the case of optimisation for multiple criteria.

The chapter is organised as follows. First, we detail our experimental setup in Section 4.2. We describe the *classes under test* used in

our study, our experiment procedure, and threats to validity inherent to this study. Thereafter, we answer the three research questions we enumerated earlier. In Section 4.4 we compare our work to relevant related work done in this topic. Finally, we summarise the chapter in Section 4.5.

4.2 EMPIRICAL STUDY

In order to evaluate the influence of the evolutionary algorithm on test suite generation, we conducted an empirical study. In this section, we describe the experimental setup as well as results.

4.2.1 *Experimental Setup*

4.2.1.1 *Selection of Classes Under Test*

A key factor of studying evolutionary algorithms on automatic test generation is the selection of classes under test. As many open source classes, for example contained in the SF110 [200] corpus, are trivially simple [118] and any algorithm easily covers each class fully not allowing us to make a comparison between algorithms, we used the selection of non-trivial classes from the DynaMOSA study [128]. This is a corpus of 117 open-source Java projects and 346 classes, selected from four different benchmarks. The complexity of classes ranges from 14 statements and 2 branches to 16,624 statements and 7,938 branches. The average number of statements is 1,109, and the average number of branches is 259.

4.2.1.2 *Unit Test Generation Tool*

We used EvoSuite [9], which already provides support for most of the search algorithms used this study, and would allow an unbiased comparison of the algorithms as the underlying implementation of the tool is the same across all algorithms. By default, EvoSuite uses a Monotonic GA described in Section 2.4.4.2. It also provides a Standard (Section 2.4.4.1) and Steady State GA (Section 2.4.4.3), Random search and Random testing (Section 2.4.2) and, more recently, MOSA and DynaMOSA (Section 2.4.4.6). For this study, we added the 1+(λ, λ) GA and the $\mu + \lambda$ EA (Sections 2.4.4.4 and 2.4.4.5, respectively) to EvoSuite. All evolutionary algorithms use a test archive.

4.2.1.3 *Experiment Procedure*

We performed two experiments to assess the performance of six evolutionary algorithms (described in Sections 2.4.2 and 2.4.4.1 to 2.4.4.6). First, we conducted a tuning study to select the best population size (μ) of four algorithms, number of mutations (λ) of $1 + (\lambda, \lambda)$ GA, and

population size (μ) and number of mutations (λ) of $\mu + \lambda$ EA, since the performance of each EA can be influenced by the parameters used [148]. Note that, Random-based approaches do not require any tuning. Following the tuning study, we then conducted a larger study to perform the comparison between search algorithms.

For both experiments we have four configurations: two search budgets, EvoSuite's default search budget (i.e., a small search budget) of 1 minute, and a larger search budget of 10 minutes to study the effect of the search budget on the coverage of resulting test suites; single-criterion optimisation (branch coverage) and multiple-criteria optimisation[1] (i.e., line, branch, exception, weak-mutation, output, method, method-no-exception, and cbranch) [10][2]. To account for the randomness of EAs, we repeated the one minute experiments 30 times, and the 10 minutes experiments 10 times. All experiments were executed on the University of Sheffield ShARC HPC Cluster [208].

For the tuning study, we randomly selected 10% (i.e., 34) of DynaMOSA's study classes [128][3] (with 15 to 1,707 branches, 227 on average) from 30 Java projects. This resulted in a total of 79,200 ($59,400$ one minute configurations, and $19,800$ ten minutes configurations) calls to EvoSuite and more than 175 days of CPU-time overall. For the second experiment, we used the remaining 312 classes[4] (346 total - 34 used to tune each EA) from the DynaMOSA study [128]. Besides the tuned μ and λ parameters, we used EvoSuite's default parameters [148].

4.2.1.4 *Experiment Analysis*

For any test suite generated by EvoSuite on any experimental configuration we measure the coverage achieved on eight criteria, alongside other metrics, such as the number of generated test cases, the length of generated test suites, number of iterations of each EA, number of fitness evaluations. As described by Arcuri et al. [148] "easy" branches are always covered independently of the parameter settings used, and several others are just infeasible. Therefore, rather than using raw coverage values, we use relative coverage [148]: Given the coverage of a class *c* in a run *r*, $c(r)$, the best and worst coverage of *c* in any run, $max(c)$ and $min(c)$ respectively, a *relative coverage* (r_c) can be defined as $\frac{c(r) - min(c)}{max(c) - min(c)}$. If the best and worst coverage of *c* is equal, i.e., $max(c) == min(c)$, then r_c is 1 (if range of $c(r)$ is between 0 and 1) or 100 (if range of $c(r)$ is between 0 and 100). In order to

1 At the time of writing this chapter, DynaMOSA did not support all the criteria used by EvoSuite.

2 Top-level method fitness function has been excluded from our study as it is subsumed by method-no-exception fitness function.

3 Class `com.yahoo.platform.yui.compressor.YUICompressor` was excluded from tuning experiments due to a bug in EvoSuite.

4 Nine classes were discarded from the second experiment due to crashes of EvoSuite.

statistically compare the performance of each EA we use the Vargha-Delaney $\hat{A}_{12}$ effect size, and the Wilcoxon-Mann-Whitney U-test with a 95% confidence level. We also consider a *relative average improvement* metric which, given two sets of coverage values: configuration A and configuration B, can be defined as $\frac{mean(A)-mean(B)}{mean(B)}$. Furthermore, we also consider the standard deviation σ and confidence intervals of the coverage achieved by each EA using bootstrapping at 95% significance level.

4.2.1.5 *Threats to Validity*

The results reported in this chapter are limited to the number and type of EAs used in the experiments. However, we believe these are representative of state-of-art algorithms, and are sufficient in order to demonstrate the influence of each algorithm on the problem. Although we used a large number of different subjects (346 complex classes from 117 open-source Java projects), also used by a previous study [128] on test generation, our results may not generalise to other subjects. The range of parameters used in the tuning experiments was limited to only 4 values per EA. Although we used common or reported as best values, different values might influence the performance of each EA. The two search budgets used in the tuning experiments and in the empirical study are based on EvoSuite's defaults (1 minute), and used by previous studies to assess the performance of EAs with a larger search budget (10 minutes) [10, 131].

4.2.2 *Parameter Tuning*

The execution of an EA requires a number of parameters to be set. As there is not a single best configuration setting to solve all problems [114] in which an EA could be applied, a possible alternative is to tune EA's parameters for a specific problem at hand to find the "best" ones. We largely rely on a previous tuning study [148] in which default values were determined for most parameters of EvoSuite. However, the main distinguishing factor between the algorithms we are considering in this study are μ (i.e., the population size) and λ (i.e., the number of mutations). In particular, we selected common values used in previous studies and reported to be the best for each EA:

- Population size of 10, 25, 50, and 100 for Standard GA, Monotonic GA, SteadyState GA, MOSA, and DynaMOSA.
- λ size of 1, 8 [124], 25, and 50 for $1 + (\lambda, \lambda)$ GA.
- μ size of 1, 7 [209], 25, and 50, and λ size of 1, 7, 25, and 50 for $\mu + \lambda$ EA.

Table 4.1: Best population / λ size of each EA per search budget, and single and multiple criteria optimisation. "Br. Cov." column reports the branch coverage per EA, and column "Over. Cov.", the overall coverage of a multiple-criteria optimisation. σ and CI columns report the standard deviation and confidence intervals, respectively, of the branch coverage per EA on single-criteria, and the overall coverage per EA on multiple-criteria.

Algorithm	$\lvert P \rvert$	Branch Cov.	Overall Cov.	σ	CI	Avg. $\hat{A}_{12}$	Better $\hat{A}_{12}$	Worse $\hat{A}_{12}$
Search budget of 60 seconds – Single-criteria								
Standard GA	10	0.83	—	0.21	[0.82,0.84]	0.52	0.75	0.24
Monotonic GA	25	0.83	—	0.20	[0.83,0.84]	0.52	0.76	0.32
Steady-State GA	100	0.81	—	0.22	[0.80,0.81]	0.50	0.72	0.32
$1 + (\lambda, \lambda)$ GA	50	0.57	—	0.28	[0.56,0.58]	0.58	0.70	N/A
$\mu + \lambda$ EA	1+7	0.84	—	0.21	[0.83,0.84]	0.55	0.74	0.21
MOSA	100	0.84	—	0.20	[0.84,0.85]	0.51	0.79	0.32
DynaMOSA	25	0.84	—	0.20	[0.84,0.85]	0.51	0.68	0.28
Search budget of 600 seconds – Single-criteria								
Standard GA	100	0.86	—	0.19	[0.85,0.87]	0.50	0.84	0.21
Monotonic GA	100	0.87	—	0.19	[0.86,0.88]	0.53	0.83	0.22
Steady-State GA	10	0.85	—	0.20	[0.84,0.86]	0.51	0.80	0.23
$1 + (\lambda, \lambda)$ GA	50	0.57	—	0.28	[0.56,0.59]	0.57	0.83	N/A
$\mu + \lambda$ EA	50+50	0.85	—	0.19	[0.84,0.86]	0.49	0.84	0.12
MOSA	50	0.86	—	0.21	[0.85,0.88]	0.53	0.88	0.18
DynaMOSA	25	0.85	—	0.21	[0.84,0.87]	0.50	0.83	0.19
Search budget of 60 seconds – Multiple-criteria								
Standard GA	100	0.78	0.88	0.14	[0.88,0.89]	0.52	0.75	0.23
Monotonic GA	100	0.78	0.88	0.14	[0.88,0.89]	0.52	0.77	0.21
Steady-State GA	100	0.74	0.86	0.14	[0.86,0.87]	0.53	0.75	0.27
$1 + (\lambda, \lambda)$ GA	50	0.65	0.81	0.16	[0.81,0.82]	0.53	0.69	0.33
$\mu + \lambda$ EA	1+7	0.79	0.89	0.13	[0.89,0.89]	0.56	0.76	0.28
MOSA	25	0.81	0.62	0.32	[0.61,0.63]	0.54	0.70	0.21
DynaMOSA	—	—	—	—	—	—	—	—
Search budget of 600 seconds – Multiple-criteria								
Standard GA	25	0.84	0.93	0.09	[0.92,0.93]	0.51	0.76	0.23
Monotonic GA	25	0.84	0.92	0.08	[0.92,0.93]	0.52	0.80	0.24
Steady-State GA	25	0.79	0.90	0.10	[0.89,0.90]	0.51	0.79	0.26
$1 + (\lambda, \lambda)$ GA	8	0.75	0.81	0.25	[0.79,0.83]	0.53	0.85	0.19
$\mu + \lambda$ EA	1+1	0.85	0.92	0.09	[0.92,0.93]	0.53	0.86	0.22
MOSA	10	0.87	0.68	0.33	[0.66,0.70]	0.54	0.86	0.12
DynaMOSA	—	—	—	—	—	—	—	—

A *N/A* effect size means there is no other configuration that achieved a statistically significantly higher coverage than the best configuration.

Thus, for Standard GA, Monotonic GA, SteadyState GA, MOSA, DynaMOSA, and $1 + (\lambda, \lambda)$ GA there are 4 different configurations; for $\mu + \lambda$, and as λ must be divisible by μ, there are 8 different configurations (i.e., $1 + 1$, $1 + 7$, $1 + 25$, $1 + 50$, $7 + 7$, $25 + 25$, $25 + 50$, $50 + 50$); i.e., a total of 32 different configurations.

Table 4.2: Branch and overall coverage, standard deviation (σ), and confidence intervals (CI) at 95% significance level per algorithm for a search budget of 60 seconds.

	Branch			Overall		
Algorithm	Cov.	σ	CI	Cov.	σ	CI
Search budget of 60 seconds – Single-criteria						
Random search	0.78	0.26	[0.75,0.81]	—	—	—
Random testing	0.72	0.30	[0.68,0.75]	—	—	—
Standard GA	0.80	0.25	[0.77,0.83]	—	—	—
Monotonic GA	0.82	0.23	[0.80,0.85]	—	—	—
Steady-State GA	0.77	0.27	[0.74,0.80]	—	—	—
$1+(\lambda,\lambda)$ GA	0.74	0.27	[0.71,0.77]	—	—	—
$\mu+\lambda$ EA	0.83	0.23	[0.80,0.86]	—	—	—
MOSA	0.84	0.23	[0.82,0.87]	—	—	—
DynaMOSA	0.85	0.22	[0.83,0.88]	—	—	—
Search budget of 60 seconds – Multiple-criteria						
Random search	0.76	0.24	[0.73,0.79]	0.65	0.21	[0.63,0.68]
Random testing	0.71	0.27	[0.68,0.74]	0.67	0.20	[0.64,0.69]
Standard GA	0.77	0.25	[0.75,0.80]	0.79	0.19	[0.77,0.82]
Monotonic GA	0.78	0.24	[0.75,0.81]	0.80	0.18	[0.78,0.82]
Steady-State GA	0.72	0.27	[0.69,0.75]	0.76	0.20	[0.74,0.78]
$1+(\lambda,\lambda)$ GA	0.53	0.30	[0.49,0.56]	0.70	0.18	[0.68,0.72]
$\mu+\lambda$ EA	0.77	0.24	[0.74,0.79]	0.79	0.18	[0.77,0.81]
MOSA	0.80	0.22	[0.78,0.83]	0.58	0.33	[0.55,0.62]
DynaMOSA	—	—	—	—	—	—

To identify the best population size of each EA, we performed a pairwise comparison of the coverage achieved by using any population size. The population size that achieved a significantly higher coverage more often was selected as the best. Table 4.1 shows that, for a search budget of 60 seconds and single-criteria, the best population size is different for almost all EAs (e.g., Standard GA works best with a population size of 10, and MOSA with a population size of 100). For a search budget of 600 seconds and multiple-criteria several EAs share the same population size, for example, the best value for Standard GA, Monotonic GA and Steady-State GA on multiple-criteria is 25. Table 4.1 also reports the average effect size of the best parameter value when compared to all possible parameter values; and the effect size of pairwise comparisons in which the best parameter was statistically significantly better/worse.

Table 4.3: Branch and overall coverage, standard deviation (σ), and confidence intervals (CI) at 95% significance level per algorithm for a search budget of 600 seconds.

Algorithm	Branch Cov.	σ	CI	Overall Cov.	σ	CI
Search budget of 600 seconds – Single-criteria						
Random search	0.80	0.27	[0.77,0.83]	—	—	—
Random testing	0.73	0.33	[0.69,0.77]	—	—	—
Standard GA	0.87	0.22	[0.85,0.90]	—	—	—
Monotonic GA	0.89	0.20	[0.87,0.92]	—	—	—
Steady-State GA	0.86	0.23	[0.84,0.89]	—	—	—
$1 + (\lambda, \lambda)$ GA	0.77	0.23	[0.75,0.80]	—	—	—
$\mu + \lambda$ EA	0.90	0.19	[0.88,0.92]	—	—	—
MOSA	0.90	0.19	[0.88,0.93]	—	—	—
DynaMOSA	0.91	0.18	[0.89,0.93]	—	—	—
Search budget of 600 seconds – Multiple-criteria						
Random search	0.70	0.28	[0.67,0.73]	0.65	0.26	[0.62,0.67]
Random testing	0.72	0.30	[0.69,0.76]	0.74	0.22	[0.71,0.76]
Standard GA	0.84	0.22	[0.82,0.87]	0.85	0.19	[0.83,0.87]
Monotonic GA	0.85	0.20	[0.83,0.87]	0.85	0.18	[0.83,0.87]
Steady-State GA	0.72	0.29	[0.69,0.76]	0.79	0.21	[0.77,0.82]
$1 + (\lambda, \lambda)$ GA	0.62	0.32	[0.58,0.65]	0.75	0.16	[0.74,0.77]
$\mu + \lambda$ EA	0.87	0.19	[0.85,0.90]	0.86	0.18	[0.84,0.88]
MOSA	0.87	0.20	[0.84,0.89]	0.71	0.33	[0.67,0.75]
DynaMOSA	—	—	—	—	—	—

4.3 EXPERIMENTAL RESULTS

Tables 4.2 and 4.3 summarises the results of the main experiment described in the previous section. For each algorithm we report the branch coverage achieved for single and multiple criteria, and the overall coverage for multiple criteria. Tables 4.2 and 4.3 also reports the standard deviation and confidence intervals (CI) of the coverage achieved (either branch or overall coverage) using bootstrapping at 95% significance level.

4.3.1 *RQ1 – Which evolutionary algorithm works best when using a test archive for partial solutions?*

Table 4.4 summarises the results of a pairwise tournament of all EAs. An EA X is considered to be better than an EA Y if it performs statistically significantly better on a higher number of comparisons, i.e., if it achieves a statistically significantly higher coverage more often. As there are 5 algorithms and 303 classes under test, we performed (5 -

Table 4.4: Pairwise comparison of all evolutionary algorithms. "Better than" and "Worse than" give the number of comparisons for which the best EA is statistically significantly (i.e., *p-value*< 0.05) better and worse, respectively. Columns $\hat{A}_{12}$ give the average effect size.

Algorithm	Tourn. Position	Branch Cov.	Overall Cov.	$\hat{A}_{12}$	Better than	$\hat{A}_{12}$	Worse than	$\hat{A}_{12}$
Search budget of 60 seconds – Single-criteria								
Standard GA	3	0.80	—	0.52	223 / 1212	0.79	149 / 1212	0.25
Monotonic GA	2	0.82	—	0.56	299 / 1212	0.78	57 / 1212	0.27
Steady-State GA	4	0.77	—	0.42	112 / 1212	0.76	401 / 1212	0.19
$1 + (\lambda, \lambda)$ GA	5	0.74	—	0.40	53 / 1212	0.73	432 / 1212	0.22
$\mu + \lambda$ EA	1	0.83	—	0.60	387 / 1212	0.79	35 / 1212	0.26
Search budget of 600 seconds – Single-criteria								
Standard GA	3	0.87	—	0.52	129 / 1212	0.87	96 / 1212	0.16
Monotonic GA	2	0.89	—	0.57	192 / 1212	0.89	20 / 1212	0.16
Steady-State GA	4	0.86	—	0.44	50 / 1212	0.80	217 / 1212	0.10
$1 + (\lambda, \lambda)$ GA	5	0.77	—	0.39	14 / 1212	0.82	258 / 1212	0.13
$\mu + \lambda$ EA	1	0.90	—	0.59	224 / 1212	0.88	18 / 1212	0.19
Search budget of 60 seconds – Multiple-criteria								
Standard GA	2	0.77	0.79	0.62	473 / 1212	0.85	98 / 1212	0.20
Monotonic GA	1	0.78	0.80	0.62	470 / 1212	0.85	95 / 1212	0.21
Steady-State GA	4	0.72	0.76	0.43	233 / 1212	0.88	503 / 1212	0.19
$1 + (\lambda, \lambda)$ GA	5	0.53	0.70	0.25	140 / 1212	0.86	896 / 1212	0.10
$\mu + \lambda$ EA	3	0.77	0.79	0.59	493 / 1212	0.84	217 / 1212	0.19
Search budget of 600 seconds – Multiple-criteria								
Standard GA	2	0.84	0.85	0.59	357 / 1212	0.93	112 / 1212	0.11
Monotonic GA	3	0.85	0.85	0.58	345 / 1212	0.93	125 / 1212	0.13
Steady-State GA	5	0.72	0.79	0.33	118 / 1212	0.94	566 / 1212	0.08
$1 + (\lambda, \lambda)$ GA	4	0.62	0.75	0.35	254 / 1212	0.91	623 / 1212	0.05
$\mu + \lambda$ EA	1	0.87	0.86	0.64	437 / 1212	0.93	85 / 1212	0.09

1) $\times$ 303 = 1,212 comparisons. For example, for a search budget of 60 seconds and single-criteria, $1 + (\lambda, \lambda)$ was statistically significantly better than on 53 comparisons, while it was statistically significantly worse on 432 comparisons out of 1,212 – which make it the worst EA. On the other hand, $\mu + \lambda$ was the one that won more tournaments (387) and lost less tournaments (just 35) – thus, being the best EA for a search budget of 60 seconds and single-criteria, and for a search budget of 600 seconds on single and multiple-criteria. While it is ranked only third for 60 seconds search budget and multiple-criteria, the coverage is only slightly lower compared to the higher ranked algorithms (0.79 vs. 0.80), with an $\hat{A}_{12}$ effect size of 0.59 averaged over all comparisons.

RQ1: In 3 out of 4 configurations, $\mu + \lambda$ EA is better than the other considered evolutionary algorithms.

Table 4.5: Comparison of evolutionary algorithms and two random-based approaches: Random search and Random testing.

Algorithm	Branch Cov.	Overall Cov.	EA vs. Random search $\hat{A}_{12}$	EA vs. Random search p	EA vs. Random search Rel. Impr.	EA vs. Random testing $\hat{A}_{12}$	EA vs. Random testing p	EA vs. Random testing Rel. Impr.
Search budget of 60 seconds – Single-criteria								
Random search	0.78	—	—	—	—	—	—	—
Random testing	0.72	—	—	—	—	—	—	—
Standard GA	0.80	—	0.62	0.26	+15.9%	0.68	0.22	+62.4%
Monotonic GA	0.82	—	0.66	0.23	+21.9%	0.71	0.20	+68.9%
Steady-State GA	0.77	—	0.51	0.27	+2.9%	0.60	0.28	+37.8%
$1+(\lambda,\lambda)$ GA	0.74	—	0.50	0.32	+1.5%	0.58	0.34	+36.1%
$\mu+\lambda$ EA	0.83	—	0.69	0.22	+23.5%	0.73	0.19	+71.8%
Search budget of 600 seconds – Single-criteria								
Random search	0.80	—	—	—	—	—	—	—
Random testing	0.73	—	—	—	—	—	—	—
Standard GA	0.87	—	0.69	0.19	+29.0%	0.73	0.16	+116.0%
Monotonic GA	0.89	—	0.73	0.16	+35.2%	0.76	0.14	+122.0%
Steady-State GA	0.86	—	0.63	0.22	+20.9%	0.71	0.19	+97.3%
$1+(\lambda,\lambda)$ GA	0.77	—	0.57	0.39	+8.4%	0.63	0.38	+63.6%
$\mu+\lambda$ EA	0.90	—	0.74	0.16	+36.5%	0.76	0.12	+128.7%
Search budget of 60 seconds – Multiple-criteria								
Random search	0.76	0.65	—	—	—	—	—	—
Random testing	0.71	0.67	—	—	—	—	—	—
Standard GA	0.77	0.79	0.79	0.20	+36.2%	0.84	0.19	+26.7%
Monotonic GA	0.78	0.80	0.80	0.21	+37.6%	0.84	0.18	+28.5%
Steady-State GA	0.72	0.76	0.72	0.23	+29.6%	0.78	0.24	+18.8%
$1+(\lambda,\lambda)$ GA	0.53	0.70	0.62	0.26	+20.1%	0.62	0.39	+9.7%
$\mu+\lambda$ EA	0.77	0.79	0.76	0.21	+35.9%	0.83	0.20	+25.8%
Search budget of 600 seconds – Multiple-criteria								
Random search	0.70	0.65	—	—	—	—	—	—
Random testing	0.72	0.74	—	—	—	—	—	—
Standard GA	0.84	0.85	0.88	0.17	+64.0%	0.83	0.20	+28.0%
Monotonic GA	0.85	0.85	0.88	0.18	+64.8%	0.83	0.20	+28.7%
Steady-State GA	0.72	0.79	0.79	0.23	+51.4%	0.71	0.29	+17.6%
$1+(\lambda,\lambda)$ GA	0.62	0.75	0.79	0.30	+49.1%	0.72	0.40	+14.0%
$\mu+\lambda$ EA	0.87	0.86	0.88	0.15	+66.1%	0.84	0.18	+30.6%

4.3.2 *RQ2 – How does evolutionary search compare to random search and random testing?*

Table 4.5 compares the results of each EA with the two random-based techniques, Random search and Random testing. Although Random search performs better than Random testing on single-criteria, the overall coverage in the multiple-criteria case is higher for Random testing than Random search. Our conjecture is that, in the multiple-criteria scenario, there are many more trivial coverage goals where the fitness function provides no guidance (thus benefiting Random

testing); in contrast, branch coverage goals seem to benefit from the test archive when generating new individuals (thus benefiting Random search).

On average, EAs achieve higher coverage (either branch coverage on single-criteria or overall coverage on multiple-criteria) than Random search and Random testing. For instance, for a search budget of 600 seconds and single-criteria, Random search covers 80% of all branches on average and $\mu + \lambda$ EA covers 90% (a relative improvement of +36.5%). This result is different to the earlier study by Shamshiri et al. [118], where Random testing achieved similar, and sometimes higher coverage. Our conjecture is that the better performance of the EAs in our evaluation is due to (1) the use of the test archive as suggested by Rojas et al. [117], and (2) the use of more complex classes in the experiment, as opposite to all classes from SF110 [200] corpus which due to the simplicity of the majority of classes, the EAs would be equally effective. Although different, our results corroborate Shamshiri et al. [118] findings that on classes where EAs benefit of guidance, EAs are more successful than Random testing.

RQ2: Evolutionary algorithms (in particular $\mu + \lambda$ EA) perform better than random search and random testing.

4.3.3 *RQ3 – How does evolution of whole test suites compare to many-objective optimisation of test cases?*

Table 4.6 compares each EA with the many-objective optimisation techniques MOSA and DynaMOSA. Our results confirm and enhance previous studies [126, 128] by evaluating four different EAs (i.e., Standard GA, Steady-State GA, $1 + (\lambda, \lambda)$ GA, and $\mu + \lambda$ EA) in addition to Monotonic GA, and show that MOSA and DynaMOSA perform better at optimising test cases than any EA at optimising test suites for single criteria. Although $\mu + \lambda$ achieves a marginally higher average coverage on single criteria (600 seconds) with a relative improvement of +1.6%, it is still slightly worse than MOSA with an average effect size of 0.49.

In the multiple-criteria scenario (in which we can only compare to MOSA), MOSA performs better than any other EA at optimising branch coverage, but the overall coverage is substantially lower compared to all other EAs. On the one hand, the lower overall coverage is expected since MOSA is not efficient for very large sets of coverage goals (this is what DynaMOSA addresses). However, the fact that branch coverage is nevertheless higher is interesting. A possible conjecture is that this is due to MOSA's slightly different fitness function for branch coverage [126], which includes the approach

Table 4.6: Comparison of evolutionary algorithms on whole test suites optimisation and many-objective optimisation algorithms of test cases.

	Branch	Overall	EA vs. MOSA			EA vs. DynaMOSA		
Algorithm	Cov.	Cov.	$\hat{A}_{12}$	p	Rel. Impr.	$\hat{A}_{12}$	p	Rel. Impr.
Search budget of 60 seconds – Single-criteria								
MOSA	0.84	—	—	—	—	—	—	—
DynaMOSA	0.85	—	—	—	—	—	—	—
Standard GA	0.80	—	0.39	0.27	-3.6%	0.37	0.28	-6.0%
Monotonic GA	0.82	—	0.43	0.26	-0.4%	0.41	0.28	-2.3%
Steady-State GA	0.77	—	0.30	0.19	-9.7%	0.28	0.19	-10.7%
$1 + (\lambda, \lambda)$ GA	0.74	—	0.31	0.26	-12.5%	0.29	0.25	-14.3%
$\mu + \lambda$ EA	0.83	—	0.46	0.28	+0.8%	0.44	0.29	-1.5%
Search budget of 600 seconds – Single-criteria								
MOSA	0.90	—	—	—	—	—	—	—
DynaMOSA	0.91	—	—	—	—	—	—	—
Standard GA	0.87	—	0.42	0.24	-3.2%	0.40	0.23	-4.6%
Monotonic GA	0.89	—	0.47	0.24	+0.2%	0.44	0.23	-1.4%
Steady-State GA	0.86	—	0.38	0.22	-3.5%	0.36	0.21	-5.1%
$1 + (\lambda, \lambda)$ GA	0.77	—	0.34	0.37	-14.3%	0.33	0.35	-15.6%
$\mu + \lambda$ EA	0.90	—	0.49	0.22	+1.6%	0.47	0.23	-0.7%
Search budget of 60 seconds – Multiple-criteria								
MOSA	0.80	0.58	—	—	—	—	—	—
DynaMOSA	—	—	—	—	—	—	—	—
Standard GA	0.77	0.79	0.71	0.18	+8737.7%	—	—	—
Monotonic GA	0.78	0.80	0.71	0.17	+9069.9%	—	—	—
Steady-State GA	0.72	0.76	0.63	0.17	+9058.6%	—	—	—
$1 + (\lambda, \lambda)$ GA	0.53	0.70	0.59	0.21	+7941.9%	—	—	—
$\mu + \lambda$ EA	0.77	0.79	0.70	0.17	+9071.2%	—	—	—
Search budget of 600 seconds – Multiple-criteria								
MOSA	0.87	0.71	—	—	—	—	—	—
DynaMOSA	—	—	—	—	—	—	—	—
Standard GA	0.84	0.85	0.64	0.19	+772.4%	—	—	—
Monotonic GA	0.85	0.85	0.64	0.20	+773.4%	—	—	—
Steady-State GA	0.72	0.79	0.52	0.19	+694.6%	—	—	—
$1 + (\lambda, \lambda)$ GA	0.62	0.75	0.56	0.27	+632.7%	—	—	—
$\mu + \lambda$ EA	0.87	0.86	0.67	0.18	+769.5%	—	—	—

level (whereas whole test suite optimisation considers only branch distances)[5].

RQ3: MOSA improves over EAs for individual criteria; for multiple-criteria it achieves higher branch coverage even though overall coverage is lower.

5 Please refer to Sections 2.4.5.1 and 2.4.5.2 for a detailed explanation of branch distance and approach level, respectively.

4.4 RELATED WORK

Although a common approach in search-based testing is to use genetic algorithms, numerous other algorithms have been proposed in the domain of nature-inspired algorithms, as no algorithm can be best on all domains [114].

Many researchers compared evolutionary algorithms to solve problems in domains outside software engineering [210–212]. Within search-based software engineering, comparative studies have been conducted in several domains such as discovery of software architectures [213], pairwise testing of software product lines [214], or finding subtle higher order mutants [215].

In the context of test data generation, Harman et al. [119] empirically compared GA, Random testing and Hill Climbing for structural test data generation. While their results indicate that sophisticated evolutionary algorithms can often be outperformed by simpler search techniques, there are more complex scenarios, for which evolutionary algorithms are better suited. Ghani et al. [216] compared Simulated Annealing (SA) and GA for the test data generation for Matlab Simulink models, and their results show that GA performed slightly better than SA. Sahin et al. [140] evaluated Particle Swarm Optimisation (PSO), Differential Evolution (DE), Artificial Bee Colony, Firefly Algorithm and Random search algorithms on software test data generation benchmark problems, and concluded that some algorithms performs better than others depending on the characteristics of the problem. For example, ABC performs better when a larger number of constraints is involved. They also concluded that Random search is effective on easy problems, while it is not satisfactory on hard problems. Varshney et al. [217] proposed a DE-based approach to generate test data that cover data-flow coverage criteria, and compared the proposed approach to Random search, GA and PSO with respect to number of generations and average percentage coverage. Their results show that the proposed DE-based approach is comparable to PSO and has better performance than Random search and GA. In contrast to these studies, we consider unit test generation, which arguably is a more complex scenario than test data generation, and in particular local search algorithms are rarely applied.

Although often newly proposed algorithms are compared to random search as a baseline (usually showing clear improvements), there are some studies that show that random search can actually be very efficient for test generation. In particular, Shamshiri et al. [118] compared GA against Random search for generating test suites, and found almost no difference between the coverage achieved by evolutionary search compared to random search. They observed that GAs covers more branches when standard fitness functions provide guidance, but most branches of the analysed projects provided no such

guidance. Similarly, Sahin et al. [140] showed that Random search is effective on simple problems.

To the best of our knowledge, no study has been conducted to evaluate several different evolutionary algorithms in a whole test suite generation context and considering a large number of complex classes. As can be seen from this overview of comparative studies, it is far from obvious what the best algorithm is, since there are large variations between different search problems.

4.5 SUMMARY

Although evolutionary algorithms are commonly applied for whole test suite generation, there is a lack of evidence on the influence of different algorithms. Our study yielded the following key results:

– The choice of algorithm can have a substantial influence on the performance of whole test suite optimisation, hence tuning is important. While EvoSuite provides tuned default values, these values may not be optimal for different flavours of evolutionary algorithms.

– EvoSuite's default algorithm, a Monotonic GA, is an appropriate choice for EvoSuite's default configuration (60 seconds search budget, multiple criteria). However, for other search budgets and optimisation goals, other algorithms such as a $\mu + \lambda$ EA may be a better choice.

– Although previous studies showed little benefit of using a GA over random testing, our study shows that on complex classes and with a test archive, evolutionary algorithms are superior to random testing and random search.

– The Many Objective Sorting Algorithm (MOSA) is superior to whole test suite optimisation; it would be desirable to extend EvoSuite so that DynaMOSA supports all coverage criteria.

Despite the fact we have provided evidence of which evolutionary algorithm achieves high coverage, an important question remains open: Are test cases generated by the best evolutionary algorithm found in our evaluation, i.e., MOSA, able to fulfil the purpose of testing — finding faults in the software under test? In the next chapter we study the effectiveness of test cases generated by MOSA and an extended version of MOSA (which besides coverage also optimises the diversity of test cases) at detecting real faults, and helping developers to find the location of the faulty code.

5

ENTROPY: A NON-FUNCTIONAL CRITERION TO IMPROVE THE DIAGNOSTIC ABILITY OF AUTOMATICALLY GENERATED UNIT TESTS

ABSTRACT

Automatic unit test generation techniques usually aim to cover structural properties of the program under test, e.g., all program branches. However, even if they exercise 100% of all lines/branches of the program under test, automatically generated unit tests might not exhibit properties one might desire, for example, ability to find faults. In this chapter we extend a coverage based approach to test generation with an additional non-functional criterion dubbed *entropy*, to improve the ability of automatically generated tests at (i) triggering the faulty behaviour of a program under test, and (ii) reducing the human effort of localising the root cause of a fault. An empirical evaluation on real faults shows that test suites optimised for coverage and *entropy* are more effective at revealing 4 out of 6 real faults, 25% more effective at localising the root cause of each real fault, and 1.5% smaller than test suites only optimised for coverage.

5.1 Introduction 77
5.2 Background 78
5.3 Entropy as a Non-Functional Criterion for Automated Test Generation 83
5.4 Empirical Study 88
5.5 Related Work 100
5.6 Summary 101

5.1 INTRODUCTION

As discussed and evaluated in the previous chapters, evolutionary algorithms are very effective at generating unit test suites optimised for code coverage. Our experiments in Chapter 3 have shown that an evolutionary algorithm can optimise several coverage criteria at the same time without sacrificing its performance, and in Chapter 4 we evaluated which evolutionary algorithm achieves the highest code coverage on object-oriented programs. However, it has been recently reported that (i) although automatically generated test cases are more effective at achieving higher levels of coverage than manually written test cases, they are not as easy to adopt by developers as one might

think [196]; and (ii) although test suites with high coverage are more likely to find faults [131], they are not necessarily more effective [218].

In order to increase the practicality of automatically generated test cases, non-functional *properties* such as the length [113] of a test suite, memory consumption [146], readability [141], or code quality [147] have been explored. In this chapter we propose the integration of a functional criterion such as branch coverage and a non-functional metric called *entropy* [219, 220] to improve the ability of automatically generated test cases at detecting and localising faults. In summary, the contributions of this chapter are as follows:

- We propose an entropy-based metric effective at fault detection and fault localisation.
- We integrate the proposed metric into the most effective evolutionary algorithm for unit test generation, MOSA.
- We empirically evaluate the effectiveness of the proposed metric at detecting and localising six real faults.

The results of our experiments showed strong statistical evidence that optimising for coverage and entropy is more effective at revealing 4 out of 6 real faults, 25% more effective at localising the root cause of each fault, and results in 1.5% smaller test suites than only optimising for coverage.

The chapter is structured as follows. Section 5.3 defines *entropy* as a fitness function and discusses different alternatives of integrating it in unit test generation. Section 5.4 details the experimental setup and discusses the results. Section 5.5 surveys the most relevant related work and Section 5.6 summarises the chapter.

5.2 BACKGROUND

To illustrate that other criteria than coverage may need to be explored to improve the diagnostic ability of automatically generated tests, consider the example in Figure 5.1 which shows a variation of the well-known triangle example [221]. There is a fault at statement c_6: method *type* declares the predicate $b == a$ but the correct condition should be $b == c$. The automatically generated test suite (T) is composed of five test cases ($t_1 - t_5$) which cover all lines, branches, and functions of the source code. However, they are not able to trigger the faulty condition and therefore not helpful at debugging and localising the faulty source code.

Suppose we have received a bug report for the *Triangle* class, what could a human developer do to identify the faulty behaviour and localise the root cause of the failure? A typical approach to address a software bug is to first write a test case that is able to reveal the faulty behaviour, otherwise all statements of the source code would have to

		T								
		t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
	`public class Triangle { ...`									
	`int type(int a, int b, int c) {`									
c1	`int type = SCALENE;`	1	1	1	1	0	0	1	1	1
c2	`if ((a==b) && (b==c))`	1	1	1	1	0	0	1	1	1
c3	`type = EQUILATERAL;`	1	0	0	0	0	0	0	1	0
c4	`else if ((a*a) == ((b*b) + (c*c)))`	0	1	1	1	0	0	1	0	1
c5	`type = RIGHT;`	0	0	1	0	0	0	1	0	0
c6	`else if ((a==b) || (b==a)) /* FAULT */`	0	1	0	1	0	0	1	0	1
c7	`type = ISOSCELES;`	0	1	0	0	0	0	0	0	0
c8	`return type;`	1	1	1	1	0	0	1	1	1
	`}`									
	`double area(int a, int b, int c) {`									
c9	`double s = (a+b+c)/2.0;`	0	0	0	0	1	1	1	1	1
c10	`return Math.sqrt(s*(s-a)*(s-b)*(s-c));`	0	0	0	0	1	1	1	1	1
	`}`									
	`}`									
	Test case outcome (pass “P”, fail “F”)	P	P	P	P	P	P	F	P	F

Figure 5.1: Triangle class adapted from [221] with tests and coverage matrix; *type* classifies triangles based on the side lengths, and *area* calculates the area of the triangle. Automatically generated test suite T is not able to trigger the fault and therefore all statements would have to be manually inspected in order to find the faulty one. However, by augmenting T with four additional test cases t_6 t_7, t_8, t_9, the faulty behaviour is detected and root cause of the failure can be automatically localised.

be manually inspected. Then, to find the location of the faulty code, a developer would need to manually debug the source code, which could be very tedious as any combination of statements covered by a triggering test is in theory faulty, or he/she could use an automated fault localisation technique such as Spectrum-Based Fault Localisation (SBFL) [222] (one the most popular automated approaches to assist developers in debugging).

5.2.1 *Spectrum-Based Fault Localisation (SBFL)*

SBFL is a popular automated approach to assist developers in debugging [223–228]. It takes as input the code coverage information of a given test suite, and produces a list of components (typically statements) ranked in order of fault suspiciousness. Although the technique still has important known limitations [226], research in SBFL has continuously advanced over the past few years and many of its notorious initial limitations are no longer a problem. For instance, 1)

it can identify multiple faults [229–232]; 2) it can aggregate faults scattered across the code [229–232]; and 3) it can quantify confidence of the diagnosis [227, 233].

Although many techniques have been proposed for automating the process of locating the root-cause of observed failures [222], it has been shown that Spectrum-based reasoning can achieve better diagnostic results (either on single-faults and multiple-faults) than other spectrum-based approaches [229]. Spectrum-based reasoning is an approach to fault localisation founded on probability theory which uses an abstraction of the software under test to generate a diagnostic. The main principles underlying the technique are based on Model-Based Diagnosis (MBD) [227, 230, 234–237], which uses *logical reasoning* to find faults.

In this chapter, we consider a component to be a program statement, without loss of generality. A *fault candidate* is a set of statements that together explain a fault. Let the symbol C denote the set of source code statements, the symbol D denote a set of *fault candidates* d each consisting of a set of one or more statements that together explain a fault. The set $D = \langle\{c_1, c_2, c_3\}\rangle$ indicates that statements c_1, c_2, and c_3 are *simultaneously* at fault, and no other. On the other hand, $D = \langle\{c_1\}, \{c_2\}, \{c_3\}\rangle$ means that either c_1, c_2, or c_3 is at fault.

The following subsections explain the two phases that comprises spectrum-based reasoning: candidate generation and candidate ranking.

5.2.2 *Candidate Generation*

In theory, there are 2^M possible candidates that could be generated for a software under test with M components (i.e., $2^M = 2^{10} = 1024$ candidates for the toy example described in Figure 5.1). However, the generation of *all* candidates may be ineffective when applied to large and real software programs where the number of components (i.e., M) is usually high. In practice, not *all* candidates are valid: 1) each d is considered valid if and only if every failing test execution involves a component from d; 2) a candidate d is considered minimal if and only if there is not any other $d' \in D$ that is a subset of d.

The problem of finding the set of minimal candidates can be defined in terms of the widely-known Minimal Hitting Set (MHS) problem [234]. However, being a NP-hard problem, the precise computation of MHS is highly demanding [238]. Thus, the usage of exhaustive search algorithms may be prohibitive for real and large software programs. In practice, previous research has found that the precise computation of D is not necessary [239]. To solve the candidate generation problem in a reasonable amount of time, approaches that relax the minimality constraint have been proposed [239]. STACCATO [239] is a low-cost heuristic for computing a relevant set of multiple-fault

candidates. As all test cases in T pass (see Figure 5.1) and a bug has been reported, STACCATO yields a *theoretical* baseline diagnostic report containing all statements in the program, i.e., $D = \langle\{c_1\},\{c_2\},\{c_3\},\{c_4\},$ $\{c_5\},\{c_6\},\{c_7\},\{c_8\},\{c_9\},\{c_{10}\}\rangle$. If on the other hand, we consider a test suite T augmented with test cases t_6, t_7, t_8, t_9 (note that are two failing test cases: t_7 and t_9), STACCATO would provide a more accurate list of likely faulty candidates, i.e., $D' = \langle\{c_1\},\{c_2\},\{c_4\},\{c_4\},\{c_6\},\{c_8\},$ $\{c_9\},\{c_{10}\}\rangle$. Note that components c_3, c_5, and c_7 are not considered valid candidates because they have not been covered by both failing test cases. Figure 5.2 shows a subset of all (valid and not valid) candidates that could be generated for a test suite T augmented with test cases t_6, t_7, t_8, t_9.

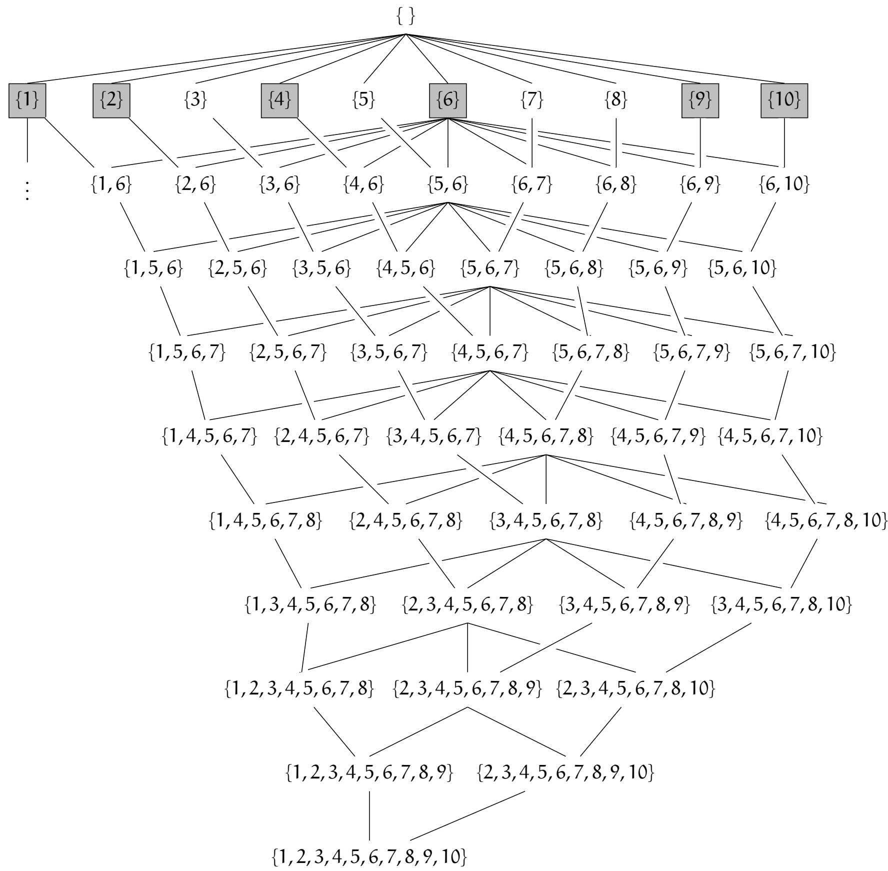


Figure 5.2: Subset of all candidates for the toy example described in Figure 5.1. The candidates in a grey box are the ones considered as valid minimal candidates for the test suite T augmented with test cases t_6, t_7, t_8, t_9.

5.2.3 *Candidate Ranking*

The candidate generation phase may result in an extensive list of diagnosis candidates. However, as not all candidates have the same prob-

ability of being the true fault explanation, techniques have been devised to assign a probability to each diagnosis candidate d, so that candidates more likely to be faulty could be inspected first.

The probability diagnosis of a candidate, $Pr(d|obs)$, is computed assuming conditional independence of all components. An observation obs_i (i.e., test case) is a tuple composed of its coverage (a_i, a column in Figure 5.1) and its outcome (e_i, pass or fail). Thus, the probability of each candidate is calculated according to Bayes rules as

$$Pr(d|obs) = Pr(d) \cdot \prod_{i \in 1..|obs|} \frac{Pr(obs_i|d)}{Pr(obs_i)}$$

where $Pr(obs_i)$ represents the probability of the observed outcome, independently of which diagnostic explanation is the correct one. The value of $Pr(obs_i)$ is a normalising factor given by

$$Pr(obs_i) = \sum_{d \in D} Pr(obs_i|d) \cdot Pr(d)$$

$Pr(d)$ estimates the probability of a candidate d being the true explanation of the faulty behaviour. Assuming that any component fails independently, the probability of a candidate d can be defined as

$$Pr(d) = \prod_{j \in d} p_j \cdot \prod_{j \in M \setminus d} (1 - p_j)$$

where p_j is the *priori* probability of a component being at fault, typically $1/1000 = 0.001$, i.e., 1 fault for every 1000 Lines of Code (LOC) [240]. $Pr(obs_i|d)$ represents the conditional probability of the observed outcome e_i produced by a test t_i (i.e., obs_i), assuming that candidate d is the actual diagnosis

$$Pr(obs_i|d) = \begin{cases} \prod_{j \in d \wedge a_{ij}=1} h_j & \text{if } e_i = 0 \\ 1 - \prod_{j \in d \wedge a_{ij}=1} h_j & \text{if } otherwise \end{cases}$$

where, a_{ij} represents the coverage of the statement j when the test i is executed. As the real values for h_j are typically not available, the values for $h_j \in [0, 1]$ are estimated by maximising $Pr(obs_i|d)$ using maximum likelihood estimation [229]. To solve the maximisation problem, a simple gradient ascent procedure [241] (bounded within the domain $0 < h_j < 1$) is applied.

As there is not any triggering test case in T (see Figure 5.1) and a bug has been reported, it is reasonable to consider that any statement is not more or less likely to be faulty than any other. Therefore, each $d \in D$ would have a probability of being faulty of $Pr(d|obs) = 1/10 = 0.1$ (assuming faults are uniformly distributed). However, if

we consider the test suite T augmented with test cases t_6, t_7, t_8, t_9 (and therefore the set of candidates previously explained, $D' = \langle\{c_1\}, \{c_2\}, \{c_4\}, \{c_6\}, \{c_8\}, \{c_9\}, \{c_{10}\}\rangle$), the procedure described above would be applied as follows[1]

$$\Pr(d'_1|obs) = \overbrace{\frac{1}{1000}}^{\Pr(d'_1)} \cdot \overbrace{\underbrace{h_1}_{t_1} \times \underbrace{h_1}_{t_2} \times \underbrace{h_1}_{t_3} \times \underbrace{h_1}_{t_4} \times \underbrace{(1-h_1)}_{t_7} \times \underbrace{h_1}_{t_8} \times \underbrace{(1-h_1)}_{t_9}}^{\Pr(obs|d'_1)}$$

...

$$\Pr(d'_4|obs) = \overbrace{\frac{1}{1000}}^{\Pr(d'_4)} \cdot \overbrace{\underbrace{h_6}_{t_2} \times \underbrace{h_6}_{t_4} \times \underbrace{(1-h_6)}_{t_7} \times \underbrace{(1-h_6)}_{t_9}}^{\Pr(obs|d'_4)}$$

...

$$\Pr(d'_7|obs) = \overbrace{\frac{1}{1000}}^{\Pr(d'_7)} \cdot \overbrace{\underbrace{h_{10}}_{t_5} \times \underbrace{h_{10}}_{t_6} \times \underbrace{(1-h_{10})}_{t_7} \times \underbrace{h_{10}}_{t_8} \times \underbrace{(1-h_{10})}_{t_9}}^{\Pr(obs|d'_7)}$$

By performing a maximum likelihood estimation, the value for each $\Pr(d'|obs)$ is as follows

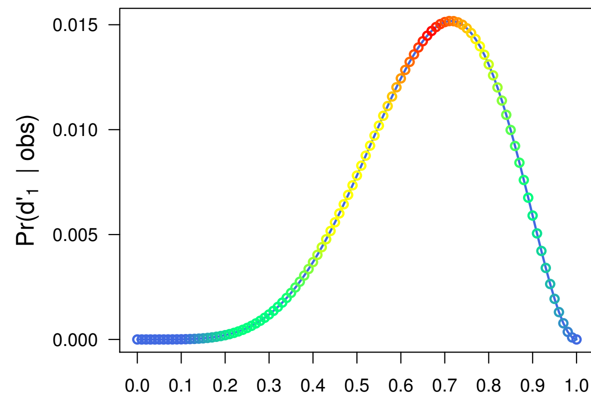
$$\Pr(d'_1|obs) = 0.04347826 \quad (h_1 = 0.01517357, \text{ see Figure 5.3a})$$
$$\Pr(d'_2|obs) = 0.04347826 \quad (h_2 = 0.01517357)$$
$$\Pr(d'_3|obs) = 0.17391300 \quad (h_4 = 0.03456000)$$
$$\Pr(d'_4|obs) = 0.34782610 \quad (h_6 = 0.06250000, \text{ see Figure 5.3b})$$
$$\Pr(d'_5|obs) = 0.04347826 \quad (h_8 = 0.01517357)$$
$$\Pr(d'_6|obs) = 0.17391300 \quad (h_9 = 0.03456000)$$
$$\Pr(d'_7|obs) = 0.17391300 \quad (h_{10} = 0.03456000, \text{ see Figure 5.3c})$$

After computing the probabilities for each $d \in D$, the candidates are ranked and shown to the user in descending order of probability to the true fault explanation.

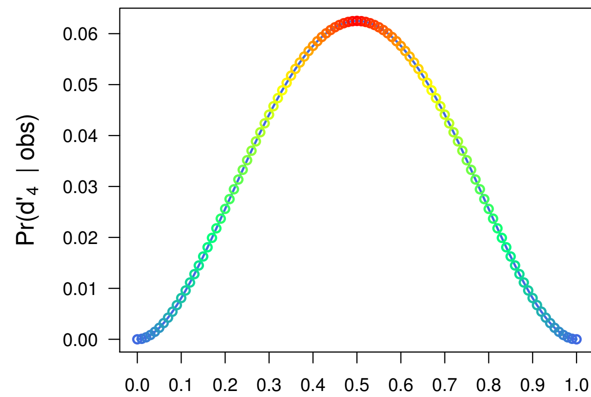
5.3 ENTROPY AS A NON-FUNCTIONAL CRITERION FOR AUTOMATED TEST GENERATION

Given a diagnostic report D, ranked by the probability of each candidate being the true fault explanation, the uncertainty in the ranking can be quantified using *entropy*, $\mathcal{H}(D)$ [219, 220] — a measure of uncertainty in a random variable [242]. For instance, the value of $\mathcal{H}(D)$ is maximum for the *theoretical* baseline diagnostic report generated

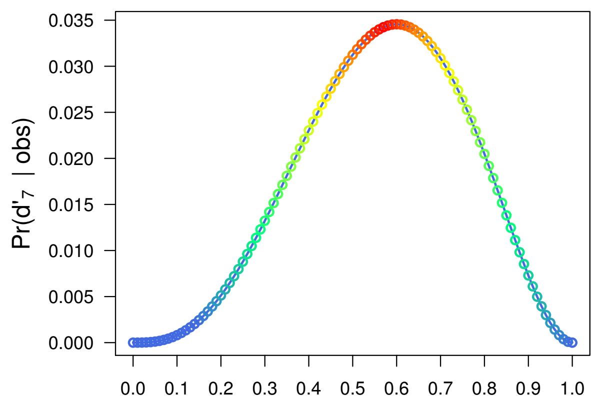
1 Although used, $\Pr(obs_i)$ factor has been omitted as it is identical for all $d' \in D'$.



(a) $\Pr(d'_1|\text{obs})$



(b) $\Pr(d'_4|\text{obs})$



(c) $\Pr(d'_7|\text{obs})$

Figure 5.3: Optimisation of h_j values.

in the previous section, $D = \{0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1\}$, because all elements in the set share the same probability of being faulty and therefore they cannot be distinguished from one another. The minimum ideal value for $\mathcal{H}$ is zero, in which case all elements

in the set can be distinguished from one another. As there are in theory 2^M different ways of representing the whole set C, the theoretical maximum value of entropy is $\log_2(M)$, where M is the number of statements. So, $\mathcal{H}(D)$ can be defined as

$$\mathcal{H}(D) = -\sum_{d_k \in D} \Pr(d_k) \cdot \log_2(\Pr(d_k)), \quad 0 \leqslant \mathcal{H} \leqslant \log_2(M)$$

thus, the entropy value of the test suite T from Figure 5.1 is

$$\mathcal{H}(D) = -10 \times \Big(0.1 \cdot \log_2(0.1)\Big) = 3.322$$

which corresponds to the maximum value. This means that the ranking suffers considerably from uncertainty, and we cannot distinguish which of the statements in the example with probability 0.1 of being faulty can explain the fault better. Therefore, in order to reveal the reported fault and to pinpoint the exact location of the root cause of the failure, entropy in the ranking must be reduced.

5.3.1 *Estimating Entropy: Coverage Density Fitness Function*

Search-based test generation algorithms as the ones described in Section 2.4 and evaluated in Chapter 4 are guided by a fitness function, which describes a desirable optimization goal. This is particularly useful when we can measure a property but have no immediate way to construct suitable test cases systematically. A fitness function takes a candidate solution (e.g., a test suite) as input, and maps it to a numeric value that estimates how close the solution is to the optimal solution. In theory, entropy ($\mathcal{H}(D)$) could be used as a fitness function. However, to generate the diagnostic report D a test suite with oracles (i.e., pass/fail verdicts) for all tests is required — this oracles are typically provided by human developers, and thus is not available during test generation. Therefore, we require a measure that could estimate entropy without the need of explicit test oracles.

The *coverage density*, which is the average percentage of statements covered by all test cases, is able to measure the relation between variety and size of a test suite, and it has been shown to be a reliable proxy for the entropy value [227, 243]. It is defined as follows

$$\bar{\rho}(T) = \frac{1}{N} \cdot \sum_{i=1}^{N} \rho(t_i), \quad 0 \leqslant \bar{\rho}(T) \leqslant 1$$

where $\rho(t_i)$ refers to the coverage density of a test case t_i

$$\rho(t_i) = \frac{|\{j \mid a_{ij} = 1 \wedge 1 \leqslant j \leqslant M\}|}{M}$$

where N and M denote the number of test cases and the number of statements, respectively. Low values of $\bar{\rho}$ mean that test cases exercise small parts of the program (sparse matrices), whereas high

values mean that test cases tend to involve most statements of the program (dense matrices). For example, the $\bar{\rho}$ value of test suite T from Figure 5.1 is 0.4. The coverage density fitness function can then be defined as follows

$$\text{fitness}(T) = |\beta - \bar{\rho}(T)|$$

where β is a value between 0.0 – 1.0. This fitness function turns the problem into a minimisation problem, i.e., the optimization aims to achieve a fitness value of zero, which is the case if a solution is found such that β is equal to $\bar{\rho}(T)$. However, what is the optimal value of $\bar{\rho}$ that could lead to a lower value of *entropy*?

A reduction in *entropy* is known as *information gain* [242], and it has been previously demonstrated [243] that the *information gain* of a test suite with uniformly distributed coverage[2] can be modelled as follows

$$IG(\bar{\rho}) = -\bar{\rho} \cdot \log_2(\bar{\rho}) - (1 - \bar{\rho}) \cdot \log_2(1 - \bar{\rho}), \quad 0 \leqslant IG(\bar{\rho}) \leqslant 1$$

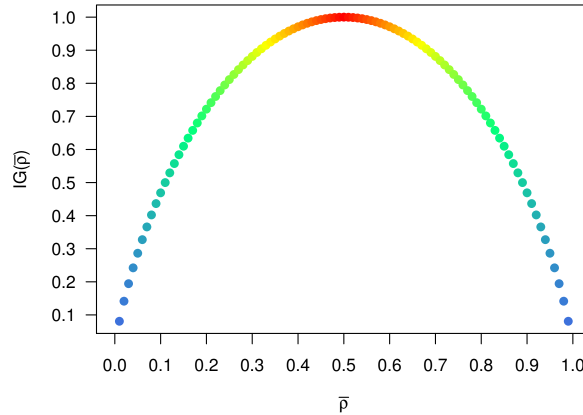


Figure 5.4: Relation between $\bar{\rho}$ and $IG(\bar{\rho})$.

As we can see in Figure 5.4, the value of $IG(\bar{\rho})$ is optimal for $\bar{\rho} = 0.5$. Hence, a technique that is able to generate a test suite such that the coverage density is $\bar{\rho} = 0.5$ (provided there is a variety of test cases) would have the capability of reducing the entropy of a diagnostic ranking, and consequently ability to diagnose a program under test. Indeed, the test suite T augmented with test cases t_6, t_7, t_8, t_9 ($\bar{\rho} = 0.5$ and $\mathcal{H}(D) = 2.437$) is able to trigger the faulty behaviour and to identify the component c_6 as the most likely to be at fault. In Section 5.4.2 we evaluate what is the ideal value of $\bar{\rho}$ for test suites with non-uniformly distributed coverage as the ideal value may be higher/lower for different systems.

2 Each test case in a test suite with uniformly distributed coverage covers a random sample of $\rho \times M$ statements, and each statement is randomly covered by $\rho \times N$ test cases. This type of test suite is commonly used as synthetic benchmark to, for example, demonstrate the best-case performance of regression testing algorithms [219, 244].

5.3.2 *Integrating Coverage Density in Evolutionary Algorithms*

Typically, the combination of functional and non-functional criteria in automatic test generation is considered as a multiple-objective problem. For instance, Ferrer et al. [144] optimised coverage and the oracle cost using multiple-objective algorithms such as NSGA-II [127] and SPEA2 [145]. Thus, the most naïve strategy to integrate a non-functional criterion such entropy would be to consider it as an additional objective to, for instance, the most common criterion, branch coverage. However, such integration raises two main problems: 1) By definition, multi-objective optimisation algorithms generate multiple solutions, i.e., some that only maximise code coverage, others that only minimise entropy, and others in between. Although we aim to optimise the entropy of a test suite, the fault could only be revealed if and only if the faulty code is covered. As entropy does not aim to cover every single line of code, it might happen that the solution that improves entropy the most does not explore the faulty code. 2) It has been reported that the combination of a functional criterion such coverage and non-functional criteria has a negative impact on the coverage achieved [144, 146]. If the combination of coverage and entropy has a detrimental effect on the overall coverage achieved, it could mean the faulty code is never exercised, and therefore the fault never revealed. For these reasons, multiple-objective optimisation algorithms are not suitable to address our problem, and therefore we do not consider them in our empirical evaluation.

Another strategy to optimise for functional and non-functional criteria is by using secondary objectives [113, 147]. As in a typical search-based test generation, the evolutionary algorithm is guided by a functional criterion (e.g., branch coverage), and during evolution the individuals with the highest coverage are selected. However, if two or more individuals of the population have the same coverage value, a secondary objective is used to break the tie. The default secondary objective of the EvoSuite test generation tool compares the *length* of two individuals with the same coverage and selects the shortest one, i.e., the one with the lowest number of statements. In this chapter we propose to replace this secondary objective with a combination of *length* and *entropy*. For a set of n non-conflicting individual secondary objectives $s_1 \dots s_n$, a combined secondary objective function sf_{comp} can be defined as

$$sf_{comp}(T) = \sum_{i=1}^{n} w_i \times s_i(T)$$

where $w_1 \dots w_n$ are weights assigned to each individual secondary objective which allow for prioritisation of the secondary objectives involved in the composition. In this chapter, we arbitrarily give the same importance to each secondary objective (i.e., $w_i = 1$) and leave

the question of what are optimal w_i values for future work. Therefore, the linear combination of *length* and *entropy* can be defined as

$$sf_{comp}(T) = \textsc{Length}(T) + |\beta - \bar{\rho}(T)| \tag{5.1}$$

The test suite T with the lowest value of sf_{comp} implies it is the shortest and the one with the lowest entropy value.

5.4 EMPIRICAL STUDY

We have conducted an empirical study to evaluate the extent to which the combination of *length* and *coverage density* as a secondary objective (see Equation 5.1) is capable of improving the diagnostic quality of automatically generated tests. In particular, our empirical study aims to answer the following research questions:

RQ1: Can optimisation of entropy improve the fault detection ability of automatically generated tests?

RQ2: Can optimisation of entropy improve the fault localisation ability of automatically generated tests?

RQ3: Does optimisation of entropy affect the coverage achieved or the number of automatically generated tests?

5.4.1 *Experimental Setup*

In our set of experiments we used the unit test generation tool EvoSuite [9] (see Section 2.6 for more details of the tool) which already supports *length* as a secondary objective. For this study, we added the combined secondary objective function (see Equation 5.1) to EvoSuite. We evaluated the fault detection and localisation effectiveness of MOSA (the most effective evolutionary algorithm for unit test generation evaluated in Chapter 4) using the default secondary objective and the proposed one on a set of six real faults.

A particular difficulty to address in our evaluation setup is the need to create test oracles: the test generation procedure needs to decide whether a test it generates passes or not. The automated generation of test oracles is challenging [20, 180, 184]. This has to do with the fact that the behaviour of the software has to be known so that the right oracles are added to the test cases. As the oracle problem is orthogonal to this chapter, we mitigated this problem by using two versions of each subject program (more details of each subject in Section 5.4.1.1). Let P be the faulty program and P' its fixed version, a test case t *passes* if $P'(t) == P(t)$; it *fails* otherwise. EvoSuite adds regression oracles to the tests which make it possible to do this comparison automatically.

Table 5.1: Details of real faults used in our experiments. For each fault we report the bug report id, the faulty CUT and its LOC. As for `Joda-Time` fault there was not a bug report id available, the hash of the commit that fixed the bug is reported.

SUT	Bug Id	CUT	LOC
Codec	99	Base64	233
Compress	114	TarUtils	61
Math	835	Fraction	182
Math	938	Line	44
Math	939	Covariance	53
Joda-Time	*"941f59"*	BasicDayOfYearDateTimeField	22

5.4.1.1 *Selection of Subject Programs*

The requirements for choosing the subject programs used in our evaluation are as follows: (1) the software programs should be developed in Java, (2) the fault must be documented, and (3) the fix should be available to validate if a test generation technique is able to trigger the faulty behaviour and therefore identify the exact place of the fault.

We selected six real faults from four large open-source libraries. For each program, we analysed recent bug reports, and selected those reports where the fix represents a change in only one statement (single-fault programs). We used the fixed version P' of a faulty version P to evaluate whether each test generation technique is able to: 1) generate a test case that reveals the faulty behaviour, and 2) pinpoint the exact location of the bug in the report, i.e., we checked if it effectively isolates the faulty statement on the top of the ranking. Note that the same programs have also been used in previous studies (e.g., [245]), but we used different faults to demonstrate that *coverage density* fitness function works regardless of whether the fault causes an undeclared exception or a wrong output. Table 5.1 provides details about our experimental subjects.

Apache Commons Codec #99

Apache Commons Codec [246] provides an API of common encoders and decoders such as Base64, Hex and URLs. As described in the *major bug* 99 [247], the method `encodeBase64String` of the class `Base64` fails because it chunks the parameter `binaryData`. This means that the second parameter of the method `newStringUtf8` called on method `encodeBase64String` should be `false` and not `true`.

Listing 5.1: Apache Commons Codec fix for bug 99.

```
--- org/apache/commons/codec/binary/Base64.java
@@ f7966c1..954d995 @@
 public static String encodeBase64String(byte[] binaryData) {
```

```
-  return StringUtils.newStringUtf8(encodeBase64(binaryData,
    true));
+  return StringUtils.newStringUtf8(encodeBase64(binaryData,
    false));
 }
```

Apache Commons Compress #114

The Apache Commons Compress [248] library defines an API for working with the most popular compressed archives such as ar, cpio, Unix dump, tar, zip, gzip, XZ, Pack200 and bzip2. The reported *major bug* 114 [249] explains that the project Apache Commons Compress fails when the class `TarUtils` receive as input a *tarfile* which contains files with special characters. A simple fix to resolve the encoding problem is to guarantee that the name of the files are treated as unsigned.

Listing 5.2: Apache Commons Compress fix for bug 114.

```
--- org/apache/commons/compress/archivers/tar/TarUtils.java
@@ 2419bb5..2d858d5 @@
 for (int i = offset; i < end; ++i) {
-  if (buffer[i] == 0) { // Trailing null
+  byte b = buffer[i];
+  if (b == 0) { // Trailing null
     break;
   }
-
-  result.append((char) buffer[i]);
+  result.append((char) (b & 0xFF)); // Allow for sign-extension
```

Apache Commons Math #835

The Apache Commons Math [250] is a library that provides self-contained mathematics and statistics functions for Java. The bug 835 [251] reports a failure when the `percentageValue()` method of the `Fraction` class multiplies a fraction value by 100, and then converts the result to a double. This causes an overflow when the numerator is greater than `Integer.MAX_VALUE/100`, and even when the value of a fraction is far below this value. A change in the order of multiplication, i.e., first convert a fraction value to a double and then multiply that value by 100, resolved the overflow problem.

Listing 5.3: Apache Commons Math fix for bug 835.

```
--- org/apache/commons/math3/fraction/Fraction.java
@@ a49e44..63a487 @@
 public double percentageValue() {
-  return multiply(100).doubleValue();
+  return 100 * doubleValue();
 }
```

Apache Commons Math #938

The *major bug* 938 [252] explains that the method `revert` from the class `Line` only maintains 10 digits of precision for the field direction. This becomes a bug when the line's position is evaluated far from the origin. A possible fix is creating a new instance of `Line` and then reverting its direction.

Listing 5.4: Apache Commons Math fix for bug 938.

```
--- org/apache/commons/math3/geometry/euclidean/threed/Line.java
@@ 43a6f1..736055 @@
 public Line revert() {
-  return new Line(zero, zero.subtract(direction));
+  final Line reverted = new Line(this);
+  reverted.direction = reverted.direction.negate();
+  return reverted;
 }
```

Apache Commons Math #939

The specification of the class `Covariance` states that it only takes a single-column matrix (i.e., N-dimensional random variable with N=1) as argument and returns a 1-by-1 covariance matrix. However, the method `checkSufficientData` throws an `IllegalArgumentException` (see *major bug* 939 [253] for detailed information) when the constructor of the class receives a 1-by-M matrix.

Listing 5.5: Apache Commons Math fix for bug 939.

```
--- org/apache/commons/math3/stat/correlation/Covariance.java
@@ 736055..49444e @@
 private void checkSufficientData(final RealMatrix matrix)
     throws MathIllegalArgumentException {
   int nRows = matrix.getRowDimension();
   int nCols = matrix.getColumnDimension();
-  if (nRows < 2 || nCols < 2) {
+  if (nRows < 2 || nCols < 1) {
     throw new MathIllegalArgumentException(
      LocalizedFormats.INSUFFICIENT_ROWS_AND_COLUMNS,
        nRows, nCols);
```

Joda-Time

Joda-Time [254] is a library for advanced *date* and *time* functionalities for the Java language. The class `BasicDayOfYearDateTimeField` provides methods to perform time calculations for a day of a year. Joda-Time bug [255] was related to the method `getMaximumValueForSet`, which returns an incorrect value. The fix of this bug consists of vali-

dating if the value of the variable `value` is between the maximum and the minimum value of the range or not.

Listing 5.6: Joda-Time fix.

```
--- org/joda/time/chrono/BasicDayOfYearDateTimeField.java
@@ a0c65a..941f59 @@
 protected int getMaximumValueForSet(long instant, int value) {
   int maxLessOne = iChronology.getDaysInYearMax() - 1;
-  return value > maxLessOne ? getMaximumValue(instant) :
    maxLessOne;
+  return (value > maxLessOne || value < 1) ?
    getMaximumValue(instant) : maxLessOne;
 }
```

5.4.1.2 *Experiment Procedure*

In order to answer our research questions we performed two studies to assess the performance of two different configurations of MOSA at (i) detecting faults and (ii) localising a fault. In summary, we evaluated the following configurations of MOSA:

– MOSA: The default configuration of the evolutionary algorithm MOSA described in Section 2.4.4.6, i.e., branch coverage as the main fitness function and test suite length as a secondary objective.

– MOSA$_{\bar{\rho}}$: The evolutionary algorithm MOSA described in Section 2.4.4.6 using branch coverage as the main fitness function, and a combination of test suite length and coverage density as a secondary objective, as described in Section 5.3.2.

Our first study is composed of two experiments. First, we conducted a preliminary experiment to select the best value of β in Equation 5.1. I.e., we ran configuration MOSA$_{\bar{\rho}}$ with $\beta = \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$ and performed a pairwise comparison of the number of real faults detected by using any β value. The β value that allowed a technique to significantly detect more faults more often was selected as the best value. In case of a tie, the number of generated test cases is used. Then, and to answer RQ1, in order to access whether the optimisation of entropy does lead to an improvement on the fault detection ability of automatically generated test cases, we performed a comparison between MOSA and MOSA$_{\bar{\rho}}$ (using the best overall value of β found in the first experiment) for each fault. A configuration A is considered more effective at detecting fault X than configuration B if it detects X in a statistically significantly higher number of runs than B.

To answer RQ2, in order to measure the success of a configuration at localising the faulty code we used the SBFL technique described

in Section 5.2.1. Then, we used a commonly used [227, 229] metric called *diagnostic quality* C_d to measure the human effort of inspecting the ranking produced by SBFL. This metric is independent of the *number of faults* M_f in the program in order enable an unbiased evaluation of the effect of M_f on C_d. As multiple explanations can be assigned with the same probability, the value of C_d for the real fault d_* is the average of the ranks that have the same probability:

$$\theta = |\{d_k|Pr(d_k) > Pr(d_*)\}|,\ 1 \leqslant k \leqslant M$$
$$\phi = |\{d_k|Pr(d_k) \geqslant Pr(d_*)\}|,\ 1 \leqslant k \leqslant M$$
$$C_d = \frac{\theta + \phi - M_f}{2}$$

A value of zero for C_d indicates an ideal diagnostic report where all M_f faulty statements appear on top of the ranking, i.e., there is no wasted effort in inspecting other statements. For example, suppose the following ranking generated by SBFL for a program with 3 statements in which only the second statement (c_2) is faulty:

$$c_2 = 0.85$$
$$c_1 = 0.10$$
$$c_3 = 0.05$$
$$C_d = \frac{0+1-1}{2} = 0$$

On the other hand, $C_d = M - M_f$ indicates that the user needs to inspect all $M - M_f$ healthy statements until reaching the M_f faulty ones — this is the worst-case outcome:

$$c_1 = 0.85$$
$$c_3 = 0.10$$
$$c_2 = 0.05$$
$$C_d = \frac{2+3-1}{2} = 2$$

To answer RQ3, we performed a statistical analysis to understand the effects of an additional criterion on the coverage achieved and on the number of automatically generated test cases.

For all experiments we used the same search budget used by a previous study on real faults [256, 257], 3 minutes. To account for the randomness of the test generation, we repeated all experiments 30 times (as suggested by Arcuri et al. [202] and Rice [258] and general advisable by previous studies on automatic test generation [113]) to take the randomness of the search-based algorithms into account. All experiments were executed on the University of Sheffield ShARC HPC Cluster [208].

5.4.1.3 *Experiment Analysis*

All data produced have been analysed following the guidelines described by Arcuri et al. [202]. In particular, we have used the Wilcoxon-Mann-Whitney U-test, the Vargha-Delaney $\hat{A}_{12}$ effect size [203], and Fisher's exact test [259]. We used the Wilcoxon-Mann-Whitney U-test to compare two different data sets, and the Vargha-Delaney $\hat{A}_{12}$ effect size to measure the probability of configuration a achieving better values than configuration b. In order to determine whether one configuration a was statistically significantly more successful than another configuration b we used Fisher's exact test. For both Wilcoxon-Mann-Whitney U-test and Fisher's exact statistical test, we consider a 95% confidence level. To also provide more information on how much better one configuration is than other, we also report the *relative improvement*. Assuming X the data set reported by configuration a, and data set Y reported by configuration b, relative improvement can be defined as $\mathrm{rel.\ impr.} = \frac{\mathrm{mean}(X) - \mathrm{mean}(Y)}{\mathrm{mean}(Y)}$. Furthermore, we also consider the standard deviation σ and confidence intervals of averaged values using bootstrapping at 95% significance level.

5.4.1.4 *Threats to Validity*

Construct Validity: The fault localisation effectiveness of each test generation configuration has been evaluated using the C_d metric, which measures diagnostic effort in terms of the position of the fault in the diagnostic report. This metric assumes that developers traverse the ranking, but that may not be the case in practice [226]. However, we argue that developers are more likely to traverse the ranking if the precision is increased.

External Validity: Although they are real and widely developed open source subjects, we have only considered five in our empirical study, all of which are libraries. Therefore, it is possible that the results for a different set of subjects with different characteristics and even with multiple-faults may produce different results.

Internal Validity: Eventual faults in the implementation of each test generation technique or in the underlying test case generation EvoSuite may invalidate the results. To mitigate this threat, we have not only thoroughly tested our scripts but also manually checked a large set of results. Furthermore, all experiments were repeated multiple times to account for the randomness of the test generation, and we verified the results between runs for consistency.

5.4.2 *Coverage Density Tuning*

As we saw in Chapter 4, the performance of each evolutionary algorithm heavily relies on the problem at hand and on several parameters such as, for example, size of the algorithm's population. In this

Table 5.2: The most effective configuration at detecting each fault. Column $\bar{\rho}$ reports the range and the average value in which configuration X performed statistically significantly better at detecting fault Y, Cov. the branch coverage of the generated test suite, #T the total number of generated test cases, T_c the number of test cases that cover the faulty statement, T_{f-c} the number of test cases that trigger the fault and cover the faulty statement.

Conf.	$\bar{\rho}$	Cov.	#T	T_{p-dc}	T_{p-c}	T_{f-c}
			Codec #99			
$MOSA_{\bar{\rho}}$	0.30-0.40 (0.37)	0.97	46	42	2	2
			Compress #114			
$MOSA_{\bar{\rho}}$	—	—	—	—	—	—
			Math #835			
$MOSA_{\bar{\rho}}$	0.30-0.40 (0.31)	0.98	50	38	10	1
			Math #938			
$MOSA_{\bar{\rho}}$	0.60-0.70 (0.61)	1.00	9	5	1	3
			Math #939			
$MOSA_{\bar{\rho}}$	0.10-0.20 (0.18)	0.96	12	10	2	1
			Joda-Time			
$MOSA_{\bar{\rho}}$	—	—	—	—	—	—

Table 5.3: The most effective configuration at detecting faults overall. Please refer to Table 5.2 for an explanation of each column.

$\bar{\rho}$	Cov.	#T	T_{p-dc}	T_{p-c}	T_{f-c}	Better than	#F
			$MOSA_{\bar{\rho}}$				
0.30-0.40 (0.33)	0.97	29	23	5	1	34 / 54	4 / 6
0.20-0.30 (0.26)	0.97	31	23	6	1	29 / 54	2 / 6
0.10-0.20 (0.18)	0.96	12	10	2	1	13 / 54	1 / 6

chapter, there is another parameter that requires an additional study, the β of Equation 5.1. As discussed in Section 5.3.1, the optimal coverage density of test suites with an uniformly distributed coverage is 0.5 (i.e., $\beta = \bar{\rho} = 0.5$). However, the coverage of automatically generated tests is not uniformly distributed by nature, i.e., some tests could cover more code than others.

Table 5.2 shows that, the value of β for which $MOSA_{\bar{\rho}}$ is able to detect a fault on a statistically significantly higher number of runs is different for almost all faults. For example, $MOSA_{\bar{\rho}}$ works statistically significantly better at detecting fault `Math #938` with a β value of 0.61, and fault `Math #939` with 0.18. However, there are some faults detected by $MOSA_{\bar{\rho}}$ on a statistically significantly higher number of runs with the same range values of β, e.g., `Codec #99`, `Math #835`.

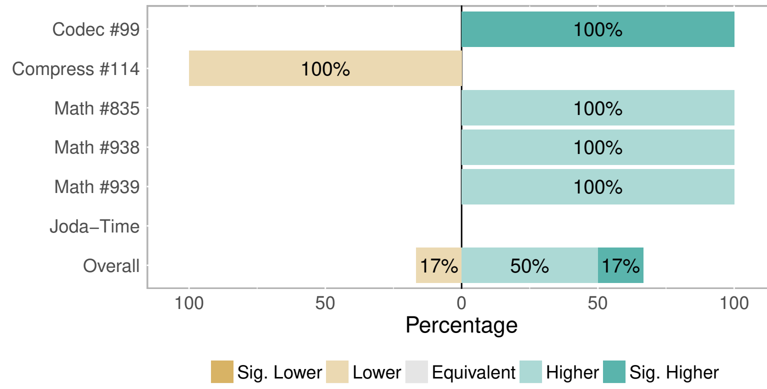


Figure 5.5: Fault detection effectiveness of $\text{MOSA}_{\bar{\rho}}$ when compared to MOSA. "Significantly higher" is the number of faults for which a technique generated at least a test case that detected the fault in a statistically significantly higher number of runs than the other; "Higher" refers to the number of faults where a technique generated at least one test case that detected the fault in a (non-statistically significantly) higher number of runs; "Equivalent" is where the fault was detected by both techniques for the same number of runs.

Note that, despite different range values of β used in our experiments, $\text{MOSA}_{\bar{\rho}}$ was not able to detect faults `Compress #114` and `Joda-Time`. The reason is that a failure could only be revealed if and only if the faulty code is covered and executed with the input that triggers it. For instance, the percentage of generated test cases that covered the faulty statement of fault `Compress #114` was only 7% of all test cases, whereas the average in Table 5.2 is 25%, minimum of 8.7% for `Codec #99`. That is, when there are only a few test cases covering the faulty code, it is less likely to find at least one that triggers the fault.

Table 5.3 summarises the overall results of $\text{MOSA}_{\bar{\rho}}$. We performed a pairwise tournament across all faults and all range values of β, a total of 54 tournaments (i.e., 9 range values of $\beta \times 6$ faults). If a particular value of β performed statistically significantly better at localising each fault than another value of β it gets a +1 point, if performed statistically significantly worse it gets -1. In the end, the β value with the highest number of points is selected as the most effective one.

> $\text{MOSA}_{\bar{\rho}}$ *works statistically significantly better at detecting faults with a* β *value between 0.30 and 0.40 (0.33, on average).*

5.4.3 *RQ1 – Can optimisation of entropy improve the fault detection ability of automatically generated tests?*

Figure 5.5 reports the fault detection effectiveness of $\text{MOSA}_{\bar{\rho}}$ vs. MOSA. $\text{MOSA}_{\bar{\rho}}$ performed statistically significantly better than MOSA for `Codec #99`, and performed better for three other faults (all

`Math` faults). The reason for such improvement of $\text{MOSA}_{\bar{\rho}}$ over MOSA can be explained by the number of generated test cases that cover the faulty code. Table 5.4 shows that for all faults, $\text{MOSA}_{\bar{\rho}}$ generated more test cases covering the faulty code, and therefore is more likely to find at least one that triggers the fault. E.g., for `Codec #99`, 4.4% of all test cases generated by MOSA covered the faulty code, whereas 8.4% of all test cases generated by $\text{MOSA}_{\bar{\rho}}$ exercised the faulty code, i.e., almost twice the number of test cases generated by MOSA.

For `Compress #114`, $\text{MOSA}_{\bar{\rho}}$ failed to generate a test case able to trigger the faulty behaviour, and MOSA generated *by chance* one triggering test case (see Listing 5.7) on a single repetition out of 30. Therefore, MOSA was better (however not statistically significantly) than $\text{MOSA}_{\bar{\rho}}$.

Listing 5.7: Triggering test case generated by MOSA for Compress #114 fault.

```java
@Test
public void test09() throws Throwable {
  byte[] byteArray0 = new byte[3];
  byteArray0[0] = (byte) (-20);
  String string0 = TarUtils.parseName(byteArray0, 0, 1691);
  assertEquals("\u00EC", string0);
}
```

> *RQ1:* $\text{MOSA}_{\bar{\rho}}$ *performs statistically significantly better at detecting 1 fault and better at detecting 3 out of 6 faults than* MOSA.

5.4.4 *RQ2 – Can optimisation of entropy improve the fault localisation ability of automatically generated tests?*

Ideally, a single trigger test case covering a single statement would be enough to pinpoint the exact location of a fault, i.e., $C_d = 0$. However, on complex classes under test such as the ones used in our experiments, it is almost impossible to cover a single statement without executing a few other statements. A test suite with only trigger test cases will not help at localising the faulty statement either, as all covered statements would be considered faulty. Non-triggering test cases, either the ones that cover or the ones that do not cover the faulty statement, could also influence the fault localisation effectiveness of a test suite. For instance, non-triggering test cases that do not cover the faulty statement could positively improve the effectiveness of a test suite, as the likelihood of the non-faulty statements being faulty would decrease. On the other hand, non-triggering test cases that cover the faulty statement would reduce the contribution of triggering test cases to pinpoint the faulty statement. Thus, an ideal test suite, i.e., a suite that is able to pinpoint the exact location of a fault,

Table 5.4: Fault localisation effectiveness of each configuration per fault. Column Cov. report the branch coverage of the generated test suite, #T the number of generated test cases, $\mathcal{H}$ the entropy value of the diagnose ranking produced by #T, C_d the number of statement that require inspection in order to find the true faulty one, $\hat{A}_{12}$ and p the probability of $MOSA_{\bar{\rho}}$ achieving a lower C_d value than MOSA, and its *p-value* respectively, and Rel. Impr. which reports how much better/worse $MOSA_{\bar{\rho}}$ is than MOSA. σ and CI columns report the standard deviation and confidence intervals (at 95% significance level), respectively, of the averaged C_d value. The "Overall" results only consider faults detected by all configurations.

Conf.	Cov.	#T	T_{p-dc}	T_{p-c}	T_{f-c}	$\mathcal{H}$	C_d	σ	CI	$\hat{A}_{12}$	p	Rel. Impr.
Codec #99												
MOSA	0.98	49.2	95.5%	2.4%	2.0%	1.9	0.60	1.34	[-0.60,1.20]	—	—	—
$MOSA_{\bar{\rho}}$	0.97	45.7	91.6%	4.5%	3.9%	1.5	0.27	0.93	[-0.06,0.49]	0.47	0.70	-55.4%
Compress #114												
MOSA	1.00	16.0	93.8%	0.0%	6.2%	2.1	0.00	0.00	[0.00,0.00]	—	—	—
$MOSA_{\bar{\rho}}$	—	—	—	—	—	—	—	—	—	—	—	—
Math #835												
MOSA	0.98	53.9	81.7%	15.3%	3.0%	2.7	7.12	10.50	[-0.06,12.94]	—	—	—
$MOSA_{\bar{\rho}}$	0.98	50.0	76.9%	20.8%	2.3%	3.0	10.81	15.23	[1.85,17.08]	0.62	0.36	+51.7%
Math #938												
MOSA	1.00	10.4	61.4%	27.7%	10.8%	3.3	5.56	6.74	[0.62,9.31]	—	—	—
$MOSA_{\bar{\rho}}$	1.00	12.0	58.3%	25.0%	16.7%	2.6	0.00	0.00	[0.00,0.00]	0.06	0.09	-100.0%
Math #939												
MOSA	0.96	10.0	63.7%	25.8%	10.5%	2.7	4.79	0.82	[4.45,5.21]	—	—	—
$MOSA_{\bar{\rho}}$	0.96	9.2	55.5%	33.6%	10.8%	2.7	5.00	0.00	[5.00,5.00]	0.55	0.10	+4.4%
Joda-Time												
MOSA	—	—	—	—	—	—	—	—	—	—	—	—
$MOSA_{\bar{\rho}}$	—	—	—	—	—	—	—	—	—	—	—	—
Overall												
MOSA	0.98	30.9	84.0%	12.1%	3.9%	2.7	4.52	4.85	[1.10,7.16]	—	—	—
$MOSA_{\bar{\rho}}$	0.98	29.2	79.0%	15.8%	5.1%	2.5	4.02	4.04	[1.70,5.64]	0.43	0.31	-24.8%

would have: (i) at least one triggering test case, (ii) ideally zero non-triggering test cases covering the faulty code, and (iii) as many as possible non-triggering test cases that cover non-faulty code to exonerate statements covered by triggering test cases of being faulty.

As we can see in Table 5.4, the test suite generated by MOSA for `Compress #144` exhibits these three properties. One test case triggers the fault, zero non-triggering test case covering the faulty code, and all the remaining test cases cover non-faulty code. Therefore, MOSA achieved a minimal C_d value of zero. However, meeting such properties or achieving a minimal C_d value is not always possible due to, for example, the structure of the program under test. For example, the `checkSufficientData` function of fault `Math #939` is a private function, and it is only executed by the constructor of the class `Covariance`. Therefore, at the top of the ranking will always be the statements of the `Covariance` constructor and the `checkSufficientData` function.

Table 5.4 also shows that the most effective configuration at ranking faulty statements is the one with the largest sample of triggering test cases. For example, $MOSA_{\bar{\rho}}$ performed better than MOSA for `Math #938` as 16.7% vs 10.8% of all test cases trigger the fault, respectively. Hence, the number of statements a developer would have to investigate in order to find the true faulty one that explains fault `Math #938` is 5.56 with tests generated by MOSA, and 0.00 with tests generated by $MOSA_{\bar{\rho}}$ (i.e., the faulty statement has the highest probability of being faulty and therefore, no effort is required to actually find it). On the other hand, $MOSA_{\bar{\rho}}$ performed worse than MOSA for `Math #835` as 2.3% vs 3.0% of all test cases trigger the fault, respectively. Hence, the number of statements would require inspection is 7.12 with MOSA, and 10.81 with $MOSA_{\bar{\rho}}$ (a relative improvement of +51.7%). Overall, a developer would have to inspect -24.8% statements with tests generated by $MOSA_{\bar{\rho}}$ than with tests generated by MOSA.

It is also worth mentioning that, low values of *entropy* lead to less statements that must be inspected by a human developer, as motivated in earlier sections. For `Math #938`, MOSA achieved a $\mathcal{H}$ value of 3.3 and a C_d value of 5.56. On the other hand, $MOSA_{\bar{\rho}}$ achieved a lower value of $\mathcal{H}$ (2.6), and therefore a lower value of C_d (0.0).

RQ2: $MOSA_{\bar{\rho}}$ *is 24.8% more effective at localising the root case of a failure (however with an effect size of 0.43) than* MOSA.

5.4.5 *RQ3 – Does optimisation of entropy affect the coverage achieved or the number of automatically generated tests?*

One of the main concerns when combining functional and non-functional criteria on automatic test generation is the effect on the number of test cases. Table 5.5 reports the coverage achieved and the number of generated test cases by configurations MOSA, and $MOSA_{\bar{\rho}}$.

As we can see, the coverage achieved by both configurations is exactly the same for all faults but `Codec #99`, for which $MOSA_{\bar{\rho}}$ achieved -1.3% coverage than MOSA. Although coverage was slightly lower for this fault, $MOSA_{\bar{\rho}}$ performed statistically significantly better than MOSA at detecting and localising it (see RQ1 and RQ2, respectively). Thus, the reduction in coverage did not influence the fault detection or localisation ability of $MOSA_{\bar{\rho}}$.

In terms of number of generated test cases, for all faults except `Math #938`, $MOSA_{\bar{\rho}}$ generated statistically significantly fewer test cases than MOSA. For faults `Codec #99`, `Math #835`, and `Math #939`, $MOSA_{\bar{\rho}}$ generated -7%, -7.2%, and -7.6% test cases, respectively, than MOSA. For fault `Math #938`, $MOSA_{\bar{\rho}}$ generated +15.7% (however, not statistically significantly) test cases than MOSA. In contrast to the non-influence of coverage on the fault detection/localisation ability

Table 5.5: Coverage achieved and number of generated test cases by MOSA, and MOSA$_{\bar{\rho}}$. $\hat{A}_{Cov}$ reports the probability of MOSA$_{\bar{\rho}}$ achieving lower coverage than MOSA (and p, its p-value), $\hat{A}_{\#T}$ reports the probability of MOSA$_{\bar{\rho}}$ generating fewer test cases than MOSA (and p, its p-value), and Rel. Impr. reports how much better/worse MOSA$_{\bar{\rho}}$ is than MOSA at achieving high coverage and at generating small test suites. σ and CI columns report the standard deviation and confidence intervals (at 95% significance level), respectively, of the averaged coverage value (i.e., column Cov.), and of the averaged number of test cases (i.e., column #T). The "Overall" results only consider faults detected by all configurations.

Conf.	Cov.	σ	CI	$\hat{A}_{Cov}$	p	Rel. Impr.	#T	σ	CI	$\hat{A}_{\#T}$	p	Rel. Impr.
						Codec #99						
MOSA	0.98	0.00	[0.98,0.98]	—	—	—	49	2.39	[47.20,50.80]	—	—	—
MOSA$_{\bar{\rho}}$	0.97	0.01	[0.97,0.97]	0.05	0.00	-1.3%	46	3.49	[44.77,46.79]	0.20	0.03	-7.0%
						Compress #114						
MOSA	1.00	0.00	[1.00,1.00]	—	—	—	16	0.00	[16.00,16.00]	—	—	—
MOSA$_{\bar{\rho}}$	—	—	—	—	—	—	—	—	—	—		
						Math #835						
MOSA	0.98	0.00	[0.98,0.98]	—	—	—	54	1.96	[52.62,55.25]	—	—	—
MOSA$_{\bar{\rho}}$	0.98	0.00	[0.98,0.98]	0.50	0.00	0.0%	50	3.14	[48.38,51.77]	0.13	0.01	-7.2%
						Math #938						
MOSA	1.00	0.00	[1.00,1.00]	—	—	—	10	0.92	[9.75,11.00]	—	—	—
MOSA$_{\bar{\rho}}$	1.00	0.00	[1.00,1.00]	0.50	0.00	0.0%	12	0.00	[12.00,12.00]	0.94	0.08	+15.7%
						Math #939						
MOSA	0.96	0.00	[0.96,0.96]	—	—	—	10	1.53	[9.32,10.63]	—	—	—
MOSA$_{\bar{\rho}}$	0.96	0.00	[0.96,0.96]	0.50	0.00	0.0%	9	0.54	[9.12,9.36]	0.37	0.05	-7.6%
						Joda-Time						
MOSA	—	—	—	—	—	—	—	—	—	—		
MOSA$_{\bar{\rho}}$	—	—	—	—	—	—	—	—	—	—		
						Overall						
MOSA	0.98	0.00	[0.98,0.98]	—	—	—	31	1.70	[29.72,31.92]	—	—	—
MOSA$_{\bar{\rho}}$	0.98	0.00	[0.98,0.98]	0.39	0.00	-0.3%	29	1.79	[28.57,29.98]	0.41	0.04	-1.5%

of each configuration, the number of test cases indeed influenced the fault localisation effectiveness of MOSA and MOSA$_{\bar{\rho}}$. For faults in which both configurations achieved the same coverage (i.e., all `Math` faults), the configuration that generated the fewest number of test cases performed worst at ranking the faulty statements.

> *RQ3: On average,* MOSA$_{\bar{\rho}}$ *achieves -0.3% coverage (with an effect size of 0.39) than* MOSA*, and it generates statistically significantly fewer test cases (-1.5% on average).*

5.5 RELATED WORK

Although there is a large body of work on automated test generation and debugging in general, there have only been few attempts at using automatic test generation techniques in the context of debugging.

Baudry et al. [260] proposed an approach to improve diagnostic accuracy using a bacteriological algorithm (similar to a GA) to select test cases from a test suite. The criterion for test selection they

proposed estimates the quality of a test for diagnosis. Their selection procedure attempts to find an optimal balance between the size of a test suite and its contribution to diagnosis. The goal of their work is similar to ours, but our contributions are complementary: One could use *entropy* for test generation and the algorithm they proposed for test selection. It remains to be evaluated if such a combination would improve the diagnostic report's accuracy.

Artzi et al. [261] use a specialized concrete-symbolic execution [35, 80] to improve fault localisation. The principle of their customized algorithm is highly similar to the Nearest Neighbours Queries algorithm proposed by Renieris et al. [262]. The approach proposed by Artzi et al. [261] generates tests that are similar to a given failing test, whereas the approach proposed by Renieris et al. [262] selects tests that are similar to a given failing one. One important difference between our work and theirs is in the assumptions: While we make no assumption about any existing test suite, they assume there is at least one fault-revealing test to seed the search. However, in practice it is possible that no fault-revealing test exists in the test suite. An important technical difference between our approaches is that their algorithm uses as input a single fault-revealing test and generates passing tests that minimises observed differences for that particular test. However, it has been shown in previous studies [263] and as we discussed in RQ2, multiple fault-revealing tests can help improve diagnostic accuracy.

Rößler et al. [245] introduced a search-based approach to identify fault candidates. Similar to the work of Artzi et al. [261], their BUGEX tool takes a failing test case as a starting point, determines a set of "facts" (e.g., executed statements, branches, program states, etc.) and then systematically tries to generate variations of the failing test which differ in individual facts. If a passing test differs in only one fact to the failing test, then that fact is assumed to be relevant for diagnosis; if the differing test also fails, then the fact is irrelevant. BUGEX is also implemented using EVOSUITE, but our approach differs in several aspects: First, we do not assume the existence of a single failing test — we can optimize a test suite also in the presence of no faults or in the presence of multiple faults. Second, BUGEX uses a white-box testing approach to minimise facts about structural aspects of a program. In contrast, our approach is only guided by *entropy*, which means it is applicable to any testing domain.

5.6 SUMMARY

Despite the fact that evolutionary algorithms are very effective at generating test suites with high levels of coverage, test suites might not exhibit properties a developer might desire, such as, for example, ability to find faults or to minimise the effort of localising the root cause

of a failure. In this chapter we proposed the use of an additional non-functional criterion, *entropy*, as a secondary objective of an automated test generation approach to improve the ability of test suites at detecting and localising faults. An empirical study on six real faults showed that: 1) the proposed approach is statistically significantly more effective at detecting one fault, and better at detecting three other faults than a baseline coverage based approach; 2) the proposed approach is 25% more effective at fault localisation than a baseline approach. Our evaluation also showed that the use of *entropy* as secondary objective has a sightly negative effect on the coverage achieved (-0.3%), but has a positive effect on the number of generated test cases (-1.5%).

So far, we have only evaluated the performance of evolutionary algorithms on a single class of a single version of a program under test. However, it is fair to accept that a developer would want to generate test cases for *all* classes of a project as all could be faulty, or just because a certain amount of coverage across all classes needs to be met. But, how to generate test cases for thousands of classes? What if the software changes in the near future, does a developer need to re-generate tests for all classes? To answer these and other questions, in the next chapter we introduce the concept of *continuous test generation* where we present several strategies that could be used to efficiently generate tests for software that is typically developed incrementally.

6

CONTINUOUS TEST GENERATION: ENHANCING CONTINUOUS INTEGRATION WITH AUTOMATED TEST GENERATION

ABSTRACT

In object oriented software development, automated unit test generation tools typically target one class at a time. A class, however, is usually part of a software project consisting of more than one class, and these are subject to changes over time. This context of a class offers significant potential to improve test generation for individual classes. In this chapter, we introduce *Continuous Test Generation* (CTG), which includes automated unit test generation during continuous integration (i.e., infrastructure that regularly builds and tests software projects). CTG offers several benefits: First, it answers the question of how much time to spend on each class in a project. Second, it helps to decide in which order to test them. Finally, it provides techniques to select which classes should be subjected to test generation in the first place. We have implemented CTG using the EvoSuite unit test generation tool, and performed experiments using eight of the most popular open source projects available on GitHub, ten randomly selected projects from the SF100 corpus, and five industrial projects. Our experiments demonstrate improvements of up to +58% for branch coverage and up to +69% for thrown undeclared exceptions, while reducing the time spent on test generation by up to +83%.

6.1 Introduction 103
6.2 Testing Whole Projects 106
6.3 Continuous Test Generation (CTG) 111
6.4 Empirical Study 113
6.5 Related Work 128
6.6 Summary 129

6.1 INTRODUCTION

As previously reviewed in Chapter 2, a number of different automated test generation techniques and tools such as EvoSuite [9] or Pex [81] to support software testers and developers have been proposed. However, even though these tools make it feasible for developers to apply automated test generation on an individual class during development, testing an entire project consisting of many classes in

an interactive development environment is still problematic: systematic unit test generation is usually too computationally *expensive* to be used by developers on entire projects. For example, the EvoSuite [9] search-based unit test suite generator requires somewhere around 2 minutes of search time to achieve a decent level of coverage on most classes (as every fitness evaluation requires costly test execution), and more time for the search to converge. While 2 minutes may not sound particularly time consuming, it is far from the instantaneous result developers might expect while writing code. Even worse, a typical software project has more than one class — for example, `JodaTime` [264] (one of the most popular Java libraries) has more than 130 classes, and consequently generating tests for 2 minutes per class would take more than 4 hours. Thus, most unit test generation techniques are based on the scenario that each class in a project is considered a unit and could be tested independently.

In practice, unit test generation may not always be performed on an individual basis, state-of-the-art unit test generation tools still lack wide adoption by industry. For instance, in industry there are often requirements on the minimum level of code coverage[1] that needs to be achieved in a software project, meaning that test generation may need to be applied to *all* classes. As the software project evolves, involving code changes in multiple sites, test generation may be repeated to maintain and improve the degree of unit testing. Yet another scenario is that, for legacy projects with large codebases, an automated test case generation tool might be applied to all classes when introduced for the first time. If the tool does not work convincingly well in such a case, then likely the tool will not be adopted.

By considering a software project and its evolution as a whole, rather than each class independently, there is the potential to use contextual information for improving unit test generation:

– When generating test cases for a set of classes, it would be suboptimal to use the same amount of computational resources for all of them, especially when there are at the same time both trivial classes (e.g., only having get and set methods) and complex classes full of non-linear predicates.

– Test suites generated for one class could be used to help the test data generation for other classes, for example using different types of seeding strategies [150].

– Finally, test suites generated for one revision of a project can be helpful in producing new test cases for a new revision.

An attractive way to exploit this potential lies in using *continuous integration* [265]. The roots of continuous integration can be traced

1 Industrial standards such as DO-178B, IEC 61508, and IEEE 1008-1987 require 100% code coverage.

back to the Extreme Programming methodology. One of the main objectives of continuous integration is to reduce the problems of "integration hell", i.e., different engineers working on the same code base at the same time, such that their changes have to be merged together. One approach to deal with such problems is to use controlled version repositories (e.g., SVN or Git) and to commit changes on a daily basis, instead of waiting days or weeks. At each new code commit, a remote server system can build the application automatically to see if there are any code conflicts. Furthermore, at each new build, the available regression test suites can be executed to verify whether any new features or bug fixes break any existing functionality; developers responsible for new failures can be automatically notified.

Continuous integration is typically run on powerful servers, and can often resort to build farms or cloud-based infrastructure to speed up the build process for large projects. It is widely adopted in industry, and several different systems are available for practitioners. The most popular ones include the open source projects Jenkins [266], CruiseControl [267], and GitLab CI [268]; and the non-open source projects Travis CI [269], Circle CI [270], and Bamboo from Atlassian [271]. The functionalities of those continuous integration systems can typically be extended with plugins. For example, at the time of this writing, Jenkins has more than 1,000 plugins [266], including plugins that measure and visualise code coverage information based on regression test suites (e.g., the Emma plugin [272]), or plugins that reports warnings collected with static analysis (e.g., FindBugs [273, 274]).

This opens doors for automated test generation tools, in order to enhance the typically manually generated regression test suites with automatically generated test cases, and it allows the test generation tools to exploit the advantages offered when testing a project as a whole. In fact, if automated oracles are available (e.g., formal post-conditions and class invariants), then a test case generation tool can be run continuously 24/7, and can report to the developers as soon as a specification condition is violated. An example of such form of "continuous" testing is discussed by Nguyen et al. [275]. Note that, *continuous testing* can also be performed locally [276, 277]. Besides running regression test suites on dedicated continuous integration servers, these suites could also be automatically run in the background on the development machines by the IDE (e.g., Eclipse). The idea would be to provide feedback to the developers as soon as possible, while they are still editing code.

In this chapter, we introduce Continuous Test Generation (CTG), which enhances continuous integration with automated test generation. This integration raises many questions on how to test the classes in a software project: For instance, *in which order* should classes be tested?, *how much time* to spend on each class?, and *which information*

can be carried over from the tests of one class to another? To provide first answers to some of these questions, we have implemented CTG as an extension to the EvoSuite test generation tool and performed experiments on a range of different software projects. In detail, the contributions of this chapter are as follows:

– We introduce the problem of generating unit tests for whole projects, and discuss in details many of its aspects.

– We describe different strategies of scheduling the order in which classes are tested to improve the performance.

– We propose a technique to incrementally test the units in a software project, leading to overall higher code coverage while reducing the time spent on test generation.

– We present a rigorous empirical study on 10 open source projects from the SF100 corpus, eight of the most popular projects on GitHub, and five industrial projects supporting the viability and usefulness of our presented techniques.

– All the presented techniques have been implemented as an extension of the EvoSuite test generation tool.

Our experiments demonstrate that, by intelligently using the information provided when viewing a software project as a whole, the techniques presented in this chapter can lead to improvements of up to +58% for branch coverage and up to +69% for thrown undeclared exceptions. At the same time, applying this test generation incrementally not only improves the test effectiveness, but also saves time — by up to +83%. However, our experiments also point out important areas of future research on CTG which are discussed later in Section 8.2.4.

The chapter is organised as follows. In Section 6.2 we introduce the problem of testing whole software projects and propose three strategies to improve the performance of automated test generation techniques. In Section 6.3 we introduce a CTG strategy, which integrates automated test generation in a continuous integration environment. Thereafter, in Section 6.4 we present the details of our empirical study and evaluate each proposed strategy at testing whole software projects. We compare our CTG strategy with related work in Section 6.5, and finally, we summarise the chapter in Section 6.6.

6.2 TESTING WHOLE PROJECTS

Test generation is a complex problem, therefore the longer an automated test generation tool is allowed to run on an individual class, the better the results. For example, given more time, a search-based approach will be able to run for more iterations, and a tool based

on DSE can explore more paths. However, the available time budget is usually limited and needs to be distributed among all individual classes of a given software project. The problem addressed in this section can thus be summarised at high level as follows:

> *Given a project X, consisting of n units, and a time budget b, how to best use an automated unit test generation tool to maximise code coverage and failure detection on X within the time limit b?*

The values for n and b will be specific to the projects on which test generation is applied. In our experiments, the values for n range from 1 to 412, with an average of 90. Estimating what b will look like is more challenging, and at the moment we can only rely on the informal feedback of how our industrial partners think they will use EvoSuite on whole projects. However, it is fair to assume that already on a project of a few hundred classes, running EvoSuite with a minimum of just a few minutes per CUT might take hours. Therefore, what constitutes a reasonable value for b will depend on the particular scenario.

If EvoSuite is run on developer machines, then running it on a whole project at each code commit might not be a feasible option. However, it could be run after the last code commit of the day until the day after. For example, on a week day, assuming a work schedule from 9 a.m. to 5 p.m., it could mean running EvoSuite for 16 hours, and 64 hours on weekends (i.e., 16 hours on Friday, 24 hours on Saturday, and 24 hours on Sunday). Given a modern multicore PC, EvoSuite could even be run on a whole project during the day, in a similar way as done with regression suites in continuous testing [276, 277]; but that could have side effects of slowing down the PC during coding and possible noise issues that might be caused by the CPU working at 100%. An alternative scenario would be a remote continuous integration system serving several applications/departments within a company. Here, the available budget b would depend on the build schedule and on the number of projects for which the continuous integration server is used. Some companies also use larger build-farms or cloud-infrastructure for continuous integration, which would allow for larger values of b, or more frequent runs of EvoSuite.

6.2.1 *Simple Budget Allocation*

The simplest, naïve approach to target a whole project is to divide the budget b equally among the n classes, and then apply a tool like EvoSuite independently on each for b/n minutes (assuming no parallel runs on different CPUs/cores). In this chapter, we call this *simple* strategy, and it is the strategy we have used in past empirical

studies of EvoSuite (e.g., [278]). However, this simple strategy may not yield optimal results. In the rest of this section, we describe different aspects of targeting whole projects that can be addressed to improve upon the *simple* strategy. Note that in principle test generation for a class can be finished before the allocated budget is used up (i.e., once 100% coverage is achieved). In this case, the time saved on such a class could be distributed on the remaining classes; that is, the schedule could be adapted dynamically during runtime. For our initial study we performed two experiments: 1) only optimising for coverage, 2) only optimising for exceptions [130] (where no test generation run would end prematurely). I.e., as exception is a unbounded fitness function (see Section 3.2.1.6 for more details), all test generation runs would spend the time budget allocated in full. However, we will consider such optimisations as future work.

6.2.2 *Smart Budget Allocation*

In the *simple* approach, each n CUT gets an equal share of the time budget b. If there are k CPUs/cores that can be used in parallel (or a distributed network of computers), then the actual amount of available computational resources is $k \times b$. For example, assuming a four core PC and a 10 minute budget, then a tool like EvoSuite could run on 40 CUTs for one minute per CUT. However, such a resource allocation would not distinguish between trivial and complex classes requiring more resources to be fully covered. This budget allocation can be modelled as an optimization problem.

Assume n CUTs, each taking a share of the total $k \times b$ budget, with b expressed as number of minutes. Assume a testing tool that, when applied on a CUT c for z minutes, obtains performance response $t(c, z) = y$, which could be calculated as the obtained code coverage and/or number of triggered failures in the CUT c. If the tool is randomized (e.g., a typical case in search-based and dynamic symbolic execution tools like EvoSuite), then y is a random variable. Let $|Z| = n$ be the vector of allocated budgets for each CUT, and $|Y_Z| = n$ the vector of performance responses $t(c, z)$ calculated once Z is chosen and the automated testing tool is run on each of the n CUTs for the given time budgets in Z. Assume a performance measure f on the entire project that should be maximised (or minimised). For example, if y represents code coverage, one could be interested in the average $f(Z) = \frac{\sum y \in Y_Z}{n}$ of all of the CUTs. Under these conditions, maximising $f(Z)$ could be represented as a search problem in which the solution space is represented by the vector Z, under two constraints: first, their total budget should not exceed the total, i.e., $\sum_{z_i \in Z} z_i \leqslant k \times b$, and, second, it should be feasible to find a "schedule" in which those n "jobs" can be run on k CPUs/cores within b minutes. A trivial con-

sequence of this latter constraint is that no value in Z can be higher than b.

Given this optimization problem definition, any optimization/search algorithm (e.g., genetic algorithms) could be used to address it. However, there are several open challenges with this approach, like for example:

- The optimization process has to be extremely efficient, as any time spent on it would be taken from the budget $k \times b$ for test generation.

- The budget allocation optimization has to be done before generating any test case for any CUT, but the values $t(c, z) = y$ are only obtained *after* executing the testing tool and the test cases are run. There is hence the need to obtain an estimate function t', as t cannot be used. This t' could be for example obtained with machine learning algorithms [279], trained and released as part of the testing tool. A further approach could also be to execute some few test cases, and use the gathered experience to predict the complexity of the CUT for future test case generation efforts.

- Even if it is possible to obtain a near perfect estimate function $t' \simeq t$, one major challenge is that its output should not represent a single, concrete value y, but rather the probability distribution of such a random variable. For example, if the response is measured as code coverage, a possibility could be that the output of $t'(c, z)$ is represented by a $|R| = 101$ vector, where each element represents the probability P of y obtaining such a code coverage value (with 1% interval precision), i.e. $R[i] = P(y == i\%)$, where $\sum r \in R = 1$. Based on how R is defined (could even be a single value representing a statistics of the random variable, like mean and median), there can be different ways to define the performance measure $f(Z)$ on the entire project.

After having described the budget allocation problem in general, in this section we present a first attempt to address it. We start our investigation of addressing whole projects with a simple to implement technique. First, each CUT will have a minimum amount of the time budget, e.g., $z \geqslant 1$ (i.e., one minute). Then the remaining budget $(k \times b) - (n \times z)$ can be distributed among the n CUTs proportionally to their number of branches (but still under the constraint $z \leqslant b$). In other words, we can estimate the difficulty of a CUT by counting its number of branches. This is an easy way to distinguish a trivial from a complex CUT. Although counting the number of branches is a coarse measure, it can already provide good results (as we will show in the empirical study in this chapter). It is conceivable that more sophisticated metrics such as, for example, cyclomatic complexity or even

techniques based on machine learning, may lead to improved budget distribution. We will discuss this in Section 8.2.4 as part of our future work.

Having a minimum amount of time per CUT (e.g., $z \geqslant 1$) is independent of whether a smart budget allocation is used. For example, if we only have one core and budget $b = 5$ minutes, it would make no sense to run EvoSuite on a project with thousands of CUTs, as only a few milliseconds would be available on average per CUT. In such cases, it would be more practical to just run EvoSuite on a subset of the classes (e.g., five) such that there is enough time (e.g., one minute) for each of those CUTs to get some usable result. Ensuring that all classes are tested would then require allocating the budget to different classes in successive runs of EvoSuite in the following days (Section 6.3.1 presents some ideas on how to use historical data to address this problem).

6.2.3 *Seeding Strategies*

After allocating the time budget Z for each of the n CUTs, the test data generation (e.g., using EvoSuite) on each of those n CUTs will be done in a certain order (e.g., alphabetically or randomly), assuming $n > k$ (i.e., more CUTs than possible parallel runs). This means that when we start to generate test cases for a CUT c, we will usually have already finished generating test suites for some other CUTs in that project, and these test suites can be useful in generating tests for c. Furthermore, there might be information available from past EvoSuite runs on the same project. This information can be exploited for *seeding*.

In general, with seeding we mean any technique that exploits previous knowledge to help solve a testing problem at hand. For example, in SBST existing test cases can be used when generating the initial population of a genetic algorithm [280], or can be included when instantiating objects [150]. Seeding is also useful in a DSE context, in particular to overcome the problem of creating complex objects [281], and the use of seeding in test suite augmentation is established for SBST and DSE-based augmentation approaches [68].

In order to make it possible to exploit information from different CUTs within a run of EvoSuite on a whole project, one needs to *sort* the execution of the n CUTs in a way that, when a class c can use test cases from another class c', then c' should be executed (i.e., generated test for) before c and if test execution for c' is currently running, then postpone the one of c till c' is finished, but only if meanwhile another class c'' can be generated tests for. I.e., if a CUT A takes as input an object of type B, then to cover A we might need B set in a specific way. For instance, in the following snippet of code, using the test cases generated for B can give us a pool of interesting instances of B.

```
public class A {
  public void foo(B b) {
    if (b.isProperlyConfigured()) {
      ... // target
    }
  }
}
```

To cover the target branch in A.foo, one could just rely on traditional SBST approaches to generate an appropriate instance of B. But, if we first generate test suites for B, then we can exploit those tests for seeding in A. For example, each time we need to generate an input for A.foo, with a certain probability (e.g., 50%) we can rather use a randomly selected instance from the seeded pool, which could speed up the search.

6.3 CONTINUOUS TEST GENERATION (CTG)

So far, we have discussed generating unit tests for all classes in a project. However, projects evolve over time: classes are added, deleted, and changed, and automated test generation can be invoked regularly during continuous integration, by extending it to CTG. CTG can exploit all the historical data from the previous runs to improve the effectiveness of the test generation.

There are two main ways in which CTG can exploit such historical data: First, we can improve the *budget allocation*, as newly introduced classes should be prioritized over old classes that have been extensively tested by CTG in previous runs. Second, the test cases generated in the previous runs can be directly used for *seeding* instead of regenerating tests for each class from scratch at every CTG run on a new software version.

6.3.1 *Budget Allocation with Historical Data*

The *Budget* allocation described in Section 6.2.2 only takes into account the complexity of a CUT. However, there are several factors that influence the need to do automated test generation when it is invoked repeatedly. Usually, a commit of a set of changes only adds/-modifies a few classes of a project.

- If a class has been changed, more time should be spent on testing it. First, modified source code is more prone to be faulty than unchanged source code [282]. Second, the modifications are likely to invalidate old tests that need to be replaced, or add new behaviour for which new tests are required [174].

  – If a class has *not* been changed, invoking automated test generation can still be useful if it can help to augment the existing test suite. However, once the test generator has reached a maximum level of coverage and cannot further improve it for a given class, invoking it again will simply waste resources.

For example, suppose a project X has two classes: a "simple" one S, and a "difficult" one D. Assume that, by applying the *Budget* allocation (see Section 6.2.2), the time budget allocated for D is twice as much than for S, i.e. $z_D = 2 \times z_S$ and $z_S = \frac{z_D}{2}$. Now, further suppose that only S has been changed since the last commit; in this case, we would like to increase the time spent on testing S, even though it is a simple one. For this, we first use an underlying basic budget assignment (e.g., *Budget* or *Budget & Seeding*), and then multiplied by a factor $h > 1$, such that the budget for S becomes $z_S = h \times \frac{z_D}{2}$. Thus, if $h = 2$ (which is the value we use in the experiments reported in this chapter), then the *modified* simple class S will receive the same amount of time as the *unchanged* difficult class D.

Given an overall maximum budget (see Section 6.2.2), the total budget should not exceed this maximum, even in the face of changed classes. That is, $\sum_{z_i \in Z} z_i \leqslant k \times b$; however, it will happen that adding a multiplication factor h for new/modified classes results in the total budget exceeding this maximum. As test generation will be invoked regularly in this scenario, it is not imperative that all classes are tested, especially the ones that have not been modified. So, one can apply a strategy to skip the testing of some unchanged classes in the current CTG execution. To do that, we rank classes according to their complexity and the fact of whether they were modified, and then select the maximum number of classes such that the total budget $k \times b$ is not exceeded.

For classes that have not been changed, at some point we may decide to stop invoking the test generator on them. A possible way to do this is to monitor the progress achieved by the test generator. If 100% coverage has been achieved, then generating more tests for the same criterion will not be possible. If less than 100% coverage has been achieved, then we can invoke test generation again. However, if after several invocations the test generator does not succeed in increasing the coverage, we can assume that all coverage goals that the test generator can feasibly cover have been reached. In the context of this chapter, we look at the last three runs of the test generator, and if there has been no improvement for the last three runs, then we stop testing a class until it is changed again.

6.3.2 *Seeding Previous Test Suites*

When repeatedly applying test generation to the same classes, the results of the previous test generation run can be used as a start-

ing point for the new run. This is another instance of seeding, as described in Section 6.2.3. There are different ways how a previous result can be integrated into a new run of a genetic algorithm. For example, in previous work where the goal was to improve upon manually written tests, Fraser et al. [150] re-used the existing test cases by modifying the search operators of EvoSuite such that whenever a new test case was generated, it was based on an existing test case with a certain probability. Xu et al. [68] considered the reuse of test cases during test suite augmentation for DSE or search-based and hybrid techniques [69], by using the old tests as starting population of the next test generation run; in this approach the success of the augmentation depends strongly on the previous tests.

The approach we took in the context of CTG is to first check which of the previous tests still compile against the new version of a CUT. For example, if from version p_n to p_{n+1} a signature (i.e., name or parameters) of a method, or a class name is modified, test cases may no longer compile and therefore are not candidates to be included in the next test suite. One the other hand, tests that still compile on the new version of the CUT can be used for seeding. I.e., instead of initialising the initial population of the genetic algorithm in EvoSuite completely randomly, we create a test suite with all valid test cases and insert it as one individual into the initial population of the new genetic algorithm. Thus, essentially applying a form of elitism between different invocations of the genetic algorithm.

6.4 Empirical Study

In this section we empirically evaluate the strategies proposed in this chapter at testing a project as a whole. In particular, we evaluate the following strategies: a *Simple* strategy (Section 6.2.1), a smart *Budget* allocation strategy (Section 6.2.2), a *Seeding* strategy (Section 6.2.3), a combination of the latter two (i.e, smart *Budget* and *Seeding* strategy at the same time, *Budget & Seeding*), and a CTG strategy (Section 6.3), and we aim at answering the following research questions:

RQ1: What are the effects of smart *Budget* allocation?

RQ2: What are the effects of *Seeding* strategies?

RQ3: How does combining *Seeding* with smart *Budget* allocation fare?

RQ4: What are the effects of using CTG for test generation?

RQ5: What are the effects of *History*-based selection and *Budget* allocation on the total time of test generation?

6.4.1 *Experimental Setup*

To answer the research questions, we performed two different types of experiments: The first one aims to identify the effects of optimizations based on testing whole projects; the second experiment considers the scenario of testing projects over time.

6.4.1.1 *Unit Test Generation Tool*

We used the EvoSuite [9] unit test generation tool, which already provides support for the *Simple* strategy [200, 278], and therefore would allow an unbiased comparison of the different strategies. For this study we implemented the remaining strategies described in this chapter as an extension of the EvoSuite tool.

6.4.1.2 *Subject Selection*

We used three different sources for case study projects (see Table 6.1): First, as an unbiased sample, we randomly selected ten projects from the SF100 corpus of classes [278] (which consists of 100 projects randomly selected from SourceForge); this results in a total of 279 classes. Second, we used five industrial software projects (1,307 classes in total) provided by one of our industrial collaborators. Due to confidentiality restrictions, we can only provide limited information on the industrial software.

To simulate evolution with CTG over several versions, we required projects with version history. Because it is complicated to obtain a full version history of compiled software versions for each project in the two previous sets (due to different repository systems and compilation methods), we additionally considered the top 15 most popular projects on GitHub[2]. We had to discard some of these projects: 1) For some (e.g., `JUnit` [283], `JNA` [284]) there were problems with EvoSuite (e.g., EvoSuite uses `JUnit` and thus cannot be applied to the `JUnit` source code without modifications). 2) Some projects (`Jedis` [285], `MongoDB Java Driver` [286]) require a server to run, which is not supported by EvoSuite yet. 3) We were unable to compile the version of `RxJava` [287] last cloned (10 March, 2014). 4) By default, classes of the `Rootbeer GPU Compiler` [288] project were compiled with an incorrect package name. 5) Finally, we removed `Twitter4J` [289], the largest project of the 15 most popular projects, as our experimental setup would not have allowed to finish the experiments in time. This leaves the following eight projects (475 classes in total), each with a version history for experimentation: `HTTP-Request` [290], `JodaTime` [264], `JSON` [291], `JSoup` [292], `Scribe` [293], `Spark` [294], `Async-HTTP-Client` [295], and `SpringSide` [296].

2 GitHub homepage `https://github.com`, accessed 11/2017.

Table 6.1: Case study software projects. For each software project we report the number of classes under test. Note that for the most popular software projects on GitHub the number of classes is not a constant, as it can change at each revision. We hence report the total number of unique classes throughout the 100 commits, and, in brackets, the number of classes at the first and last commits.

Project	# CUTs
Open source projects randomly selected from* SF100 *corpus	
tullibee	18
a4j	45
gaj	10
rif	13
templateit	19
jnfe	51
sfmis	19
gfarcegestionfa	48
falselight	8
water-simulator	48
Industrial software projects	
projectA	245
projectB	122
projectC	412
projectD	211
projectE	317
Most popular software projects on GitHub	
HTTP-Request	1 [1:1]
JodaTime	135 [133:132]
JSon	37 [16:25]
JSoup	45 [41:45]
Scribe	79 [65:78]
Spark	34 [21:30]
Async-HTTP-Client	81 [71:75]
SpringSide	63 [23:60]

6.4.1.3 *Experiment Procedure*

For each open source project of the SF100 corpus and industrial project, we ran EvoSuite with four different strategies: *Simple* (Section 6.2.1), smart *Budget* allocation (Section 6.2.2), *Seeding* strategy (Section 6.2.3), and a combination of the latter two (i.e, smart *Budget* and *Seeding* strategy at the same time, *Budget & Seeding*). For the open source projects from GitHub we ran EvoSuite with the same

four strategies, but also with another strategy, a *History* strategy (Section 6.3.1) which used seeding of previous test suites (Section 6.3.2). The open source subjects were run on the University of Sheffield Iceberg HPC Cluster [201]. Each cluster node was running a Linux distribution, using six cores (12 considering hyper-threading) at 2.6 GHz. On the other hand, the industrial case study was run on the development machine of one of the software engineers employed by our industrial partner. The machine was running Windows 7, with six cores (12 considering hyper-threading) at 3.07 GHz. In contrast to the experiments on the cluster, that were using all the 12 virtual cores per node, in the industrial case study we only used six parallel CTG runs at any given time. The reason was that we could not occupy all the computational resources of that machine.

As we have described, when running EvoSuite on a whole project, there is the question of how long to run it. This depends on the available computational resources and how EvoSuite will be used in practice (e.g., during the day while coding, or over the weekend). In this chapter, due to the high cost of running the experiments, we could not consider all these different scenarios. So, we decided for one setting per case study that could resemble a reasonable scenario. In particular, for all the case studies we allowed an amount of time proportional to the number of classes in each project, i.e., three minutes per CUT $\times$ | CUTs |. For the smart *Budget* allocation, we allowed a minimum amount of time $z \geqslant 1$ minute (see Section 6.2.2).

Unlike the other strategies, the *History* strategy requires different versions of the same project. As considering the full history would not be feasible, we limited the experiments to the last 100 commits of each project, i.e., we considered the latest 100 consecutive commits of each project. Note that, project `JSON` only has 65 commits in its entire history.

For the experiments, we configured *History* to use the *Budget* allocation as baseline because the average branch coverage on the first set of experiments (10 projects randomly selected from SF100 corpus) achieved an highest relative improvement on that approach. The maximum time for test generation was calculated for *History* for each commit in the same way as for other strategies proportional to the number of CUTs in the project (three minutes per CUT $\times$ | CUTs |).

On the open source projects from SF100, each experiment was repeated 50 times with different random seeds to take into account the randomness of the algorithms. As we applied *History* with a time window of 100 commits to the GitHub projects, we only ran EvoSuite five times on these eight projects. On the industrial systems we were only able to do a single run. Running experiments on real industrial case studies presents many challenges, and that is one of the reasons why they are less common in the software engineering literature. Even if it was not possible to run those experiments as rig-

orously as in case of the open source software, they do provide extra valuable information to support the validity of our results.

6.4.1.4 *Measurements*

As primary measurement of success of test generation we use branch coverage (previously defined in Section 2.4.5.2). However, branch coverage is only one possible measure to quantify the usefulness of an automatically generated test suite [12, 297]. In the presence of automated oracles (e.g., formal specifications such as pre/post-conditions), one would also want to see if any fault has been found. Unfortunately, automated oracles are usually unavailable. One could look at program crashes, but that is usually not feasible for unit testing. However, at unit level it is possible to see if any exception has been thrown in a method of the CUT, and then check whether that exception is declared as part of the method signature (i.e., using the Java keyword `throws`).

As a second measurement we used undeclared exceptions (previously defined in Section 3.2.1.6). If an exception is declared as part of a method signature, then throwing such an exception during execution would be part of normal, expected behaviour. On the other hand, finding an undeclared exception would point to a unit level bug. Such a bug might not be critical (e.g., impossible to throw by the user through the application interfaces such as a GUI), and could even simply point to "implicit" preconditions. For example, some exceptions might be considered as normal if a method gets the wrong inputs (e.g., a null object) but, then, the developers might simply fail to write a proper method signature. This is the case when an exception is explicitly thrown with the keyword `throw`, but then it is missing from the signature.

Whether a thrown exception represents a real fault is an import question for automated unit testing. In particular, it is important to develop techniques to filter out "uninteresting" exceptions that likely are just due to violated implicit preconditions. However, regardless of how many of these exceptions are caused by real bugs, a technique that finds more of these exceptions would be better [257]. For this reason, tools like EvoSuite not only try to maximise code coverage, but also the number of unique, undeclared thrown exceptions for each method in the CUTs, and experiments have shown that this can reveal faults [130].

For the first set of experiments the overall time per project was fixed. In the second set of experiments on CTG we also look at the time spent on test generation.

6.4.1.5 *Analysis Procedure*

The experiments carried out in this chapter are very different than previous uses of EvoSuite. In previous empirical studies, each CUT was targeted independently from the other CUTs in the same project. That was to represent scenarios in which EvoSuite is used by practitioners on the classes they are currently developing. On the other hand, here when targeting whole projects there are dependencies: e.g., in the smart *Budget* allocation, the amount of time given to each CUT depends also on the number of branches of the other CUTs. When there are dependencies, analysing the results of each CUT separately might be misleading. For example, how to define what is the branch coverage on a whole project?

Assume a project P composed of $|P| = n$ CUTs, where the project can be represented as a vector $P = \{c_1, c_2, \ldots, c_n\}$. Assume that each CUT c has a number of testing targets $\gamma(c)$, of which $k(c)$ are actually covered by applying the analysed testing tool. Because tools like EvoSuite are randomized, the scalar $k(c)$ value will be represented by a statistics (e.g., the mean) on a sample of runs (e.g., 50) with different random seeds. For example, if the tool was run r times, in which each time we obtained a number of covered targets $k_i(c)$, then $k(c) = \frac{\sum_{i=1}^{i=r} k_i(c)}{r}$. If we want to know the coverage for a CUT c, then we can use $cov(c) = \frac{k(c)}{\gamma(c)}$, i.e., number of covered targets divided by the number of targets. But what would be the coverage on P, i.e., the mean coverage obtained over the r runs, as is commonly used in the literature? A typical approach is to calculate the average of those coverage values averaged over all the r runs:

$$avg(P) = \frac{1}{n} \sum_{c \in P} \frac{k(c)}{\gamma(c)} \ .$$

However, in this case, all the CUTs have the same weight. The coverage on a trivially small CUT would be as important as the coverage of a large, complex CUT. An alternative approach would be to consider the absolute coverage on the project per run:

$$\mu(P_i) = \frac{\sum_{c \in P} k_i(c)}{\sum_{c \in P} \gamma(c)} \ ,$$

and, with that, consider the average on all the r runs:

$$\mu(P) = \frac{1}{r} \sum_{i=1}^{i=r} \mu(P_i) \ .$$

With $\mu(P)$, we are actually calculating the average ratio of how many targets in total have been covered over the number of all possible targets. The statistics $avg(P)$ and $\mu(P)$ can lead to pretty different results. Considering the type of problem addressed in this chapter,

we argue that $\mu(P)$ is a more appropriate measure to analyse the data of our empirical analyses.

All the data from these empirical experiments have been statistically analysed following the guidelines discussed by Arcuri et al. [202]. In particular, we used the Wilcoxon-Mann-Whitney U-test and the Vargha-Delaney $\hat{A}_{12}$ effect size [203]. The Wilcoxon-Mann-Whitney U-test is used when algorithms (e.g., result data sets X and Y) are compared (in R this is done with $wilcox.test(X, Y)$). In our case, what is compared is the distribution of the values $\mu(P_i)$ for each project P. For the statistical tests, we consider a 95% confidence level.

Given a performance measure W (e.g., branch coverage), $\hat{A}_{xy}$ measures the probability that running algorithm x yields higher W values than running algorithm y. If the two algorithms are equivalent, then $\hat{A}_{xy} = 0.5$. This effect size is independent of the raw values of W, and it becomes a necessity when analysing the data of large case studies involving artefacts with different difficulty and different orders of magnitude for W. E.g., $\hat{A}_{xy} = 0.7$ entails one would obtain better results 70% of the time with x.

Beside the standardised Vargha-Delaney $\hat{A}_{12}$ statistics, to provide more information we also considered the relative improvement ρ. Given two data sets X and Y, the relative average improvement can be defined as:

$$\rho(X, Y) = \frac{mean(X) - mean(Y)}{mean(Y)} .$$

Finally, we also reported the standard deviation σ and confidence intervals (CI) using bootstrapping at 95% significance level. Note that, because experiments on the industrial software projects were only performed once, and experiments on the most popular software projects on GitHub only performed a few times; we only reported the σ and CI for experiments on the 10 open source projects randomly selected from SF100 corpus.

6.4.1.6 *Threats to Validity*

Threats to *internal validity* might come from how the empirical study was carried out. To reduce the probability of having faults in our testing framework, it has been carefully tested. Furthermore, randomized algorithms are affected by chance. To cope with this problem, we repeated each experiment (50 times for the SF100 experiments and five times for the GitHub experiments) and followed rigorous statistical procedures to evaluate their results.

To cope with possible threats to *external validity*, the SF100 corpus was employed as case study, which is a collection of 100 Java projects randomly selected from SourceForge [278]. From SF100, 10 projects were randomly chosen. Although the use of SF100 provides high confidence in the possibility to generalize our results to other open source

software as well, we also included on our experiments some of the most popular Java projects from GitHub.

As open source software represents only one face of software development, in this chapter we also used five industrial systems. However, the selection of those systems was constrained by the industrial partners we collaborate with. Results on these systems might not generalize to other industrial systems.

The strategies presented in this chapter have been implemented in a prototype that is based on the EvoSuite tool, but any other tool that can automatically handle the subjects of our empirical study could be used. We chose EvoSuite because it is a fully automated tool, and recent competitions for JUnit generation tools [16, 185–188] suggest that it represents the state of the art.

To allow reproducibility of the results (apart from the industrial case study), all 18 subjects and EvoSuite are freely available from our webpage at `www.evosuite.org`.

6.4.2 *Testing Whole Projects*

The first set of experiments considers the effects of generating unit tests for whole projects. Table 6.2 shows the results of the experiments on the 10 open source projects randomly selected from SF100 corpus. The results in Table 6.2 are based on branch coverage. The *Simple* strategy is used as point of reference: the results on the other strategies (smart *Budget*, *Seeding* and their combination, *Budget & Seeding*) are presented relatively to the *Simple* strategy, on a per project basis. For each strategy compared to *Simple*, we report the $\hat{A}_{12}$ effect size, and also the relative improvement ρ.

Table 6.3 presents the results on the number of unique pairs exception/method for each CUT, grouped by project. For each run, we calculated the sum of all unique pairs on all CUTs in a project, and averaged these results over the 50 runs. In other words, Table 6.3 is structured in the same way as Table 6.2, with the only difference that the results are for found exceptions instead of branch coverage.

The results on the industrial experiments were analysed in the same way as the open source software results. Table 6.4 shows the results for branch coverage. However, due to confidentiality restrictions, no results on the `thrown` exceptions are reported.

The results in Table 6.2 clearly show that a smart *Budget* allocation significantly improves branch coverage. For example, for the project `sfmis` the branch coverage goes from 35.8% to 46.7% (a relative improvement of +30.6%). The $\hat{A}_{12} = 1$ means that, in *all* the 50 runs with smart *Budget* allocation the coverage was higher than in *all* the 50 runs with *Simple* strategy. However, there are two projects in which it seems it provides slightly worse results; in those cases, however, the results are not statistically significant. If we look at the results on the

Table 6.2: Branch coverage results for the 10 open source projects randomly selected from the SF100 corpus. For each project we report the branch coverage achieved by each strategy, the standard deviation (σ), and confidence intervals using bootstrapping at 95% significance level. For all strategies, but the *Simple* strategy, we report the effect sizes ($\hat{A}_{12}$ and relative average improvement) compared to the *Simple* strategy. Effect sizes $\hat{A}_{12}$ that are statistically significant are reported in bold. Results on the open source case study are based on 50 runs per configuration.

	Simple			Budget			Seeding			Budget & Seeding		
Project	Cov.			Cov.	$\hat{A}_{12}$	ρ	Cov.	$\hat{A}_{12}$	ρ	Cov.	$\hat{A}_{12}$	ρ
tullibee	39.1%	–	–	43.5%	**0.89**	+11.3%	39.6%	0.56	+1.1%	43.9%	**0.92**	+12.1%
a4j	62.5%	–	–	64.4%	**0.86**	+3.0%	55.3%	**0.00**	-11.5%	55.2%	**0.00**	-11.7%
gaj	66.5%	–	–	65.6%	0.46	-1.4%	67.5%	0.54	+1.5%	67.2%	0.53	+1.0%
rif	25.3%	–	–	25.0%	0.48	-1.4%	25.7%	0.58	+1.4%	24.8%	0.45	-2.0%
templateit	20.1%	–	–	24.6%	**0.97**	+22.4%	20.3%	0.53	+0.8%	24.9%	**0.97**	+23.7%
jnfe	38.7%	–	–	51.7%	**0.96**	+33.5%	43.9%	**0.64**	+13.4%	51.6%	**0.96**	+33.3%
sfmis	35.8%	–	–	46.7%	**1.00**	+30.6%	36.2%	0.55	+1.1%	46.3%	**0.99**	+29.3%
gfarcegestionfa	25.2%	–	–	33.4%	**0.96**	+32.5%	23.8%	0.43	-5.4%	33.1%	**0.95**	+31.5%
falselight	6.1%	–	–	6.2%	0.51	+2.0%	6.1%	0.50	0.0%	6.1%	0.50	0.0%
water-simulator	3.1%	–	–	3.8%	**0.75**	+19.1%	3.2%	0.53	+1.4%	4.0%	**0.78**	+27.2%

	Cov.	σ	CI	Cov.	σ	CI	Cov.	σ	CI	Cov.	σ	CI
tullibee	39.1%	3.2	[38.2,40.1]	43.5%	2.0	[43.0,44.1]	39.6%	3.1	[38.8,40.4]	43.9%	2.1	[43.3,44.5]
a4j	62.5%	1.0	[62.3,62.8]	64.4%	1.4	[64.0,64.8]	55.3%	2.2	[54.8,56.1]	55.2%	2.1	[54.7,55.9]
gaj	66.5%	4.9	[65.2,67.7]	65.6%	5.8	[64.1,67.3]	67.5%	4.1	[66.4,68.6]	67.2%	3.7	[66.3,68.3]
rif	25.3%	1.6	[24.9,25.8]	25.0%	1.6	[24.6,25.5]	25.7%	1.5	[25.3,26.2]	24.8%	1.5	[24.5,25.3]
templateit	20.1%	1.1	[19.8,20.4]	24.6%	2.0	[24.1,25.2]	20.3%	1.2	[20.0,20.6]	24.9%	2.0	[24.4,25.5]
jnfe	38.7%	3.4	[37.7,39.5]	51.7%	2.2	[51.2,52.3]	43.9%	8.1	[41.7,46.2]	51.6%	2.3	[51.0,52.3]
sfmis	35.8%	1.3	[35.4,36.2]	46.7%	0.9	[46.5,47.0]	36.2%	1.8	[35.7,36.7]	46.3%	1.7	[45.9,46.9]
gfarcegestionfa	25.2%	5.9	[23.6,26.8]	33.4%	1.4	[33.1,33.9]	23.8%	6.0	[22.3,25.5]	33.1%	1.5	[32.8,33.6]
falselight	6.1%	0.9	[6.0,6.4]	6.2%	0.0	[6.2,6.2]	6.1%	0.9	[6.0,6.4]	6.1%	0.9	[6.0,6.4]
water-simulator	3.1%	0.9	[2.9,3.4]	3.8%	0.9	[3.6,4.1]	3.2%	0.9	[3.0,3.5]	4.0%	1.0	[3.8,4.3]

industrial case study in Table 6.4, there is a large improvement on all five projects. In particular, for one of those projects, the relative improvement for branch coverage was as high as +67.6%.

The results in Table 6.3 are slightly different. Although the smart *Budget* allocation still provides significantly better results on a higher number of projects (statistically better in five out of 10; and equivalent results in two subjects), there are three cases in which results are statistically worse. In two of those latter cases, the branch coverage was statistically higher (Table 6.2), and our conjecture is that the way exceptions are included in EvoSuite's fitness function (see [130]) means that test suites with higher coverage (as achieved by the *Budget* allocation) would be preferred over test suites with more exceptions. In this case, improving EvoSuite's optimization strategy (e.g., by using multi-objective optimization) may lead to better results with respect to both measurements. For the `rif` project (a framework for remote method invocation) the decrease in the number of exceptions is not significant (10.66 to 9.60) and neither is the decrease in coverage (25.3% to 25.0%). In this case, it seems that the use of the number of branches is not a good proxy measurement of the test complexity. This suggests that further research on measurements other than branches as proxy for complexity would be important. On the other

Table 6.3: Thrown exception results for the 10 open source projects randomly selected from the SF100 corpus. For each project we report the total number (i.e., sum of the averages over 50 runs for each CUT) of undeclared thrown exceptions, the standard deviation (σ), and confidence intervals using bootstrapping at 95% significance level. For all strategies, but the *Simple* strategy, we report the effect sizes ($\hat{A}_{12}$ and relative ratio difference) compared to the *Simple* strategy. Effect sizes $\hat{A}_{12}$ that are statistically significant are reported in bold. Results on the open source case study are based on 50 runs per configuration.

	Simple			Budget			Seeding			Budget & Seeding		
Project	Exc.			Exc.	$\hat{A}_{12}$	ρ	Exc.	$\hat{A}_{12}$	ρ	Exc.	$\hat{A}_{12}$	ρ
tullibee	23.46	–	–	29.36	**0.88**	+25.1%	23.46	0.49	0.0%	29.06	**0.92**	+23.8%
a4j	88.96	–	–	93.58	**0.77**	+5.1%	87.20	0.41	-2.0%	88.22	0.47	-0.9%
gaj	30.28	–	–	29.74	0.40	-1.8%	31.26	**0.64**	+3.2%	30.32	0.50	+0.1%
rif	10.66	–	–	9.60	**0.28**	-10.0%	11.10	0.59	+4.1%	9.44	**0.25**	-11.5%
templateit	18.48	–	–	31.14	**0.97**	+68.5%	19.05	0.56	+3.1%	30.66	**0.98**	+65.9%
jnfe	89.84	–	–	94.46	**0.90**	+5.1%	92.88	**0.64**	+3.3%	94.34	**0.89**	+5.0%
sfmis	30.58	–	–	35.02	**0.93**	+14.5%	31.24	0.59	+2.1%	35.08	**0.89**	+14.7%
gfarcegestionfa	53.70	–	–	51.10	**0.33**	-4.9%	51.66	0.41	-3.8%	50.60	**0.29**	-5.8%
falselight	1.52	–	–	1.42	0.45	-6.6%	1.58	0.53	+3.9%	1.42	0.45	-6.6%
water-simulator	45.74	–	–	43.10	**0.21**	-5.8%	45.88	0.52	+0.3%	43.46	**0.23**	-5.0%
	Cov.	σ	CI	Cov.	σ	CI	Cov.	σ	CI	Cov.	σ	CI
tullibee	23.46	1.9	[23.0,24.0]	29.36	3.4	[28.4,30.4]	23.46	2.5	[22.8,24.1]	29.06	3.0	[28.2,29.9]
a4j	88.96	4.8	[87.6,90.2]	93.58	4.4	[92.4,94.8]	87.20	5.2	[85.9,88.6]	88.22	5.6	[86.7,89.8]
gaj	30.28	1.8	[29.8,30.8]	29.74	1.4	[29.3,30.1]	31.26	1.8	[30.8,31.8]	30.32	1.3	[30.0,30.7]
rif	10.66	1.3	[10.3,11.0]	9.60	1.2	[9.3,9.9]	11.10	1.2	[10.7,11.5]	9.44	1.1	[9.1,9.8]
templateit	18.48	3.5	[17.5,19.4]	31.14	4.3	[30.1,32.3]	19.05	2.9	[18.3,19.8]	30.66	3.9	[29.5,31.7]
jnfe	89.84	3.1	[88.9,90.6]	94.46	2.9	[93.7,95.2]	92.88	5.2	[91.3,94.2]	94.34	2.8	[93.5,95.1]
sfmis	30.58	2.3	[30.0,31.2]	35.02	2.0	[34.5,35.6]	31.24	2.4	[30.7,32.0]	35.08	2.6	[34.4,35.9]
gfarcegestionfa	53.70	7.9	[51.7,56.0]	51.10	5.9	[49.7,52.6]	51.66	8.4	[49.6,54.0]	50.60	5.2	[49.3,52.0]
falselight	1.52	0.5	[1.4,1.7]	1.42	0.5	[1.3,1.6]	1.58	0.5	[1.5,1.7]	1.42	0.5	[1.3,1.6]
water-simulator	45.74	3.0	[44.9,46.6]	43.10	2.1	[42.5,43.7]	45.88	3.1	[45.0,46.8]	43.46	1.9	[42.9,44.0]

hand, we would like to highlight that for the `templateit` project the relative improvement was +68.5%.

RQ1: Smart Budget allocation improves performance significantly in most of the cases.

Regarding input *Seeding*, in Table 6.2 there is one case in which it gives statistically better results, but also one in which it gives statistically worse results. On the industrial case study (Table 6.4), it seems to provide better results (although this observation is based only on one run). Regarding the number of `thrown` exceptions, there are two projects in which it gives statistically better results (Table 6.3). Unlike the *Budget* allocation, the usefulness of seeding will be highly dependent on the specific project under test. If there are many dependencies between classes and many branches depend on specific states of parameter objects, then *Seeding* is likely to achieve better results. If this is not the case, then the use of *Seeding* may adversely affect the search, e.g., by reducing the diversity, thus exhibiting lower overall coverage in some of the projects. However, note that the actual *Seeding* implemented in EvoSuite for these experiments is simplistic. Thus, a main

Table 6.4: Branch coverage results for the industrial case study. For each project we report the branch coverage achieved by each strategy. For all strategies, but the *Simple* strategy, we report the effect sizes ($\hat{A}_{12}$ and relative average improvement) compared to the *Simple* strategy. Results on the industrial case study are based on one single run.

	Simple	Budget			Seeding			Budget & Seeding		
Project	Cov.	Cov.	$\hat{A}_{12}$	ρ	Cov.	$\hat{A}_{12}$	ρ	Cov.	$\hat{A}_{12}$	ρ
projectA	23.6%	28.3%	1.00	+19.8%	24.2%	1.00	+2.4%	28.8%	1.00	+21.9%
projectB	13.0%	21.9%	1.00	+67.6%	15.6%	1.00	+19.7%	21.2%	1.00	+62.2%
projectC	30.4%	41.3%	1.00	+35.8%	30.3%	0.00	-0.2%	41.5%	1.00	+36.4%
projectD	72.5%	87.9%	1.00	+21.2%	72.7%	1.00	+0.2%	86.0%	1.00	+18.5%
projectE	23.9%	28.5%	1.00	+19.0%	24.1%	1.00	+0.8%	28.8%	1.00	+20.1%

conclusion from this result is that further research is necessary on *how* to best exploit this additional information during the search.

RQ2: Input Seeding may improve performance, but there is a need for better seeding strategies to avoid negative effects.

Finally, we analyse what happens when input *Seeding* is used together with the smart *Budget* allocation. For most projects, either performance improves by a little (compared to just using smart *Budget* allocation), or decreases by a little. Overall, when combined together, results are slightly worse than when just using the *Budget* allocation. This is in line with the conjecture that *Seeding* used naively can adversely affect results: Suppose that seeding on a particular class is bad (for example as is the case in the `a4j` project), then assigning significantly more time to such a class means that, compared to *Budget*, significantly more time will be wasted on misguided seeding attempts, and thus the relative performance will be worse. Note also that the overall result is strongly influenced by one particular project that is problematic for input *Seeding* (i.e., `a4j` with $\hat{A}_{12} = 0$ in Table 6.2). This further supports the need for smarter seeding strategies.

RQ3: Seeding with Budget allocation improves performance, but Seeding strategies may negatively affect improvements achieved by Budget allocation.

6.4.3 *Continuous Test Generation*

The second set of experiments considers the effects of CTG over time. Figure 6.1 plots the overall branch coverage achieved over the course of 100 commits. We denote the strategy that uses *Seeding* from previous test suites and allocation based on *History*. In most of the projects the higher coverage of the *History* strategy achieves clearly higher

Table 6.5: Coverage results over time for the most popular software projects on GitHub. For each project we report the "time coverage": the average branch coverage over all classes in a project version, averaged over all 100 commits. These time coverages are averaged out of the five repeated experiments. We compare the "*History*" strategy with the "*Simple*", "*Budget*", and "*Budget & Seeding*" ones, and report the effect sizes ($\hat{A}_{12}$ over the five runs and relative average improvement). Effect sizes $\hat{A}_{12}$ that are statistically significant are reported in bold.

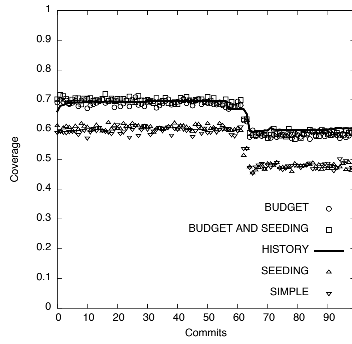
	Simple	Budget	Budget &	History						
Project	Cov.	Cov.	Seeding Cov.	Cov.	$\hat{A}_s$	ρ	$\hat{A}_b$	ρ	$\hat{A}_{b\&s}$	ρ
HTTP-Request	0.25	0.24	0.25	0.39	**1.00**	+57.97%	**1.00**	+58.69%	**1.00**	+56.77%
JodaTime	0.55	0.62	0.61	0.65	**1.00**	+17.82%	0.80	+4.85%	**0.92**	+6.06%
JSon	0.58	0.64	0.65	0.86	**1.00**	+49.19%	**1.00**	+33.72%	**1.00**	+32.02%
JSoup	0.37	0.43	0.42	0.56	**1.00**	+51.18%	**1.00**	+31.74%	**1.00**	+33.73%
Scribe	0.83	0.85	0.87	0.85	**1.00**	+1.76%	0.48	+0.02%	**0.00**	-2.41%
Spark	0.38	0.40	0.40	0.50	**1.00**	+31.38%	**1.00**	+25.39%	**1.00**	+24.32%
Async-HTTP-Client	0.55	0.64	0.65	0.65	**1.00**	+18.90%	0.80	+1.32%	0.80	+0.14%
SpringSide	0.47	0.47	0.47	0.50	0.60	+5.90%	0.60	+5.62%	0.60	+6.13%

coverage, and this coverage gradually increases with each commit. The coverage increase is also confirmed when looking at the results throughout history; to this extent, Table 6.5 summarises the results similarly to the previous experiment, and compares against the baseline strategies *Simple*, *Budget*, and *Budget & Seeding*. In all projects, the coverage was higher than using the *Simple* strategy (only on `SpringSide` is this result not significant). Compared to *Budget*, there is an increase in all projects (significant for four) but for `Scribe`, coverage is essentially the same. Compared to *Budget & Seeding*, there is a significant increase in five projects, and an insignificant increase in two projects. Interestingly, for the `Scribe` project *History* leads to significantly lower coverage (-2%) than *Budget & Seeding*. This shows that seeding of input values is very beneficial on `Scribe` (where 69% of the classes have dependencies on other classes in the project), and indeed on average the benefit of input *Seeding* is higher than the benefit of the *History* strategy. However, in principle *History* can also be combined with *Budget & Seeding*.

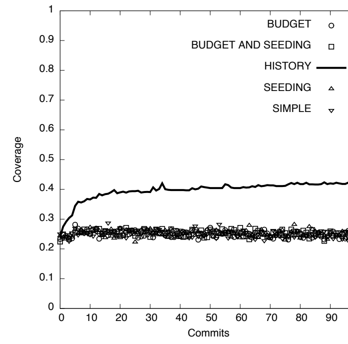
RQ4: CTG achieves higher coverage than testing each project version individually, and coverage increases further over time.

Figure 6.2 shows the time spent on test generation. Note that the strategies (*Simple*, *Seeding*, *Budget*, *Budget & Seeding*) were always configured to run with the same fixed amount of time. During the first call of CTG, the same amount of time was consumed for the *History* strategy, but during successive commits this time reduces gradually as fewer classes need further testing.

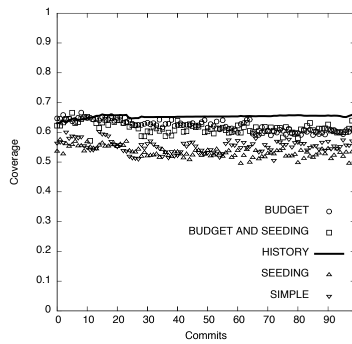
RQ5: CTG reduces the time needed to maximise the code coverage of unit test suites for entire projects.



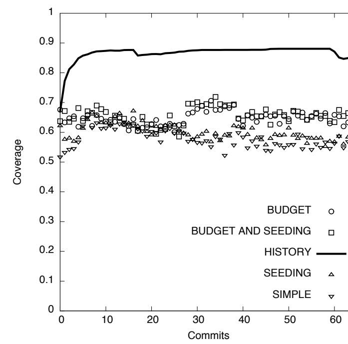
(a) Async-HTTP-Client.



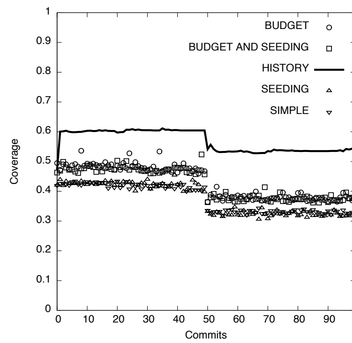
(b) HTTP-Request.



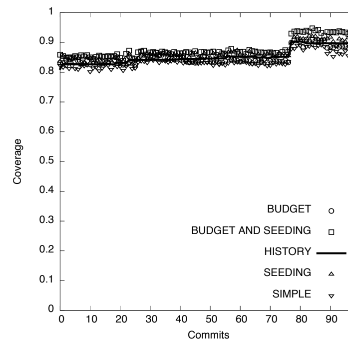
(c) Joda Time.



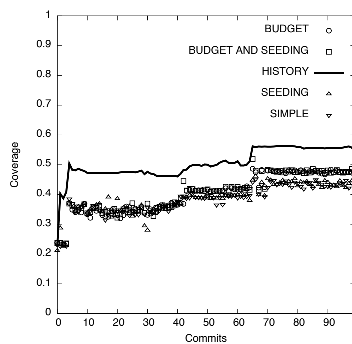
(d) JSON.



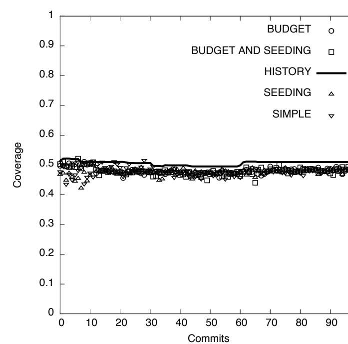
(e) JSoup.



(f) Scribe.

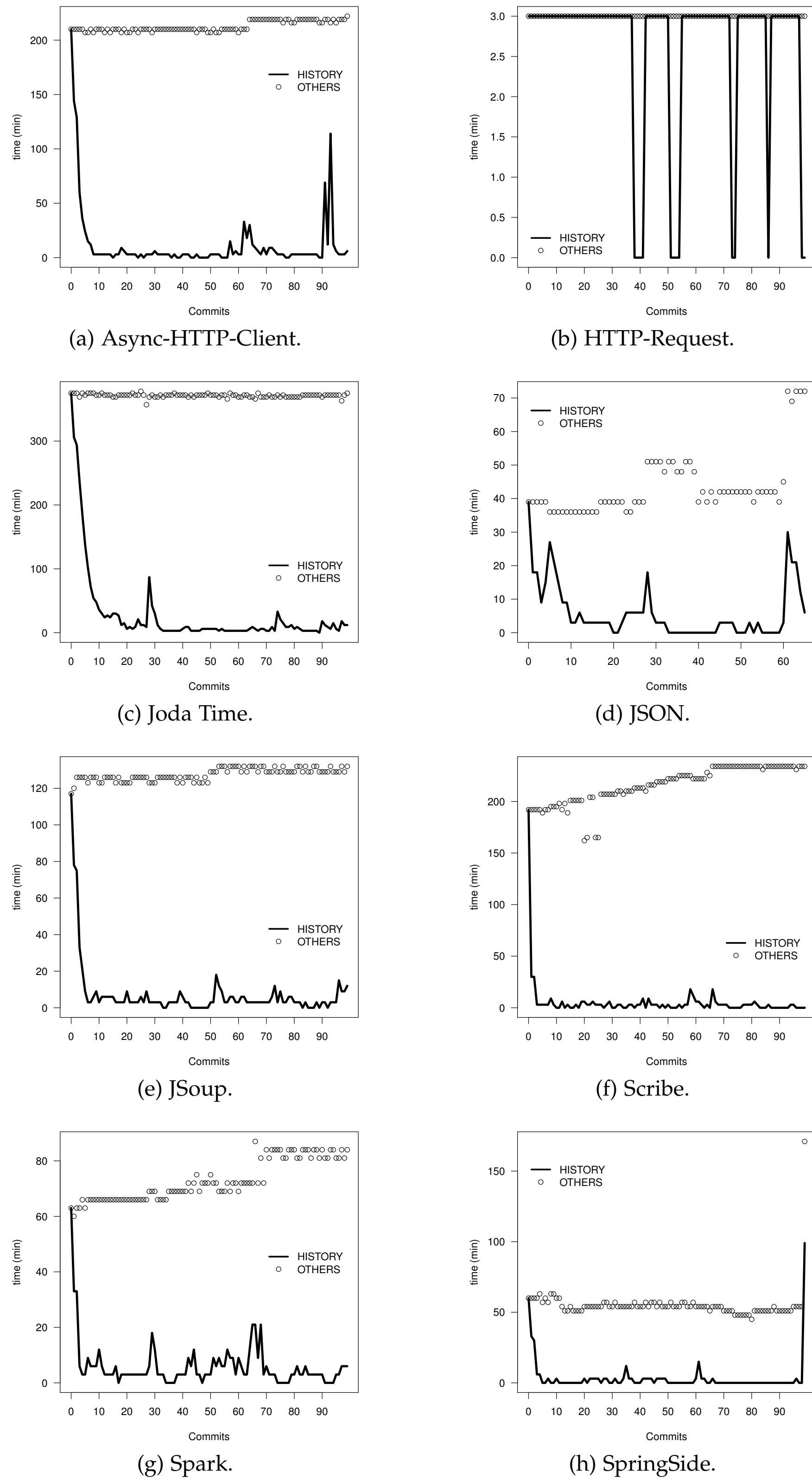


(g) Spark.



(h) SpringSide.

Figure 6.1: Branch coverage results over the course of 100 commits for the GitHub open source case study.



(a) Async-HTTP-Client.

(b) HTTP-Request.

(c) Joda Time.

(d) JSON.

(e) JSoup.

(f) Scribe.

(g) Spark.

(h) SpringSide.

Figure 6.2: Time spent on test generation for the GitHub open source case study over the course of 100 commits.

Let us now look at some of the examples in more detail. `Async-HTTP-Client` exhibits two interesting events during its history (see Figure 6.1a): From the first commit until commit number 63 all strategies have a constant coverage value. At commit 64, 20 classes were changed and three new classes were added. Although this affected the coverage of *History* and also other strategies, *History* only increase its time for test generation briefly from 18 minutes at commit 63, to 30 minutes on commit 64, on average (compared to 219 minutes for a full test generation run). Figure 6.2a further shows a large increase of the test generation time at commit 93, although the coverage does not visibly change. In this commit, several classes were changed at that time, but only *cosmetic* changes happen to the source code (commit message "Format with 140 chars lines"). As EvoSuite apparently had already reached its maximum possible coverage on these classes, no further increase was achieved. We can observe similar behaviour in the plots of `JSoup` (Figure 6.1e), where a major change occurred at commit 50 with the introduction of a new class (`org.jsoup.parser.TokeniserState`), which adds 774 new branches to the 2,594 previously existing branches. `JSoup` project at commit 50 also has a slightly coverage reduction. Although *History strategy* on `JSoup` maintains its coverage between commits 1 and 49, at commit 50 drops to 0.54 (on average). Before that commit the complexity was 2,594 number of branches, but at commit 50 a new class named `org.jsoup.parser.TokeniserState` with a complexity of 774 was added to the project, increasing the global complexity of the project to 3,368 branches, and decreasing its coverage. Although the observed coverage reduction at commit 50 (0.54), this represents 32% (on average) more coverage than the second highest coverage at that commit (0.36 from *Budget and Seeding*). In Figure 6.2e, we observed the same time reduction as on Async-HTTP-Client project.

The `HTTP-Request` subject reveals a nice increase over time, although the time plot (Figure 6.2b) shows only small improvement (13% less time in total). This is because this project consists only of a single class. Consequently, most commits will change that particular class, leading to it being tested more often. In the commits where the class was not tested, no source code changes were performed (e.g., only test classes or other project files were changed, not source code). Thus, `HTTP-Request` is a good example to illustrate how using previous test suites for seeding gradually improves test suites over time, independently of the time spent on the class. Because this project has only one class, the *Seeding* strategy has similar results (on average) to the *Simple* strategy. A similar behaviour can also be observed for `JSON` (see Figure 6.1d), where *History* leads to a good increase in coverage over time. There is a slight bump in the coverage plot at commit 61 (Figure 6.2d), where 13 new classes were added to the project.

`JodaTime`, `Scribe`, and `SpringSide` are examples of projects with only a small increase in coverage (Figures 6.1c, 6.1f and 6.1h, respectively). Although these projects differ in size, it seems that their classes are all relatively easy for EvoSuite, such that additional time or the seeding has no further beneficial effect. For example, 72% of the classes in `SpringSide` have less than 17 branches. However, in all three cases the reduction in test generation time is very large (Figures 6.2c, 6.2f and 6.2h respectively).

Finally, `Spark` shows interesting behaviour where *all* approaches lead to increased coverage over the course of time (Figure 6.1g). This is because during the observed time window of 100 commits the project was heavily refactored. For example, some complex classes were converted into several simpler classes, increasing the time spent for non-History based strategies (Figure 6.2g), up to a maximum of 84 minutes on the last commit. This project also illustrates nicely why applying seeding blindly does not automatically lead to better results: For example, at commit 30 there are only 9 out of 25 classes that actually have dependencies, and many of the dependencies are on the class `ResponseWrapper` — which EvoSuite struggles to cover. As a consequence, there is no improvement when using seeding. This suggests that there is not a single optimal seeding strategy, but that seeding needs to take external factors such as dependencies and achieved coverage into account.

6.5 RELATED WORK

Continuous test generation is closely related to *test suite augmentation*: Test suite augmentation is an approach to test generation that considers code changes and their effects on past test suites. Some test suite augmentation techniques aim to restore code coverage in test suites after changes by producing new tests for new behaviour (e.g. [68]), while other approaches explicitly try to exercise changed code to reveal differences induced by the changes (e.g., [22, 178, 179]); Shamshiri et al. [298] have been extending the EvoSuite tool in this direction. Although test suite augmentation is an obvious application of CTG, there are differences: First, CTG answers the question of *how to implement* test suite augmentation (e.g., how to allocate the computational budget to individual classes). Second, while CTG can benefit from information about changes, it can also be applied *without* any software changes. Third, CTG is not tied to an individual coverage criterion; for example, one could apply CTG such that once coverage of one criterion is saturated, test generation can target a different, more rigorous criterion. Finally, the implementation as part of continuous integration makes it possible to automatically notify developers of any faults found by automated oracles such as assertions or code contracts. Some of the potential benefits of performing test suite aug-

mentation continuously have also been identified in the context of software product-lines [299].

6.6 summary

In this chapter, the scope of unit test generation tools like EvoSuite is extended: Rather than testing classes in isolation, we consider whole projects in the context of continuous integration. This permits many possible optimizations, and our EvoSuite-based prototype provides Continuous Test Generation (CTG) strategies targeted at exploiting complexity and/or dependencies among the classes in the same project. To validate these strategies, we carried out a rigorous evaluation on a range of different open source and industrial projects, totalling 2,061 classes. The experiments overall confirm significant improvements on the test data generation: up to +58% for branch coverage and up to +69% for thrown undeclared exceptions, while reducing the time spent on test generation by up to +83%.

Although our immediate objective in our current experiments lies in improving the quality of generated test suites, we believe that the use of CTG could also have more far reaching implications. For example, regular runs of CTG will reveal testability problems in code, and may thus lead to improved code and design. The use of CTG offers great incentive to include assertions or code contracts, which would be automatically and regularly exercised.

To reduce the gap between what approaches are proposed by researchers and what is actually used by engineers in practice, in the next chapter we present three new plugins for the EvoSuite tool. These plugins allow praticioners to simple execute EvoSuite from an IDE, or to use the CTG strategies discussed in this chapter in a continuous integration system.

7

UNIT TEST GENERATION DURING SOFTWARE DEVELOPMENT: EVOSUITE PLUGINS FOR MAVEN, INTELLIJ AND JENKINS

ABSTRACT

Different techniques to automatically generate unit tests for object oriented classes have been proposed, but how to integrate these tools into the daily activities of software development is a little investigated question. In this chapter, we report on our experience in supporting industrial partners by introducing the EvoSuite automated JUnit test generation tool in their software development processes. The first step consisted of providing a plugin to the Apache Maven build infrastructure. The move from a research-oriented point-and-click tool to an automated step of the build process has implications on how developers interact with the tool and generated tests, and therefore, we produced a plugin for the popular IntelliJ Integrated Development Environment (IDE). As build automation is a core component of Continuous Integration (CI), we provide a further plugin to the Jenkins CI system, which allows developers to monitor the results of EvoSuite and integrate generated tests in their source tree. In this chapter, we discuss the resulting architecture of the plugins, and the challenges arising when building such plugins. Although the plugins described are targeted for the EvoSuite tool, they can be adapted and their architecture can be reused for other test generation tools as well.

7.1 Introduction 131
7.2 Unit Test Generation in Build Automation 133
7.3 IDE Integration of Unit Test Generation 137
7.4 Continous Test Generation 139
7.5 Lessons Learnt 142
7.6 Summary 147

7.1 INTRODUCTION

As described in the previous chapters, the EvoSuite tool automatically generates JUnit tests for Java software [9, 113, 130, 189]. Given a Class Under Test (CUT), EvoSuite creates sequences of calls that maximise testing criteria such as line and branch coverage, while at the same time generating JUnit assertions to capture the current behaviour of the CUT. Although our previous experiments on open-

source projects and industrial systems have shown that EvoSuite can successfully achieve good code coverage — how should it be integrated in the development process of the software engineers?

In order to answer this question, the interactions between a test generation tool and a software developer have been subjected to a number of different controlled empirical studies and observations [134, 196, 300]. However, the question of integrating test generation into the development process goes beyond the interactions of an individual developer with the tool: In an industrial setting, several developers work on the same, large code base, and a test generation tool should smoothly integrate into the current processes and tool chains of the software engineers.

There are different ways of using the EvoSuite test generation tool. The most basic way of doing it, is by running EvoSuite from the command line. If the tool is compiled and assembled in a standalone executable jar (e.g., `evosuite.jar`), then it can be called on a CUT (e.g., `org.Foo`) as follows:

```
$ java -jar evosuite.jar org.Foo
```

However, in a typical Java project the full classpath needs to be specified (e.g., as a further command line input). This is necessary to tell the tool where to find the bytecode of the CUT and of all its dependency classes. For example, if the target project is compiled in a folder called `build`, then to execute EvoSuite on the CUT, one can use the following command:

```
$ java -jar evosuite.jar -class org.Foo -projectCP build
```

where the option `-class` is used to specify the CUT, and the option `-projectCP` is used for specifying the classpath.

This approach works fine if EvoSuite is used in a "static" context, e.g., when the classpath does not change, and a user tests the same specific set of classes several times. A typical example of such a scenario is the running of experiments on a set of benchmarks in an academic context [200] — which is quite different from an industrial use case. An industrial software system might have hundreds, if not thousands, of entries on the classpath, which might frequently change when developers push new changes to the source repository (e.g., Git, Mercurial or SVN). Thus, manually specifying long classpaths for every single submodule is not a viable option.

Usability can be improved by integrating the test generation tool directly into an IDE. For example, EvoSuite has an Eclipse plugin [9] which includes a jar version of EvoSuite. Test generation can be activated by the developer by selecting individual classes, and the classpath is directly derived from the APIs of the IDE itself. However, this approach does not scale well to larger projects with many classes and frequent changes. Furthermore, EvoSuite requires changes to

the build settings that have to be consistent for all developers of a software project, as EvoSuite's simulation of the *environment* of the CUT requires inclusion of a dependency jar file (containing mocking infrastructure, for example, the Java API of the file system [301] and networking [302]).

To overcome these problems, we have developed a set of plugins for common software development infrastructure in industrial Java projects. In particular, in this chapter we present a plugin to control EvoSuite from Apache Maven [303] (Section 7.2), as well as plugins for IntelliJ IDEA [304] (Section 7.3), and Jenkins CI [266] (Section 7.4) to interact with the Apache Maven plugin. Additionally, in Section 7.5 we discuss lessons we learnt while developing those plugins, and finally, we summarise the chapter in Section 7.6.

7.2 UNIT TEST GENERATION IN BUILD AUTOMATION

Nowadays, the common standard in industry to compile and assemble Java software is to use automated build tools. Maven is perhaps the currently most popular one (an older one is Ant, whereas the more recent Gradle is currently gaining momentum). Integrating a test generation tool into an automated build tool consists of supporting execution of generated tests, as well as generation of new tests.

7.2.1 *Integrating Generated Tests in Maven*

In order to make tests deterministic and isolate them from the environment, EvoSuite requires the inclusion of a runtime library [301]. When using a build tool like Maven, it is easy to add third-party libraries. For example, the runtime dependency for the generated tests of EvoSuite can be easily added (and automatically downloaded) for example by copy&pasting the following entry into the `pom.xml` file defining the build:

```
<dependency>
  <groupId>org.evosuite</groupId>
  <artifactId>evosuite-standalone-runtime</artifactId>
  <version>1.0.5</version>
  <scope>test</scope>
</dependency>
```

Once this is set, the generated tests can use this library, which is now part of the classpath. This is important because, when a software project is compiled and packaged (e.g., with the command `mvn package`), all the test cases are executed as well to validate the build.

However, when we generated test cases for one of our industrial partners for the first time, building the target project turned into

mayhem: some generated tests failed, as well as some of the existing manual tests (i.e, the JUnit tests manually written by the software engineers), breaking the build. The reason is due to how classes are instrumented: The tests generated by EvoSuite activate a Java Agent to perform runtime bytecode instrumentation, which is needed to replace some of the Java API classes with our mocks [301]. The instrumentation is done when the tests are run, and can only be done when a class is loaded for the first time. On one hand, if the manual existing tests are run first before the EvoSuite ones, the bytecode of the CUTs would be already loaded, and instrumentation cannot take place, breaking (i.e., causing them to fail) all the generated tests depending on it. On the other hand, if manual tests are run last, they will use the instrumented versions, and possibly fail because they do not have the simulated environment configured for them.

There might be different ways to handle this issue, as for example forcing those different sets of tests to run on independently spawned JVMs. However, this might incur some burden on the software engineers' side, who would need to perform the configuration, and adapt (if even possible) all other tools used to report and visualise the test results (as we experienced). Our solution is twofold: (1) each of our mocks has a rollback functionality [302], which is automatically activated after a test is finished, so running manual tests after the generated ones is not a problem; (2) we created a *listener* for the Maven test executor, which forces the loading and instrumentation of all CUTs before *any* test is run, manual tests included. Given this solution, engineers can run all the tests in any order, and in the same classloader/JVM. This is achieved by simply integrating the following entry into the `pom.xml` where the Maven test runner is defined (i.e., in `maven-surefire-plugin`):

```
<property>
 <name>listener</name>
 <value>org.evosuite.runtime.InitializingListener</value>
</property>
```

7.2.2 *Generating Tests with Maven*

The configuration options discussed so far handle the case of running generated tests, but there remains the task of generating these tests in the first place. Although invoking EvoSuite on the local machine of a software engineer from an IDE may be a viable scenario during software development, it is likely not the best solution for legacy systems. When using EvoSuite for the first time on a large industrial software system with thousands of classes, it is more reasonable to run EvoSuite on a remote dedicated server, as it would be a very time consuming task. To simplify the configuration of this (e.g., to

avoid manually configuring classpaths on systems with dozens of `pom.xml` files in a hierarchy of submodules) and to avoid the need to prepare scripts to invoke EvoSuite accordingly, we implemented a Maven plugin with an embedded version of EvoSuite. For example, to generate tests for all classes in a system using 64 cores, a software engineer can simply type:

```
$ mvn -Dcores=64 evosuite:generate
```

To get an overview of all execution goals, the EvoSuite Maven plugin can be called as follows:

```
$ mvn evosuite:help
```

or as follows:

```
$ mvn evosuite:help -Ddetail=true -Dgoal=generate
```

to get the list of parameters of, e.g., the `generate` goal. In particular, it is possible to configure aspects such as number of cores used (`cores`), memory allocated (`memoryInMB`), or time spent per class (`timeInMinutesPerClass`).

It is further possible to influence how the time is allocated to individual classes using the `strategy` parameter. For instance, the `simple` strategy described in Section 6.2.1 allocates the time specified in the `timeInMinutesPerClass` per class. By default, EvoSuite used the `budget` strategy described in Section 6.2.2, which allocates a time-budget proportional to the complexity of a class. As a proxy to complexity, EvoSuite uses the number of branches to determine whether class A is more complex than class B. That is, classes with more branches have more time available to be tested. First, EvoSuite determines the maximum and the minimum time budget available. The minimum time budget is the minimum time per class (by default 1 minute) multiplied by the total number of classes. The maximum time budget is `timeInMinutesPerClass` multiplied by the total number of classes. The difference between maximum and minimum time budget is called `extraTime` and it is used to give more time to more complex classes. Assuming there is an `extraTime` of e, the time budget per branch is $\frac{e}{|branches|}$. Then, each CUT C has a time budget of $minTimeBudgetPerClass + (timePerBranch \times |branches_C|)$. Further implemented strategies are the experimental `seeding` strategy (Section 6.2.3), where EvoSuite tries to test classes in the order of dependencies to allow re-use of Java objects, and the `history` strategy (Section 6.3), where EvoSuite exploits the fact the projects evolve over time. To get an overview of tests generated so far, one can use:

```
$ mvn evosuite:info
```

By default, EvoSuite stores tests in the `.evosuite/evosuite-tests` hidden folder. Once the developer has inspected the tests and decided

to integrate them into the source folder, this can be done using the following command:

```
$ mvn evosuite:export
```

The export command copies the generated tests to another folder, which can be set with the `targetFolder` option (default value is `src/test/java`). Tests will only be executed by the `mvn test` command once they are in `src/test/java` (unless Maven is configured otherwise). Once exported, the coverage of manually written (if any) and automatically generated tests can be measured using EvoSuite. This can be done using the following command:

```
$ mvn evosuite:coverage
```

The coverage command instruments all classes under `src/main/java` (for typical Maven projects) and runs all test cases from `src/test/java`. EvoSuite executes all test cases using the JUnit API on all classes, and determines the coverage achieved on all of EvoSuite's target code coverage criteria. Future improvements of this option will try to re-use `maven-surefire`[1] plugin to run the test cases instead of directly using the JUnit API.

To enable the EvoSuite plugin, the software engineer would just need to copy&paste the following plugin declaration to the root `pom.xml` file:

```
<plugin>
  <groupId>org.evosuite.plugins</groupId>
  <artifactId>evosuite-maven-plugin</artifactId>
  <version>1.0.5</version>
  <executions>
    <execution>
      <goals>
        <goal>prepare</goal>
      </goals>
      <phase>process-test-classes</phase>
    </execution>
  </executions>
</plugin>
```

By doing this, there is no further need to do any installation or manual configuration: Maven will automatically take care of it. Note: if a plugin is not in the Maven Central Repository [306] one needs to add the URL of the server where the plugin is stored, but that needs to be done just once (e.g., in a corporate cache using a repository manager like Nexus [307]).

Once the EvoSuite Maven plugin is configured by editing the `pom.xml` file (which needs to be done only once), if an engineer wants to

1 The Maven Surefire plugin [305] is used during the test phase of a maven project to execute all unit tests.

generate tests on a new server, then it is just a matter of uploading the target system there (e.g., `git clone` if Git is used as source repository manager), and then executing `mvn evosuite:generate`. That is all that is needed to generate tests with EvoSuite's default configuration (some parameters can be added to specify the number of cores to use, for how long to run EvoSuite, if only a subset of classes should be tested, etc.).

7.3 IDE INTEGRATION OF UNIT TEST GENERATION

Once the generated unit tests require a runtime dependency to run, embedding EvoSuite within an IDE plugin (as in the past we did for Eclipse) becomes more difficult because of potential EvoSuite version mismatches: the IDE plugin could use version X, whereas the project could have dependency on Y. Trying to keep those versions aligned is not trivial: a software engineer might work on different projects at the same time, each one using a different version; a software engineer pushing a new version update in the build (e.g., by changing the dependency version in the `pom.xml` file and then committing the change with Git) would break the IDE plugin of all his/her colleagues, who would be forced to update their IDE plugin manually; etc.

Our solution is to keep the IDE plugin as lightweight as possible, and rely on the build itself to generate the tests. For example, the IDE plugin would just be used to select which are the CUTs, and what parameters to use (e.g., how long the search should last). Then, when tests need to be generated, the IDE plugin just spawns a process that calls `mvn evosuite:generate`. By doing this, it does not matter what version of EvoSuite the target project is configured with, and updating it will be transparent to the user. Furthermore, every time a new version of EvoSuite is released, there is no need to update the IDE plugin, just the `pom.xml` file (which needs to be done only once and just by one engineer).

However, to achieve this, the interfaces between the IDE plugin and the Maven plugin need to be *stable*. This is not really a problem in automated test data generation, because in general there are only few parameters a user is really interested into: for what CUTs should tests be generated for, and what resources to use (e.g., memory, CPU cores, and time).

This approach worked well for some of our industrial partners, but not for all of them: For example, some use Gradle to build their software rather than Maven. Furthermore, relying on a build tool does not work when no build tool is used, e.g. when a new project is created directly in the IDE. To cope with the issue of handling build tools for which we have no plugin (yet), or handling cases of no build tool

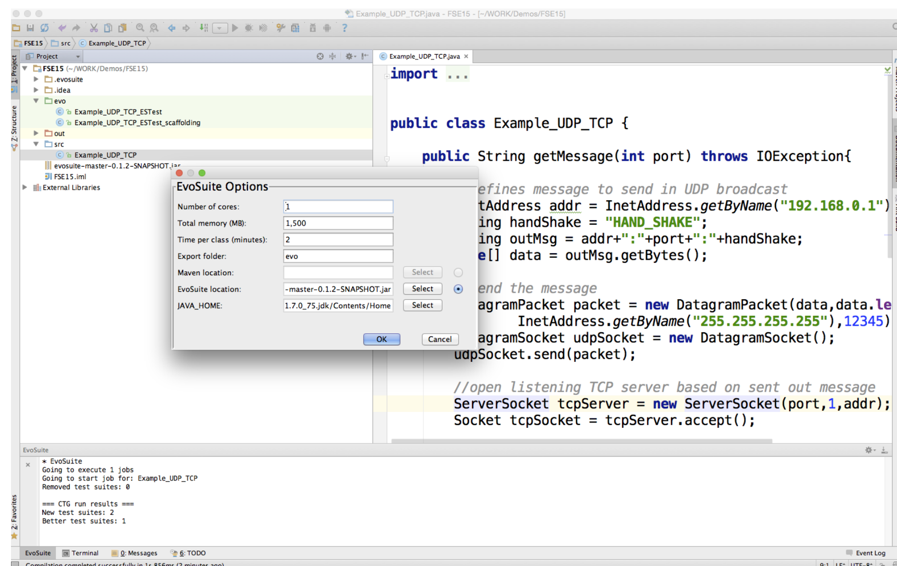


Figure 7.1: Screenshot of the EvoSuite plugin for IntelliJ IDEA, when applied on the code example from Arcuri et al. [302].

at all, we also found it necessary to have the option of using an external command line EvoSuite executable, which the IDE plugin calls on a separate spawned process. As the corresponding jar file does not need to be part of the build, it can be simply added directly to the source repository (e.g., Git) without needing to change anything regarding how the system is built. In this way, all developers in the same project will use the same version, and do not need to download and configure it manually.

Regarding the runtime dependency for the generated tests, this is not a problem for build tools like Ant/Ivy and Gradle, as they can make use of Maven repositories. However, when no build tool is employed, the runtime dependency needs to be added and configured manually (as for any other third-party dependency). Note: the EvoSuite executable could be used as runtime dependency as well (it is a superset of it), but it would bring many new third-party libraries in the build. This might lead to version mismatch problems if some of these libraries are already used in the project.

This architecture is different from what we originally had considered for our Eclipse plugin. To experiment with it, we started a new plugin for a different IDE, namely IntelliJ IDEA. This was further motivated by the fact that most of our industrial partners use IntelliJ and not Eclipse. Figure 7.1 shows a screenshot of applying EvoSuite to generate tests for the motivating example used in [302]. A user can select one or more classes or packages in the project view, right click on them, and start the action `Run EvoSuite`. This will show a popup dialog, in which some settings (e.g., for how long to run EvoSuite) can be chosen before starting the test data generation. Progress is shown in a tool window view.

While working on the IntelliJ plugin we found out that, in general, embedding and executing a test data generation tool on the same JVM of the plugin (as we did with Eclipse) is not a viable option. If you are compiling a project with Java 8, for example, that does *not* mean that the IDE itself is running on Java 8 (recall that IDEs like IntelliJ, Eclipse and NetBeans are written in Java and execute in a JVM). For example, up to version 14, IntelliJ for Mac used Java 6, although IntelliJ can be used to develop software for Java 8. The reason is due to some major performance GUI issues in the JVM for Mac in both Java 7 and Java 8. An IDE plugin will run in the same JVM of the IDE, and so needs to be compiled with a non-higher version of Java. In our case, as EvoSuite is currently developed/compiled for Java 8, calling it directly for the IDE plugin would crash it due to bytecode version mismatch. Thus, the test data generation tool has to be called on a spawned process using its own JVM.

7.4 CONTINOUS TEST GENERATION

Although generating tests on demand (e.g., by directly invoking the Maven/IntelliJ plugins) on the developer's machine is feasible, there can be many reasons for running test generation on a remote server. In particular, running EvoSuite on many classes repeatedly after source code changes might be cumbersome. To address this problem, we introduced the concept of Continuous Test Generation (CTG) in Section 6.3, where Continous Integration (CI) (e.g., Jenkins and Bamboo) is extended with automated test generation. In a nutshell, a remote server will run EvoSuite at each new code commit using the EvoSuite Maven plugin. Only the new tests that improve upon the existing regression suites will be sent to the users using a plugin to the Jenkins CI system.

7.4.1 *Invoking EvoSuite in the Context of CTG*

To repeatedly invoke EvoSuite during CTG, the `history` strategy needs to be set on the Maven plugin. As previously described in Section 6.3.1, this strategy changes the budget allocation such that more time is spent on new or modified classes than old classes, under the assumption that new or modified code is more likely to be faulty [282]. Furthermore, instead of starting each test generation from scratch, the `history` strategy re-uses previously generated test suites as a seed when generating the initial population of the Genetic Algorithm, to start test generation with some code coverage, instead of trying to cover goals already covered by a previous execution of CTG (see Section 6.3.2 for more details).

The EvoSuite Maven plugin creates a folder called `.evosuite` under the project folder where all the files generated and/or used dur-

ing test generation are kept. To be independent of any Source Control Management (SCM), we have implemented a very simple mechanism to check which classes (i.e., Java files) have changed from one commit to another one. Under `.evosuite`, CTG creates two files: `hash_file` and `history_file`. Both files are based on a two column format, and are automatically created by the EvoSuite Maven plugin. The first file contains as many rows as there are Java files in the Maven project, and each row is composed of the full path of each Java file and its hash. The hash value allows EvoSuite to determine whether a Java file has been changed. Although this precisely identifies which Java files (i.e., classes) have been changed, it does not take into account whether the change was in fact a source change or just, for example, a JavaDoc change. In Section 8.2 we propose different strategies that could be explored in the future to improve this feature. The second file (`history_file`) just keeps the list of new/modified classes. A class is considered *new* if there is no record of that class on the `hash_file`. A class is considered as *modified*, if its current hash value is different from the value on `hash_file`. Similar to Git output, the first column of `history_file` is the status of the Java file: "A" means added, and "M" means modified. The second column is the full path of the Java file. Each CTG call also creates a temporary folder (under the `.evosuite` folder) named with the format `tmp_year_month_day_hour_minutes_seconds`. All files (such as .log, .csv, .java files, etc) generated by EvoSuite during each test generation will be saved in this temporary folder.

At the end of each test generation, the best test suites will be copied to a folder called `best-tests` under `.evosuite`. This folder will only contain test suites that improve over already existing tests, i.e., manually written (if any) and automatically generated tests. In order to be copied to the `best-tests` folder, a test suite for a CUT needs to either be (a) generated for a class that has been added or modified, (b) achieve a higher code coverage than the existing tests, or (c) cover at least one additional coverage goal that is not covered by the existing tests.

7.4.2 *Accessing Generated Tests from Jenkins*

Once CTG is part of the build process (e.g., through the Maven plugin), then integrating it in a CI system becomes easier. We have developed a plugin for the Jenkins CI system which allows developers to:

– Visualize code coverage and time spent on test generation;

– Get statistic values such as coverage per criterion, number of testable classes, number of generated test cases, total time spent on test generation per project, module, build, or class;

– View the source code of the generated test suites per class;

– Commit and push the new generated test suites to a specific branch[2].

The Jenkins plugins relies on information produced by the underlying Maven plugin, which generates a file named `project_info.xml` with detailed information. Consequently, reproducing the functionality of the Jenkins plugin for other CI platforms should be straightforward.

Currently, the EvoSuite Jenkins plugin is available for download on our webpage at `www.evosuite.org/downloads`. To install it, the administrator of a Jenkins instance has to go to "Manage Jenkins" menu, and then "Manage Plugins" option. On the "Advanced" tab there are three different options to install plugins: "HTTP Proxy Configuration", "Upload Plugin", and "Update Site". The "Upload Plugin" option should be used to upload and install the `evosuite.hpi` file previously downloaded from our webpage. Once installed, the EvoSuite Jenkins plugin runs as a "post-build" step, in which the output of the EvoSuite Maven plugin is displayed on the CI web interface. This is similar to the type of architecture used by other plugins such as Emma [272] (a widely used Java tool for code coverage): the Emma Maven plugin needs to be added to the `pom.xml` project descriptor, and then it needs to be called as part of the CI build. To enable the EvoSuite Jenkins plugin, users just have to access the "configure" page of their project and add EvoSuite as one of the "post-build" actions. As shown in Figure 7.2, there are three options to configure EvoSuite:

1. *Automatic commits*: The plugin can be configured to automatically commit newly generated test suites to the Git repository. If this option is deactivated, then the generated test suites will remain on the CI system and users can still use the CI web interface to access the generated test suites of each class.

2. *Automatic push*: The plugin can be configured to automatically push commits of generated tests to a remote repository.

3. *Branch name*: To minimise interference with mainstream development, it is possible to let the plugin push to a specific branch of the repository.

Consequently, when the development team of a project is already running a CI server like Jenkins and is using a build tool like Maven, then adding and configuring the EvoSuite Jenkins plugin is a matter of a few minutes. This is in fact an essential property of a successful technology transfer from academic research to industrial practice, as

2 At the time of writing this chapter, EvoSuite just supported Git repositories.

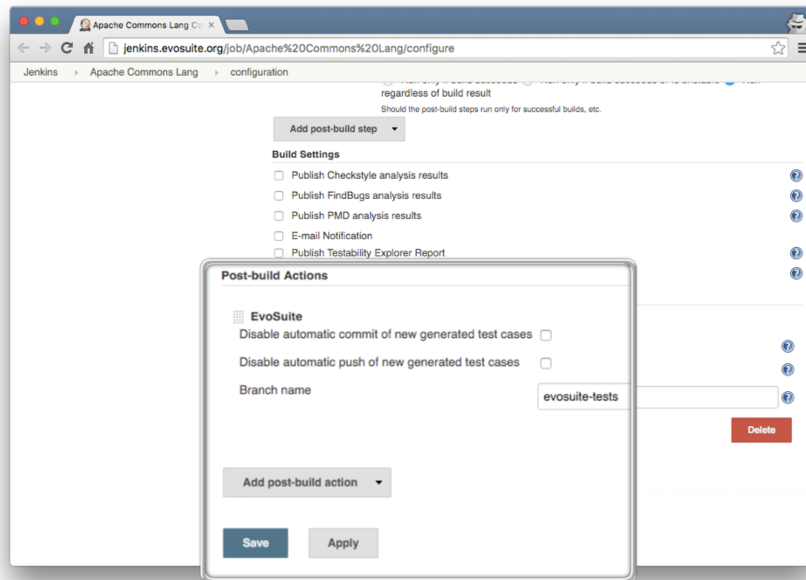


Figure 7.2: Configuring the EvoSuite Jenkins plugin.

"trying out" those novel techniques becomes a simple, low risk activity. At any rate, support for other CI systems (e.g., Bamboo) and build tools (e.g., Gradle) is possible and should be developed in the future to support more projects / systems.

Once configured, and after the first execution of CTG on the project under test, a coverage plot will be shown on the main page of the project, as shown in Figure 7.3. In this plot, the x-axis represents the commits, and y-axis represents the coverage achieved by each criterion. The plot is clickable and redirects users to the selected build (see Figure 7.4).

On the project dashboard, users also have access to a button called "EvoSuite Project Statistics", which redirect them to a statistics page, where the overall coverage, the coverage per criterion, and the time spent on test generations is reported (see Figure 7.5). Similarly, on the build and module pages (and in addition to coverage values) the number of generated test cases is also reported. On the class page (see Figure 7.6) users can also view the source code of the generated test suite.

7.5 LESSONS LEARNT

While developing the plugins for Maven, IntelliJ IDEA and Jenkins, we learnt several important lessons, which we discuss in this section.

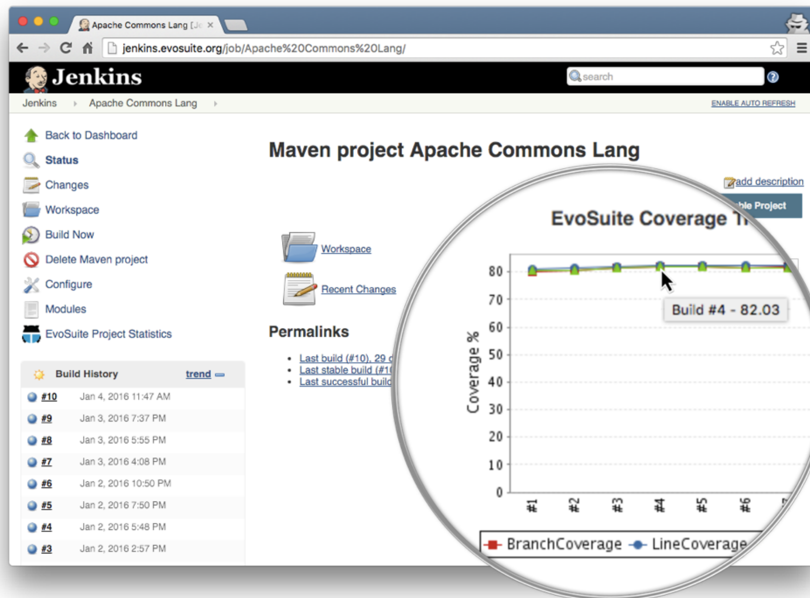


Figure 7.3: Jenkins dashboard with EvoSuite plugin applied on Apache Commons Lang project.

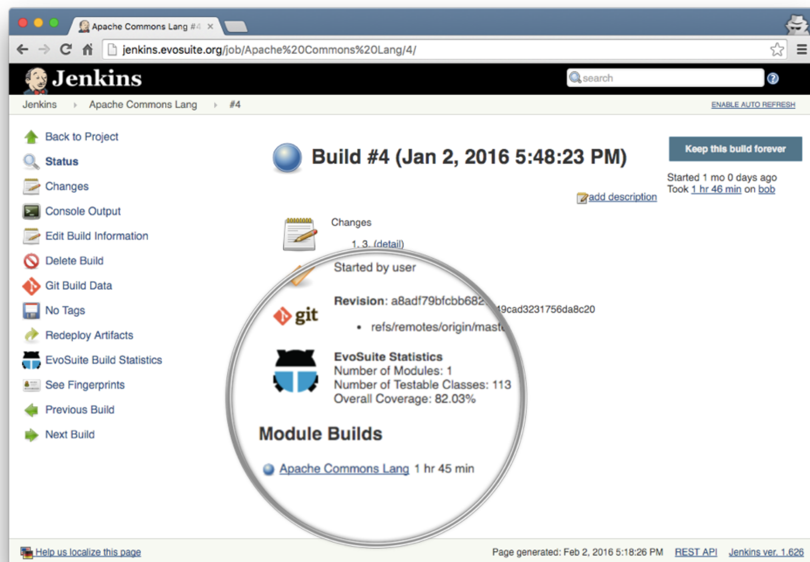


Figure 7.4: Jenkins build dashboard with EvoSuite statistics like, for example, *number of testable classes*, or *overall coverage*.

7.5.1 *Lightweight Plugins*

Developing a plugin is usually a very time consuming and tedious task — not necessarily because of specific technical challenges, but rather due to a systematic lack of documentation. Most tools we anal-

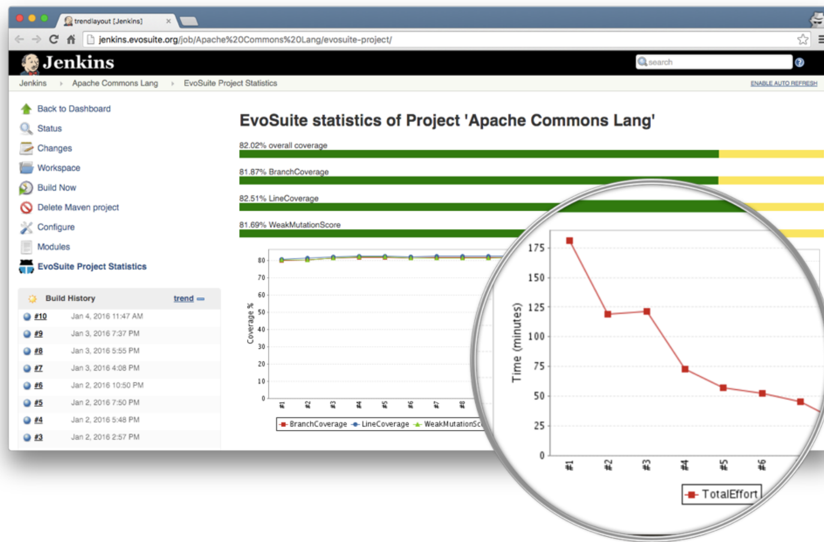


Figure 7.5: EvoSuite statistics such as *overall and coverage achieved by each criterion*, and *time spend on generation* of a project.

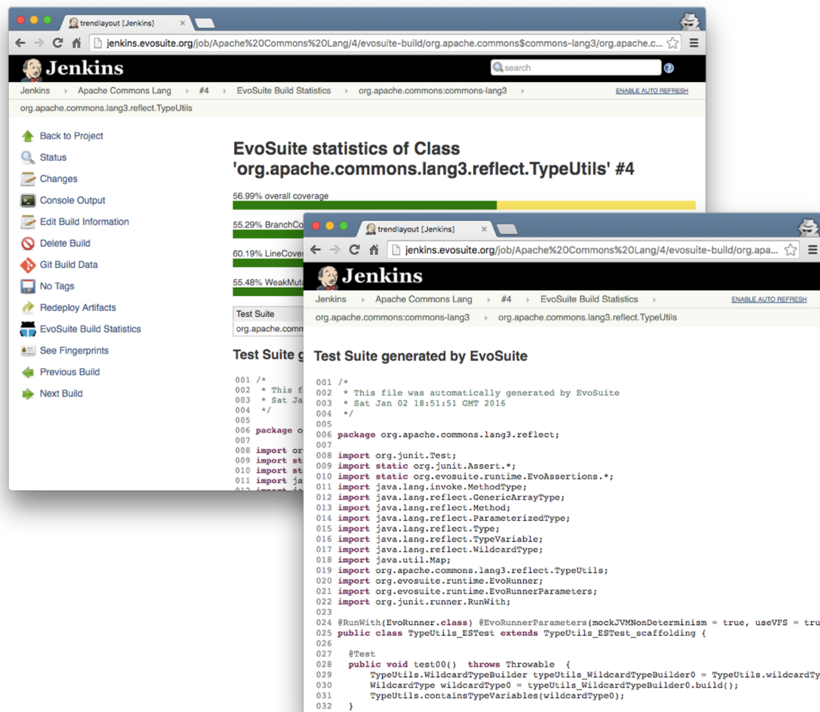


Figure 7.6: EvoSuite statistics of a class and the source code of the generated test suite.

ysed provide some tutorials on how to write plugins, but these are very basic. API documentation in form of JavaDocs is usually very scarce, if it exists at all. For example, at the time of writing this chapter, IntelliJ IDEA does not even have browsable JavaDoc documenta-

tion. The "recommended" way to learn how to develop plugins for IntelliJ IDEA is to check out its source code, and also to study other already existing open-source plugins for it. The same happened during the development of the Jenkins plugin: Although there are more than 1,000 Jenkins plugins [266] (at the time of writing this chapter) and the documentation to setup the IDE (Eclipse or IntelliJ IDEA) to develop and build a Jenkins plugin is very complete, the documentation of, for example, how to keep data from one build to another is very limited. To our surprise, Jenkins web interface is not created as a typical webpage. I.e., instead of building all the web interface elements from different files (e.g., .html, .css, .js, etc.) every time a page is loaded, Jenkins deserialises all the data previously generated by a build. This is of course a feature that speed up Jenkins, but it took us a while to understand it and properly use it, due to the lack of documentation.

Often, adding even some very basic functionalities requires hours if not days of studying the source code of those tools, or asking questions on their developers' forums (in this regard, IntelliJ's forum was very useful). To complicate matters even more, the APIs of these tools are not really meant for maintainability (e.g., backward compatibility to previous versions, as usually done for widely used libraries), and can drastically change from release to release.

The lesson here is that plugins should be as lightweight as possible, where most of the functionalities should rather be in the test data generation tools. A plugin should be just used to start the test data generation with some parameters, and provide feedback on when the generation is finished, or issue warnings in case of errors.

Another lesson learnt is that, at least in our cases, it pays off to run the test data generation tools in a separated JVM. This is not only for Java version mismatch issues (recall Section 7.3), but also for other technical details. The first is related to the handling of classloaders: EvoSuite heavily relies on classloaders, for example to load and instrument CUTs, and also to infer the classpath of the JVM that started EvoSuite automatically (this is needed when EvoSuite spawns client processes). When run from command line, the classloader used to load EvoSuite's entry point would be the system classloader, which usually is an instance of `URLClassLoader`. A `URLClassLoader` can be queried to obtain the classpath of the JVM (e.g., to find out which version of Java was used, and its URL on the local file system). However, this is practically never the case in plugins, where classes are usually loaded with custom classloaders. If a tool relies on the system classloader, then running it inside a plugin will simply fail (as it was in our case with EvoSuite).

Another benefit of running a test data generation tool on a separate process is revealed when there are problems, like a crash or hanging due to an infinite loop or deadlock. If such problems happen in a

spawned process, then that will not have any major side effects on the IDE, and the software engineers will not need to restart it to continue coding. As generating tests is a time consuming activity (minutes or even hours, depending on the number of CUTs), a couple of seconds of overhead due to a new JVM launch should be negligible.

Furthermore, there are more subtle corner cases we encountered: During a demonstration of EvoSuite with the Eclipse plugin, we decided to switch off the wifi connection just a minute before the demo started, in order to avoid other programs interfering with the demo, e.g., an incoming Skype call. Unfortunately, to the amusement of the audience, this had the side effect of making the EvoSuite Eclipse plugin to not working any more, although running EvoSuite from command line was perfectly fine. Following debugging investigations led to us to the culprit: the *localhost* host name resolution. EvoSuite uses RMI to control its spawn client processes. This implies opening a registry TCP port on the local host, which resulted in the IP address of the wifi network card. This mapping was cached in the JVM when Eclipse started. Switching off the wifi did not update the cache, and then EvoSuite, which was running in the same JVM of Eclipse, was using this no longer valid IP address. This problem would not have happened if EvoSuite was started in its own JVM. (Note, however, that a simple fix to this issue was to hardcode the address `127.0.0.1` instead of leaving the default resolution of the *localhost* variable).

7.5.2 *Compile Once, Test Everywhere*

Java is a very portable language. Thanks to Java, we have been able to apply EvoSuite and its plugins on all major operating systems, including Mac OS X, Linux, Solaris and Windows. However, this was not straightforward.

Among academics, Mac and Linux systems are very common. The latter is particularly the case because clusters of Linux computers are often used for research experiments. However, in industry Windows systems are the most common ones, and when we applied EvoSuite it turned out that initially our plugins did not work for that operating system.

A common issue is the handling of file paths, where Mac and Linux use "/" as path delimiter, whereas Windows uses "\". However, this issue is simple to fix in Java by simply using the constant `File.separator` when creating path variables. Another minor issues is the visualisation of the GUI: for example, we noticed some small differences between Mac and Windows in the IntelliJ plugin pop-up dialogs. To resolve this problem one needs to open the plugin on both operating systems, and perform layout modifications until the pop-up dialogs are satisfactory in both systems.

However, there were also some more complex problems. In particular, Windows has limitations when it comes to start new processes: Process cannot take large inputs as parameter (e.g., typically max 8191 characters). In test data generation, large inputs are common, for example to specify the full classpath of the CUT, and the lists of CUTs to test. A workaround is to write such data to a file on disk, and use the name and path of this file as input to the process; the process will then read from this file and apply its configurations. However, this approach does not work for the classpath, as that is an input to the JVM process, and not the Java program the JVM is running. Fortunately, this is a problem faced by all Java developers working on Windows, and there are many forums/blogs discussing workarounds. The solution we chose in EvoSuite is that, when we need to spawn a process using a classpath C, we rather create a "pathing jar" on the fly. A pathing jar is a jar file with no data but a manifest configuration file, where the property `Class-Path` is set to C (after properly escaping it). Then, instead of using C as classpath when spawning a new process, the classpath will just point to the generated pathing jar.

Another major issues we faced when running EvoSuite on Windows is the termination of spawned processes, although this might simply be a limitation of the JVM: Commands like `Process.destroy` (to kill a spawned process) and `System.exit` (to terminate the execution of the current process) do not work reliably on Windows, resulting in processes that are kept on running indefinitely. This is challenging to debug, but fortunately, as it affects all Java programmers working on Windows, there are plenty of forums/blogs discussing it. In particular, in Windows one has to make sure that all streams (*in*, *out* and *err*) between a parent and a spawned process are closed before attempting a `destroy` or a `exit` call.

To be on the safe side and to avoid the possibility of EvoSuite leaving orphan processes, the entry point of EvoSuite (e.g., IntelliJ or Maven plugins) starts a TCP server, and gives its port number as input to all the spawned processes. Each spawned process will connect to the entry point, and check if the connection is alive every few seconds. If the connection goes down for any reason, then the spawned process will terminate itself. This approach ensures that, when a user stops EvoSuite, no spawned process can be left hanging, as the TCP server in the entry point will not exist any more. The benefit of this approach is that it is operating system agnostic, as it does not rely on any adhoc operating-sytem specific method to make sure that no spawned process is left hanging.

7.6 SUMMARY

In this chapter, we presented three plugins we developed for EvoSuite to make it usable from Maven, IntelliJ IDEA and Jenkins. This

was done in order to improve the integration of EvoSuite into the development process for large industrial software projects. We discussed the motivations for our architectural choices, based on our experience in starting to apply EvoSuite among our industrial partners, and presented technical details and lessons learnt.

The architecture of our plugins is not specific to EvoSuite, and could in principle be reused for other test data generation tools, e.g., Randoop [41], jTExpert [190], GRT [308] and T3i [309]. However, to this end we would need to formalize the names of the input parameters (e.g., how to specify the classes to test and how many cores could be used at most) that are passed to those tools, and they would then need to be updated to use this information.

EvoSuite and its plugins are freely available for download. Their source code is released under the LGPL open-source license, and it is hosted on GitHub. For more information, visit our webpage at: `www.evosuite.org`.

8 CONCLUSIONS & FUTURE WORK

In this chapter we first summarise our main contributions, and we then outline how these contributions could be further enhanced.

8.1 Summary of Contributions 149
8.2 Future Work 151

8.1 SUMMARY OF CONTRIBUTIONS

As we previously described in Chapter 1, the problem considered in this thesis is the use of search-based algorithms to automatically generate unit test cases for object-oriented software that is, typically, developed continuously. Although search-based techniques have been successfully applied, their applicability is an open question. In particular, this thesis aims to explore and investigate the following research questions:

– Which coverage criteria shall be used to guide the search in order to produce the best test cases? How can a search-based algorithm efficiently optimise several coverage criteria?

– Which search-based algorithm works best at generating unit tests for single and multiple criteria?

– How can search-based testing be improved to automatically generate unit test cases that are able to detect software faults, and to help developers to find the location of the faulty code?

– Which components (e.g., a class in Java) in a software should be subjected to test generation? In which order should components be tested? How much time should be allocated to test each component?

– How can a test generation tool be integrated in the developers' processes?

The main contributions of this thesis are fivefold, and they are summarised in the following sections.

8.1.1 *Optimisation of Multiple Coverage Criteria*

In Chapter 2 we showed that, typically, search-based test generation approaches use evolutionary search algorithms that are guided by

coverage criteria such as branch coverage or statement coverage to generate tests. However, developers may expect that automated test generation approaches would exercise several properties of the software under test simultaneous, as not even manually-written test cases only aim to, for example, cover all branches. Thus, in Chapter 3, we defined six coverage criteria, and proposed a simple approach to simultaneously optimise these coverage criteria, in addition to the three other criteria previously used independently in the literature (i.e., branch coverage, statement coverage, and weak mutation). An empirical evaluation on 650 open-source Java classes showed that the optimisation of all criteria is effective and efficient in terms of coverage achieved and computational cost required to compute all criteria.

8.1.2 *Evolutionary Algorithms for Test Suite Generation*

Given the different coverage criteria defined in Chapters 2 and 3 and the approach to optimise all of them defined in Chapter 3, in Chapter 4 we performed an empirical evaluation of seven different evolutionary algorithms and two random approaches to understand the influence of each one at optimising test suites for an individual coverage objective (i.e., branch coverage), and for multiple coverage objectives (i.e., all criteria defined in the previous contribution). Our results showed that 1) evolutionary algorithms outperform both random-based techniques, and 2) the MOSA algorithm works better for the test generation problem than the other evolutionary algorithms considered in our study.

8.1.3 *Diagnostic Ability of Automatically Generated Unit Tests*

A software fault can only be detected if and only if there is at least one test case that covers the faulty code with the input that triggers the faulty behaviour. Therefore, given the fact that MOSA was the evolutionary algorithm evaluated in the previous contribution that achieved the highest coverage, in Chapter 5 we performed an evaluation of MOSA and an extended version of MOSA (which optimises the diversity of coverage-based generated test cases) on six real faults. Our results showed that the proposed extension of MOSA is more effective at detecting four out of six faults, and could reduce the time developers spend at localising the faulty code by 25%.

8.1.4 *Continuous Test Generation*

Experiments on automated test generation techniques (as the ones described in Chapters 2 to 4) consist of applying a tool to an entire software project, and to allocate the same amount of time to every component (e.g., class in Java). In practice, even if one would restrict this

test generation to code that has been changed since the last time of test generation, the computational effort (e.g., CPU time and memory used) to generate tests may exceed what developers are prepared to use their own computers for while they are working on them. Therefore, in Chapter 6, we presented a novel approach called Continuous Test Generation (CTG) to alleviate this problem.

CTG is the synergy of automated test generation with continuous integration: Tests could be generated during every nightly build, but resources are focused on the most important classes, and test suites are built incrementally over time. CTG supports the application of *test suite augmentation*, but most importantly: 1) addresses the time-budget allocation problem of individual classes; 2) it is not tied to an individual coverage criterion; 3) it is applicable for incremental test generation, even if the system under test did not change; and 4) it leads to overall higher code coverage while reducing the computational time spent on test generation.

8.1.5 *The EvoSuite Toolset*

EvoSuite is a search-based tool that uses a genetic algorithm to automatically generate test suites for Java classes. By default, EvoSuite provides a command line version and an Eclipse plugin. However, in order to increase its adoption and usage by practitioners, in Chapter 7 we introduced three new plugins for EvoSuite: a plugin for the Apache Maven build infrastructure, for the IntelliJ IDE, and for the Jenkins CI system. Note that, these three plugins provide support for all contributions previously described.

8.2 FUTURE WORK

In this section we suggest several recommendations for future work.

8.2.1 *Coverage Criteria*

Although the optimisation of multiple coverage criteria proposed in Chapter 3 allows the exploration of multiple properties of the software under test, an important question that remains to be answered in the future is, which selection of criteria matches the expectations of practitioners? Are there some criteria that practitioners would be more likely to use than others? To address these questions controlled experiments with real software testers will have to be conducted.

8.2.2 *Hyper-heuristics Search Algorithms*

Considering the variation of results achieved by each evolutionary algorithm evaluated in Chapter 4 with respect to different configurations and classes under test, it would be of interest to use these insights to develop *hyper-heuristics* [310] that select and adapt the optimal algorithm to the specific problem at hand.

8.2.3 *Oracle Problem*

One of the greatest challenges in automatic test generation is the process of automatically verifying whether a test case reveals a fault — this is typically known as the *oracle problem* (more details in Section 2.5.4). Therefore, in order to avoid it, studies automatic test generation are typically performed on a regression scenario. That is, it is assumed there is a *golden* version of the software under test that is correct, and test cases are automatically generated for it. The oracles of those test cases exercise the behaviour of the *golden* version. These test cases are then executed against future versions of the software to verify that no regression faults have been introduced. (This is exactly the process we used in Chapter 5 to overcome the lack of accurate oracles.) Although an effective process in a regression scenario, ideally what a software engineer really would like to is to generate test cases for the current version of the software under test and find faults in it. However, due to the lack of an accurate technique to automatically generate oracles that are able reveal faults on the current version, software engineers may have to be asked to manually provide them. To do so, automated test generation techniques would have to generate test cases that are easy for human developers to understand. Otherwise, the process of understanding what each test case does and provide an oracle would be very tedious and time consuming, in particular for large test suites.

Readability could be the key to reduce the cost of asking developers to manually provide an oracle [141, 142]. However, how can an automated test generation technique optimise coverage, entropy, and readability simultaneously is still an open question. Unlike the combination of multiple coverage criteria we proposed in Chapter 3 or the integration of coverage and *entropy* we proposed in Chapter 5, the integration of readability may require a dedicated multi-objective optimisation algorithm (e.g., NSGA-II [127]), as the test case with the highest coverage may be the least readable test, and the most readable one may be the one with the lowest coverage or worst entropy. Therefore, further empirical studies would need to be performed to assess the most efficient approach of generating test cases that achieve high coverage, low entropy, and that are readable simultaneously.

8.2.4 *Scheduling Classes for Testing*

The continuous test generation prototype described in Sections 6.3 and 7.3 at this point is only a proof of concept, and there remains much potential for further improvements. In particular, there is potential to further improve the time budget scheduler that is responsible for allocating a certain amount of time each class under test is allowed to be tested.

8.2.4.1 *Complexity Metrics*

As some classes may require more time to be tested than other, our continuous test generation approach allocates for each class a time budget proportional to its complexity, i.e., number of branches. Thus, complex classes have a larger time budget than classes that are less complex. However, this measure does not distinguish, for example, nested branches and normal branches (branches with depth one). We propose to look at other metrics or combinations of different metrics [311] which might improve the accuracy of the time budget scheduler.

8.2.4.2 *Adaptive Time Budget*

Although two or more classes under test could have the same complexity (according to some metrics), the effort to test each one could be different, as some could require, e.g., the creation of complex objects. For instance, assume there are two classes, A and B, both with exactly same complexity. On a first invocation of a test generation tool, only 30% of the code of class A is covered and 80% of the code of class B is covered. Given that both classes are equally complex and were tested for the same time budget, the coverage achieved might mean that class A is more difficult to test than class B. Therefore, a future invocation of the test generation tool could explore this information and allow more time to test class A.

8.2.4.3 *Repository's History*

The abundant information available in a source control management could also be explored to improve the time budget scheduler. For instance, *fault prediction* [312] approaches have been used to estimate the probability of a component (e.g., file) being faulty in the future based on historical data from a version control system [313]. Such probability could also be used in automatic test generation to, for example, allocate a time budget proportional to the *faulty* probability of each class. In such a scenario, classes that are more likely of being faulty would be tested for longer than classes that are less likely to be faulty.

8.2.4.4 *Detection of Code Changes*

As described in 7.3, our continuous test generation identifies which Java files (i.e., classes) have been changed, and feeds this information to the time budget scheduler. However, this approach has a drawback, it is not able to distinguish between a documentation (e.g., JavaDoc) change and a source-code change. To efficiently use the time spend on testing, future work should investigate alternatives to improve the detection of changes that are not source-code related. For instance, the Abstract Syntax Tree (AST) of a current modified class and it previous version can be compared to identify whether the structure of the class has been modified.

8.2.5 *The EvoSuite Unit Test Generation Tool*

Although EvoSuite [9] is now a mature and advance unit test generation tool and its effectiveness has been evaluated on open source as well as industrial software in terms of code coverage [117, 200], fault finding effectiveness [256, 257], and effects on developer productivity [134, 196], there are still some functionalities that could be further improved and much potential for additional functionalities.

8.2.5.1 *Java's 64k Method Limit*

According to the JVM specification, the virtual machine code of a method can not exceed 65,535 bytes long. As EvoSuite instruments the bytecode of a class under test (i.e., injects custom code into the class to keep track of, for example, which branch instructions have been covered or not) occasionally, the length of an (already long) method exceeds the limit specified. A potential way of addressing this would be to stop instrumenting before the limit is reached, at the price of limited search guidance; a more effective solution would involve identifying parts of the code that are worth instrumenting thus reducing the overhead of the instrumentation.

8.2.5.2 *Flaky Tests*

Unstable tests, also known as "flaky" tests, i.e., tests that either do not compile or fail due to environment dependencies such as system time, are still a challenge for test generation tools. In order to address this problem, once the search is completed, EvoSuite applies various optimisations to reduce the length and improve the readability of the generated tests. For example, statements that do not contribute to increase coverage are removed, and a minimised set of effective test assertions is selected using mutation analysis. It is known for previous experiments that minimised tests are less likely to be flaky. However, occasionally, due to the generation of long tests, EvoSuite's minimisation phase runs out of time and hence EvoSuite reverts the

resulting test suite to its previous, unminimised version, which as mentioned is more likely to be flaky. A *partial minimisation* approach to select and minimise only a reduced subset of tests, or a more efficient minimisation approach based on *delta-debugging* [314], could be explored to alleviate this issue.

8.2.5.3 *Build Tools*

In order to attract software engineers (either researchers or practitioners working on industrial projects) to use EvoSuite, we must keep improving and updating all current build tools, and add support for new ones if possible. For instance, Gradle build tool[1] is gaining momentum in industry and it seems already very popular in projects on GitHub. The proposed architecture for the IntelliJ plugin described in Section 7.3 already paved the way for a simple and straightforward way of creating new plugins on top of EvoSuite's API.

8.2.5.4 *Continuous Integration Systems*

In Section 7.4 we presented the first prototype version of the EvoSuite Jenkins plugin, and although it is usable, there is much potential for additional functionalities. For example, although coverage of existing test cases is measured, this is not yet used in coverage visualizations. In particular, it would be helpful to see in detail which parts of all classes under test are covered by existing tests, which parts are covered by newly generated tests, and which parts are not yet covered at all. Furthermore, support to other SCMs besides the already supported Git would be beneficial, and support to other continuous integration systems, would also be a plus.

---

1 Gradle homepage `http://gradle.org`, accessed 11/2017.

BIBLIOGRAPHY

[1] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems". *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 249–265. ISBN: 978-1-931971-16-4 (cit. on p. 1).

[2] A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265. ISSN: 1460-244X (cit. on p. 1).

[3] F. C. Williams. "Early computers at Manchester University". *Radio and Electronic Engineer* 45.7 (July 1975), pp. 327–331 (cit. on p. 1).

[4] John W. Tukey. "The Teaching of Concrete Mathematics". *The American Mathematical Monthly* 65.1 (1958), pp. 1–9 (cit. on p. 1).

[5] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. 3rd. Wiley Publishing, 2011. ISBN: 1118031962, 9781118031964 (cit. on pp. 2, 11, 15, 19).

[6] Edsger W. Dijkstra. "Structured Programming". Ed. by O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. London, UK, UK: Academic Press Ltd., 1972. Chap. Chapter I: Notes on Structured Programming, pp. 1–82. ISBN: 0-12-200550-3 (cit. on pp. 2, 11).

[7] *Guava issue #2924*. Aug. 2017. URL: `https://github.com/google/guava/issues/2924` (visited on 11/2017) (cit. on p. 2).

[8] *Patch for Guava issue #2924*. Oct. 2017. URL: `https://github.com/google/guava/commit/a8f4ebc429d01150d3e35980373cb8c9c123aeaa` (visited on 11/2017) (cit. on p. 3).

[9] Gordon Fraser and Andrea Arcuri. "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software". *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ESEC/FSE '11. Szeged, Hungary: ACM, 2011, pp. 416–419. ISBN: 978-1-4503-0443-6 (cit. on pp. 4, 31, 42, 48, 55, 63–65, 88, 103, 104, 114, 131, 132, 154).

[10] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. "Combining Multiple Coverage Criteria in Search-Based Unit Test Generation". *Search-Based Software Engineering: 7th International Symposium, SSBSE 2015, Bergamo, Italy, September 5-7, 2015, Proceedings*. Ed. by Márcio Barros and Yvan Labiche. **Best Paper with industry-relevant SBSE results**. Cham: Springer International Publishing, 2015, pp. 93–108. ISBN: 978-3-319-22183-0 (cit. on pp. 8, 66, 67).

[11] José Campos, Yan Ge, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. "An Empirical Evaluation of Evolutionary Algorithms for Test Suite Generation". *Proceedings of the 9th International Symposium Search-Based Software Engineering (SSBSE)*. Ed. by Tim Menzies and Justyna Petke. **Distinguished Paper Award**. Cham: Springer International Publishing, 2017, pp. 33–48. ISBN: 978-3-319-66299-2 (cit. on p. 8).

[12] José Campos, Rui Abreu, Gordon Fraser, and Marcelo d'Amorim. "Entropy-based Test Generation for Improved Fault Localization". *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. ASE'13. Silicon Valley, CA, USA: IEEE Press, 2013, pp. 257–267. ISBN: 978-1-4799-0215-6 (cit. on pp. 9, 117).

[13] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. "Continuous Test Generation: Enhancing Continuous Integration with Automated Test Generation". *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. Vasteras, Sweden: ACM, 2014, pp. 55–66. ISBN: 978-1-4503-3013-8 (cit. on p. 9).

[14] José Campos, Gordon Fraser, Andrea Arcuri, and Rui Abreu. "Continuous Test Generation on Guava". *Search-Based Software Engineering: 7th International Symposium, SSBSE 2015, Bergamo, Italy, September 5-7, 2015, Proceedings*. Ed. by Márcio Barros and Yvan Labiche. Cham: Springer International Publishing, 2015, pp. 228–234. ISBN: 978-3-319-22183-0 (cit. on p. 9).

[15] Andrea Arcuri, José Campos, and Gordon Fraser. "Unit Test Generation During Software Development: EvoSuite Plugins for Maven, IntelliJ and Jenkins". *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Apr. 2016, pp. 401–408 (cit. on p. 9).

[16] Gordon Fraser, José Miguel Rojas, José Campos, and Andrea Arcuri. "EvoSuite at the SBST 2017 Tool Competition". *Proceedings of the 10th International Workshop on Search-Based Software Testing*. SBST '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 39–41. ISBN: 978-1-5386-2789-1 (cit. on pp. 9, 42, 120).

[17] "IEEE Standard Glossary of Software Engineering Terminology". *IEEE Std 610.12-1990* (Dec. 1990), pp. 1–84 (cit. on p. 11).

[18] Hong Zhu, Patrick A. V. Hall, and John H. R. May. "Software Unit Test Coverage and Adequacy". *ACM Comput. Surv.* 29.4 (Dec. 1997), pp. 366–427. ISSN: 0360-0300 (cit. on pp. 13, 30, 62).

[19] Yue Jia and Mark Harman. "An Analysis and Survey of the Development of Mutation Testing". *IEEE Transactions on Software Engineering* 37.5 (Sept. 2011), pp. 649–678. ISSN: 0098-5589 (cit. on p. 13).

[20] Gordon Fraser and Andreas Zeller. "Mutation-driven Generation of Unit Tests and Oracles". *IEEE Transactions on Software Engineering* 38.2 (2012), pp. 278–292. ISSN: 0098-5589 (cit. on pp. 14, 23, 30, 31, 42, 43, 58, 88).

[21] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold. "Prioritizing Test Cases for Regression Testing". *IEEE Transactions on Software Engineering* 27.10 (Oct. 2001), pp. 929–948. ISSN: 0098-5589 (cit. on pp. 14, 39).

[22] K. Taneja and Tao Xie. "DiffGen: Automated Regression Unit-Test Generation". *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 407–410. ISBN: 978-1-4244-2187-9 (cit. on pp. 14, 128).

[23] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. "Evaluating and Improving Fault Localization". *Proceedings of the 39th International Conference on Software Engineering*. ICSE '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 609–620. ISBN: 978-1-5386-3868-2 (cit. on p. 14).

[24] Rene Just, Franz Schweiggert, and Gregory M. Kapfhammer. "MAJOR: An Efficient and Extensible Tool for Mutation Analysis in a Java Compiler". *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 612–615. ISBN: 978-1-4577-1638-6 (cit. on p. 14).

[25] René Just. "The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java". *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. San Jose, CA, USA: ACM, 2014, pp. 433–436. ISBN: 978-1-4503-2645-2 (cit. on p. 14).

[26] Rene Just, Gregory M. Kapfhammer, and Franz Schweiggert. "Do Redundant Mutants Affect the Effectiveness and Efficiency of Mutation Analysis?" *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. ICST '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 720–725. ISBN: 978-0-7695-4670-4 (cit. on p. 14).

[27] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. 1st ed. New York, NY, USA: Cambridge University Press, 2008. ISBN: 0521880386, 9780521880381 (cit. on p. 14).

[28] J. H. Andrews, L. C. Briand, and Y. Labiche. "Is Mutation an Appropriate Tool for Testing Experiments?" *Proceedings of the 27th International Conference on Software Engineering*. ICSE '05. St. Louis, MO, USA: ACM, 2005, pp. 402–411. ISBN: 1-58113-963-2 (cit. on p. 14).

[29] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. "Are Mutants a Valid Substitute for Real Faults in Software Testing?" *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, pp. 654–665. ISBN: 978-1-4503-3056-5 (cit. on p. 14).

[30] Jeff Offutt and Aynur Abdurazik. "Generating Tests from UML Specifications". *Proceedings of the 2nd international conference on The unified modeling language: beyond the standard*. UML'99. Fort Collins, CO, USA: Springer-Verlag, 1999, pp. 416–429. ISBN: 3-540-66712-1 (cit. on p. 14).

[31] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. "Random Testing: Theoretical Results and Practical Implications". *IEEE Transactions on Software Engineering* 38.2 (Mar. 2012), pp. 258–277. ISSN: 0098-5589 (cit. on pp. 14, 15).

[32] Christoph Csallner and Yannis Smaragdakis. "JCrasher: An Automatic Robustness Tester for Java". *Software-Practice & Experience* 34 (11 2004), pp. 1025–1050. ISSN: 0038-0644 (cit. on pp. 14, 15).

[33] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. "Feedback-Directed Random Test Generation". *Proceedings of the 29th international conference on Software Engineering*. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 75–84. ISBN: 0-7695-2828-7 (cit. on pp. 14–16).

[34] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner, and Lisa Ling Liu. "Automatic Testing of Object-Oriented Software". *Proceedings of the 33rd conference on Current Trends in Theory and Practice of Computer Science*. SOFSEM '07. Harrachov, Czech Republic: Springer-Verlag, 2007, pp. 114–129. ISBN: 978-3-540-69506-6 (cit. on p. 15).

[35] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing". *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '05. Chicago, IL, USA: ACM, 2005, pp. 213–223. ISBN: 1-59593-056-6 (cit. on pp. 15, 21, 101).

[36] Paolo Tonella. "Evolutionary Testing of Classes". *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '04. Boston, Massachusetts, USA: ACM, 2004, pp. 119–128. ISBN: 1-58113-820-2 (cit. on pp. 15, 23–25, 37).

[37] Patrice Godefroid, Michael Y. Levin, and David A Molnar. "Automated Whitebox Fuzz Testing". *Proceedings of the Network and Distributed System Security Symposium*. NDSS '08. San Diego, California, USA: The Internet Society, 2008 (cit. on pp. 15, 21).

[38] John Regehr. "Random Testing of Interrupt-driven Software". *Proceedings of the 5th ACM International Conference on Embedded Software*. EMSOFT '05. Jersey City, NJ, USA: ACM, 2005, pp. 290–298. ISBN: 1-59593-091-4 (cit. on p. 15).

[39] T. A. Thayer, M. Lipow, and E. C. Nelson. *Software Readability*. North Holland, Amsterdam, 1978 (cit. on pp. 15, 19).

[40] Joe W. Duran and S.C. Ntafos. "An Evaluation of Random Testing". *IEEE Transactions on Software Engineering* SE-10.4 (July 1984), pp. 438–444. ISSN: 0098-5589 (cit. on p. 15).

[41] Carlos Pacheco and Michael D. Ernst. "Randoop: Feedback-directed Random Testing for Java". *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*. OOPSLA '07. Montreal, Quebec, Canada: ACM, 2007, pp. 815–816. ISBN: 978-1-59593-865-7 (cit. on pp. 15, 148).

[42] L.J. White and E.I. Cohen. "A Domain Strategy for Computer Program Testing". *IEEE Transactions on Software Engineering* 6.3 (1980), pp. 247–257. ISSN: 0098-5589 (cit. on p. 16).

[43] George B. Finelli. "NASA Software failure characterization experiments". *Reliability Engineering & System Safety* 32.1-2 (1991), pp. 155–169. ISSN: 0951-8320 (cit. on p. 16).

[44] M.J.P. van der Meulen, P.G. Bishop, and R. Villa. "An Exploration of Software Faults and Failure Behaviour in a Large Population of Programs". *Proceedings of the 15th International Symposium on Software Reliability Engineering*. ISSRE '04. Nov. 2004, pp. 101–112 (cit. on p. 16).

[45] Christoph Schneckenburger and Johannes Mayer. "Towards the Determination of Typical Failure Patterns". *Fourth International Workshop on Software Quality Assurance: In Conjunction with the 6th ESEC/FSE Joint Meeting*. SOQUA '07. Dubrovnik, Croatia: ACM, 2007, pp. 90–93. ISBN: 978-1-59593-724-7 (cit. on p. 16).

[46] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. "Adaptive Random Testing: The ART of Test Case Diversity". *Journal of Systems and Software* 83.1 (Jan. 2010), pp. 60–66. ISSN: 0164-1212 (cit. on p. 16).

[47] T.Y. Chen, H. Leung, and I.K. Mak. "Adaptive Random Testing". *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*. Ed. by MichaelJ. Maher. Vol. 3321. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 320–329. ISBN: 978-3-540-24087-7 (cit. on p. 16).

[48] Tsong Yueh Chen and Robert Merkel. "Quasi-random Testing". *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ASE '05. Long Beach, CA, USA: ACM, 2005, pp. 309–312. ISBN: 1-58113-993-4 (cit. on p. 16).

[49] A.F. Tappenden and J. Miller. "A Novel Evolutionary Approach for Adaptive Random Testing". *IEEE Transactions on Reliability* 58.4 (Dec. 2009), pp. 619–633. ISSN: 0018-9529 (cit. on pp. 16, 18, 19).

[50] Hongmei Chi and Edward L. Jones. "Computational Investigations of Quasirandom Sequences in Generating Test Cases for Specification-based Tests". *Proceedings of the 38th Conference on Winter Simulation*. WSC '06. Monterey, California: Winter Simulation Conference, 2006, pp. 975–980. ISBN: 1-4244-0501-7 (cit. on p. 16).

[51] Ali Shahbazi, Andrew F. Tappenden, and James Miller. "Centroidal Voronoi Tessellations - A New Approach to Random Testing". *IEEE Transactions on Software Engineering* 39.2 (Feb. 2013), pp. 163–183. ISSN: 0098-5589 (cit. on p. 16).

[52] Kwok Ping Chan, Tsong Yueh Chen, and Dave Towey. "Restricted Random Testing". *Proceedings of the 7th International Conference on Software Quality*. ECSQ '02. London, UK, UK: Springer-Verlag, 2002, pp. 321–330. ISBN: 3-540-43749-5 (cit. on pp. 16, 18).

[53] Huai Liu, Xiaodong Xie, Jing Yang, Yansheng Lu, and Tsong Yueh Chen. "Adaptive Random Testing Through Test Profiles". *Software-Practice & Experience* 41.10 (Sept. 2011), pp. 1131–1154. ISSN: 0038-0644 (cit. on pp. 17, 18).

[54] Kwok Ping Chan, Tsong Yueh Chen, Fei-Ching Kuo, and Dave Towey. "A Revisit of Adaptive Random Testing by Restriction". *Proceedings of the 28th Annual International Computer Software and Applications Conference - Volume 01*. COMPSAC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 78–85. ISBN: 0-7695-2209-2-1 (cit. on p. 17).

[55] Kwok Ping Chan, T. Y. Chen, and Dave Towey. "Forgetting Test Cases". *Proceedings of the 30th Annual International Computer Software and Applications Conference - Volume 01*. COMPSAC '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 485–494. ISBN: 0-7695-2655-1 (cit. on p. 17).

[56] T.Y. Chen, R. Merkel, P.K. Wong, and G. Eddy. "Adaptive Random Testing Through Dynamic Partitioning". *Proceedings of the 4th International Conference on Quality Software*. QSIC '04. Sept. 2004, pp. 79–86 (cit. on pp. 17, 19).

[57] T.Y. Chen and D.H. Huang. "Adaptive Random Testing by Localization". *Proceedings of the 11th Asia-Pacific Software Engineering Conference*. APSEC '04. Nov. 2004, pp. 292–298 (cit. on p. 17).

[58] Johannes Mayer. "Adaptive Random Testing by Bisection and Localization". *Proceedings of the 5th International Conference on Formal Approaches to Software Testing*. FATES '05. Edinburgh, UK: Springer-Verlag, 2006, pp. 72–86. ISBN: 3-540-34454-3, 978-3-540-34454-4 (cit. on p. 18).

[59] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. "ARTOO: Adaptive Random Testing for Object-oriented Software". *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. Leipzig, Germany: ACM, 2008, pp. 71–80. ISBN: 978-1-60558-079-1 (cit. on p. 18).

[60] Yu Lin, Xucheng Tang, Yuting Chen, and Jianjun Zhao. "A Divergence-Oriented Approach to Adaptive Random Testing of Java Programs". *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*. ASE '09. Nov. 2009, pp. 221–232 (cit. on p. 18).

[61] I. K. Mak. "On the Effectiveness of Random Testing". MA thesis. Australia: University of Melbourne, Department of Computer Science, 1997 (cit. on p. 19).

[62] T. Y. Chen, T. H. Tse, and Y. T. Yu. "Proportional Sampling Strategy: A Compendium and Some Insights". *Journal of Systems and Software* 58.1 (Aug. 2001), pp. 65–81. ISSN: 0164-1212 (cit. on p. 19).

[63] Kwok Ping Chan, Tsong Yueh Chen, and Dave Towey. "Restricted Random Testing: Adaptive Random Testing by Exclusion". *International Journal of Software Engineering and Knowledge Engineering* 16.4 (2006), pp. 553–584. ISSN: 0218-1940 (cit. on p. 19).

[64] Johannes Mayer and Christoph Schneckenburger. "An Empirical Analysis and Comparison of Random Testing Techniques". *Proceedings of the 2006 ACM/IEEE International Symposium on*

Empirical Software Engineering. ISESE '06. Rio de Janeiro, Brazil: ACM, 2006, pp. 105–114. ISBN: 1-59593-218-6 (cit. on p. 19).

[65] Andrea Arcuri and Lionel Briand. "Adaptive Random Testing: An Illusion of Effectiveness?" *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA '11. Toronto, Ontario, Canada: ACM, 2011, pp. 265–275. ISBN: 978-1-4503-0562-4 (cit. on p. 19).

[66] L.A. Clarke. "A System to Generate Test Data and Symbolically Execute Programs". *IEEE Transactions on Software Engineering* SE-2.3 (Sept. 1976), pp. 215–222. ISSN: 0098-5589 (cit. on p. 19).

[67] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. "Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing Nasa Software". *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. ISSTA '08. Seattle, WA, USA: ACM, 2008, pp. 15–26. ISBN: 978-1-60558-050-0 (cit. on p. 19).

[68] Zhihong Xu, Yunho Kim, Moonzoo Kim, Gregg Rothermel, and Myra B. Cohen. "Directed Test Suite Augmentation: Techniques and Tradeoffs". *ACM Symposium on the Foundations of Software Engineering (FSE)*. FSE '10. Santa Fe, New Mexico, USA: ACM, 2010, pp. 257–266. ISBN: 978-1-60558-791-2 (cit. on pp. 19, 41, 110, 113, 128).

[69] Zhihong Xu, Yunho Kim, Moonzoo Kim, and Gregg Rothermel. "A Hybrid Directed Test Suite Augmentation Technique". *IEEE International Symposium on Software Reliability Engineering (ISSRE)*. ISSRE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 150–159. ISBN: 978-0-7695-4568-4 (cit. on pp. 19, 113).

[70] Cristian Zamfir and George Candea. "Execution Synthesis: A Technique for Automated Software Debugging". *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. Paris, France: ACM, 2010, pp. 321–334. ISBN: 978-1-60558-577-2 (cit. on p. 19).

[71] Corina S. Păsăreanu and Willem Visser. "A Survey of New Trends in Symbolic Execution for Software Testing and Analysis". *International Journal on Software Tools for Technology Transfer (STTT)* 11.4 (Oct. 2009), pp. 339–353. ISSN: 1433-2779 (cit. on pp. 19, 20).

[72] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. "Symbolic Execution for Software Testing in Practice: Preliminary Assessment". *Proceedings of the 33rd International Con-*

ference on Software Engineering. ICSE '11. Honolulu, HI, USA: ACM, 2011, pp. 1066–1071. ISBN: 978-1-4503-0445-0 (cit. on pp. 19, 20).

[73] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. ISBN: 3-540-78799-2, 978-3-540-78799-0 (cit. on pp. 20, 21).

[74] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. "RWset: Attacking Path Explosion in Constraint-based Test Generation". *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 351–366. ISBN: 3-540-78799-2, 978-3-540-78799-0 (cit. on p. 20).

[75] Rupak Majumdar and Ru-Gang Xu. "Reducing Test Inputs Using Information Partitions". *Proceedings of the 21st International Conference on Computer Aided Verification*. CAV '09. Grenoble, France: Springer-Verlag, 2009, pp. 555–569. ISBN: 978-3-642-02657-7 (cit. on p. 20).

[76] Raul Santelices and Mary Jean Harrold. "Exploiting Program Dependencies for Scalable Multiple-path Symbolic Execution". *Proceedings of the 19th International Symposium on Software Testing and Analysis*. ISSTA '10. Trento, Italy: ACM, 2010, pp. 195–206. ISBN: 978-1-60558-823-0 (cit. on p. 20).

[77] Dawei Qi, Hoang D.T. Nguyen, and Abhik Roychoudhury. "Path Exploration Based on Symbolic Output". *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE '11. Szeged, Hungary: ACM, 2011, pp. 278–288. ISBN: 978-1-4503-0443-6 (cit. on pp. 20, 21).

[78] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. "Directed Symbolic Execution". *Proceedings of the 18th International Conference on Static Analysis*. SAS'11. Venice, Italy: Springer-Verlag, 2011, pp. 95–111. ISBN: 978-3-642-23701-0 (cit. on p. 21).

[79] Saswat Anand, Alessandro Orso, and Mary Jean Harrold. "Type-dependence Analysis and Program Transformation for Symbolic Execution". *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'07. Braga, Portugal: Springer-Verlag, 2007, pp. 117–133. ISBN: 978-3-540-71208-4 (cit. on pp. 21, 22).

[80] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A Concolic Unit Testing Engine for C". *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. ESEC/FSE-13. Lisbon, Portugal: ACM, 2005, pp. 263–272. ISBN: 1-59593-014-0 (cit. on pp. 21, 101).

[81] Nikolai Tillmann and Jonathan De Halleux. "Pex: White Box Test Generation for .NET". *Proceedings of the 2Nd International Conference on Tests and Proofs*. TAP'08. Prato, Italy: Springer-Verlag, 2008, pp. 134–153. ISBN: 3-540-79123-X, 978-3-540-79123-2 (cit. on pp. 21, 31, 103).

[82] Rupak Majumdar and Koushik Sen. "Hybrid Concolic Testing". *Proceedings of the 29th International Conference on Software Engineering*. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 416–426. ISBN: 0-7695-2828-7 (cit. on p. 21).

[83] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. "Statically-directed Dynamic Automated Test Generation". *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA '11. Toronto, Ontario, Canada: ACM, 2011, pp. 12–22. ISBN: 978-1-4503-0562-4 (cit. on p. 21).

[84] Saswat Anand and Mary Jean Harrold. "Heap Cloning: Enabling Dynamic Symbolic Execution of Java Programs". *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 33–42. ISBN: 978-1-4577-1638-6 (cit. on p. 22).

[85] Patrice Godefroid and Daniel Luchaup. "Automatic Partial Loop Summarization in Dynamic Test Generation". *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA '11. Toronto, Ontario, Canada: ACM, 2011, pp. 23–33. ISBN: 978-1-4503-0562-4 (cit. on pp. 22, 23).

[86] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. "Loop-extended Symbolic Execution on Binary Programs". *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. ISSTA '09. Chicago, IL, USA: ACM, 2009, pp. 225–236. ISBN: 978-1-60558-338-9 (cit. on p. 23).

[87] Phil McMinn. "Search-based Software Test Data Generation: A Survey". *Software Testing, Verification and Reliability* 14.2 (June 2004), pp. 105–156. ISSN: 0960-0833 (cit. on pp. 23, 24, 30–32).

[88] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. "Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications". *De-*

partment of Computer Science, King's College London, Tech. Rep. TR-09-03 (2009) (cit. on pp. 23, 24).

[89] Mark Harman and Bryan F. Jones. "Search-based Software Engineering". *Information and Software Technology* 43.14 (2001), pp. 833–839. ISSN: 0950-5849 (cit. on p. 23).

[90] John Clarke, Mark Harman, R Hierons, B Jones, M Lumkin, K Rees, M Roper, and M Shepperd. "The Application of Metaheuristic Search Techniques to Problems in Software Engineering". *SEMINAL (Software Engineering using Metaheuristic INnovative ALgorithms) technical report SEMINAL-TR-01-2000* (2000) (cit. on p. 23).

[91] J. Clarke et al. "Reformulating software engineering as a search problem". *IEE Proceedings - Software* 150.3 (June 2003), pp. 161–175. ISSN: 1462-5970 (cit. on p. 23).

[92] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. "Search-based Software Engineering: Trends, Techniques and Applications". *ACM Comput. Surv.* 45.1 (Dec. 2012), 11:1–11:61. ISSN: 0360-0300 (cit. on pp. 23, 25).

[93] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. "Search Based Software Engineering: Techniques, Taxonomy, Tutorial". Ed. by Bertrand Meyer and Martin Nordio. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 1–59. ISBN: 978-3-642-25230-3 (cit. on pp. 23, 25, 35).

[94] W. Miller and D. L. Spooner. "Automatic Generation of Floating-Point Test Data". *IEEE Transactions on Software Engineering* 2.3 (May 1976), pp. 223–226. ISSN: 0098-5589 (cit. on p. 23).

[95] Phil McMinn. "Search-Based Software Testing: Past, Present and Future". *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. ICSTW '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 153–163. ISBN: 978-0-7695-4345-1 (cit. on pp. 23, 26, 27, 34).

[96] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gal, S. Katsikas, and K. Karapoulios. "Application of Genetic Algorithms to Software Testing". *Proceedings of the 5th International Conference on Software Engineering and its Applications*. Toulouse, France: 1992, pp. 625–636 (cit. on pp. 23, 24).

[97] B. Korel. "Automated Software Test Data Generation". *IEEE Transactions on Software Engineering* 16.8 (Aug. 1990), pp. 870–879. ISSN: 0098-5589 (cit. on pp. 23, 24, 30–32, 37).

[98] B.F. Jones, H.-H. Sthamer, and D.E. Eyres. "Automatic Structural Testing using Genetic Algorithms". *Software Engineering Journal* 11.5 (Sept. 1996), pp. 299–306. ISSN: 0268-6961 (cit. on p. 23).

[99] R. P. Pargas, M. J. Harrold, and R. R. Peck. "Test-Data Generation Using Genetic Algorithms". *Software Testing, Verification and Reliability* 9 (4 Dec. 1999), pp. 263–282 (cit. on pp. 23, 24).

[100] Joachim Wegener, Andre Baresel, and Harmen Sthamer. "Evolutionary Test Environment for Automatic Structural Testing". *Information and Software Technology* 43.14 (2001), pp. 841–854. ISSN: 0950-5849 (cit. on pp. 23, 30–32).

[101] Oliver Bühler and Joachim Wegener. "Evolutionary Functional Testing". *Computers and Operations Research* 35.10 (Oct. 2008), pp. 3144–3160. ISSN: 0305-0548 (cit. on pp. 23, 30).

[102] Wasif Afzal, Richard Torkar, and Robert Feldt. "A Systematic Review of Search-based Testing for Non-functional System Properties". *Information and Software Technology* 51.6 (June 2009), pp. 957–976. ISSN: 0950-5849 (cit. on pp. 23, 30).

[103] Gordon Fraser and Andrea Arcuri. "Achieving Scalable Mutation-based Generation of Whole Test Suites". *Empirical Software Engineering* (2014), pp. 1–30. ISSN: 1382-3256 (cit. on pp. 23, 30, 35, 42, 43, 56, 58).

[104] Nigel Tracey, John Clark, and Keith Mander. "Automated Program Flaw Finding Using Simulated Annealing". *SIGSOFT Software Engineering Notes* 23.2 (Mar. 1998), pp. 73–81. ISSN: 0163-5948 (cit. on pp. 23, 27).

[105] Nigel Tracey, John Clark, Keith Mander, and John McDermid. "Automated Test-data Generation for Exception Conditions". *Software Practice & Experience* 30.1 (Jan. 2000), pp. 61–79. ISSN: 0038-0644 (cit. on pp. 23, 53).

[106] Nigel Tracey, John Clark, John McDermid, and Keith Mander. "Systems Engineering for Business Process Change". Ed. by Peter Henderson. New York, NY, USA: Springer-Verlag New York, Inc., 2002. Chap. A Search-based Automated Test-data Generation Framework for Safety-critical Systems, pp. 174–213. ISBN: 1-85233-399-5 (cit. on pp. 23, 37).

[107] B. Korel. "Dynamic Method of Software Test Data Generation". 2 (Dec. 1992), pp. 203–213 (cit. on p. 24).

[108] M Ross, CA Brebbia, G Staples, and J Stapleton. "The Problematics of Testing Object-Oriented Software". 2 (1994), pp. 411–426 (cit. on p. 24).

[109] A. Arcuri and X. Yao. "On Test Data Generation of Object-Oriented Software". *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*. TAICPART-MUTATION 2007. Sept. 2007, pp. 72–76 (cit. on p. 24).

[110] Stefan Wappler and Frank Lammermann. "Using Evolutionary Algorithms for the Unit Testing of Object-oriented Software". *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*. GECCO '05. Washington DC, USA: ACM, 2005, pp. 1053–1060. ISBN: 1-59593-010-8 (cit. on p. 25).

[111] Stefan Wappler and Joachim Wegener. "Evolutionary Unit Testing of Object-oriented Software Using Strongly-typed Genetic Programming". *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*. GECCO '06. Seattle, Washington, USA: ACM, 2006, pp. 1925–1932. ISBN: 1-59593-186-4 (cit. on p. 25).

[112] Gordon Fraser and Andrea Arcuri. "Evolutionary Generation of Whole Test Suites". *Proceedings of the 2011 11th International Conference on Quality Software*. QSIC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 31–40. ISBN: 978-0-7695-4468-7 (cit. on p. 25).

[113] Gordon Fraser and Andrea Arcuri. "Whole Test Suite Generation". *IEEE Transactions on Software Engineering* 39.2 (2013), pp. 276–291. ISSN: 0098-5589 (cit. on pp. 25, 29, 42, 51, 63, 64, 78, 87, 93, 131).

[114] David H Wolpert and William G Macready. "No free lunch theorems for optimization". *IEEE transactions on evolutionary computation* 1.1 (1997), pp. 67–82 (cit. on pp. 25, 67, 75).

[115] Gordon Fraser and Andrea Arcuri. "Handling Test Length Bloat". *Software Testing, Verification and Reliability (STVR)* 23.7 (2013), pp. 553–582. ISSN: 1099-1689 (cit. on pp. 25, 36, 43).

[116] Dean C Karnopp. "Random search techniques for optimization problems". *Automatica* 1.2-3 (1963), pp. 111–121 (cit. on p. 26).

[117] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. "A Detailed Investigation of the Effectiveness of Whole Test Suite Generation". *Empirical Software Engineering* (2016) (cit. on pp. 26, 64, 73, 154).

[118] Sina Shamshiri, José Miguel Rojas, Gordon Fraser, and Phil McMinn. "Random or Genetic Algorithm Search for Object-Oriented Test Suite Generation?" *Proceedings of the Conference on Genetic and Evolutionary Computation*. ACM. 2015, pp. 1367–1374 (cit. on pp. 26, 37, 64, 65, 73, 75).

[119] Mark Harman and Phil McMinn. "A Theoretical & Empirical Analysis of Evolutionary Testing and Hill Climbing for Structural Test Data Generation". *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. ISSTA '07. London, United Kingdom: ACM, 2007, pp. 73–83. ISBN: 978-1-59593-734-6 (cit. on pp. 26, 75).

[120] Fred Glover. "Tabu Search-Part I". *ORSA Journal on Computing* 1.3 (1989), pp. 190–206 (cit. on p. 27).

[121] E. Diaz, J. Tuya, and R. Blanco. "Automated Software Testing using a Metaheuristic Technique based on Tabu Search". *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*. ASE '03. Montreal, Canada: IEEE, Oct. 2003, pp. 310–313 (cit. on p. 27).

[122] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. "Optimization by Simulated Annealing". *Science* 220.4598 (1983), pp. 671–680 (cit. on p. 27).

[123] J. H. Holland. "Genetic Algorithm and the Optimal Allocation of Trials". *SiAM Journal of Computing* 2.2 (June 1973), pp. 88–105 (cit. on p. 28).

[124] Benjamin Doerr, Carola Doerr, and Franziska Ebel. "From black-box complexity to designing new genetic algorithms". *Theoretical Computer Science* 567 (2015), pp. 87–104 (cit. on pp. 29, 67).

[125] Aram Ter-Sarkisov and Stephen R Marsland. "Convergence Properties of (μ+ λ) Evolutionary Algorithms". *AAAI*. 2011 (cit. on p. 29).

[126] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. "Reformulating branch coverage as a many-objective optimization problem". *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*. IEEE. 2015, pp. 1–10 (cit. on pp. 29, 30, 64, 73).

[127] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and Tanaka Meyarivan. "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II". *International Conference on Parallel Problem Solving From Nature*. Springer. 2000, pp. 849–858 (cit. on pp. 29, 35, 87, 152).

[128] Annibale Panichella, Fitsum Kifetew, and Paolo Tonella. "Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets". *IEEE Transactions on Software Engineering* (2017) (cit. on pp. 30, 65–67, 73).

[129] Mark Harman and John Clark. "Metrics Are Fitness Functions Too". *Proceedings of the Software Metrics, 10th International Symposium*. METRICS '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 58–69. ISBN: 0-7695-2129-0 (cit. on p. 30).

[130] Gordon Fraser and Andrea Arcuri. "1600 Faults in 100 Projects: Automatically Finding Faults While Achieving High Coverage with EvoSuite". *Empirical Software Engineering* (2013), pp. 1–29. ISSN: 1382-3256 (cit. on pp. 31, 43, 108, 117, 121, 131).

[131] Gregory Gay. "The Fitness Function for the Job: Search-Based Generation of Test Suites That Detect Real Faults". *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Mar. 2017, pp. 345–355 (cit. on pp. 31, 64, 67, 78).

[132] A. Arcuri. "It Does Matter How You Normalise the Branch Distance in Search Based Software Testing". *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. Apr. 2010, pp. 205–214 (cit. on p. 32).

[133] Mark Harman, Lin Hu, Robert M. Hierons, André Baresel, and Harmen Sthamer. "Improving Evolutionary Testing By Flag Removal". *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '02. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002, pp. 1359–1366. ISBN: 1-55860-878-8 (cit. on p. 34).

[134] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. "Automated Unit Test Generation during Software Development: A Controlled Experiment and Think-Aloud Observations". *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. 2015 (cit. on pp. 34, 64, 132, 154).

[135] Richard A. DeMillo and A. Jefferson Offutt. "Constraint-Based Automatic Test Data Generation". *IEEE Transactions on Software Engineering* 17.9 (Sept. 1991), pp. 900–910. ISSN: 0098-5589 (cit. on p. 35).

[136] Leonardo Bottaci. "A Genetic Algorithm Fitness Function for Mutation Testing" (2001) (cit. on p. 35).

[137] Mark Harman and Phil McMinn. "A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search". *IEEE Transactions on Software Engineering* 36.2 (Mar. 2010), pp. 226–247. ISSN: 0098-5589 (cit. on p. 35).

[138] Alison Watkins and Ellen M. Hufnagel. "Evolutionary Test Data Generation: A Comparison of Fitness Functions: Research Articles". *Software Practice Experience* 36.1 (Jan. 2006), pp. 95–116. ISSN: 0038-0644 (cit. on p. 35).

[139] André Baresel, Harmen Sthamer, and Michael Schmidt. “Fitness Function Design To Improve Evolutionary Structural Testing”. *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO ’02. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002, pp. 1329–1336. ISBN: 1-55860-878-8 (cit. on pp. 35, 36).

[140] Omur Sahin and Bahriye Akay. “Comparisons of metaheuristic algorithms and fitness functions on software test data generation”. *Applied Soft Computing* 49 (2016), pp. 1202–1214. ISSN: 1568-4946 (cit. on pp. 35, 75, 76).

[141] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. “Modeling Readability to Improve Unit Tests”. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. **ACM SIGSOFT Distinguished Paper Award**. Bergamo, Italy: ACM, 2015, pp. 107–118. ISBN: 978-1-4503-3675-8 (cit. on pp. 35, 57, 78, 152).

[142] Ermira Daka, José Campos, Jonathan Dorn, Gordon Fraser, and Westley Weimer. “Generating Readable Unit Tests for Guava”. *Search-Based Software Engineering: 7th International Symposium, SSBSE 2015, Bergamo, Italy, September 5-7, 2015, Proceedings*. Ed. by Márcio Barros and Yvan Labiche. Cham: Springer International Publishing, 2015, pp. 235–241. ISBN: 978-3-319-22183-0 (cit. on pp. 35, 152).

[143] S. Afshan, P. McMinn, and M. Stevenson. “Evolving Readable String Test Inputs Using a Natural Language Model to Reduce Human Oracle Cost”. *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. Mar. 2013, pp. 352–361 (cit. on pp. 35, 42).

[144] Javier Ferrer, Francisco Chicano, and Enrique Alba. “Evolutionary Algorithms for the Multi-objective Test Data Generation Problem”. *Softw. Pract. Exper.* 42.11 (Nov. 2012), pp. 1331–1362. ISSN: 0038-0644 (cit. on pp. 35, 36, 87).

[145] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. “SPEA2: Improving the Strength Pareto Evolutionary Algorithm”. *Proceedings of the Evolutionary Methods for Design, Optimisation and Control with Application to Industrial Problems*. Vol. 3242. EUROGEN 2001 103. International Center for Numerical Methods in Engineering, 2001 (cit. on pp. 35, 87).

[146] Mark Harman, Kiran Lakhotia, and Phil McMinn. “A Multi-objective Approach to Search-based Test Data Generation”. *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*. GECCO ’07. London, England: ACM, 2007, pp. 1098–1105. ISBN: 978-1-59593-697-4 (cit. on pp. 35, 36, 62, 78, 87).

[147] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. “Automatic Test Case Generation: What if Test Code Quality Matters?” *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. Saarbrücken, Germany: ACM, 2016, pp. 130–141. ISBN: 978-1-4503-4390-9 (cit. on pp. 36, 78, 87).

[148] Andrea Arcuri and Gordon Fraser. “Parameter tuning or default values? An empirical investigation in search-based software engineering”. *Empirical Software Engineering* 18.3 (2013), pp. 594–623 (cit. on pp. 36, 66, 67).

[149] M. Miraz, P.L. Lanzi, and L. Baresi. “Improving Evolutionary Testing by Means of Efficiency Enhancement Techniques”. *Proceedings of the 2010 IEEE Congress on Evolutionary Computation*. CEC ’10. Barcelona, Spain: IEEE, July 2010, pp. 1–8. ISBN: 978-1-4244-6909-3 (cit. on p. 36).

[150] Gordon Fraser and Andrea Arcuri. “The Seed is Strong: Seeding Strategies in Search-Based Software Testing”. *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. ICST ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 121–130. ISBN: 978-0-7695-4670-4 (cit. on pp. 36, 43, 104, 110, 113).

[151] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. “Seeding Strategies in Search-based Unit Test Generation”. *Software Testing, Verification and Reliability* 26.5 (Aug. 2016), pp. 366–401. ISSN: 0960-0833 (cit. on pp. 36, 43).

[152] Koushik Sen and Gul Agha. “CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools”. *Proceedings of the 18th International Conference on Computer Aided Verification*. CAV’06. Seattle, WA: Springer-Verlag, 2006, pp. 419–423. ISBN: 3-540-37406-X, 978-3-540-37406-0 (cit. on p. 37).

[153] Kobi Inkumsah and Tao Xie. “Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution”. *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*. ASE ’08. L’Aquila: IEEE, Sept. 2008, pp. 297–306. ISBN: 978-1-4244-2187-9 (cit. on p. 37).

[154] Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan De Halleux. “FloPSy: Search-based Floating Point Constraint Solving for Symbolic Execution”. *Proceedings of the 22Nd IFIP WG 6.1 International Conference on Testing Software and Systems*. ICTSS ’10. Natal, Brazil: Springer-Verlag, 2010, pp. 142–157. ISBN: 3-642-16572-9, 978-3-642-16572-6 (cit. on p. 37).

[155] J.P. Galeotti, G. Fraser, and A Arcuri. "Improving Search-based Test Suite Generation with Dynamic Symbolic Execution". *Proceedings of the 24th International Symposium on Software Reliability Engineering*. ISSRE '13. Nov. 2013, pp. 360–369 (cit. on pp. 37, 42).

[156] Andreas Zeller. "Yesterday, My Program Worked. Today, It Does Not. Why?" *SIGSOFT Software Engineering Notes* 24.6 (Oct. 1999), pp. 253–267. ISSN: 0163-5948 (cit. on p. 37).

[157] S. Yoo and M. Harman. "Regression Testing Minimization, Selection and Prioritization: A Survey". *Software Testing, Verification & Reliability* 22.2 (Mar. 2012), pp. 67–120. ISSN: 0960-0833 (cit. on pp. 37, 38).

[158] Sebastian Elbaum, Gregg Rothermel, and John Penix. "Techniques for Improving Regression Testing in Continuous Integration Development Environments". *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, pp. 235–245. ISBN: 978-1-4503-3056-5 (cit. on p. 37).

[159] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. "A Methodology for Controlling the Size of a Test Suite". *ACM Transactions on Software Engineering and Methodology* 2.3 (July 1993), pp. 270–285. ISSN: 1049-331X (cit. on p. 38).

[160] V. Chvatal. "A Greedy Heuristic for the Set-Covering Problem". *Mathematics of Operations Research* 4.3 (Aug. 1979), pp. 233–235. ISSN: 0364-765X (cit. on p. 38).

[161] Gregg Rothermel, Mary Jean Harrold, Jeffery von Ronne, and Christie Hong. "Empirical Studies of Test-Suite Reduction". *Software Testing, Verification and Reliability* 12.4 (2002), pp. 219–249. ISSN: 1099-1689 (cit. on p. 38).

[162] Gregg Rothermel and Mary Jean Harrold. "Analyzing Regression Test Selection Techniques". *IEEE Transactions on Software Engineering* 22.8 (Aug. 1996), pp. 529–551. ISSN: 0098-5589 (cit. on p. 38).

[163] Gregg Rothermel and Mary Jean Harrold. "A Safe, Efficient Regression Test Selection Technique". *ACM Transactions on Software Engineering and Methodology* 6.2 (Apr. 1997), pp. 173–210. ISSN: 1049-331X (cit. on p. 38).

[164] S. S. Yau and Z. Kishimoto. "A Method for Revalidating Modified Programs in the Maintenance Phase". *Proceedings of International Computer Software and Applications Conference*. Oct. 1987 (cit. on p. 39).

[165] Kurt Fischer, Farzad Raji, and Andrew Chruscicki. "A methodology for retesting modified software". *Proceedings of the National Telecommunications Conference B-6-3*. 1981, pp. 1–6 (cit. on p. 39).

[166] F. I. Vokolos and P. G. Frankl. "Empirical Evaluation of the Textual Differencing Regression Testing Technique". *Proceedings of the International Conference on Software Maintenance*. ICSM '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 44–. ISBN: 0-8186-8779-7 (cit. on p. 39).

[167] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. "Comparing and Combining Test-suite Reduction and Regression Test Selection". *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: ACM, 2015, pp. 237–247. ISBN: 978-1-4503-3675-8 (cit. on p. 39).

[168] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. "Test Case Prioritization: An Empirical Study". *Proceedings of the IEEE International Conference on Software Maintenance*. ICSM '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 179–. ISBN: 0-7695-0016-1 (cit. on p. 39).

[169] Zheng Li, M. Harman, and R.M. Hierons. "Search Algorithms for Regression Test Case Prioritization". *IEEE Transactions on Software Engineering* 33.4 (Apr. 2007), pp. 225–237. ISSN: 0098-5589 (cit. on p. 39).

[170] Jung-Min Kim and Adam Porter. "A History-based Test Prioritization Technique for Regression Testing in Resource Constrained Environments". *Proceedings of the 24th International Conference on Software Engineering*. ICSE '02. Orlando, Florida: ACM, 2002, pp. 119–129. ISBN: 1-58113-472-X (cit. on p. 39).

[171] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. "TimeAware Test Suite Prioritization". *Proceedings of the 2006 International Symposium on Software Testing and Analysis*. ISSTA '06. Portland, Maine, USA: ACM, 2006, pp. 1–12. ISBN: 1-59593-263-1 (cit. on p. 39).

[172] Gordon Fraser and Franz Wotawa. "Test-case Prioritization with Model-checkers". *Proceedings of the 25th Conference on IASTED International Multi-Conference: Software Engineering*. SE'07. Innsbruck, Austria: ACTA Press, 2007, pp. 267–272 (cit. on p. 39).

[173] Mats Skoglund and Per Runeson. "A Case Study on Regression Test Suite Maintenance in System Evolution". *Proceedings of the 20th IEEE International Conference on Software Maintenance*. ICSM '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 438–442. ISBN: 0-7695-2213-0 (cit. on p. 40).

[174] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. "Understanding Myths and Realities of Test-suite Evolution". *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE '12. Cary, North Carolina: ACM, 2012, 33:1–33:11. ISBN: 978-1-4503-1614-9 (cit. on pp. 40, 111).

[175] Brian Robinson, Michael D. Ernst, Jeff H. Perkins, Vinay Augustine, and Nuo Li. "Scaling Up Automated Test Generation: Automatically Generating Maintainable Regression Unit Tests for Programs". *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 23–32. ISBN: 978-1-4577-1638-6 (cit. on p. 40).

[176] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. "ReAssert: Suggesting Repairs for Broken Unit Tests". *2009 IEEE/ACM International Conference on Automated Software Engineering*. Nov. 2009, pp. 433–444 (cit. on p. 40).

[177] M. Mirzaaghaei, F. Pastore, and M. Pezze. "Supporting Test Suite Evolution through Test Case Adaptation". *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. Apr. 2012, pp. 231–240 (cit. on p. 40).

[178] Alessandro Orso and Tao Xie. "BERT: BEhavioral Regression Testing". *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*. WODA '08. Seattle, Washington: ACM, 2008, pp. 36–42. ISBN: 978-1-60558-054-8 (cit. on pp. 41, 128).

[179] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. "Test-Suite Augmentation for Evolving Software". *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 218–227. ISBN: 978-1-4244-2187-9 (cit. on pp. 41, 128).

[180] Edward Miller and William E. Howden. *Software Testing and Validation Techniques*. 2nd ed. New York, USA: IEEE Comp. Soc. Press, 1981 (cit. on pp. 41, 88).

[181] E.T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. "The Oracle Problem in Software Testing: A Survey". *IEEE Transactions on Software Engineering* PP.99 (Nov. 2014). ISSN: 0098-5589 (cit. on pp. 41, 42).

[182] Phil McMinn, Mark Stevenson, and Mark Harman. "Reducing Qualitative Human Oracle Costs Associated with Automatically Generated Test Data". *Proceedings of the First International Workshop on Software Test Output Validation*. STOV '10.

Trento, Italy: ACM, 2010, pp. 1–4. ISBN: 978-1-4503-0138-1 (cit. on p. 42).

[183] M. Harman, Sung Gon Kim, K. Lakhotia, P. McMinn, and Shin Yoo. "Optimizing for the Number of Tests Generated in Search Based Test Data Generation with an Application to the Oracle Cost Problem". *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, Workshops Proceedings*. Apr. 2010, pp. 182–191 (cit. on p. 42).

[184] Fabrizio Pastore, Leonardo Mariani, and Gordon Fraser. "CrowdOracles: Can the Crowd Solve the Oracle Problem?" *Proceedings of the 6th International Conference on Software Testing, Verification and Validation*. ICST '13. Luxembourg: IEEE Computer Society, 2013, pp. 342–351 (cit. on pp. 42, 88).

[185] Sebastian Bauersfeld, Tanja E. J. Vos, Kiran Lakhotia, Simon Poulding, and Nelly Condori. "Unit Testing Tool Competition". *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. ICSTW '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 414–420. ISBN: 978-0-7695-4993-4 (cit. on pp. 42, 120).

[186] Gordon Fraser and Andrea Arcuri. "EvoSuite at the SBST 2013 Tool Competition". *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. ICSTW '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 406–409. ISBN: 978-0-7695-4993-4 (cit. on pp. 42, 120).

[187] Gordon Fraser and Andrea Arcuri. "EvoSuite at the Second Unit Testing Tool Competition". *Future Internet Testing*. Ed. by Tanja E.J. Vos, Kiran Lakhotia, and Sebastian Bauersfeld. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 95–100. ISBN: 978-3-319-07784-0 (cit. on pp. 42, 120).

[188] Gordon Fraser and Andrea Arcuri. "EvoSuite at the SBST 2016 Tool Competition". *Proceedings of the 9th International Workshop on Search-Based Software Testing*. SBST '16. Austin, Texas: ACM, 2016, pp. 33–36. ISBN: 978-1-4503-4166-0 (cit. on pp. 42, 120).

[189] Gordon Fraser and Andrea Arcuri. "EvoSuite: On the Challenges of Test Case Generation in the Real World". *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. ICST '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 362–369. ISBN: 978-0-7695-4968-2 (cit. on pp. 42, 131).

[190] Abdelilah Sakti, Gilles Pesant, and Yann-Gaël Guéhéneuc. "Instance Generator and Problem Representation to Improve Object Oriented Code Coverage". *IEEE Transactions on Software Engineering* 41.3 (Mar. 2015), pp. 294–313. ISSN: 0098-5589 (cit. on pp. 43, 148).

[191] Gordon Fraser and Andrea Arcuri. "Automated Test Generation for Java Generics". *Software Quality. Model-Based Approaches for Advanced Software and Systems Engineering*. Ed. by Dietmar Winkler, Stefan Biffl, and Johannes Bergsmann. Vol. 166. Lecture Notes in Business Information Processing. Springer International Publishing, 2014, pp. 185–198. ISBN: 978-3-319-03601-4 (cit. on p. 43).

[192] Andrea Arcuri and Gordon Fraser. "Java Enterprise Edition Support in Search-Based JUnit Test Generation". *Search Based Software Engineering: 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings*. Ed. by Federica Sarro and Kalyanmoy Deb. Cham: Springer International Publishing, 2016, pp. 3–17. ISBN: 978-3-319-47106-8 (cit. on p. 43).

[193] Andrea Arcuri, Gordon Fraser, and René Just. "Private API Access and Functional Mocking in Automated Unit Test Generation". *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. ICSE '17. IEEE, Mar. 2017, pp. 126–137. ISBN: 978-1-5090-6031-3 (cit. on p. 43).

[194] Jeremias Rößler, Andreas Zeller, Gordon Fraser, Cristian Zamfir, and George Candea. "Reconstructing Core Dumps". *Proceedings of the 6th International Conference on Software Testing, Verification and Validation*. ICST '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 114–123. ISBN: 978-0-7695-4968-2 (cit. on p. 43).

[195] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. "Does Automated White-Box Test Generation Really Help Software Testers?" *Proc. of ISSTA'13*. ACM, 2013, pp. 291–301 (cit. on pp. 48, 52).

[196] Gordon Fraser, Matthew Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. "Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study". *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2015) (cit. on pp. 48, 64, 78, 132, 154).

[197] Nan Li, Xin Meng, Jeff Offutt, and Lin Deng. "Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (Experience Report)". *Proc. of ISSRE'13*. IEEE, 2013, pp. 380–389 (cit. on p. 48).

[198] Konrad Jamrozik, Gordon Fraser, Nikolai Tillman, and Jonathan de Halleux. "Generating Test Suites with Augmented Dynamic Symbolic Execution". *Proceedings of the 7th International Conference on Tests & Proofs*. Ed. by Margus Veanes and Luca Viganò. Vol. 7942. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 152–167. ISBN: 978-3-642-38915-3 (cit. on p. 50).

[199] Nadia Alshahwan and Mark Harman. "Coverage and Fault Detection of the Output-uniqueness Test Selection Criteria". *Proc. of ISSTA'14*. ACM, 2014, pp. 181–192 (cit. on p. 52).

[200] Gordon Fraser and Andrea Arcuri. "A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite". *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24.2 (Dec. 2014), 8:1–8:42. ISSN: 1049-331X (cit. on pp. 55, 56, 63, 65, 73, 114, 132, 154).

[201] The University of Sheffield. *Iceberg HPC Cluster*. URL: `https://www.sheffield.ac.uk/cics/research/hpc/iceberg` (visited on 11/2017) (cit. on pp. 56, 116).

[202] Andrea Arcuri and Lionel Briand. "A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering". *Software Testing, Verification and Reliability (STVR)* 24.3 (2014), pp. 219–250. ISSN: 1099-1689 (cit. on pp. 56, 60, 93, 94, 119).

[203] András Vargha and Harold D. Delaney. "A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong". *Journal of Educational and Behavioral Statistics* 25.2 (2000), pp. 101–132. eprint: `https://doi.org/10.3102/10769986025002101` (cit. on pp. 56, 94, 119).

[204] Sreedevi Sampath, Renee Bryce, and Atif Memon. "A Uniform Representation of Hybrid Criteria for Regression Testing". *IEEE Transactions on Software Engineering* 39.10 (Oct. 2013), pp. 1326–1344. ISSN: 0098-5589 (cit. on p. 62).

[205] Dennis Jeffrey and Neelam Gupta. "Improving Fault Detection Capability by Selectively Retaining Test Cases During Test Suite Reduction". *IEEE Transactions on Software Engineering (TSE)* 33.2 (2007), pp. 108–123. ISSN: 0098-5589 (cit. on p. 62).

[206] Shin Yoo and Mark Harman. "Pareto Efficient Multi-objective Test Case Selection". *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. ISSTA '07. London, United Kingdom: ACM, 2007, pp. 140–150. ISBN: 978-1-59593-734-6 (cit. on p. 62).

[207] Shin Yoo and Mark Harman. "Using Hybrid Algorithm For Pareto Efficient Multi-Objective Test Suite Minimisation". *Journal of Systems and Software* 83.4 (2010), pp. 689–701. ISSN: 0164-1212 (cit. on p. 62).

[208] The University of Sheffield. *ShARC HPC Cluster*. URL: `https://www.sheffield.ac.uk/cics/research/hpc/sharc` (visited on 11/2017) (cit. on pp. 66, 93).

[209] Thomas Jansen, Kenneth A De Jong, and Ingo Wegener. "On the choice of the offspring population size in evolutionary algorithms". *Evolutionary Computation* 13.4 (2005), pp. 413–440 (cit. on p. 67).

[210] Aniruddha Basak and Jason Lohn. "A comparison of evolutionary algorithms on a set of antenna design benchmarks". *2013 IEEE Conference on Evolutionary Computation*. Ed. by Luis Gerardo de la Fraga. Vol. 1. Cancun, Mexico, June 2013, pp. 598–604 (cit. on p. 75).

[211] M. Wolfram, A. K. Marten, and D. Westermann. "A comparative study of evolutionary algorithms for phase shifting transformer setting optimization". *2016 IEEE International Energy Conference (ENERGYCON)*. Apr. 2016, pp. 1–6 (cit. on p. 75).

[212] Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele. "Comparison of multiobjective evolutionary algorithms: Empirical results". *Evolutionary computation* 8.2 (2000), pp. 173–195 (cit. on p. 75).

[213] Aurora Ramírez, José Raúl Romero, and Sebastián Ventura. "A Comparative Study of Many-objective Evolutionary Algorithms for the Discovery of Software Architectures". *Empirical Softw. Engg.* 21.6 (Dec. 2016), pp. 2546–2600. ISSN: 1382-3256 (cit. on p. 75).

[214] Roberto Erick Lopez-Herrejon, Javier Ferrer, Francisco Chicano, Alexander Egyed, and Enrique Alba. "Comparative analysis of classical multi-objective evolutionary algorithms and seeding strategies for pairwise testing of Software Product Lines". *Proceedings of the IEEE Congress on Evolutionary Computation, CEC*. 2014, pp. 387–396 (cit. on p. 75).

[215] Elmahdi Omar, Sudipto Ghosh, and Darrell Whitley. "Comparing Search Techniques for Finding Subtle Higher Order Mutants". *Proceedings of the Conference on Genetic and Evolutionary Computation*. GECCO '14. Vancouver, BC, Canada: ACM, 2014, pp. 1271–1278. ISBN: 978-1-4503-2662-9 (cit. on p. 75).

[216] K. Ghani, J. A. Clark, and Y. Zhan. "Comparing algorithms for search-based test data generation of Matlab Simulink models". *2009 IEEE Congress on Evolutionary Computation*. May 2009, pp. 2940–2947 (cit. on p. 75).

[217] S. Varshney and M. Mehrotra. "A differential evolution based approach to generate test data for data-flow coverage". *2016 International Conference on Computing, Communication and Automation (ICCCA)*. Apr. 2016, pp. 796–801 (cit. on p. 75).

[218] Laura Inozemtseva and Reid Holmes. "Coverage is Not Strongly Correlated with Test Suite Effectiveness". *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 435–445. ISBN: 978-1-4503-2756-5 (cit. on p. 78).

[219] Alberto Gonzalez-Sanchez, Rui Abreu, Hans-Gerhard Gross, and Arjan J.C. van Gemund. "Spectrum-Based Sequential Diagnosis". *Proceedings of the 25th AAAI International Conference on Artificial Intelligence (AAAI'11)*. Aug. 2011, pp. 189–196. ISBN: 978-1-57735-507-6 (cit. on pp. 78, 83, 86).

[220] Shin Yoo, Mark Harman, and David Clark. "Fault Localization Prioritization: Comparing Information-theoretic and Coverage-based Approaches". *ACM Transactions on Software Engineering and Methodology* 22.3 (July 2013), 19:1–19:29. ISSN: 1049-331X (cit. on pp. 78, 83).

[221] Hiralal Agrawal, Joseph R. Horgan, Edward W. Krauser, and Saul London. "Incremental Regression Testing". *Proceedings of the Conference on Software Maintenance*. ICSM '93. Washington, DC, USA: IEEE Computer Society, 1993, pp. 348–357. ISBN: 0-8186-4600-4 (cit. on pp. 78, 79).

[222] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. "A Survey on Software Fault Localization". *IEEE Transactions on Software Engineering* 42.8 (Aug. 2016), pp. 707–740. ISSN: 0098-5589 (cit. on pp. 79, 80).

[223] James A. Jones, Mary Jean Harrold, and John Stasko. "Visualization of Test Information to Assist Fault Localization". *Proceedings of the 24th International Conference on Software Engineering*. ICSE '02. Orlando, Florida: ACM, 2002, pp. 467–477. ISBN: 1-58113-472-X (cit. on p. 79).

[224] Raul Santelices, James A. Jones, Yanbing Yu, and Mary Jean Harrold. "Lightweight Fault-localization Using Multiple Coverage Types". *Proceedings of the 31st International Conference on Software Engineering*. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 56–66. ISBN: 978-1-4244-3453-4 (cit. on p. 79).

[225] Shaimaa Ali, James H. Andrews, Tamilselvi Dhandapani, and Wantao Wang. "Evaluating the Accuracy of Fault Localization Techniques". *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. ASE '09. IEEE

Computer Society, 2009, pp. 76–87. ISBN: 978-0-7695-3891-4 (cit. on p. 79).

[226] Chris Parnin and Alessandro Orso. "Are Automated Debugging Techniques Actually Helping Programmers". *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA '11. Toronto, Ontario, Canada: ACM, 2011, pp. 199–209. ISBN: 978-1-4503-0562-4 (cit. on pp. 79, 94).

[227] A. Gonzalez-Sanchez, R. Abreu, H.-G. Gross, and A.J.C. van Gemund. "Prioritizing Tests for Fault Localization through Ambiguity Group Reduction". *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 83–92 (cit. on pp. 79, 80, 85, 93).

[228] E. Alves, M. Gligoric, V. Jagannath, and M. d'Amorim. "Fault-Localization Using Dynamic Slicing and Change Impact Analysis". *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*. Nov. 2011, pp. 520–523 (cit. on p. 79).

[229] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. "Spectrum-Based Multiple Fault Localization". *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 88–99. ISBN: 978-0-7695-3891-4 (cit. on pp. 80, 82, 93).

[230] W. Mayer and M. Stumptner. "Evaluating Models for Model-Based Debugging". *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 128–137. ISBN: 978-1-4244-2187-9 (cit. on p. 80).

[231] Franz Wotawa. "Bridging the Gap Between Slicing and Model-based Diagnosis". *Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering*. SEKE '08. San Francisco, CA, USA: Knowledge Systems Institute Graduate School, 2008, pp. 836–841. ISBN: 1-891706-22-5 (cit. on p. 80).

[232] W. Eric Wong, Vidroha Debroy, Yihao Li, and Ruizhi Gao. "Software Fault Localization Using DStar (D*)". *Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability*. SERE '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 21–30. ISBN: 978-0-7695-4742-8 (cit. on p. 80).

[233] Johan De Kleer. "Diagnosing Multiple Persistent and Intermittent Faults". *Proceedings of the 21st International Joint Conference on Artificial Intelligence*. IJCAI '09. Pasadena, California, USA:

Morgan Kaufmann Publishers Inc., 2009, pp. 733–738 (cit. on p. 80).

[234] J de Kleer and B C Williams. "Diagnosing multiple faults". *Artif. Intell.* 32.1 (Apr. 1987), pp. 97–130. ISSN: 0004-3702 (cit. on p. 80).

[235] Alexander Feldman, Gregory Provan, and Arjan Van Gemund. "Computing Minimal Diagnoses by Greedy Stochastic Search". *Proceedings of the 23rd national conference on Artificial intelligence - Volume 2*. AAAI'08. Chicago, Illinois: AAAI Press, 2008, pp. 911–918. ISBN: 978-1-57735-368-3 (cit. on p. 80).

[236] Alexander Feldman and Arjan van Gemund. "A Two-step Hierarchical Algorithm for Model-based Diagnosis". *Proceedings of the 21st national conference on Artificial intelligence*. AAAI. Boston, Massachusetts: AAAI Press, 2006, pp. 827–833. ISBN: 978-1-57735-281-5 (cit. on p. 80).

[237] Franz Wotawa, Markus Stumptner, and Wolfgang Mayer. "Model-Based Debugging or How to Diagnose Programs Automatically". *Proceedings of the 15th international conference on Industrial and engineering applications of artificial intelligence and expert systems*. IEA/AIE '02. London, UK, UK: Springer-Verlag, 2002, pp. 746–757. ISBN: 3-540-43781-9 (cit. on p. 80).

[238] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990. ISBN: 0716710455 (cit. on p. 80).

[239] Rui Abreu and Arjan J. C. van Gemund. "A Low-Cost Approximate Minimal Hitting Set Algorithm and its Application to Model-Based Diagnosis". *Proceedings of the 8th Symposium on Abstraction, Reformulation, and Approximation*. SARA'09. 2009 (cit. on p. 80).

[240] John Carey, Neil Gross, Marcia Stepanek, and Otis Port. "Software Hell". *Business Week*. 1999, pp. 391–411 (cit. on p. 82).

[241] M. Avriel. *Nonlinear Programming: Analysis and Methods*. Dover Books on Computer Science Series. Dover Publications, 2003. ISBN: 9780486432274 (cit. on p. 82).

[242] R.A Johnson. "An Information Theory Approach to Diagnosis". *Reliability and Quality Control, IRE Transactions on* RQC-9.1 (Apr. 1960), pp. 35–35. ISSN: 0097-4552 (cit. on pp. 83, 86).

[243] Alberto Gonzalez-Sanchez, Eric Piel, Hans-Gerhard Gross, and Arjan J. C. van Gemund. "Prioritizing Tests for Software Fault Localization". *Proceedings of the 10th International Conference on Quality Software*. QSIC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 42–51. ISBN: 978-0-7695-4131-0 (cit. on pp. 85, 86).

[244] Alberto Gonzalez-Sanchez, Hans-Gerhard Gross, and Arjan J. C. van Gemund. "Modeling the Diagnostic Efficiency of Regression Test Suites". *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. ICSTW '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 634–643. ISBN: 978-0-7695-4345-1 (cit. on p. 86).

[245] Jeremias Rößler, Gordon Fraser, Andreas Zeller, and Alessandro Orso. "Isolating Failure Causes Through Test Case Generation". *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ISSTA 2012. Minneapolis, MN, USA: ACM, 2012, pp. 309–319. ISBN: 978-1-4503-1454-1 (cit. on pp. 89, 101).

[246] *Apache Commons Codec*. Apr. 2003. URL: `https://commons.apache.org/proper/commons-codec` (visited on 11/2017) (cit. on p. 89).

[247] *Apache Commons Codec — Bug report #99*. Mar. 2010. URL: `https://issues.apache.org/jira/browse/CODEC-99` (visited on 11/2017) (cit. on p. 89).

[248] *Apache Commons Compress*. Nov. 2003. URL: `https://commons.apache.org/proper/commons-compress` (visited on 11/2017) (cit. on p. 90).

[249] *Apache Commons Compress — Bug report #114*. May 2010. URL: `https://issues.apache.org/jira/browse/COMPRESS-114` (visited on 11/2017) (cit. on p. 90).

[250] *Apache Commons Math*. May 2003. URL: `http://commons.apache.org/proper/commons-math` (visited on 11/2017) (cit. on p. 90).

[251] *Apache Commons Math — Bug report #835*. July 2012. URL: `https://issues.apache.org/jira/browse/MATH-835` (visited on 11/2017) (cit. on p. 90).

[252] *Apache Commons Math — Bug report #938*. Mar. 2013. URL: `https://issues.apache.org/jira/browse/MATH-938` (visited on 11/2017) (cit. on p. 91).

[253] *Apache Commons Math — Bug report #939*. Mar. 2013. URL: `https://issues.apache.org/jira/browse/MATH-939` (visited on 11/2017) (cit. on p. 91).

[254] *Joda-Time*. Dec. 2003. URL: `http://www.joda.org/joda-time` (visited on 11/2017) (cit. on p. 91).

[255] *Joda-Time — Bug fixed in revision 941f59*. Apr. 2006. URL: `https://github.com/JodaOrg/joda-time/commit/941f593f7fe5654b07eac7c9f1998b894329f28e` (visited on 11/2017) (cit. on p. 91).

[256] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. "Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges". *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 201–211. ISBN: 978-1-5090-0025-8 (cit. on pp. 93, 154).

[257] M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Jānis Benefelds. "An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application". *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. ICSE-SEIP '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 263–272. ISBN: 978-1-5386-2717-4 (cit. on pp. 93, 117, 154).

[258] J.A. Rice. *Mathematical Statistics and Data Analysis*. Advanced series. Cengage Learning, 2007. ISBN: 9780534399429 (cit. on p. 93).

[259] R. A. Fisher. "On the Interpretation of χ^2 from Contingency Tables, and the Calculation of P". English. *Journal of the Royal Statistical Society* 85.1 (1922), pp. 87-94. ISSN: 09528385 (cit. on p. 94).

[260] Benoit Baudry, Franck Fleurey, and Yves Le Traon. "Improving Test Suites for Efficient Fault Localization". *Proceedings of the 28th International Conference on Software engineering*. ICSE. Shanghai, China: ACM, 2006, pp. 82–91. ISBN: 1-59593-375-1 (cit. on p. 100).

[261] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. "Directed Test Generation for Effective Fault Localization". *Proceedings of the 19th international symposium on Software testing and analysis*. ISSTA '10. Trento, Italy: ACM, 2010, pp. 49–60. ISBN: 978-1-60558-823-0 (cit. on p. 101).

[262] Manos Renieris and Steven P. Reiss. "Fault Localization With Nearest Neighbor Queries". *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*. ASE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 30–39 (cit. on p. 101).

[263] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. "A Practical Evaluation of Spectrum-based Fault Localization". *Journal of Systems and Software (JSS)* 82.11 (Nov. 2009), pp. 1780–1792. ISSN: 0164-1212 (cit. on p. 101).

[264] *Joda-Time*. Revision: 58069. Dec. 2003. URL: `https://github.com/JodaOrg/joda-time` (visited on 11/2017) (cit. on pp. 104, 114).

[265] Martin Fowler and Matthew Foemmel. "Continuous Integration". *Thought-Works* (2006) (cit. on p. 104).

[266] *Jenkins*. URL: `https://jenkins.io` (visited on 11/2017) (cit. on pp. 105, 133, 145).

[267] *CruiseControl*. URL: `http://cruisecontrol.sourceforge.net` (visited on 11/2017) (cit. on p. 105).

[268] *GitLab Continuous Integration & Deployment*. URL: `https://about.gitlab.com/features/gitlab-ci-cd` (visited on 11/2017) (cit. on p. 105).

[269] *Travis Continuous Integration*. URL: `https://travis-ci.org` (visited on 11/2017) (cit. on p. 105).

[270] *CircleCI: Continuous Integration and Delivery*. URL: `https://circleci.com` (visited on 11/2017) (cit. on p. 105).

[271] *Bamboo - Continuous integration, deployment & release management*. URL: `https://www.atlassian.com/software/bamboo` (visited on 11/2017) (cit. on p. 105).

[272] *Emma Plugin for Jenkins*. URL: `https://wiki.jenkins.io/display/JENKINS/Emma+Plugin` (visited on 11/2017) (cit. on pp. 105, 141).

[273] David Hovemeyer and William Pugh. "Finding Bugs is Easy". *ACM SIGPLAN Notices* 39.12 (Dec. 2004), pp. 92–106. ISSN: 0362-1340 (cit. on p. 105).

[274] *FindBugs Plugin for Jenkins*. URL: `https://wiki.jenkins.io/display/JENKINS/FindBugs+Plugin` (visited on 11/2017) (cit. on p. 105).

[275] Cu D Nguyen, Anna Perini, Paolo Tonella, and FB Kessler. "Automated Continuous Testing of Multi-Agent Systems". *The fifth European workshop on Multi-agent systems*. 2007 (cit. on p. 105).

[276] David Saff and Michael D. Ernst. "Reducing Wasted Development Time via Continuous Testing". *Proceedings of the 14th International Symposium on Software Reliability Engineering*. ISSRE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 281–. ISBN: 0-7695-2007-3 (cit. on pp. 105, 107).

[277] David Saff and Michael D. Ernst. "An Experimental Evaluation of Continuous Testing During Development". *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '04. Boston, Massachusetts, USA: ACM, 2004, pp. 76–85. ISBN: 1-58113-820-2 (cit. on pp. 105, 107).

[278] Gordon Fraser and Andrea Arcuri. "Sound Empirical Evidence in Software Testing". *ACM/IEEE International Conference on Software Engineering (ICSE)*. ICSE '12. Zurich, Switzerland: IEEE Press, 2012, pp. 178–188. ISBN: 978-1-4673-1067-3 (cit. on pp. 108, 114, 119).

[279] Thomas M. Mitchell. *Machine Learning*. 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997. ISBN: 0070428077, 9780070428072 (cit. on p. 109).

[280] S. Yoo and M. Harman. "Test Data Regeneration: Generating New Test Data from Existing Test Data". *Software Testing, Verification and Reliability (STVR)* 22.3 (May 2012), pp. 171–201. ISSN: 0960-0833 (cit. on p. 110).

[281] Suresh Thummalapenta, Jonathan de Halleux, Nikolai Tillmann, and Scott Wadsworth. "DyGen: Automatic Generation of High-coverage Tests via Mining Gigabytes of Dynamic Traces". *International Conference on Tests and Proofs*. TAP '10. Málaga, Spain: Springer-Verlag, 2010, pp. 77–93. ISBN: 3-642-13976-0, 978-3-642-13976-5 (cit. on p. 110).

[282] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. "Predicting Fault Incidence Using Software Change History". *IEEE Transactions on Software Engineering (TSE)* 26.7 (July 2000), pp. 653–661. ISSN: 0098-5589 (cit. on pp. 111, 139).

[283] *JUnit 4*. Dec. 2000. URL: `https://github.com/junit-team/junit4` (visited on 11/2017) (cit. on p. 114).

[284] *Java Native Access (JNA)*. Oct. 1998. URL: `https://github.com/java-native-access/jna` (visited on 11/2017) (cit. on p. 114).

[285] *Jedis*. June 2010. URL: `https://github.com/xetorthio/jedis` (visited on 11/2017) (cit. on p. 114).

[286] *The Java driver for MongoDB*. Jan. 2009. URL: `https://github.com/mongodb/mongo-java-driver` (visited on 11/2017) (cit. on p. 114).

[287] *RxJava: Reactive Extensions for the JVM*. Mar. 2012. URL: `https://github.com/ReactiveX/RxJava` (visited on 11/2017) (cit. on p. 114).

[288] *Rootbeer GPU Compiler*. Aug. 2012. URL: `https://github.com/awesomenix/rootbeer1` (visited on 11/2017) (cit. on p. 114).

[289] *Twitter4J*. June 2007. URL: `https://github.com/yusuke/twitter4j` (visited on 11/2017) (cit. on p. 114).

[290] *HTTP-Request*. Revision: b9d13. Oct. 2011. URL: `https://github.com/kevinsawicki/http-request` (visited on 11/2017) (cit. on p. 114).

[291] *JSON*. Revision: 4d86b. Dec. 2010. URL: `https://github.com/stleary/JSON-java` (visited on 11/2017) (cit. on p. 114).

[292] *jsoup: Java HTML Parser*. Revision: 80158. Jan. 2010. URL: https://github.com/jhy/jsoup (visited on 11/2017) (cit. on p. 114).

[293] *ScribeJava: Simple OAuth library for Java*. Revision: 26792. Sept. 2010. URL: https://github.com/scribejava/scribejava (visited on 11/2017) (cit. on p. 114).

[294] *Spark: A Tiny Web Framework for Java*. Revision: f1f06. May 2011. URL: https://github.com/perwendel/spark (visited on 11/2017) (cit. on p. 114).

[295] *Async HTTP Client*. Revision: 95886. Feb. 2010. URL: https://github.com/AsyncHttpClient/async-http-client (visited on 11/2017) (cit. on p. 114).

[296] *SpringSide: Spring Framework*. Revision: a11fc. Feb. 2012. URL: https://github.com/springside/springside4 (visited on 11/2017) (cit. on p. 114).

[297] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. "Comparing Non-adequate Test Suites Using Coverage Criteria". *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. ISSTA 2013. Lugano, Switzerland: ACM, 2013, pp. 302–313. ISBN: 978-1-4503-2159-4 (cit. on p. 117).

[298] Sina Shamshiri, Gordon Fraser, Phil Mcminn, and Alessandro Orso. "Search-Based Propagation of Regression Faults in Automated Regression Testing". *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. ICSTW '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 396–399. ISBN: 978-0-7695-4993-4 (cit. on p. 128).

[299] Zhihong Xu, Myra B. Cohen, Wayne Motycka, and Gregg Rothermel. "Continuous Test Suite Augmentation in Software Product Lines". *Proceedings of the 17th International Software Product Line Conference*. SPLC '13. Tokyo, Japan: ACM, 2013, pp. 52–61. ISBN: 978-1-4503-1968-3 (cit. on p. 129).

[300] Mariano Ceccato, Alessandro Marchetto, Leonardo Mariani, Cu D. Nguyen, and Paolo Tonella. "Do Automatically Generated Test Cases Make Debugging Easier? An Experimental Assessment of Debugging". *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2015) (cit. on p. 132).

[301] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. "Automated unit test generation for classes with environment dependencies". *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. ACM. 2014, pp. 79–90 (cit. on pp. 133, 134).

[302] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. "Generating TCP/UDP Network Data for Automated Unit Test Generation". *ACM Symposium on the Foundations of Software Engineering (FSE)*. 2015 (cit. on pp. 133, 134, 138).

[303] *Apache Maven*. URL: `https://maven.apache.org` (visited on 11/2017) (cit. on p. 133).

[304] *IntelliJ IDEA*. URL: `https://www.jetbrains.com/idea` (visited on 11/2017) (cit. on p. 133).

[305] *Maven Surefire Plugin*. URL: `http://maven.apache.org/surefire/maven-surefire-plugin` (visited on 11/2017) (cit. on p. 136).

[306] *Maven Central Repository*. URL: `https://search.maven.org` (visited on 11/2017) (cit. on p. 136).

[307] *Nexus Repository, Firewall, Lifecycle*. URL: `https://www.sonatype.com/products-overview` (visited on 11/2017) (cit. on p. 136).

[308] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Masami Hagiya, Yoshinori Tanabe, and Mitsuharu Yamamoto. "GRT at the SBST 2015 Tool Competition". *Proceedings of the Eighth International Workshop on Search-Based Software Testing*. SBST '15. Florence, Italy: IEEE Press, 2015, pp. 48–51 (cit. on p. 148).

[309] I. S. Wishnu B. Prasetya. "T3i: A Tool for Generating and Querying Test Suites for Java". *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: ACM, 2015, pp. 950–953. ISBN: 978-1-4503-3675-8 (cit. on p. 148).

[310] Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. "Hyper-heuristics: a survey of the state of the art". *Journal of the Operational Research Society* 64.12 (Dec. 2013), pp. 1695–1724. ISSN: 1476-9360 (cit. on p. 152).

[311] K. Herzig, S. Just, A. Rau, and A. Zeller. "Predicting defects using change genealogies". *Proceedings of the IEEE 24th International Symposium on Software Reliability Engineering*. ISSRE '13. Pasadena, CA: IEEE, 2013, pp. 118–127 (cit. on p. 153).

[312] Cagatay Catal and Banu Diri. "A Systematic Review of Software Fault Prediction Studies". *Expert Syst. Appl.* 36.4 (May 2009), pp. 7346–7354. ISSN: 0957-4174 (cit. on p. 153).

[313] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. "Predicting the Location and Number of Faults in Large Software Systems". *IEEE Transactions on Software Engineering* 31.4 (Apr. 2005), pp. 340–355. ISSN: 0098-5589 (cit. on p. 153).

[314] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. "Efficient Unit Test Case Minimization". *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*. ASE '07. Atlanta, Georgia, USA: ACM, 2007, pp. 417–420. ISBN: 978-1-59593-882-4 (cit. on p. 155).

DECLARATION

This thesis contains original work undertaken at The University of Sheffield, UK between 2013 and 2017.

Sheffield, UK, November 2017

José Carlos Medeiros de Campos

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

`https://bitbucket.org/amiede/classicthesis/`

Final Version as of November 2017.