MONASH University

Using Defect Prediction to Improve the Bug Detection Capability of Search-Based Software Testing

Anjana Perera

Doctor of Philosophy

A Thesis Submitted for the Degree of Doctor of Philosophy at
Monash University in 2022
Faculty of Information Technology

To my family, Sandamali, Ramani, Ananda and Anitha

Copyright notice

Abstract

Automated test generators, such as search-based software testing (SBST) techniques, replace the tedious and expensive task of manually writing test cases. SBST techniques are effective at generating tests with high code coverage. However, is high code coverage sufficient to maximise the number of bugs detected? In fact, the existing SBST approaches that are only guided by coverage have limitations when it comes to detecting bugs. We argue that SBST needs to be focused to search for test cases in likely defective areas rather than in likely non-defective areas of the code in order to maximise the likelihood of detecting bugs. Defect predictors give useful information about bug-prone areas in software. We leverage them to inform SBST where it should concentrate the search for test cases.

We formulate the objective of this thesis: *Improve the bug detection capability of SBST by incorporating defect prediction information.* We achieve the main objective via three research studies that aim at addressing the following three research objectives (RO); RO1) develop an approach that allocates time budget to classes for test generation based on defect prediction, RO2) understand the impact of imprecision in defect prediction for guiding SBST, and RO3) develop an SBST technique that uses defect prediction to guide the search process to likely defective areas. To this end, we introduce two SBST approaches guided by defect prediction, i.e., defect prediction guided SBST (SBST_{DPG}) (RO1) and predictive many objective sorting algorithm (PreMOSA) (RO3), and present a comprehensive experimental analysis of the impact of defect prediction imprecision on the bug detection performance of SBST (RO2).

The primary finding of this thesis is that defect prediction improves the effectiveness and efficiency of SBST in terms of detecting bugs. Our experimental evaluations on the Defects4J benchmark demonstrate the proposed SBST approaches guided by defect prediction detect up to 13.1% more bugs on average than the state-of-the-art SBST. In particular, our recommendations to improve the bug detection performance of SBST are to use class level defect prediction to allocate time budgets for test generation such as SBST_{DPG} (RO1) and to use method level defect prediction to guide the search process in SBST such as PreMOSA (RO3). We find that the recall of the defect predictor, which is indicative of false negatives, has a significant impact on the bug detection effectiveness of SBST with a large effect size, while the effect of precision, which is indicative of false alarms, is not of meaningful practical significance (RO2). We recommend that SBST techniques must handle the potential false negatives, i.e., missed bugs, in the predictions. In the context of combining defect prediction and SBST, our recommendation for practice is to increase the recall while maintaining precision at an acceptable level, e.g., 75%, if the SBST technique does not handle the potential false negatives. If SBST

handles the potential false negatives (e.g., PreMOSA), then it is beneficial to further improve the defect predictor performance only when there is a tight time budget for test generation. When there is a reasonably large time budget and SBST handles the potential false negatives, we recommend practitioners to not focus on improving the defect predictor performance beyond an acceptable level, e.g., recall and precision $\geq 75\%$.

Declaration

This thesis is an original work of my research and contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Signature: _____

Print Name: Anjana Perera

Date: Tuesday 6^{th} September, 2022

Publications during enrolment

Publications arising from this thesis are listed as follows.

1) Perera, A., Aleti, A., Böhme, M. and Turhan, B., 2020, September. Defect prediction guided search-based software testing. In 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 448-460). IEEE.

DOI: 10.1145/3324884.3416612
Pre-print: https://arxiv.org/abs/2109.12645

2) Perera, A., Aleti, A., Turhan, B. and Böhme, M., 2022. An experimental assessment of using theoretical defect predictors to guide search-based software testing. IEEE Transactions on Software Engineering, to appear.

DOI: 10.1109/TSE.2022.3147008 [Open Access]

3) Perera, A., 2020, December. Using defect prediction to improve the bug detection capability of search-based software testing. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (pp. 1170-1174).

DOI: 10.1145/3324884.3415286
Pre-print: https://arxiv.org/abs/2206.06549

The following papers are currently under review at the respective venues.

1) Perera, A., Turhan, B., Aleti, A. and Böhme, M., 2022. On the impact of imprecision in defect prediction for guiding search-based software testing. Under review in ACM Transactions on Software Engineering and Methodology.

The following publications are not part of the thesis, but were produced in parallel to the research described in the thesis.

1) Perera, A., Aleti, A., Tantithamthavorn, C., Jiarpakdee, J., Turhan, B., Kuhn, L. and Walker, K., 2022. Search-based fairness testing for regression-based machine learning systems. Empirical Software Engineering, 27(3), pp.1-36.

DOI: 10.1007/s10664-022-10116-7 [Open Access]

The following papers are not part of the thesis, but were produced in parallel to the research described in the thesis and are currently under review.

1) Martinez, M., Kechagia, M., Perera, A., Petke, J., Sarro, F., Aleti, A., 2022. Test-based patch clustering for automated program repair. Under review in Empirical Software Engineering.

Pre-print: https://arxiv.org/abs/2207.11082

Acknowledgements

I am immensely grateful to my supervisors, Associate Professor Aldeida Aleti, Professor Burak Turhan and Dr. Marcel Böhme, for their guidance, advice and encouragement. They were not only mentors to me, but also sponsors who provided me opportunities to grow as a researcher, and I am indebted to them for their support throughout the PhD journey.

I would like to express my deepest gratitude to my panel members, Associate Professor Ron Steinfeld, Dr. Li Li, Associate Professor Yuan-Fang Li and Dr. Chakkrit Tantithamthavorn, for their invaluable feedback and appreciation during the milestone seminars.

I owe my gratitude to Professor John Grundy for supporting me to refine my research topic and objectives in the initial part of the PhD.

My gratitude extends to my thesis examiners, Associate Professor Gregory Gay and Professor Phil McMinn, for reading my thesis and providing invaluable feedback and suggestions.

I would like to extend my sincere thanks to the past and present staff at the Graduate Research Student Services team and the Operations team at Faculty of Information Technology of Monash University. I am also grateful to Julie Holden for her insightful feedback and suggestions on better communicating my research.

I would also like to thank fellow PhD students and friends, Samodha Pallewatta, Malika Ratnayake, Danushka Liyanage, Alexander Ek, Shashi Jayaweera, Evelina Blomquist, Daniel Ehrenreich and Phoebe Heung for their support, companionship and the interesting conversations, technical and otherwise.

Last but not least, this endeavour would not have been possible without the unconditional support and love of my wife, Sandamali Kankanam Arachchige, who was always with me during the peaks and troughs of this journey supporting me in every possible way. I am deeply indebted to my parents, Ramani Seneviratne and Ananda Perera (who unfortunately passed away in 2013), and my brother, Anitha Perera, for guiding me to come this far in life.

This research was supported by the Faculty of Information Technology Research Scholarship and Faculty of Information Technology International Postgraduate Research Scholarship from the Faculty of Information Technology of Monash University.

Contents

List of Figures

List of Tables

Abbreviations

SDLC	Software Development Life Cycle
SBSE	Search-Based Software Engineering
GA	Genetic Algorithm
SBST	Search-Based Software Testing
RIP	Reachability, Infection and Propagation
RO	Research Objective
CI	Continuous Integration
CUT	Class Under Test
SQA	Software Quality Assurance
CDG	Control Dependency Graph
ITS	Issue Tracking System
VCS	Version Control Systems
TWR	Time Weighted Risk
MCC	Matthews Correlation Coefficient
BADS	Budget Allocation based on Defect Scores
DP	Defect Predictor
RQ	Research Question

Chapter 1

Introduction

In the wake of automating many manual and laborious tasks, software has become a key component in almost every system, e.g., online banking, trading, self-driving cars, airplanes, hospitals. It is not surprising to learn that software has direct or indirect impacts on the lives of humans, animals and other living things. What would happen if a software system that we highly rely on fails to function as we expect? In October 2018, a Lion Air Boeing 737 MAX 8 airplane crashed causing death to everyone on board. After 5 months of that incident, an Ethiopian Airlines Boeing 737 MAX airplane crashed resulting in deaths of everyone on board. Although the causes for these two crashes leading to a shocking number of fatalities were due to several reasons, it was found out that the fault was originated because of a bug in a part of the Boeing's flight management computer software [1]. The consequences of software failures are not only limited to fatalities. In August 2012, a defective software component in the Knight Capital's trade execution system caused a huge financial loss of $440 million to the company in just 45 minutes [2].

Software testing is an important process in the software development life cycle (SDLC) to ensure the delivery of high quality software products to the market. Various automated test generation techniques have been proposed in the literature [3–5] since manual test generation is a difficult and time consuming task, especially with the software systems becoming sophisticated and large. Automated test generators automatically generate tests for a given program to achieve a given testing goal such as maximising structural coverage.

Harman and Jones [6] coined the emerging research area search-based software engineering (SBSE), which uses metaheuristic search algorithms such as genetic algorithms (GA) or simulated annealing [7] to solve software engineering problems. Search-based software testing (SBST) is a sub-area of SBSE, which specifically focuses on tackling software testing problems such as test data generation. Research on SBST dates back to 1976 [8], and since then it has had a growing interest by the research community [9]. SBST techniques [9] have been very successful in automated test generation, and are widely used not only in academia, but also in the industry (e.g., Facebook [10, 11]).

SBST techniques use search methods to automatically generate high quality test cases for a particular system [12, 13]. These techniques focus on code coverage, and previous work show that they are very effective at achieving high coverage [14–17]. They can even cover more code than manually written test cases [18]. However, according to the reachability, infection and propagation (RIP) model [19–22], covering the buggy code is necessary but not sufficient to detect the bugs in the code. In fact, the results from previous studies show that the SBST techniques guided only by coverage have limitations when it comes to detecting bugs [23, 24]. For example, EvoSuite [3], a state-of-the-art SBST tool, could detect only 23% of the bugs, on average, from the Defects4J dataset [25], when it is given a three minutes time budget per class and using branch coverage as criterion [23]. For SBST, it is difficult to directly aim at generating test cases that detect bugs because it is computationally expensive to assess if a test case has detected a bug (i.e., semantic bugs) during the search process. To detect bugs, previous work in SBST resorted to coverage-based test generation when automated oracles were not available (e.g., semantic bugs) [23, 24, 26].

SBST techniques guided only by coverage search for tests to cover the whole code base by assuming every part of the code is equally important to cover. They have no guidance in terms of where the buggy code is likely to be located, and hence spend most of the search effort in non-buggy code which constitutes a greater portion of the code base. Only a small fraction of the code base constitutes the buggy code [27]. For example, a project may have hundreds or thousands of classes, but only a few classes may be buggy. In those few buggy classes, the bug is most likely to be located in a few buggy methods. This means that it is ineffective for SBST techniques in terms of detecting bugs to try to search for test cases that cover the whole code base when only a small portion of it

is actually buggy. We expect SBST techniques to have better bug detection capability if the search for tests targets more the buggy areas in the code.

Defect predictors are well-studied techniques for estimating the bug-prone areas in software. The predictions can be coarse-grained like package [28] and file/class [29, 30] levels, or fine-grained like method level [29, 31, 32]. They use various features related to metrics like code size [33], code complexity [34], change history [35] and organisation [36] to predict whether a package, file or method is defective. Defect predictors have been shown to be effective at locating bugs in software [29, 37, 38]. As a result of their efficacy, organisations use defect predictors to help developers in code reviews [39, 40] and to focus their limited testing efforts on likely buggy parts in code [41]. In addition, defect prediction has been successfully used to inform other automated testing techniques, e.g., Paterson et al. [42] proposed a test case prioritisation strategy.

We hypothesise that we can improve the bug detection capability of SBST by informing SBST of the areas in the code which are likely to be buggy. It is not known where the buggy code is prior to running tests. We argue that defect prediction can be used to identify bug-prone areas in the code and inform the SBST techniques where the bug is likely to be. Then SBST can be guided to search for more tests covering the likely buggy areas in order to increase the likelihood of detecting the bug. Therefore, we formulate the main research objective of this thesis as to improve the bug detection capability of SBST by incorporating defect prediction information.

To achieve the main research objective, we devise three research objectives (RO) and conduct three research studies to address them. The three research objectives are

RO1) to develop an approach that allocates time budget to classes for test generation based on defect prediction,

RO2) to understand the impact of imprecision in defect prediction for guiding SBST, and

RO3) to develop an SBST technique that uses defect prediction to guide the search process to likely defective areas.

The first objective aims at improving the bug detection performance of the test suites generated by SBST by addressing the problem of time budget allocation using coarse-grained defect prediction (i.e., at class level) as guidance for the SBST technique. In the second one, we systematically investigate the impact of the errors in defect predictions on the bug detection performance of SBST. The third objective aims at improving the bug detection performance of SBST using fine-grained defect prediction (i.e., at method level) inside the search process of SBST to guide the search for tests to likely buggy areas in code.

The proposed time budget allocation approach in RO1 (Chapter 5) can be used to allocate the available time budget for test generation of a whole project in the context of continuous integration (CI) with maximising the chances of detecting bugs. The comprehensive experimental analysis conducted in RO2 (Chapter 6) provides insights into the types of prediction errors that have an impact on the bug detection performance of SBST. The recommendations we make based on the second study can be used to build better SBST techniques that handle the potential errors with significant impact. We also provide actionable conclusion for the defect prediction research community to focus on when improving defect predictors in the context of combining them with SBST. The proposed SBST technique in RO3 (Chapter 7) can be used together with the time budget allocation approach proposed in RO1 to optimally utilise the allocated time budget to a class by further guiding the search for tests towards buggy areas in a class.

1.1 Research Problem

The results from previous studies show that SBST techniques only guided by coverage, despite achieving high code coverage [14–17], are not as effective in detecting bugs [23, 24]. A test oracle is needed to determine if a test case has detected a bug. When the automated oracles are not available, which is usually the case for semantic bugs, SBST techniques only focus on achieving coverage criteria because it is difficult for them to directly target generating tests that detect bugs. However, in order to detect bugs, coverage of buggy code alone is not sufficient according to the RIP model.

SBST tools like EvoSuite employ search methods such as GA to generate a test suite for a class under test (CUT) according to a given test goal like maximise statement

coverage, branch coverage, method coverage, or a combination of the three. One of the crucial parameters that has to be tuned in the GA is the time budget, which is used as a stopping criterion for the GA. Previous studies investigated the improvement of bug detection performance of SBST caused by increasing the time budget [24, 26, 43]. This is based on the hypothesis that allocating a higher time budget allows the search method to further explore the search space of possible test inputs, thus increasing the probability of finding the optimum. Despite the increase in bug detection effectiveness when increasing the time budget, this idea of allocating large amount of time budget has limitations in its practicability [44].

Real world projects are usually very large and can have thousands of classes [45]. If an SBST tool runs test generation for 10 minutes per class as done in [43], then it will take at least 166 hours to finish the task for the whole project having thousands of classes. Even the CI systems may not be able to allocate such large amount of resources (e.g., 166 hours run-time) as they also have other processes competing for the available resources. Given the limited computational resources available in practice [44] and the expectation of faster feedback cycles from testing in modern software development practices (e.g., agile), it is not viable to allocate such large amount of time budgets like 10 minutes per class for test generation.

SBST techniques use fitness functions to evaluate the fitness of generated tests and they are based on the test goals (e.g., branch, method, line, exception and weak mutation coverage) that need to be achieved. The generated test suites for different test goals may capture different behaviour of the program under test. Previous work investigated the bug detection performance of SBST when it is using different fitness functions based on single and combinations of test goals [24]. The results indicate that the most effective fitness function is based on branch coverage since it aims at thoroughly exploring the program structure. However, SBST could only detect on average 25.24% of bugs from the Defects4J dataset when using branch coverage based fitness function and 10 minutes of time budget. This suggests that the bug detection performance of SBST is still not impressive even after using a large time budget and the most effective fitness function.

We argue that one main limitation in SBST techniques that focus on code coverage is that they have no guidance in terms of where the buggy code is likely to be located in the code base. The buggy code constitutes only a smaller portion of the code base [27].

Hence, SBST techniques are more likely to waste the search efforts by trying to cover the whole code base. We expect SBST techniques to achieve a better bug detection performance, if they target the search for tests more in the buggy areas in the code.

In conclusion, SBST techniques have been shown to struggle in terms of detecting bugs despite achieving impressive code coverage results. While increasing the time budget allocated for the search and branch coverage based fitness function indicated increased bug detection performance, there is still more room for improvement and also these ideas have limitations in practicability. We argue that SBST techniques should focus more on searching for tests in likely defective areas in code in order to effectively and efficiently detect bugs especially given the resource constrained nature in practice.

1.2 Research Objectives

The main research objective of this thesis is to improve the bug detection capability of SBST by incorporating defect prediction information. In order to detect a bug, a test case has to satisfy all three conditions of the RIP model. SBST techniques are very effective at satisfying the reachability condition of the RIP model, i.e., reaching the buggy code. Despite covering the buggy code, SBST techniques are shown to be not as effective at generating tests that can cause an incorrect program state (infect) and then propagating that to a failure of the program. SBST techniques that target high code coverage search for test cases to cover the entire code base. However, the buggy code constitutes a smaller portion of the whole code base. For example, Wattanakriengkrai et al. [27] reported only a 2%-28% of files are buggy in the systems they studied and as little as 1%-3% of lines in a buggy file are buggy. As we discussed in Section 1.1, we argue that it is likely effective for SBST to concentrate the search for tests in buggy code when it comes to bug detection. This way it can increase the likelihood of successful infection and propagation of the bug. We hypothesise that SBST techniques can use the information of buggy locations as given by defect predictors to focus the search for tests more in the likely buggy code in order to increase its bug detection capability.

Figure 1.1 shows an overview of our main research objective. The requirement is to generate test suites with improved bug detection for a project containing classes. We

plan to use defect predictors to extract characteristics of the code that can derive defectiveness and predict the future defective code in the system. We study how this location information about bugs can be used to inform SBST techniques in order to increase bug detection. This thesis makes contributions to the SBST knowledge by proposing approaches for SBST to exploit buggy location information given by defect predictors.

To achieve our main research objective, we set out three research objectives, which we will describe in the next part of the section.

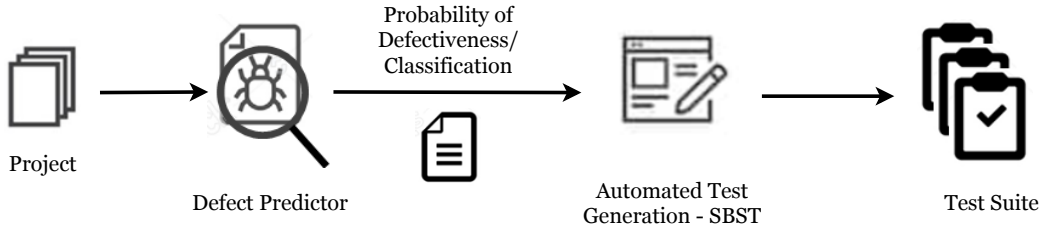

FIGURE 1.1: Main research objective overview

1.2.1 Time Budget Allocation

SBST tools such as EvoSuite generate test suites for each class in the project separately. In object-oriented design, a general rule of thumb is to design highly cohesive and loosely coupled classes. Hence, unlike in within a class, there is not much heuristics to be derived from a different class which can be used in another class to guide the test generation. Also, SBST will run into scalability issues if it is to aggregate all the coverage targets from multiple classes in a project. For example, it will require large memory for the larger test suites generated for multiple classes during run-time. Thus, SBST tools usually run test generation for each class independently.

As we discussed in Section 1.1, one of the crucial parameters that needs to be tuned carefully is the time budget allocated for the search of tests. Time budget is a stopping criterion for the GA. Allocating a higher time budget allows SBST to extensively explore and exploit the search space of possible test inputs, thereby increasing the chances of finding the optimal tests. For example, previous work showed the improved bug detection performance of EvoSuite when the time budget is increased from 2 minutes to 10 minutes [24]. However, due to the resource constrained nature in the development environments, it is not practical to allocate higher time budgets for every class in a project.

For small projects, it is feasible to run test generation individually for each class in the project outside of working hours in developer machines. Real-world projects, however, are usually very large, e.g., a modern car has millions of lines of code and thousands of classes [45], and they require a significant amount of resources (e.g., time) to run the test generation tools for each class in the project. Even in an open source project like Apache Commons Math [46], there are around 800 classes. In a project like this, it would take at least 13-14 hours to run test generation with spending just one minute per each class.

Modern software development practices, e.g., agile, expect faster feedback cycles from testing. Hence, the most viable option for running frequent test generation is to adapt SBST in the CI systems. However, CI systems are already dealing with high demands for computational resources from the existing processes in the system such as regression testing, code quality checks, integration testing and project builds. If SBST to be successfully adapted in the CI systems, one major challenge it has is to use minimal resources possible such that it does not idle or interrupt other processes that are already in the system. Due to this high demand and the limited availability of computational resources in organisations, the development environments are usually resource constrained. Therefore, it is necessary to optimally utilise the available computational resources, e.g., time budget, to generate test suites for a whole project with maximising the chances of detecting bugs.

Not every class in a project is buggy. Allocating an equal time budget to all the classes in a project is a sub-optimal strategy in terms of bug detection. In an ideal scenario, SBST should run test generation on all the buggy classes and non-buggy classes should be left out from test generation. However, the practitioners do not know which classes are buggy prior to running tests. We plan to use class level defect prediction to get information of the probability of defectiveness of classes in a project. Using this information, we differentiate classes in a project in order to allocate the available time budget to those classes. Our aim is to develop a time budget allocation approach guided by defect prediction for SBST to improve the bug detection performance of the test suites generated. Therefore, we formulate the following research objective;

RO1: Develop an approach that allocates time budget to classes for test generation based on defect prediction.

1.2.2 Impact of Defect Predictor Imprecision

There is a plethora of defect predictors which have been proposed over the past 40 years [47]. Often, the predictions produced by defect predictors are not perfectly accurate. For example, Zimmermann et al. [48] found that only 21 out of 622 cross-project defect predictor combinations to have recall, precision and accuracy greater than 75%. In their systematic literature review, Hall et al. [49] reported defect predictor performances from as low as 5% and 25% to as high as 95% and 85% for precision and recall, respectively. Hosseini et al. [50] also reported similar findings in their systematic literature review of cross-project defect predictors. This shows that the defect predictors have a wavering performance and we cannot expect their predictions to be accurate all the time.

Defect prediction researchers usually aim at elevating both recall and precision. A lower recall and precision can significantly hamper the benefits of defect predictors for the developers who usually manually inspect or test the predicted buggy code to find bugs. Previous work report the developers' opinions about the defect predictor performance [39, 41, 47]. Poor recall of the defect predictor means that there are higher false negatives (i.e., labelling buggy code as non-buggy). This can lead the developers to completely miss bugs, since they are not likely to inspect the code with non-buggy label which usually constitutes a large portion of the code base. Poor precision means there are higher false positives (i.e., wrongly labelling non-buggy code as buggy). False positives cause developers to waste their precious time on inspecting non-buggy code, which eventually leads to losing trust on the defect predictor [39, 41]. In the eyes of the developers, higher precision is more important compared to higher recall in a defect predictor, because higher precision means lower false positives [47] and less waste of their efforts.

The impact of the defect prediction errors on the bug detection performance of SBST guided by defect prediction has not been studied before. Hence, the impact of false negatives (indicated by recall) and false positives (indicated by precision) on the SBST

techniques is unknown. False negatives may result in SBST techniques not generating tests for buggy areas in code because they are not labelled as buggy by the predictor. This could lead the SBST techniques to miss bugs. On the other hand, false positives may not be as important in the context of combining defect prediction and SBST, since searching for tests in false positives may not be a significant burden to the automated test generation techniques in contrast to a developer manually inspecting the false positives. To answer this question, *"What is the impact of imprecise predictions on the bug detection performance of SBST?"*, we aim at investigating the impact of defect prediction imprecision for guiding SBST. Therefore, we formulate the following research objective;

> RO2: Understand the impact of imprecision in defect prediction for guiding search-based software testing.

1.2.3 Guiding the Search Process with Defect Prediction

Coverage is often used to define the fitness function used in SBST techniques [14, 15, 51]. During the search process, test cases with high coverage are considered of higher quality, and the aim of the search process is to generate test cases that maximise coverage. The existing SBST techniques, therefore, treat all the coverage targets in the CUT as equally important to cover. However, only one or a few methods in a class are buggy, hence, it is likely to be ineffective in terms of bug detection to search for tests to cover targets that contain non-buggy methods. The generated test suites by existing SBST techniques, despite having high code coverage, have only a few test cases that at least cover the buggy methods. In the context of detecting bugs, we identify this as a main limitation in SBST techniques guided only by coverage.

As we discussed under RO1 (Section 1.2.1), due to the usual resource limitations in practice, requirement of frequent feedback from testing in modern software development practices, and the high demand for the resources in CI systems, it is very important for SBST techniques to optimally utilise the available time budget for test generation. In RO1, we propose a defect prediction guided time budget allocation technique that allocates higher time budgets to highly likely to be buggy classes. Despite receiving higher time budget for test generation, it is still sub-optimal to spend a majority of the

allocated time budget to search for test cases that cover non-buggy targets in the class. Therefore, it is necessary to significantly limit the search resources spent on covering non-buggy targets and increase the test coverage for the buggy targets in the CUT in order to increase the chances of detecting the bugs.

We hypothesise that augmenting coverage information used by SBST techniques with defect prediction information improves the performance of SBST in terms of bug detection. We argue that coverage guidance alone is not sufficient to effectively guide the search process to find tests that detect bugs. We plan to use method level defect prediction to get information of which methods are likely to be buggy and which methods are not. Using this information, the SBST technique can exploit the targets that contain likely buggy methods to increase the test coverage for those targets. Our aim is to develop an SBST technique that uses defect prediction information inside the search process along with coverage information to guide the search for test cases towards likely buggy targets in the class. Therefore, we formulate the following research objective;

> RO3: Develop an SBST technique that uses defect prediction to guide the search process to likely defective areas.

1.3 Research Contributions

This section lists the contributions produced in this research to knowledge and practice.

1.3.1 Contributions to Knowledge

1. We demonstrate class level defect prediction can be used to guide SBST to efficiently and effectively detect bugs through time budget allocation in a resource constrained environment. The proposed solution combines defect prediction at class level and SBST by allocating higher time budgets to highly likely to be defective classes. It is experimentally evaluated using 434 real bugs from six open source Java projects (which took roughly 34,600 CPU-hours).

2. Through a comprehensive experimental analysis involving 420 real bugs from six open source Java projects (which took roughly 180,750 CPU-hours), we demonstrate the recall of the defect predictor has a significant impact on the bug detection

performance of SBST with a large effect size. On the other hand, the impact of precision of the defect predictor is not of practical significance. We identify that SBST techniques must handle potential false negatives in the predictions when defect predictors are used to guide the search for tests. For SBST, it is important to be informed of most of the buggy targets even at the expense of acceptable level of false positives. In the context of combining defect prediction and SBST, we recommend the researchers to target higher recall while having a sufficiently high precision, instead of trying to elevate both recall and precision. A replication package is made available to be used by other studies and can be found at https://doi.org/10.6084/m9.figshare.16564146.

3. We demonstrate method level defect prediction can be used to guide the search process in SBST along with coverage information to improve the bug detection performance of SBST. The proposed SBST technique approaches the test generation problem as a many-objective optimisation problem and uses buggy methods predictions to decide which objectives (i.e., coverage targets) to prioritise in the search process. It exploits the likely buggy targets while exploring the likely non-buggy targets with a lesser priority and successfully accounts for false negatives in the defect predictions. The proposed solution is experimentally evaluated using 420 real bugs from six open source Java projects and using theoretical defect predictors with acceptable performance as recommended by Zimmermann et al. [48] (which took roughly 48,800 CPU-hours). A replication package is made available to be used by other studies and can be found at https://doi.org/10.6084/m9.figshare.19027778.

1.3.2 Contributions to Practice

1. We develop a novel time budget allocation approach for SBST to run test generation with improved bug detection performance. The proposed approach is publicly available as a tool at https://github.com/SBST-DPG/sbst-dpg. When practitioners want to run test generation with SBST for a large project in a CI system or developer machines, they can leverage our proposed approach to allocate time budgets to classes in the project.

2. We develop a novel SBST technique that augments defect prediction information with coverage information to guide the search process towards buggy areas in the CUT with improved bug detection performance. The proposed SBST technique is implemented in the EvoSuite framework and is publicly available at https://github.com/premosa-sbst/evosuite. We recommend practitioners to use our proposed technique with a method level defect predictor with an acceptable performance (i.e., recall and precision $\geq 75\%$) in place of an SBST technique only guided coverage when either a large or a tight time budget is available in a resource constrained environment. Moreover, this technique can be used together with the proposed time budget allocation approach to further improve the bug detection performance.

1.4 Thesis Structure

The organisation of the thesis following Chapter 1, the introduction, is as follows.

Chapter 2 describes the background of SBST and defect prediction that would be needed to understand the rest of the thesis. The proposed SBST approaches in the thesis address the test generation problem as a many-objective optimisation problem. This chapter presents the optimisation problem in many-objective formulation. We also describe the current state-of-the-art SBST technique for unit testing, dynamic many objective sorting algorithm (DynaMOSA) [14], which is used in the benchmark methods in our studies.

Chapter 3 presents a review of the SBST and defect prediction literature. We identify the research gap in related work that we address in this thesis by using defect prediction as guidance. This is followed by a review of previous work in using defect prediction for automated software testing and positioning our research in this area of work.

Chapter 4 describes the methodological aspects of the research conducted in this thesis to address the three research objectives.

Chapters 5, 6 and 7 are contribution chapters and each of them presents the research studies conducted to address the three research objectives described in Section 1.2. In particular, Chapter 5 presents the proposed time budget allocation approach to address the RO1. Chapter 6 presents the study designed to systematically investigate the impact

of defect prediction imprecision on the bug detection performance of SBST in order to achieve the RO2. Chapter 7 presents the proposed SBST technique guided by defect prediction to achieve the RO3.

Chapter 8 discusses the findings of this research and their implications to knowledge and practice.

Chapter 9 discusses potential ideas and directions for future work.

Chapter 2

Background

2.1 Introduction

In order to ensure the delivery of high quality software, development teams exercise software quality assurance (SQA) activities, e.g., code review and software testing. Bugs can be introduced to the code in different times during development, and they can reside in the software systems for long periods like two years [52]. In order to detect these bugs, developers have to exhaustively test everywhere in the system. In the time of fast-paced software development and with software systems becoming sophisticated and large, it is not viable to conduct exhaustive testing. To adapt to this rapid changes in software development, practitioners can leverage defect prediction models [39] to predict the areas in the software (e.g., files) that are likely to be buggy. Once these areas have been identified, they can prioritise their limited SQA resources to the most risky parts of the code.

Manually writing tests is labour intensive, expensive, difficult and error-prone task. In order to address these problems, researchers have studied automating the test generation task. So far, various automated test generation techniques have been introduced [3–5, 53]. Automated test generation techniques that apply search-based optimisation techniques belong to the area of search-based software testing (SBST) [3]. With the use of these techniques, practitioners are able to focus their efforts to work on creative tasks while letting the automated techniques take care of the labour intensive and monotonous test generation tasks.

In this thesis, we argue that SBST techniques should focus the test generation efforts more towards the likely buggy areas in software to increase the chances of detecting bugs. To do that, we propose to leverage defect prediction to inform the SBST techniques of the likely buggy areas. In this chapter, we describe the background of SBST and defect prediction in detail.

2.2 Search-Based Software Testing

Harman and Jones formally established the search-based software engineering (SBSE) research area in 2001 [6]. SBSE is a sub-area of software engineering where software engineering problems are formulated as search problems and metaheuristic search algorithms like genetic algorithms (GA) [54], ant colony optimisation [55], particle swarm optimisation [56] and simulated annealing [57] are applied to search for the optimal solutions. SBSE research extends to all phases of the software engineering process; software testing and debugging [58, 59], requirements analysis [7], software design [60], software maintenance [61] and project management [62].

SBST is a sub-area of SBSE where research in SBST focuses on solving software testing problems by applying metaheuristic search methods. Majority of the research work in SBSE is applied in the context of software testing. According to a survey in 2012, SBST research takes up 54% of the research done in SBSE [63]. Even though SBSE emerged as a key research area in 2001, SBST research dates back to 1976 [8]. From 1976 to 2013, the number of publications in SBST has shown a polynomial rise suggesting a growing interest in the research community [9]. Each year since 2008, the research community has been conducting a dedicated SBST workshop, i.e., International Workshop on Search-Based Software Testing, co-locating with major software engineering conferences, i.e., International Conference on Software Engineering (ICSE) and International Conference on Software Testing, Verification and Validation (ICST), with the aim of bringing together the research and industrial communities from SBST. SBST research has made its way to the industry as well, for example, evolutionary testing of autonomous parking system at Daimler [64] and search-based prediction of faults at Ericsson [65]. One of the biggest successes of SBST research is the Sapienz tool which is deployed in production at Facebook [10]. Sapienz is a search-based android testing tool that optimises test sequences to maximise coverage and fault revelation.

SBST research has been applied to solve a plethora of software testing problems such as functional testing [64, 66], safety testing [67], security testing [68], integration testing [69], robustness testing [70], exception testing [71, 72], structural testing [53, 73], fairness testing [74], energy testing [75], test prioritisation [76], unit testing [14, 23, 26, 73] and regression testing [44]. Its use extends to various applications; self-driving cars [66, 77, 78], healthcare software [74], web applications [68], mobile applications [10, 11, 75], cyber-physical systems [76], data processing systems [70], short-term conflict alert systems of air traffic controllers [67], and financial software [26]. Test data generation for unit testing is one of the well studied problems in SBST [79]. In particular, structural coverage of unit testing has been a main focus [14, 15, 44, 51, 73, 80, 81]. Manually writing tests is a time consuming task and SBST techniques have been able to automate this process. They have been shown to achieve higher code coverage than manually written tests, suggesting their effectiveness [18].

2.2.1 Search-Based Software Testing for Unit Test Generation

SBST techniques use meta heuristics search algorithms to search for the optimal solution in a large solution space for specific criteria (e.g., maximise branch coverage). The search algorithms can be global solvers like genetic algorithms [3, 14, 51], particle swarm optimisation [82] and ant colony optimisation [83] or local solvers like simulated annealing [84], hill climbing [85] and Korel's Alternating Variable Method [86]. By far genetic algorithms have been commonly used in the area of SBST for test case generation. It is based on the Darwinian evolution and survival of the fittest theorem [87].

2.2.1.1 Problem Representation

To apply a genetic algorithm to a test generation problem, the valid solutions to the problem need to be represented, this is known as problem representation. Each solution in the search space is called a *chromosome* or an *individual*. GA evolves a collection of individuals, called *population*, through a series of *generations* to find the optimal solution. An individual can be a test case [14, 15] or a test suite [3, 51] depending on the problem representation chosen by the SBST technique. A test case is a sequence of statements $t = \{s_1, \ldots, s_l\}$, where l is the length of the test case. A statement s_i, where $i \in [1, l]$, can be of any of the following types; i) primitive statements, ii) constructor

statements, iii) field statements, iv) method statements, and v) assignment statements. A test program that executes the program under test can be synthesised using statements from the above five types. Primitive statements represent the assignment of primitive values to variables, e.g., int var0 = 100;. Constructor statements represent creating new instances of a given class by calling a constructor, e.g., Foo foo0 = new Foo();. Field statements represent accessing public member fields of instances/classes, e.g., int var1 = foo0.bar;. Method statements represent invoking methods on instances or calling static methods, e.g., int var2 = foo0.baz();. Assignment statements represent assignment of values to elements of arrays and public member fields of instances/classes, e.g., var3[0] = 100;. A test suite is a collection of test cases $T = \{t_1, \ldots, t_n\}$, where n is the size of the test suite.

2.2.1.2 The Search Steps and Genetic Operators

Figure 2.1 shows the main steps of genetic algorithm which are *initialisation*, *evaluation*, *selection*, *crossover*, *mutation* and *reinsertion* [88]. In the initialisation step, a set of randomly generated M number of individuals is created as the initial population. Upon creating the initial population, GA enters a loop comprising of the steps evaluation, selection, crossover, mutation and reinsertion. These steps are performed until termination criteria is met, e.g., maximum time budget, test goal and maximum number of generations. Termination criteria decides when the search for tests should stop. For example, the search can stop once the allocated time budget runs out or the test goal, e.g., maximum branch coverage, has been met. In practice, maximum time budget is more suitable as a stopping condition than maximum number of generations since the testing activities are largely dictated by the availability of computational resources. Different SBST techniques take different time to perform the same number of generations given the same computational power.

In the evaluation step, GA evaluates the fitness of all the individuals in the population using a fitness function. The purpose of the fitness function is to guide the search towards the optimal solution in order to meet the objective, e.g., maximise branch coverage. Fitness of an individual measures the extent that individual satisfies the objective, and it is used in selection and reinsertion steps to decide whether to select or drop the individual. Branch distance and approach level are widely used to formulate

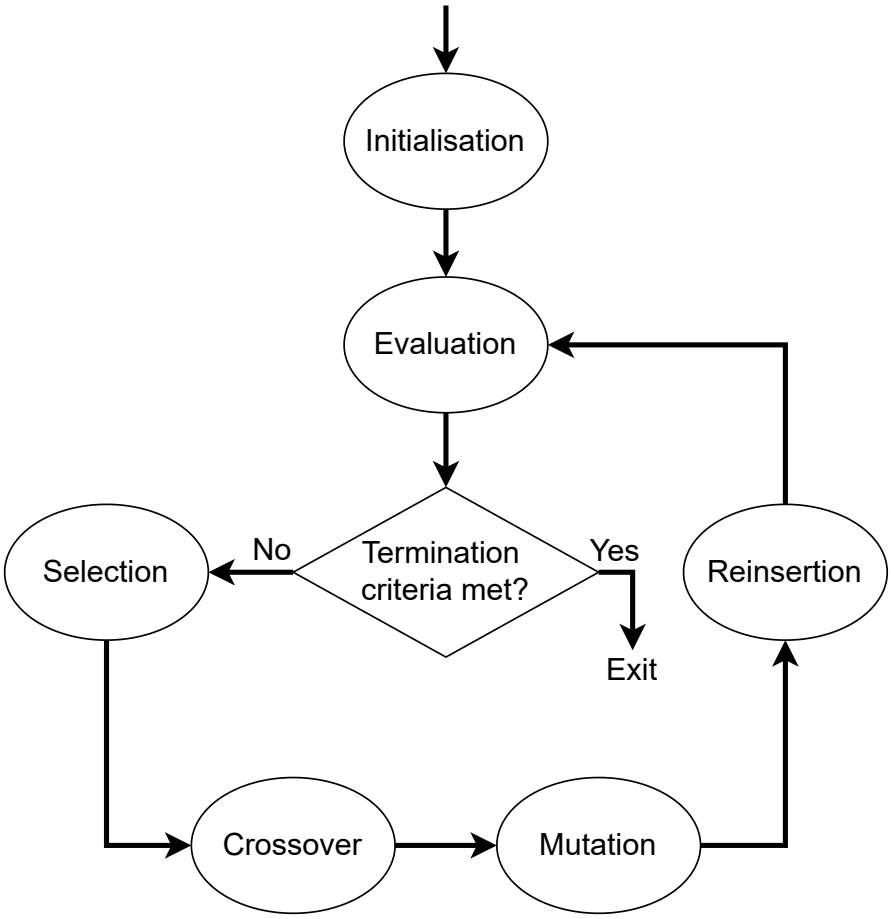

FIGURE 2.1: Overview of the main steps of a genetic algorithm.

fitness functions targeted at maximising branch coverage as an objective [15, 89]. Branch distance [86, 87] is calculated at the predicate where the execution path diverges away from the desired path to the target branch. It measures how close it is to switch the outcome at the predicate (TRUE → FALSE or FALSE → TRUE) causing the test case to execute the desired alternative branch. Branch distance is used to guide the search to find inputs that evaluate the desired branches (TRUE or FALSE) at the control dependent nodes of the target branch. Approach level is calculated based on the distance (i.e., number of control dependent nodes) between the branch where the execution diverges from the desired execution path and the branch under consideration. If the execution of a test case is closer to reaching the target branch in terms of the control dependent nodes in the control flow graph, then it is rewarded with a lower approach level. A lower approach level means better fitness of an individual. Approach level is used to guide the search to find tests that execute along the desired path to the target branch.

Selection is the process of selecting parents from the current population to generate

offspring solutions through crossover and mutation. Usually, the parent selection operators give higher chance for the fitter individuals to be selected for reproduction than selecting individuals with lower fitness, e.g., tournament selection [88] and roulette wheel selection [90].

In the crossover step, the two selected parent chromosomes are crossed over to create two child chromosomes. There are different types of crossover operators such as single point crossover and uniform crossover. For example, single point crossover splices the two parent chromosomes at a selected crossover point and form two child chromosomes by fusing the front of one parent to the end of the other. The crossover operator helps the search to exploit existing information in the current population.

The mutation step randomly modifies the two child chromosomes produced from the crossover step and produces two new child chromosomes. For example, mutation operator adds, modifies and removes statements to/from test cases [15] with a given mutation probability. It introduces diversity into the search by helping the search to explore new areas of the search space and also prevents the search from becoming trapped in local optima.

Reinsertion step is about selecting individuals from the current population and the offspring population to form a new population in the next iteration (i.e., generation) in GA. The *elitism* strategy has been used as the reinsertion operator in the existing GA based approaches in SBST [3, 14]. Elitism strategy chooses the fittest M individuals from the current population and offspring and forms a new population to be used in the next iteration.

2.2.2 EvoSuite

We use the state-of-the-art SBST tool EvoSuite [3] in this thesis. EvoSuite is an automated test generation framework that generates JUnit test suites for Java classes. It was first proposed by Fraser and Arcuri [3] in 2011, since then it has gained growing interest in the SBST community [14, 15, 23, 51, 91]. Its effectiveness has been evaluated on open source and as well as industrial software projects in terms of the code coverage [14, 15, 51, 80, 81] and bug detection [23, 26]. The existing SBST techniques like whole test suite [89], whole test suite with archive [51], MOSA [15] and DynaMOSA [14]

are implemented in EvoSuite, hence, making it easier to compare the SBST techniques without having any confounding effects due to different implementations or use of tools. Furthermore, EvoSuite won 6 out of 7 of the SBST unit testing tool competitions [92–97]. To date, EvoSuite is actively maintained, and its source code and releases are readily available to use at GitHub [98] and their website [99]. Given the maturity of EvoSuite, we decide to use it as the SBST tool in our thesis.

2.2.3 Single Objective Formulation

The test generation problem can be formulated in two ways; i) single objective formulation [3, 51], and ii) many-objective formulation [14, 15]. In the latter one, the test generation problem is approached as a many objective optimisation problem whereas the single objective formulation approaches the test generation problem as optimising test suites for a single objective [3, 51]. Dynamic many objective sorting algorithm (DynaMOSA) [14] is the state-of-the-art SBST technique and a many-objective approach. We use DynaMOSA as the SBST technique in benchmark methods in our thesis. In this section, we discuss the existing single objective approaches and in the next section, we describe the DynaMOSA approach in detail.

2.2.3.1 Single Target Strategy

Tonella [100] introduced an approach based on GA (also known as *one goal at a time* [51]) to automatically generate unit tests for object-oriented programs by giving only one coverage target at a time for the search. For example, if the objective is to maximise branch coverage, then the single target strategy runs GA separately for each branch in the program. There are two major drawbacks in this approach. Since it considers only one target (e.g., branch) at a time, if an infeasible branch is selected as the next target to cover, then there is the risk of wasting the time budget on that infeasible branch. This could lead to missing test generation for other branches in the program. The other major drawback of this approach is that it misses the targets covered by accident (i.e., collateral coverage) while focusing the search to cover another target. It is important for SBST techniques to pay attention to the collateral coverage since a test case that covers a target also covers the targets along its execution path in the control flow graph.

2.2.3.2 Whole Test Suite

Fraser and Arcuri [3] introduced *whole test suite (WTS)* approach that optimises test suites considering all the coverage targets at the same time. The objective function in WTS is an aggregation of all the coverage targets into a single scalar objective function, which is known as sum scalarisation. By aggregating all the targets into a single objective, WTS is able to mitigate the drawbacks in the single target strategy. In particular, WTS simultaneously focuses on all the targets of the program, hence the infeasible targets do not prevent the search from finding tests to cover feasible targets. The collateral coverage is automatically included in the aggregated objective function.

WTS defines the problem of finding a test suite that satisfies all the targets as follows. Let $U = \{u_1, \ldots, u_k\}$ be the set of k coverage targets of the program under test. Then, WTS needs to find a test suite $T = \{t_1, \ldots, t_n\}$ that minimises the fitness function $f_U(T)$,

$$f_U(T) = \sum_{u \in U} d(u, T) \tag{2.1}$$

where $d(u, T)$ is the distance for the target $u \in U$ according to a distance function. If the target $u \in U$ is covered by the test suite T, then $d(u, T)$ is equal to zero.

For branch coverage problem, the distance $d(b, T)$ for the branch $b \in B$, where $B = \{b_1, \ldots, b_k\}$ is the set of branches of the program under test, is defined as follows;

$$d(b, T) = \begin{cases} 0 & \text{if the branch has been covered,} \\ v(d_{min}(b, T)) & \text{if the predicate has been executed at least twice,} \\ 1 & \text{otherwise.} \end{cases} \tag{2.2}$$

where $(d_{min}(b, T)$ is the minimum branch distance for the branch $b \in B$ and is computed according to the branch distance computation schemes [90]. $v(x)$ is a normalising function and it normalises the minimum branch distance to a value within the range 0 to 1.

Finally, WTS aggregates these single branch distances for each branch $b \in B$ to the following single fitness function.

$$f(T) = |M| - |M_T| + \sum_{b \in B} d(b, T) \tag{2.3}$$

where M is the set of methods without branches (branchless methods), and M_T denotes the set of branchless methods which are covered by the test suite T.

WTS achieves higher branch coverage than the single target strategy [3, 89]. However, it is not guaranteed that the targets covered by WTS subsume the targets covered by the single target strategy. The fitness function in WTS (e.g., Equation. (2.3)) treats all the coverage targets equally and the search receives the same reward when covering either of the targets. Hence, WTS is more likely to keep cover trivial targets than to try covering nontrivial targets. Single target approach, on the other hand, may get to spend a significant time budget to cover nontrivial targets at the expense of missing many trivial targets.

2.2.3.3 Archive-Based Whole Test Suite

Rojas et al. [51] introduced *archive-based whole test suite* (WSA) approach which addresses the problem of the lack of tests generated to cover nontrivial targets by WTS. In WTS, if the search finds a new test suite T_{new} by modifying a test suite T_{old} and T_{new} covers previously uncovered n targets and loses coverage of more than n previously covered targets, then according to the fitness function in Equation.(2.1), T_{new} is not considered any better than T_{old} despite covering new targets. WSA addresses this problem by removing a target from the fitness function once it is covered. This way the search will not be rewarded anymore for covering that target again. It is rewarded only when it tries to cover uncovered targets. WSA adapts the archiving technique proposed in [15] to retain the test cases covering the targets that are removed from the fitness function. At the end of the search, this archive of test cases forms the final test suite. The fitness function in WSA is as follows;

$$f_U(T) = \sum_{u \in U \setminus C} d(u, T) \tag{2.4}$$

where C denotes the set of covered targets.

One limitation in the WSA approach is that it may lose guidance to cover certain targets once targets are removed from the fitness function after they are covered. For example, the fitness function for branch coverage is an aggregation of branch distances for each branch of the program under test. Branch distance provides guidance to the search to find a test that executes the respective branch only at its control dependent node. Figure 2.2 shows a control dependency graph (CDG) for a program and b_i, where $i \in [1, 6]$, is a branch. The branch distance of branch b_6 provides guidance only at the node D. The guidance provided by the approach level, i.e., guidance to find tests that execute along the desired path to the target branch, is already provided by the aggregated branch distances in the fitness function. For example, the branch distances of b_1 and b_3 provide guidance for the search to find a test that executes b_6 by covering b_1 and b_3. When WSA removes covered targets from the fitness function, there is the chance of that guidance being removed. For example, once WSA finds a test case to cover b_5, hence b_1 and b_3 are also covered, b_1, b_3 and b_5 are removed from the fitness function. This means that the updated fitness function does not have approach level guidance for b_6 anymore. As a result, WSA is likely to not cover the targets that are deep down in the CDG.

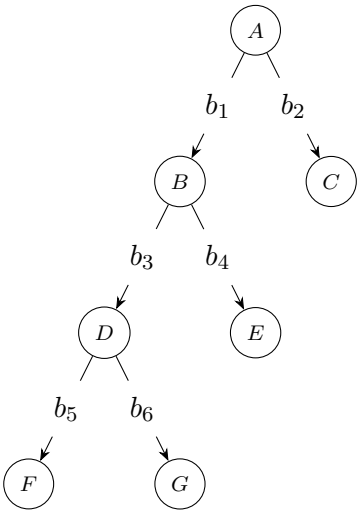

FIGURE 2.2: Control dependency graph

2.2.4 Many-Objective Formulation

The test generation problem can be approached as a many objective optimisation problem where each target of the program under test is represented as an objective. The many objective solvers simultaneously optimise test cases to cover multiple targets. By formulating the test generation problem as a many-objective problem, the limitations in single objective approaches such as problems of sum scalarisation [14] and lack of guidance in the single fitness functions can be mitigated. Previous work showed that many objective solvers like MOSA and DynaMOSA are more effective at achieving high branch, statement and strong mutation coverage than the single objective approaches (e.g., single target and whole test suite approaches) [14, 15, 81]. According to the reachability, infection and propagation (RIP) principle [19–22], it is necessary for a test to cover the buggy code in order to detect the bug. Previous work indicates that mutation coverage significantly correlates with the bug detection of the test suites [101]. Therefore, in this thesis, we use many objective formulation and many objective solvers to approach the test generation problem. In particular, we use the state-of-the-art SBST technique DynaMOSA in the benchmark methods and propose an SBST technique which addresses the test generation as a many objective optimisation problem.

2.2.4.1 Optimisation Problem

In many objective formulation, the objectives to be optimised are the individual distances for each target in the program under test. The problem of finding a set of test cases that satisfy all the targets can be formulated as follows. Let $U = \{u_1, \ldots, u_k\}$ be the set of k coverage targets and we need to find a set of non-dominated test cases $T = \{t_1, \ldots, t_n\}$ that minimise the fitness functions $f_i(t)$ for all the targets $u_i \in U$,

$$f_i(t) = d(u_i, t) \tag{2.5}$$

where $d(u_i, t)$ is the distance from covering the target $u_i \in U$ with respect to the test case t.

For branch coverage problem, the fitness function $f_i(t)$ of the branch $b_i \in B$, where $B = \{b_1, \ldots, b_k\}$ is the set of branches of the program under test, is defined as follows.

$$f_i(t) = al(b_i, t) + d(b_i, t) \tag{2.6}$$

where $al(b_i, t)$ is the approach level and $d(b_i, t)$ is the normalised branch distance of the branch b_i for the test case t. In many objective optimisation, each target is considered a separate objective. Hence the approach level guidance provided by the aggregated fitness function in the single objective formulation does not exist anymore. Therefore, the approach level is added to each fitness function.

For a given test case t, the fitness is a vector of k values ($\langle f_1, \ldots, f_k \rangle$), where f_i ($i \in [1, k]$) represents the distance of t from covering the target $u_i \in U$. If a test case t covers a target u_i, then the corresponding fitness f_i of t is zero.

To maximise the coverage of multiple targets, we need to find a set of non-dominated test cases $T = \{t_1, \ldots, t_n\}$ where for each $t_j \in T$, $\exists u_i \in U$ such that $f_i(t_j) = 0$. A set of test cases are said to be non-dominated if each test case in the set is better on at least one coverage target and worse on the remaining targets when compared to other test cases in the set.

To evaluate individual test cases, we can use Pareto dominance (Definition 2.1) and Pareto optimality (Definition 2.2) of test cases [102].

Definition 2.1. Pareto Dominance. A test case t_i **dominates** another test case t_j, if, and only if, the values of the fitness vector satisfy the following conditions:

$$\forall x \in \{1, \ldots, k\} \;\; f_x(t_i) \leq f_x(t_j)$$

and

$$\exists y \in \{1, \ldots, k\} \;\; s.t. \;\; f_y(t_i) < f_y(t_j)$$

The definition above states that a test case t_i dominates another test case t_j, if, and only if, t_i is closer to cover at least one coverage target and not worse in terms of covering other targets when compared to t_j.

Definition 2.2. Pareto Optimality. A test case is **Pareto optimal**, if, and only if, it is not dominated by any other test case in the space of all possible test cases.

The definition above states that a Pareto optimal test case is better on covering one or more targets and can be worse on covering the remaining targets when compared to all possible test cases.

The solution to the many-objective problem is a set of Pareto optimal test cases. Unlike in usual many-objective optimisation problems where there are trade-offs in the objective space, in the context of test generation, the optimal test cases are the ones which cover at least one target, i.e., objective, (i.e., $\exists u_i \in U \ s.t. \ f_i(t) = 0$). Therefore, these test cases that cover at least one target form the final test suite and represent a sub-set of the Pareto optimal test cases.

2.2.4.2 Many Objective Sorting Algorithm

Panichella et al. [15] proposed many objective sorting algorithm (MOSA), which formulates the test generation problem as a many objective optimisation problem and produces a set of Pareto optimal test cases. It starts with a set of randomly generated test cases as the initial population. It generates new population of test cases by applying crossover and mutation operators. Test cases are selected to the next generation using a ranking algorithm called preference sorting algorithm which is based on 'preference criterion' and the non-dominance relation of test cases. According to preference criterion, for each target $u_i \in U$, the test case that is closest to cover u_i is selected to the first non-dominated front. All the test cases in the first non-dominated front are selected to the next generation. MOSA maintains an archive of test cases generated during the evolution process that forms the final test suite. The archive contains the shortest test cases for each covered target.

2.2.4.3 Dynamic Many Objective Sorting Algorithm

DynaMOSA [14] is the successor of MOSA and stands as the state-of-the-art SBST technique. A main limitation in MOSA is that it tries to cover all the targets from the beginning of the search while most of the targets are not reachable until their control

dependent targets are covered. DynaMOSA addresses this problem by introducing a method called dynamic selection of targets.

A subset of the targets that are included in the search process cannot be covered until their control dependent targets are covered. Assume a test generation scenario which uses branch coverage as the optimisation criterion. In the beginning of the search, U contains all the branches as shown in the CDG (Figure 2.2) as the set of targets to search for test cases. However, branches like b_5 and b_6 cannot be covered until their control dependent branches b_1 and b_3 are covered. Likewise, b_3 cannot be covered until b_1 is covered. Therefore, it is inefficient to search for tests to cover such targets, e.g., b_5, while their control dependent targets are still uncovered. It will unnecessarily increase the computational complexity of the SBST technique because of the added objectives to the search without any added benefit.

To address this, DynaMOSA dynamically selects targets to search for test cases only when their control dependent targets are covered. For example, b_5 and b_6 are selected to the search process only if b_1 and b_3 are covered. At the start of the search, DynaMOSA selects the set of targets $U^* \subseteq U$ that do not have control dependencies. At any given time in the search, DynaMOSA optimises test cases to cover only the targets in U^*. Once a new population is generated, DynaMOSA needs to update U^* with new targets if their control dependent targets are covered. Since DynaMOSA aims at maximising code coverage, it also removes the covered targets from U^* to allow itself to focus more on uncovered targets. Unlike in WSA, removing covered targets from U^* does not cause DynaMOSA to lose guidance to cover the targets that are deep down in the CDG. This is because it uses approach level in the fitness function of each target.

2.3 Defect Prediction

Defect prediction techniques predict the areas in software that are likely to be defective, in other words bug-prone areas. The exact locations of the bugs are not known prior to testing. However, there are factors which characterise the code and can correlate with the likelihood of software components being defective. For example, code complexity is one such factor, which indicates a component with higher number of entities and higher number of dependencies between them is highly likely to be defective [34]. The

concept behind defect prediction is that mathematical models can be built using the factors correlating with the defectiveness of the code to estimate if the code is likely to be defective in the future.

2.3.1 Constructing Defect Prediction Models

To construct a defect prediction model, a defect dataset comprising of metrics to characterise the code with respect to their defectiveness and bug history of the code needs to be created first [103]. Previous studies have considered a wide range of metrics such as code size [33], code complexity [34], object-oriented metrics [104], organisational metrics [36], change history related metrics [35] and code churns [38] to characterise the code when building defect predictors. Dam et al. [105] introduced an approach to automatically learn semantic and syntactic features that can derive defectiveness of the code instead of manually selecting the features to use, and it was shown to outperform the defect predictors built with traditional software metrics.

These metrics and the bug history of the code can be extracted from the issue tracking systems (ITS) like Jira [106] and BugZilla [107] and version control systems (VCS) like Git [108], CVS [109] and SVN [110] maintained by development teams for the software projects. For example, ITS can be used to retrieve the bug reports and extract information like the release version of a bug when it was reported [34, 103]. VCS can be used to identify when a bug was fixed by checking commit messages with keywords like "fixed" and "bug" followed up by "#" or a number [34]. Previous studies have used the SZZ algorithm [111, 112] to find the bug introducing changes for a given bug fixing commit and bug report, which helps to identify the bug history of the code to build defect datasets.

Code snapshots at the versions of interest can be retrieved from the VCS. Code size, code complexity and object-oriented metrics can be extracted from the code snapshots. For example, CK tool [113] is a Java code metrics calculator that extracts a large set of metrics including the Chidamber/Kemerer (CK) metrics [114] using the source code of a software project. VCS contains code change information that can be used to extract change history related metrics, code churns and ownership metrics by mining the software repositories. For example, recent changes to the code by a new author can indicate the code is highly likely to be defective [30]. This information can be extracted

by examining the usernames of the authors and diffs of commit history of the code from the VCS.

Upon creating a defects dataset, a defect prediction model is constructed using a machine learning technique [32, 38] or a statistical approach [36, 37]. The prediction model relates the software metrics and defectiveness of code in the defects dataset. Majority of previous work used machine learning techniques to build defect prediction models, e.g., naive Bayes [115], logistic regression [116], random forest [117], support vector machine [118], J48 [119] and C4.5 [120]. The defect prediction models can then be used to predict bugs in future versions of the code.

In a setting where a defects dataset is not available to be used as a training dataset to train a machine learning technique, cache-based [29, 121] or risk estimation based prediction techniques [30, 39] can be used to construct prediction models. For example, FixCache [29] maintains a cache of the most bug-prone software entities. In particular, when FixCache finds a bug fixing revision in the VCS it determines the bug introducing change and fetches the entity and nearby entities (temporal and spatial locality) at the revision where the bug was introduced into the cache. The cache is used to predict bugs in future revisions. Lewis et al. [39] and Freitas [30] developed defect predictors based on risk estimation techniques. In particular, they used a time weighted risk (TWR) formula to calculate an estimation of the probability of defectiveness of an entity. For example, the TWR formula takes the timestamps of the bug fixing commits for a given entity and calculates a time-weighted risk for that entity in terms of its defectiveness.

2.3.2 Types of Defect Predictors

Defect predictors give predictions at different levels of granularity, e.g., coarse-grained prediction levels such as package [28], file [29] and class [104] levels and fine-grained levels such as method [32] and line [27] levels. The output of the defect predictor can be in various forms; probability of the code being defective [30], number of defects (i.e., defects density) [122], binary classification of buggy or not buggy [31], ranking of code according to the defectiveness [39]. The probability of the code being defective and the number of defects can be converted to a binary classification using a threshold [32, 36]. For example, a threshold of 0.5 was used to determine if the code is likely to be buggy or not [36]. In particular, if the probability of defectiveness is greater than 0.5, then

the code is labelled as buggy, and non-buggy otherwise. Similarly, a threshold of 1 was used to convert the defect density to a binary classification [32]. In particular, if defects density is greater than or equal to 1, then the code is labelled as buggy, and non-buggy otherwise. Defectiveness ranking is useful when the limited SQA resources and activities need to be prioritised for a part of the code. Usually, the top-n riskiest entities (e.g., files) are selected according to the ranking given by the defect predictor to conduct SQA activities [39]. The probability of defectiveness can also be used to rank the entities, where the entity with the highest probability is the riskiest and the one with lowest probability is the least risky entity [42].

2.4 Summary

In this chapter, we discuss how SBST fits in the broad area of software engineering. We give an overview of the main steps of applying a genetic algorithm to a test generation problem. We discuss the existing SBST approaches that formulate the test generation problem as single objective optimisation problem and their limitations. Then we present the optimisation problem in many-objective formulation and describe the state-of-the-art DynaMOSA in detail, which is used in the benchmark methods in the thesis. The proposed SBST approaches in our thesis address the test generation problem as a many-objective optimisation problem. Finally, we give an overview of the construction of defect prediction models and discuss different types of defect predictors available.

Chapter 3

Related Work

3.1 Introduction

This chapter presents a review of the search-based software testing (SBST) literature. In particular, we discuss the SBST techniques for unit testing and limitations of SBST approaches, including the state-of-the-art SBST technique, in terms of detecting bugs. This is followed by a review of the defect prediction literature. We discuss the advantages and limitations of different types of defect predictors categorised by the prediction technique used (e.g., machine learning), output type, metrics used, and granularity of the predictions in the context of using them to guide SBST. Finally, we present a review of the previous work in defect prediction for automated software testing.

3.2 Search-Based Software Testing

As we discussed in Section 2.2, SBST has been used in plethora of software testing problems and applications [53, 64, 66, 66–71, 73–78]. Test data generation is one of the main problems where SBST is applied [123]. In particular, SBST for unit testing is the most well studied test generation problem in the literature [79]. Among those studies, structural coverage of unit tests [14, 15, 44, 51, 73, 80, 81] has received larger attention compared to bug detection [23, 24, 26] from the SBST researchers. In this thesis, we pay a particular attention to the bug detection capability of the SBST for unit testing.

SBST techniques use meta heuristics search algorithms to search for high quality test cases for a specific criteria (e.g., maximise branch coverage). Mainly, the test generation problem is formulated in two ways; i) single objective formulation [3, 51], and ii) many-objective formulation [14, 15], which we introduced in Sections 2.2.3 and 2.2.4. Whole test suite approaches [3, 51] use single objective formulation where they optimise test suites to satisfy a single objective composed of all the coverage targets. Many objective sorting algorithms [14, 15] use many objective formulation where they simultaneously optimise test cases to satisfy all the coverage targets. Previous work shows that these many objective sorting algorithms are more effective and efficient than whole test suite approaches in terms of code coverage and mutation coverage [14, 15, 81].

Dynamic many objective sorting algorithm (DynaMOSA) [14] approaches the test generation problem as a many objective optimisation problem and is considered the state-of-the-art SBST technique. It is better at achieving high code coverage than many objective sorting algorithm (MOSA) [15] and archive-based whole test suite (WSA) [51]. DynaMOSA exploits the control dependency of the targets in the program and narrows down the search to a subset of the coverage targets at a time. This makes DynaMOSA less computationally complex than its predecessor MOSA. According to the reachability, infection and propagation (RIP) model, it is necessary but not sufficient to cover the buggy code in order to detect a bug. Achieving a high code coverage by the test suite does not guarantee effective bug detection. One main limitation in DynaMOSA is that it targets at covering all the targets in the program while only a few of them are actually buggy. We argue that it is likely to be ineffective in terms of bug detection to search for tests in non-buggy targets which constitute a larger portion of the code base.

Results from previous work show that SBST techniques have limitations in terms of bug detection [23, 24, 26]. For example, EvoSuite, a state-of-the-art SBST tool, could only detect on average 23% of the bugs from the Defects4J dataset [23]. However, in the same study, Randoop [4], a feedback-directed random test generation technique, detected on average 15% of the bugs, which indicates that despite its poor bug detection performance, SBST is still better than random testing techniques. Similar results can be observed in the study by Almasi et al. [26].

Salahirad et al. [24] studied the bug detection effectiveness of SBST when it is using different fitness functions based on single and combinations of test goals. Test suites

generated for different test goals may capture different behaviours of the program under test, hence SBST may detect different bugs when it is guided by different test goals. Branch coverage based fitness function was shown to be the most effective because it is focused on thoroughly exploring the program structure. However, SBST could only detect on average 25% of the bugs from the Defects4J dataset when using branch coverage fitness function and 10 minutes time budget, suggesting code coverage alone is not enough to effectively detect bugs.

As described in Section 2.2.1.2, time budget is used in SBST techniques to determine when the genetic algorithm should stop searching for tests. Higher time budget means the search method is able to extensively explore and exploit the search space of possible test inputs, thereby increasing the chances of finding better test cases with respect to the coverage criterion used. Previous work studied the improvement of bug detection effectiveness of SBST by increasing the time budget [24, 26, 43]. The results from those studies indicate that it is beneficial to increase the time budget in order for SBST to detect more bugs. However, this approach has limitations in its practicability. In particular, Almasi et al. [26] allocated 15 minutes, and Gay [43] and Salahirad et al. [24] allocated 10 minutes time budget per class. In practice, the practitioners do not know the buggy classes prior to running test generation. In the absence of a time budget allocation approach, they have to allocate a fix time budget for every class in the project. Given the industrial projects are usually very large with thousands of classes and the requirement of frequent testing, it is not viable to allocate such large time budgets for every class in a project, unless there is a lot of computational resources available in an organisation, which is usually not the case [44]. We argue that in a resource constrained environment, SBST should receive higher time budgets to search for tests in likely defective classes at the expense of the time budgets for the likely non-defective classes in order to properly utilise the available limited time budget.

3.3 Defect Prediction

Defect predictors employ machine learning techniques [32, 38], statistical approaches [36, 37], cache-based techniques [29, 121] or risk estimation techniques [30, 39] to predict bug-prone areas in the code. Majority of previous work in defect prediction used machine learning techniques such as naive Bayes, logistic regression, random forest and support

vector machine [49]. They train a model from a defect dataset containing bug history of code and features that can derive the defectiveness of the code. Cache-based or risk estimation based prediction techniques do not require a defect dataset, hence they are easy to apply to an industrial setting where a training dataset is not available. We employ Schwa [30], a defect predictor based on risk estimation, to guide the time budget allocation approach for SBST in the study conducted to achieve RO1. The proposed approach does not depend on the prediction model used by the defect predictor. Hence, any defect predictor can be used in place of Schwa for the task at hand.

The output of a defect predictor can be in the form of a binary classification [31], probability of defectiveness [30], defectiveness ranking [39] or defect density of the code [122]. Defect predictors employing machine learning techniques such as logistic regression give a probability of the code being defective [36]. Some defect prediction models are trained to output the number of defects (i.e., defect density) in the code [122]. This can be converted to a binary classification using a threshold [36]. For example, if the probability is greater than the threshold, the code is labelled as buggy, and not buggy otherwise. On the other hand, machine learning techniques such as support vector machine directly gives a binary classification. Freitas [30] proposed a defect prediction technique based on risk estimation and they used a time weighted risk formula to calculate the probability of code being defective. Yang et al. [124] proposed a learn-to-rank approach to construct defect prediction models that output a ranking of the code according to their defectiveness. Majority of the previous work studied defect predictors that output a binary classification [49]. However, defect predictors with a continuous output like probability of defectiveness provide more information to differentiate the code than what classification provides. We use a defect predictor, Schwa [30], that outputs the probability of defectiveness of classes in a project in RO1 as it allows us to differentiate classes based on how likely they are to be defective. In RO2 and RO3, we simulate defect prediction as we need to control the defect prediction performance to certain levels. Since we use theoretical defect predictors in these two research objectives, we resort to the generic defect predictor, which is the one that outputs a binary classification.

As we discussed in Section 2.3.1, previous work on defect prediction have considered a wide range of metrics to characterise the code with respect to their defectiveness [33–36, 38, 104, 105]. Graves et al. [125] showed that the number of changes and particularly the recent changes to the code are effective indicators of future defects. Kim et al. [29]

followed the observation that bugs occur in software change history as bursts, hence they argue that recent changes to the code and recent faults in the code are likely to introduce bugs in the future. Rahman et al. [121] proposed a simple approach, which was eventually implemented by the Google Engineering team [39, 40], that orders files by the number of bug fix commits in a file, and found out that its performance is quite similar to the more complex approach FixCache [29]. Furthermore, they showed that the files that have been recently involved in a bug fix are likely to contain further bugs. Freitas [30] developed an enhanced version of this approach called Schwa, which predicts defects in programs by using three metrics; recent changes, recent bug fixes, and recent new authors [36] to the code.

The effectiveness of the metrics used to indicate the defect proneness of the code is dependent on the context they are applied in [37]. For example, Nagappan et al. [36] showed that the organisational metrics are the best indicators of the defect proneness of the Windows Vista project. Caglayan et al. [37] demonstrated a counter-case to this in their replication study involving a large-scale enterprise software. We propose approaches and design studies that are not dependent on the metrics used in the defect predictors.

Defect predictors estimate the likelihood of a package [28], file [29], class [104], method [32] or line [27] is defective. There is plethora of defect predictors working at coarse-grained levels such as package, file and class levels [49]. Whereas there are only few defect predictors that work at fine-grained levels such as method and line levels. This is because obtaining bug history and features to derive defectiveness are more challenging tasks for methods and lines compared to packages, files or classes [31, 126]. While there are many coarse-grained defect predictors with effective predictive power, their use in practice has limitations because of the size of the units they make predictions for [41]. In particular, a class (or file) may have thousands of lines of code. By looking at the prediction for such class, it is hard for a developer to find the bug through code review or manual testing. In fact, Hata et al. [31] showed that the fine-grained predictors outperform coarse-grained predictors in terms of the effort required to find bugs manually. Our thesis investigates leveraging defect predictors at both coarse-grained and fine-grained levels to inform SBST techniques of the defective areas in code in order to improve the bug detection performance.

Defect predictors have been shown to be effective at locating bugs in software [29, 37, 38]. In particular, defect predictors with more than 85% of recall and precision have been reported in the literature [32, 38]. Due to their efficacy, they are used in practice to assist developers in code reviews [39, 40] and manual testing [41]. With the help of the defect predictors, developers can make decisions on where to prioritise their test efforts [127]. However, finding bugs by code reviews or manual testing may become a laborious task when the size of the code that is predicted buggy is large or complex. Previous work proposed to use model-agnostic techniques from explainable AI to generate explanations for the defect prediction models and their predictions, e.g., most important characteristics that contributed to the final prediction [128]. This further helps developers to find bugs by understanding why the code is flagged as buggy by the predictor [127]. In this thesis, we propose to use defect prediction to inform an automated testing technique which produces bug detecting test cases. The automated testing step has the potential to complement or to substitute the manual testing/code review step, hence can further reduce the workload of the developers.

Defect predictors are not perfectly accurate. While there are defect predictors with higher recalls and precision, their performance can vary, e.g., from as low as 5% and 25% to as high as 95% and 85% for recall and precision, respectively [49]. Hosseini et al. [50] also reported similar findings in their systematic literature review of cross-project defect predictors. The false negatives and false positives in the predictions can have an impact on the usefulness of defect predictors in practice. Developers may miss bugs because of false negatives [29]. False positives, on the other hand, mislead the developers into thinking that non-buggy code is buggy. This can result in a waste of developers' time and can lead them to not trust the defect predictor, especially when the predictions go against their perceptions of the code [39, 41, 47]. When an automated testing technique uses defect predictions, false positives may not be a significant burden in contrast to a developer manually inspecting non-buggy code for bugs. False negatives, on the other hand, may cause the automated techniques to completely miss the bugs. However, there is the potential to mitigate this by allowing the automated technique to explore the likely non-buggy code for bugs, which would be an expensive task if it is done manually. This thesis investigates the impact of false positives and false negatives on the bug detection performance of SBST. We propose defect prediction guided SBST techniques that handles the potential errors in the predictions to mitigate their impact

on the final outcome, i.e., bug detection of the test cases.

Zimmermann et al. [48] recommended a defect predictor with recall, precision and accuracy greater than 75% is a strong defect predictor, and vice versa. They found out that only 21 out of 622 cross-project defect predictor combinations are strong defect predictors according to this criterion. Following this recommendation, we define an acceptable defect predictor is one that has a recall and precision greater than 75%. In particular, in RO2, we systematically investigate the impact of imprecision of defect prediction in the range of acceptable defect predictors. In RO3, we experimentally assess the proposed SBST technique using acceptable defect predictors.

We use an off-the-shelf defect predictor in the proposed SBST approach in RO1 and demonstrate real defect predictors can be leveraged to guide SBST to detect bugs efficiently and effectively. In RO3, we experimentally assess the proposed SBST technique using theoretical defect predictors with lower bound and upper bound performances in the acceptable range. To do this, we simulate defect prediction following the assumption of uniform distribution of defect prediction errors similar to Herbold [129]. The performance of real defect predictors cannot be controlled to any desired values. To systematically investigate the impact of defect prediction imprecision in the acceptable range (RO2), we simulate defect prediction for different levels of performance. Using a real defect predictor in the experimental evaluation demonstrates the viability of the proposed approach in practice [42, 130–132]. However, it limits the findings of the study to a specific defect predictor built with one learner and one set of metrics.

3.4 Defect Prediction in Automated Software Testing

As we discussed in Section 3.3, defect predictors have been used in the industry to support developers in code reviews [39, 40] and in testing [41] because of their efficacy in locating bugs in software. While the main assumption of defect prediction is to provide useful information to developers [39], prediction outcomes have also been used successfully in automated testing techniques [42, 131–133].

Previous work that leveraged defect prediction to guide automated software testing techniques have used defect prediction either i) in the test generation phase [130, 133] or ii)

post test generation phase [42, 131, 132]. In particular, Hershkovich et al. [130, 133] proposed QUADRANT which uses defect prediction to select classes to run test generation. G-clef [42] is a test prioritisation strategy that uses a defect predictor and prioritises test cases in terms of their likelihood of finding bugs. FaRM [131] is a mutant selection technique that selects and ranks fault revealing mutants using prediction models. FLUCCS [132] is a fault localisation approach that ranks methods according to their likelihood of being faulty using defect prediction models. G-clef, FARM and FLUCCS are applied after the test generation step, i.e., G-clef and FaRM can be used to prioritise and select test cases, and FLUCCS can be applied to localise the fault once a bug is detected through test generation.

QUADRANT [130] is a defect prediction guided class selection strategy to run test generation with an automated testing technique like SBST. It uses a class level defect predictor, and code complexity, object-oriented and change history related metrics are used to derive the defectiveness of the code [134]. QUADRANT ranks classes according to a score which is a combination of the probability of defectiveness and/or lines of code in the class. Only the top-n classes are selected for test generation, and in the experiments, they have considered $n \in [1, 15]$. Usually, the industrial projects are very large and may have thousands of classes [45]. Selecting only a few classes for test generation and leaving out the other classes may have consequences in terms of bug detection. This is because defect predictors are not perfectly accurate and there is a reasonable chance that the buggy class not to be ranked in the top-n classes when n is significantly smaller compared to the total number of classes in a project. For example, the average relative ranking position of the buggy classes by the defect predictor used in G-clef is 13%, which means if the project has 1000 classes, the defect predictor ranks the buggy classes at 130^{th} position on average [42]. In contrast, this thesis proposes a defect prediction guided time budget allocation approach that allocates time budgets to classes for test generation while accounting for potential errors in the predictions.

The size of the regression test suites increases over the time as new test cases are added, for example, when development teams add new features to the software. To reduce the time taken to detect bugs (i.e., regressions) in software, researchers have opted for test case prioritisation strategies. Paterson et al. [42] introduced G-clef which is a test case prioritisation strategy that uses a class level defect predictor based on change history related metrics [30] and prioritises test cases in terms of the buggy rank of the classes

they cover. It was shown to be more effective at reducing the number of test cases required to find bugs compared to the existing prioritisation strategies. We employ the same defect predictor as used in G-clef to inform a time budget allocation approach in this thesis. In contrast to G-clef, our thesis focuses on improving bug detection capability of an automated test generation technique.

Mutation testing is expensive due to the larger number of mutants that need to be analysed. FaRM is a mutant selection technique that selects and ranks fault revealing mutants, i.e., the killable mutants and the ones that lead to test cases revealing bugs, and aims at revealing most of the bugs by analysing the smallest possible number of mutants [131]. FaRM uses a prediction model that captures the characteristics that derive the ability of the mutants to reveal bugs. The prediction model used in FaRM utilises source code metrics. It was shown to outperform the state-of-the-art mutant selection and mutant prioritisation methods in terms of revealing bugs. FaRM can be applied as a post-test generation task. In contrast, our thesis specifically focuses on generating tests with the guidance of defect prediction.

Once the tests (automatically generated or manually written) detect bugs, developers need to analyse the programs and test executions to find the root cause of the observed failures. Fault localisation techniques help developers to reduce their debugging efforts by highlighting the program elements (e.g., lines) which are most likely to contain the faults. FLUCCS [132] is such fault localisation approach that combines the existing spectrum-based fault localisation (SBFL) approach with defect prediction. They use a method level defect predictor that utilise code size, complexity, code churns and change-related metrics to rank methods according to their likelihood of being faulty. FLUCCS was shown to outperform the state-of-the-art SBFL techniques in terms of locating faults at the top and within the top ten methods. FLUCCS also works at the post-test generation phase, for example, once the tests generated by SBST detect a bug, FLUCCS can be applied to localise the fault with the aid of SBST generated tests.

3.5 Summary

The test generation problem is one of the main problems addressed by SBST research. Among the studies in test generation, structural coverage of unit testing has been a main

focus. In fact, several SBST techniques have been proposed with the aim of maximising structural coverage of unit tests and they have shown to be effective at achieving high code coverage, sometimes covering more code than manually written tests. However, SBST techniques are not as effective in terms of detecting bugs as indicated by the results of previous work. We argue that this is mainly due to SBST having no guidance towards the defective areas in a program.

Various types of defect predictors have been proposed over the past 40 years [47]. They give predictions at different levels of granularity, e.g., coarse-grained like file level and fine-grained like method level. Both coarse-grained and fine-grained predictions can provide useful information to SBST techniques in different ways. Due to their efficacy, defect predictors have been used to support developers in preforming manual tasks like code reviews and testing. Lack of explainability is one of the limitations in typical defect prediction outcomes when they are used by humans. Having SBST techniques to consume defect prediction outcomes and produce bug detecting tests has the potential to complement or to substitute the manual tasks and will be able to reduce the workload of the developers and the explainability issue to a certain extent. When using defect prediction to inform SBST techniques, one of the main concerns to look for is that the predictions are not perfectly accurate. This calls for an investigation of the impact of the defect prediction imprecision on guiding SBST and handling for those potential errors having a significant impact. Defect predictors have been used in automated testing techniques to inform those techniques of the likely defective areas in programs. To the best of our knowledge, this thesis is the first to investigate the use of defect prediction to guide the SBST techniques towards the likely defective areas.

Chapter 4

Methodology

In Chapter 3, we discussed the limitations of search-based software testing (SBST) techniques in the context of bug detection and the potential of defect predictors to provide additional guidance for SBST techniques to increase the bug detection performance of them. The ultimate goal of the thesis is to improve the bug detection capability of SBST by incorporating defect prediction information. To achieve that, we formulate three research objectives, which we introduced in Chapter 1, and conduct three research studies to address them. This chapter presents the methodological aspects of the research conducted in this thesis; the research method used to address the research objectives and validate the solutions proposed in each research objective, and the validity threats to the research studies conducted.

To evaluate the proposed contributions of the thesis, we design a set of experiments. The design of experiments describes the details to run a set of experiments under controlled conditions to evaluate the performance of SBST in terms of bug detection when using defect prediction for guidance. The experiments consist of generating test suites for programs containing bugs. These programs with bugs are the benchmark subjects of the experiments, which are taken from a well-studied dataset, Defects4J benchmark.

To perform an experimental comparison between a proposed approach in the thesis and the existing SBST approaches, we use the state-of-the-art SBST technique, DynaMOSA, in the benchmark methods. The SBST approaches need to be allocated a time budget for test generation. In each contribution chapter, we describe the specific time budgets allocated in the particular experiments. The proposed approaches and the benchmark

methods are implemented within the state-of-the-art SBST tool, EvoSuite, hence, any confounding effects due to different implementations or use of tools are mitigated in the experiments. The parameter settings for the SBST techniques, defect predictors and the proposed approaches are described in the particular contexts under each contribution chapter. We mainly use the number of detected bugs as the performance measure. In addition, time to generate the first bug detecting test case is also used. To compare the performance of the proposed approaches against the benchmark methods and draw conclusions from the results, we conduct statistical tests.

In this thesis, we consider generating tests to detect bugs not only limited to regressions, but also the bugs that are introduced to the system at various times. Previous work that evaluate EvoSuite in terms of bug detection considered test suite generation for a regression testing scenario [23, 26, 44, 135]. They assume the current version of the software works correctly and generate test suites to capture the behaviour of that version. Then, these test suites are used to detect regression bugs introduced in the next commit. The bug survival time, i.e., the time between the introduction of the bug and when the bug fix is performed, can be as long as 2 years or as short as a few days [52]. Given a software system, there can be bugs that are introduced to the system at various times in the past and not yet detected. Therefore, contrary to the previous works, we focus on an application scenario of generating tests to detect bugs that exist in the system.

In the next sections, we describe the benchmark subjects, benchmark methods, performance measures and the bug detection evaluation procedure used in this thesis. The design of experiments is based on the research objectives, which allows us to validate the particular approaches and test the respective hypotheses. This section describes the aspects that are common to all the experiment designs in this thesis. The other specific details are described under each contribution chapter.

4.1 Experimental Subjects

We use the version 1.5.0 of the Defects4J dataset [25, 136] as our benchmark subjects to conduct experimental evaluations for our proposed approaches. It contains 438 bugs that are from manually validated bug fixes from six real-world open source Java projects. We remove four deprecated bugs from the original dataset [136] since they are not

reproducible under Java 8, which is required by our test generation tool used, i.e., EvoSuite. This results in the following bugs that are not part of the experiments: Lang-2, Closure-63, Closure-93 and Time-21 are removed. The 434 bugs are from the following projects; JFreeChart (26 bugs), Closure Compiler (174 bugs), Apache commons-lang (64 bugs), Apache commons-math (106 bugs), Mockito (38 bugs), and Joda-Time (26 bugs).

For each bug, the Defects4J benchmark gives a buggy version and a fixed version of the program. The difference between these two versions of the program is the applied patch to fix the bug, which indicates the location of the bug. The Defects4J benchmark also provides a test execution framework to perform tasks like running the generated tests against the other version of the program (buggy/fixed) to check if the tests are able to detect the bug and fixing the flaky test suites [136]. We use this framework to determine if the test suites generated by SBST detects the bug or not (discussed in Section 4.4).

Defects4J is widely used as the benchmark subjects in research on automated unit test generation [23, 43], automated program repair [137], fault localisation [138], test case prioritisation [42], etc. This makes Defects4J a suitable benchmark for evaluating our proposed approaches, as it allows us to compare our results to existing work. Defects4J is actively maintained and more recently another 401 bugs from 11 open-source projects were added to the existing 438 bugs. Bugs.jar [139] and Bears benchmark [140] are another two bugs datasets that are mainly targeted to be used in automated program repair studies. Bugs.jar contains 1158 real bugs from eight open-source Java projects and Bears benchmark contains 251 real bugs from 72 open-source Java projects. Unlike Defects4J, both datasets have not been used in the automated test generation studies. This can be because of their lack of support provided to fix test suites containing flaky test cases and evaluate test suites for bug detection, whereas the test execution framework of Defects4J enables researchers to easily perform these tasks. More recently, Herbold et al. [141] created another bug dataset containing 2371 real bugs from 28 open-source Java projects by manually validating bug fixes through crowd sourcing. Given Defects4J is actively maintained, its maturity as a framework to support experiments, and its use in a wide array of automated software testing areas including automated test generation, we decide to use Defects4J dataset as the benchmark subjects in this thesis. We identify validating the proposed approaches in this thesis against other bugs datasets [139–141] as future work to support the external validity of our findings and increase the generalisability.

4.2 Benchmark Methods

The main objective of the thesis is to use defect prediction to guide the SBST techniques to likely defective areas in code, which results in improved bug detection performance of the generated test suites. To determine if the objective is achieved, we use the state-of-the-art SBST technique, DynaMOSA, in the benchmark methods to compare against the bug detection effectiveness and efficiency of our proposed approaches. As discussed in Section 2.2.4, DynaMOSA is better at achieving high code coverage and mutation coverage compared to the existing SBST techniques. In order to detect a bug, it is necessary to reach the buggy code according to the RIP principle. Previous work indicates that mutation coverage significantly correlates with the bug detection of the test suites [101]. Therefore, we choose DynaMOSA in the benchmark methods to measure the improvements by our proposed approaches. In each contribution chapter, we describe the specific benchmark methods used in the particular contexts.

4.3 Detecting Bugs with Search-Based Software Testing Techniques

In order to detect a bug, a test case must satisfy the conditions of the *reachability*, *infection* and *propagation* (RIP) model [19–22]. In addition, it must also have a test oracle to *reveal* the failure [142]. DynaMOSA and our proposed SBST technique in RO3 are implemented in the state-of-the-art SBST tool, EvoSuite. Tests generated by EvoSuite satisfy all three conditions of the RIP model. However, they do not have test oracles, hence are incapable of revealing bugs without test oracle inserted by humans or automated tools [143]. We will explain this more with an example.

Figure 4.1 shows the buggy code snippet and the applied patch for `NumberUtils` class from Lang-16 bug in Defects4J [136]. The buggy method, `createNumber`, takes a parameter of String type, `str`, and returns a `java.lang.Number` object with the value specified in the input `str`. If the value in `str` cannot be converted, then it throws a `NumberFormatException`. For example, if the method is called with the input `str="0Xa"`, then it is expected to return a `java.lang.Integer` object with the value `10`. However, such input does not execute the `true` branch of the `if` condition at line 458 because the

buggy method only checks for hexadecimal notations starting with `"0x"` and not `"0X"`. As a result, the execution continues along the `false` branch of the `if` condition at line 458 and throws a `NumberFormatException` instead of the expected `java.lang.Integer` object. This bug is fixed by modifying the `if` condition at line 458 as shown in the diff in Figure 4.1.

```
444  444     public static Number createNumber(String str)
                     throws NumberFormatException {
445  445       if (str == null) {
446  446         return null;
447  447       }
448  448       if (StringUtils.isBlank(str)) {
449  449         throw new NumberFormatException("A blank string is not a valid
                   number");
450  450       }
451  451       if (str.startsWith("--")) {
                   ...
456  456         return null;
457  457       }
458       -     if (str.startsWith("0x") || str.startsWith("-0x")) {
     458  +     if (str.startsWith("0x") || str.startsWith("-0x") || str.startsWith("0X") ||
                   str.startsWith("-0X")) {
459  459         return createInteger(str);
460  460       }
                 ...
594  594     }
```

FIGURE 4.1: Buggy code and patch from Lang-16 bug

Assume a test generation scenario for the buggy version of the `NumberUtils` class in our example. Figure 4.2 shows a sample test case generated by EvoSuite during the search process. The execution of the test case *reaches* the buggy code, i.e., line 458. The execution of the buggy statement causes an incorrect internal program state (*infection*), i.e., a valid argument to `str` (`str="0Xa"`) must not cause the program to throw a `NumberFormatException`. The incorrect internal program state is *propagated* to an incorrect final state (failure) of the program, i.e., at line 3 in the test case, `createNumber("0Xa")` call should output a `java.lang.Integer` object with the value `10`, instead a `NumberFormatException` is thrown. EvoSuite does not have test oracles, hence it generates assertions in the tests assuming the program under test is correct. For example, Figure 4.3 shows the final test case generated by EvoSuite for the test case shown in Figure 4.2, which is not able to *reveal* the bug since the test case does not fail when it is executed against the buggy program.

```
1  …
2  String string0 = “0Xa”;
3  NumberUtils.createNumber(string0);
4  …
```

FIGURE 4.2: Test case generated by EvoSuite during the search for the buggy version of NumberUtils class from Lang-16

```
1  public void test001()  throws Throwable  {
2      // time taken = 83090
3      try {
4          NumberUtils.createNumber("0Xa");
5          fail("Expecting exception: NumberFormatException");
6      } catch(NumberFormatException e) {
7          //
8          // 0Xa is not a valid number.
9          //
10         verifyException("org.apache.commons.lang3.math.NumberUtils", e);
11     }
12 }
```

FIGURE 4.3: Final test case with assertions by EvoSuite for the buggy version of NumberUtils class from Lang-16

In the ideal scenario, if oracle automation [143] exists, the generated test cases can reveal the bugs. Without oracle automation, the best EvoSuite can do is to propagate the incorrect state of the program to the output. The scope of this thesis is to improve the bug detection capability of the test suites generated by SBST guided by defect prediction and not oracle automation. Therefore, in this thesis, we consider that DynaMOSA or the proposed SBST approaches detect a bug if they generate a test case that propagates the internal error to the output of the program. Nevertheless, we remind the readers that neither DynaMOSA nor our proposed SBST approaches are able to reveal existing bugs in a program without the aid of oracle automation. Finally, this limitation is not only applicable to our proposed SBST approaches and DynaMOSA, but also to other SBST techniques [3, 15, 51] in this space as well.

4.4 Bug Detection Evaluation Procedure

To determine if a bug is detected by a test suite generated by an SBST approach, we perform the following procedure in the experiments. We use the test execution

framework in Defects4J [136] for this task.

First, the flaky test cases are removed from the test suites using the 'fix test suite' script in Defects4J test execution framework as described in [23]. A test case should produce the same output when it is run against the version of the program it was created in the first place. If not, the test case is considered a flaky test case. For example, a test case containing an assertion that refers to the system time will produce different outputs when it is run at different times. Hence, that test case is flaky and needs to be removed from the test suite. To identify and remove the flaky tests from a test suite, 'fix test suite' first removes all uncompilable test classes until the test suite compiles. Then, the test suite is executed against the version of the program the test suite was created, i.e., buggy version in our case. If the execution reveals flaky tests, then they will be removed, and the test suite will be re-compiled and re-executed. This is repeated until the test suite executes against the buggy version and produces the same output for five consecutive runs (i.e., without detecting any flaky tests).

We use the fixed versions of the programs as the test oracles [131]. If a test suite running against the buggy version of a program produces a different output compared to what it produces when it is run against the fixed version, then it means the test suite detects the bug. We use the 'run bug detection' script in Defects4J test execution framework and fixed version of the program as the test oracle to determine if a test suite detects a bug. EvoSuite generates assertions assuming the program under test is correct, therefore the generated tests should always produce the same execution result when they are run against the buggy version. A test suite is considered broken, if it is not compilable or produces a different execution result when it is run against the buggy version. The test suite is considered it has missed detecting the bug, if the test execution results are same when it is run against the buggy and fixed versions of the program, if the results are different, then it is considered as it has detected the bug.

4.5 Performance Measures

In order to compare the proposed approaches against the benchmark SBST approaches, we define performance measures. The main performance measure used in this thesis is

the number of bugs detected. We define the measure number of bugs detected, $d(B)$, in Equation. (4.1).

$$d(B) = \sum_{b_i \in B} d(b_i) \tag{4.1}$$

where

- $B = \{b_1, \ldots, b_n\}$ is the set of bugs from the benchmark subjects and n is the number of bugs in the dataset.
- $d(b_i)$ denotes whether the bug b_i is detected by the test suite and is calculated as in Equation. (4.2). We consider a test suite detects a bug, if it detects the bug in any of the buggy classes of that bug.

$$d(b_i) = \begin{cases} 1 & \text{if } \sum\limits_{b_{ij} \in B_i} d(b_{ij}) > 1 \\ 0 & \text{if } \sum\limits_{b_{ij} \in B_i} d(b_{ij}) = 0 \end{cases} \tag{4.2}$$

where

- $B_i = \{b_{i1}, \ldots, b_{1k}\}$ is the set of buggy classes of the bug b_i and k is the number of buggy classes of b_i.
- $d(b_{ij})$ denotes whether the test suite detects the bug b_i in buggy class b_{ij} and is calculated as in Equation. (4.3).

$$d(b_{ij}) = \begin{cases} 1 & \text{if the test suite detects the bug } b_i \text{ in buggy class } b_{ij} \\ 0 & \text{if the test suite misses the bug } b_i \text{ in buggy class } b_{ij} \\ & \text{or is broken} \end{cases} \tag{4.3}$$

We repeat the test generation runs for each buggy class and for each approach to account for the non-deterministic behaviour of genetic algorithms used in SBST. We use mean and median number of bugs detected over a number of independent runs to report and

compare the bug detection performance of the SBST approaches. We also plot the number of bugs detected as boxplots and violin plots to visualise the distribution. We conduct sound statistical tests to derive conclusions from the results of the experiments. To check for statistical significance of the differences of the number of bugs detected by two approaches, we employ non-parametric Mann-Whitney U-Test with the significance level (α) 0.05 [144]. To compute the effect size of the differences, we conduct Vargha and Delaney's $\widehat{A}_{12}$ statistical test [145].

In the context of comparing the number of bugs detected, the $\widehat{A}_{12}$ statistic measures the probability that running SBST approach A yields higher $d(B)$ values than running approach B. We interpret the magnitude of $\widehat{A}_{12}$ statistic as shown in Table 4.1 [15]. We consider the effect size is negligible if $\widehat{A}_{12} \in (0.42, 0.58)$. If $\widehat{A}_{12} = 0.50$, then the two approaches are equivalent.

TABLE 4.1: Interpretation of the magnitude of $\widehat{A}_{12}$ statistic for approach A vs. B.

Interpretation	A > B	B > A
Large	$\widehat{A}_{12} \geq 0.75$	$\widehat{A}_{12} \leq 0.25$
Medium	$0.65 \leq \widehat{A}_{12} < 0.75$	$0.25 < \widehat{A}_{12} \leq 0.35$
Small	$0.58 \leq \widehat{A}_{12} < 0.65$	$0.35 < \widehat{A}_{12} \leq 0.42$

We also use the measure time to generate bug detecting test to compare the proposed SBST technique against the benchmark SBST technique in RO3. It is defined as the time to generate the first test case that can detect a bug. We use this measure to evaluate the SBST techniques in terms of the efficiency of detecting bugs. To check for statistical significance of the differences of the time to generate bug detecting tests by two approaches, we employ Wilcoxon signed-rank test [144] and its effect size, r [146]. We describe this measure in detail and the steps to calculate it and conduct statistical tests in Section 7.4.

In RO2, we conduct two-way ANOVA test to statistically analyse the effects of recall and precision of the defect predictor on the bug detection effectiveness of SBST. Prior to conducting two-way ANOVA test, we conduct necessary statistical tests to check our data holds the assumptions of the test. To check if the observed effects are of practical significance, we compute the epsilon squared effect size ($\widehat{\epsilon}^2$) [147] of the variations in number of bugs detected with respect to recall and precision. To further analyse which groups of recall and precision are significantly different from each other and their effect

size, we conduct the Tukey's Honestly-Significant-Difference test [148] as a post hoc test and compute the Cohen's d effect size, respectively. We describe this in detail in Section 6.3.

Previous work on defect prediction have used various performance measures to measure and compare the predictive power of defect predictors such as recall, precision, accuracy, probability of false alarms, area under the curve (AUC), F-measure, G-measure and Matthews correlation coefficient (MCC) [149]. Out of these measures, recall and precision have been widely used in previous work to report the performance of defect predictors [49, 50]. In this thesis, we mainly use recall and precision to characterise the performance of the defect predictors used in the proposed SBST approaches. For completeness, we report the accuracy and MCC of the defect predictors as well.

As shown in the confusion matrix in Table 4.2, a true positive is when buggy code is correctly labelled as buggy, a true negative is when non-buggy code is correctly labelled as non-buggy, a false positive is when non-buggy code is incorrectly labelled as buggy, and a false negative is when buggy code is incorrectly labelled as non-buggy. Recall is the rate of the defect predictor correctly identifying buggy code and is calculated as in Equation. (4.4). A defect predictor with recall closer to 1 means that it identifies most of the buggy code correctly. Recall is indicative of the false negatives by the defect predictor. Poor recall of the defect predictor means there are higher false negatives, and lower false negatives otherwise. Precision is the rate of the correct buggy code labels by the defect predictor and is calculated as in Equation. (4.5). A defect predictor with precision closer to 1 means that most of the buggy code labels produced by the defect predictor are correct. Precision is indicative of the false positives by the defect predictor. Poor precision of the defect predictor means there are higher false positives, and lower false positives otherwise.

Accuracy is the rate of correct labels by the defect predictor and is calculated as in Equation. (4.6). Accuracy can be biased in the case of highly imbalanced datasets, which is usually a commonplace situation in the context of defect prediction [150]. Accuracy closer to 1 means the defect predictor correctly labels most of the code. However, if the actual buggy code is significantly smaller compared to the non-buggy code, then the accuracy can still be closer to 1 even if the defect predictor labels all the buggy code incorrectly. MCC is the Pearson correlation for a contingency table [151, 152]. MCC

is an unbiased performance metric and is calculated as in Equation. (4.7). MCC closer to +1 means the defect predictor is better at classification and equal to 0 means the classification is random. MCC closer to -1 means it is better at perverse classification, hence swapping the states of the output of the defect predictor will make it a better predictor.

TABLE 4.2: Confusion Matrix.

	Actual buggy	**Actual non-buggy**
Predicted buggy	True Positive (TP)	False Positive (FP)
Predicted non-buggy	False Negative (FN)	True Negative (TN)

$$\text{Recall} = \frac{TP}{TP + FN} \tag{4.4}$$

$$\text{Precision} = \frac{TP}{TP + FP} \tag{4.5}$$

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN} \tag{4.6}$$

$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{4.7}$$

4.6 Threats to Validity

This section discusses the validity threats to the research studies conducted to achieve the main goal of the thesis. The types of validity threats considered in this thesis are construct validity, internal validity, conclusion validity and external validity. In this section, we particularly focus on the validity threats that are common across all the studies conducted. The more specific threats to the individual studies are discussed under each contribution chapter.

4.6.1 Construct Validity

As we discussed in Section 4.5, we use recall and precision to characterise the performance of the defect predictors used in our studies. Recall and precision can be biased in the case of highly imbalanced datasets, which is usually a commonplace situation for defect datasets as there are only a few number of buggy entities compared to non-buggy ones [150]. Recall and precision are indicative of the false negatives and false positives in the predictions, respectively. Hence, they are more suitable measures to characterise defect predictors when we investigate the impact of defect prediction imprecision on SBST in RO2. Furthermore, they have been widely used in previous work to report the performance of defect predictors [49, 50] and often preferred by practitioners [47] to measure the performance. For completeness, we use MCC, which is an unbiased performance metric, to report the defect predictor performance as well.

We only consider the labelled bugs in the Defects4J dataset in our experimental studies, which is likely smaller than the set of actual bugs in the dataset. In order to check how many actual bugs are detected by the SBST approaches in the evaluations, we have to manually validate all the generated test suites, which is not a feasible task given the large number of test suites generated in the experiments. For example, there are $41,200$, $144,600$ and $36,150$ test suites generated by all the SBST approaches in the experimental studies conducted for RO1, RO2, and RO3, respectively. Therefore, in line with previous work [24, 43], we choose to conduct the experimental studies considering only the labelled bugs in Defects4J dataset.

4.6.2 Internal Validity

A threat to internal validity exists from the use of the term *experiment* in our thesis. According to the hallmarks characterised by Ayala et al. [153], our studies in the thesis correspond to *experiments with limited control*. This is because we use a retrospective repository (i.e., Defects4J) as the dataset, hence our experimental designs do not fully cover the *control* hallmark [153]. We rely on Defects4J benchmark for the accuracy of the dataset, hence our studies may be prone to different biases as we do not have any chance of control over the dataset collection. The only way to guarantee the maximum level of control is to use prospective repositories. However, all the available bugs datasets

are retrospective repositories [25, 139–141]. In fact, in their study, Ayala et al. [153] reported that only one out 254 studies use prospective repositories.

To limit the influence of randomness of genetic algorithm used in SBST approaches on our results, we repeat the test generation runs, for example 25 times. The SBST approaches are compared considering the mean and median performance and results of the statistical tests.

The proposed SBST approaches and the baseline approaches are implemented within the EvoSuite framework. Therefore, any confounding effects on the results due to different implementations or use of tools are mitigated in our experimental studies.

4.6.3 Conclusion Validity

To account for any threats to the conclusion validity, we derive conclusions from the experimental results after conducting sound statistical tests; non-parametric Mann-Whitney U-Test, Vargha and Delaney's $\widehat{A}_{12}$ statistic, two-way ANOVA test, epsilon squared effect size, Tukey's Honestly-Significant-Difference test, Cohen's d effect size, Welch ANOVA test, Games-Howell post-hoc test, Wilcoxon signed-rank test, and its effect size, r.

4.6.4 External Validity

As we described in Section 4.1, we use real bugs from Defects4J dataset as experimental subjects. These bugs are extracted by manually validating bug fixes from six open source projects. These open source projects may not represent all program characteristics, especially industrial projects. Nevertheless, Defects4J dataset has been widely used in the related literature, including previous work in automated test generation, as a benchmark [23, 42, 43, 137, 138, 154]. We identify validating the proposed approaches and replicating our studies on other bugs datasets [139–141] as future work to increase the generalisability of our results. To enable other researchers to easily replicate our studies, replication packages and prototype tools of the proposed approaches are made publicly available.

As we described in Section 2.2.2, we implement the proposed SBST approaches within the EvoSuite framework and conduct experimental studies. EvoSuite generates JUnit test suites for Java programs. Thus, we may not be able to generalise our results to other programming languages. However, the concept of using defect prediction to guide SBST for improved bug detection is not language dependent and can be applied to other programming languages as well.

Chapter 5

Time Budget Allocation

5.1 Introduction

Search-based software testing (SBST) techniques have been shown to be more effective not only at detecting bugs, but also at achieving high code and mutation coverage compared to other automated test generation techniques such as random search and dynamic symbolic execution (DSE) [23, 26, 155]. SBST techniques are guided by fitness functions based on coverage criteria such as line, branch and weak mutation coverage, and they are good at efficiently generating test suites with high coverage. As a result, SBST techniques are capable of detecting more bugs compared to the random search or DSE approaches. For instance, EvoSuite detected on average 23% of the bugs from the Defects4J dataset, and Randoop [4], a feedback-directed random test generation technique, detected only 15% of the bugs on average while generating substantially larger test suites than EvoSuite [23]. Similar results can be observed in the study conducted by Almasi et al. [26] on an industrial project. More recently, EvoSuite was shown to achieve significantly higher mutation score than EvoSuiteDSE [156], which uses a pure dynamic symbolic execution approach [155]. Given that mutation coverage significantly correlates with the bug detection of the test suites [101], SBST can be considered as the better candidate for bug detection improvement among random search and dynamic symbolic execution techniques.

SBST tools like EvoSuite generate test suites for each class in the project separately. The time budget allocated for the genetic algorithm to search for test cases is an important

parameter that needs to be tuned carefully. The SBST technique is able to extensively explore and exploit the search space of possible test inputs with a large time budget. As a result, it is more likely to detect bugs with a larger time budget compared to a smaller time budget. While it is beneficial for the genetic algorithm to have higher time budgets allocated, the resource constrained nature in practice prevents every class in a project from receiving higher time budgets.

In practice, the time budget allocated for test generation depends on several factors; i) size of the project, ii) frequency of test generation runs, and iii) availability of computational resources. Industrial projects are usually very large and can have thousands of classes [45]. To run test generation for such projects having thousands of classes, it will take at least 16-17 CPU-hours to finish the task with spending just one minute per each class. Modern software development practices such as agile requires faster and frequent feedback from testing. This makes it difficult to run SBST on developer machines as it may run frequently for a long duration and slow them down as a result [44]. Instead, practitioners can opt for continuous integration (CI) systems to provide the opportunity for SBST tools to cater this requirement of running frequent test generation for large projects. However, the CI systems already have high demands from the existing processes in the system and the available computational resources in CI systems are usually limited in practice. This prompts the necessity of SBST using minimal resources possible such that the practitioners will be able to run them in the developer machines or deploy them in the CI systems without disrupting the existing processes. Therefore, this raises the question, 'How should we optimally utilise the available computational resources, in this case time budget, to generate test suites for a whole project with maximising the chances of detecting bugs?'.

Previous work that studied the bug detection performance of SBST considered SBST approaches used in their studies to focus only on high code coverage [23, 24, 26, 135]. Hence, budget allocation for classes in a project to maximise bug detection has been overlooked by previous studies. They allocated a fix time budget to test generation for each buggy class. Since the buggy classes are not known prior to running tests, in practice all the classes in the project have to be allocated the same time budget. Usually, most classes are not buggy, hence we argue that this is a sub-optimal strategy. Only Campos et al. [44] proposed a budget allocation approach to maximise the branch coverage. They used the number of branches in classes to decide how much time budget

should be allocated for each of the classes in a project. This approach will not be suitable for the task of maximising bug detection, since high code coverage alone is not sufficient to detect the maximum number of bugs.

Ideally, SBST should run test generation for the buggy classes and non-buggy classes can be left out from test generation. However, the buggy classes are not known to practitioners prior to running tests. We use defect prediction that works at class level to get information of the probability of defectiveness of classes in a project. We differentiate the classes in a project based on this information and allocate time budget to classes accordingly. Therefore, in this chapter, we aim to achieve the following research objective;

> RO1: Develop an approach that allocates time budget to classes for test generation based on defect prediction.

To achieve this research objective, we propose a time budget allocation approach called, defect prediction guided SBST (SBST$_{DPG}$), and demonstrate its improved efficiency and effectiveness in terms of detecting bugs through an experimental evaluation. SBST$_{DPG}$ leverages Schwa [30] which gives the probabilities of defectiveness of the classes in a project. We introduce a module named budget allocation based on defect scores (BADS), which takes the probabilities of defectiveness of classes and outputs time budgets to them. Upon receiving the time budgets, SBST$_{DPG}$ runs test generation for the classes using the state-of-the-art SBST tool EvoSuite with DynaMOSA. We use Defects4J dataset as benchmark subjects to experimentally evaluate SBST$_{DPG}$ against the state-of-the-art SBST approach in terms of bug detection efficiency and effectiveness.

5.2 Motivation

This example illustrates the limitation of SBST focusing only on high code coverage when using it for detecting bugs. Figure 5.1 shows the buggy code snippet and the applied patch for `MathUtils` class from Math-94 bug in Defects4J. The `if` condition at line 412 is placed to check if either `u` or `v` is zero. This is a classic example of a bug due to an integer overflow. Assume the method is called with the following inputs `MathUtils.`*gcd*`(1073741824, 1032)`. Then, the `if` condition at line 412 is expected to

be evaluated to `false` since both `u(1073741824)` and `v(1032)` are non-zeros. However, the multiplication of `u` and `v` causes an integer overflow to zero, and the `if` condition at line 412 is evaluated to `true`. As shown in the diff, the applied patch rectifies this issue by individually checking if `u` or `v` are zero at line 412. To detect this bug, a test should not only cover the `true` branch of the `if` condition at line 412, but also pass the non-zero arguments `u` and `v` such that their multiplication causes an integer overflow to zero.

```
411 411     public static int gcd(int u, int v) {
412         -     if (u * v == 0) {
        412 +     if ((u == 0) || (v == 0)) {
413 413             return (Math.abs(u) + Math.abs(v));
414 414         }
415 415         …
416 416     }
```

FIGURE 5.1: Buggy code and patch from Math-94 bug

The fitness function for the `true` branch of the `if` condition at line 412 is $u*v/(u*v+1)$, and it tends to reward the test inputs `u` and `v` whose multiplication is closer to zero more than the ones whose multiplication is closer to causing an integer overflow to zero. For an example, suppose we have two test cases T_1 and T_2 as shown in the Figures 5.2 and 5.3, respectively, in the current iteration of the genetic algorithm. The fitness of T_1 and T_2 are $6/(6+1) = 0.143$ and $14997485/(14997485+1) \approx 0.999$ with respect to the `true` branch at line 412. Thus, T_1 is considered fitter when compared with T_2, while the T_2 is closer to detect the bug than T_1. An SBST technique focusing on high code coverage is more likely to find a test case with both/either `u` and/or `v` $= 0$ and consider this code as covered than find a test case with a pair of `u` and `v` causing an integer overflow to zero.

```
1  …
2  int int0 = 2;
3  int int1 = 3;
4  MathUtils.gcd(int0, int1);
5  …
```

FIGURE 5.2: Sample test case T_1

```
1  …
2  int int0 = 12085;
3  int int1 = 1241;
4  MathUtils.gcd(int0, int1);
5  …
```

FIGURE 5.3: Sample test case T_2

In a situation like this, we can increase the chances of detecting the bug by allowing the search method to extensively explore and exploit the search space of possible test inputs and generate more than one test case (test inputs) for such branches. We propose to use defect predictions at class level to allocate high time budgets for highly likely to be defective classes. Our proposed approach identifies `MathUtils` class as a highly likely to be defective class with the aid of a defect predictor and allocates a large time budget for test generation with SBST. This allows the SBST technique to extensively explore and exploit the search space of possible test cases that cover the `true` branch at line 412, which in turn increases the chances of finding test cases that can detect the bug.

5.3 Defect Prediction Guided Search-Based Software Testing

Defect prediction guided SBST (SBST_{DPG}) (see Figure 5.4) uses defect scores of each class produced by a defect predictor to focus the search towards the defective areas of a program. Existing SBST approaches allocate the available time budget equally for each class in the project [14, 23, 72, 80]. Usually, most classes are not buggy, hence we argue that this is a sub-optimal strategy. Ideally, valuable resources should be spent in testing classes that are likely to be buggy, hence we employ a defect predictor, known as Schwa [30], to calculate the likelihood that a class in a project is defective. Our approach has three main modules: i) Defect Predictor (DP), ii) **B**udget **A**llocation Based on **D**efect **S**cores **(BADS)**, and iii) Search-Based Software Testing (SBST).

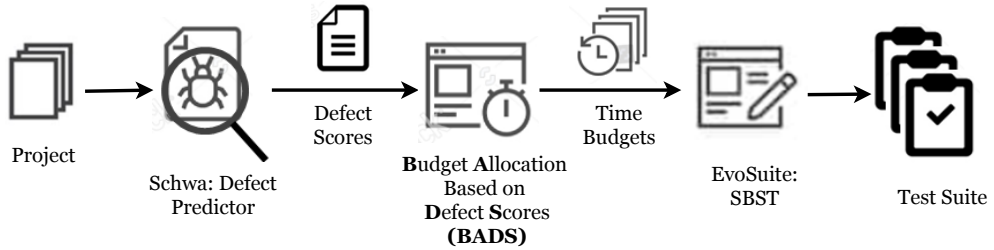

FIGURE 5.4: Defect Prediction Guided SBST Overview

5.3.1 Defect Predictor

The defect predictor gives a probability of defectiveness (defect score) for each class in the project. The vector **s** represents this output. In our implementation of SBST_{DPG},

we choose (a) the level of granularity of the defect predictor to be the class level, and (b) the Schwa [30] as the defect predictor module.

Schwa is an enhanced version of the cache-based prediction model used by the Google Engineering team [39, 40]. Paterson et al. [42] successfully applied Schwa as the defect predictor in G-clef to inform a test case prioritisation strategy of the classes that are likely to be buggy. Certainly other defect prediction approaches proposed in the literature (e.g., FixCache [29] and Change Bursts [38]) would also be suitable for the task at hand. A strength of Schwa is its simplicity, and that it does not require training a classifier which makes it easy to apply to an industrial setting where training data is not always available (like the one we study).

Schwa uses the following three measures which have been shown to be effective at producing defect predictions in the literature (see Section 3.3); i) *Revisions* - timestamps of revisions (recent changes are likely to introduce faults), ii) *Fixes* - timestamps of bug fix commits (recent bug fixes are likely to introduce new faults), and iii) *Authors* - timestamps of commits by new authors (recent changes by the new authors are likely to introduce faults). The Schwa tool extracts this information through mining a version control system such as Git [108]. The tool is readily available to use as a Python package at Pypi [157]. Therefore, given the robustness of this tool and its approach, we decide to use it as the defect predictor module in our approach.

Schwa [158] starts with extracting the three metrics; *Revisions* (R_c), *Fixes* (F_c), and *Authors* (A_c) for all classes $c \in C$ in the project. For each timestamp, it calculates a time weighted risk (TWR) [39] using the Equation. (5.1).

$$TWR(t_i) = \frac{1}{1 + \exp(-12t_i + 2 + (1 - TR) * 10)} \tag{5.1}$$

The quantity t_i is the timestamp normalised between 0 and 1, where 0 is the normalised timestamp of the oldest commit under consideration and 1 is the normalised timestamp of the latest commit. The number of commits that Schwa tracks back in version history of the project (n) is a configurable parameter and it can take values from one commit to all the commits. The parameter $TR \in [0, 1]$ is called the Time Range and it allows to change the importance given to the older commits. The time weighted risk formula scores recent timestamps higher than the older ones (see Figure 5.5).

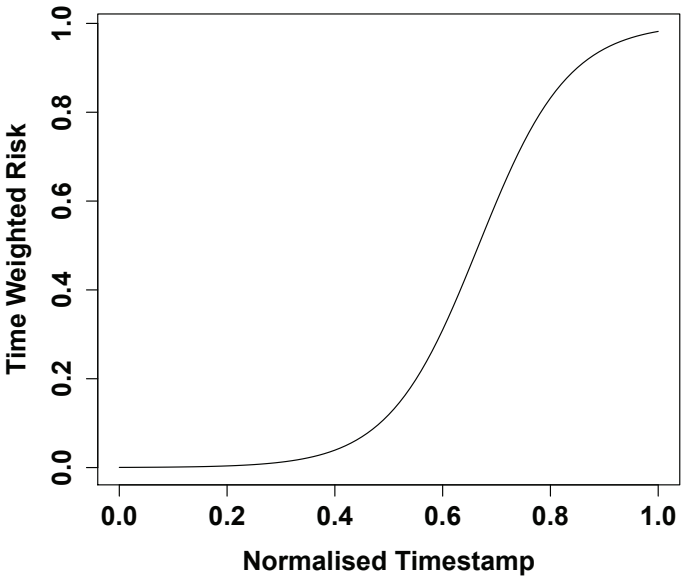

FIGURE 5.5: Time Weighted Risk ($TR = 0.4$)

Once Schwa calculated the TWRs, it aggregates these TWRs per each metric, and calculates a weighted sum s_c for each class $c \in C$ in the project as in Equation. (5.2).

$$\begin{aligned} s_c = w_r * \sum_{t_i \in R_c} TWR(t_i) + w_f * \sum_{t_i \in F_c} TWR(t_i) \\ + w_a * \sum_{t_i \in A_c} TWR(t_i) \end{aligned} \tag{5.2}$$

The sum $\sum_{t_i \in R_c} TWR(t_i)$ is the total of the time weighted risks of the *Revisions* metric for class c. Similarly, $\sum_{t_i \in F_c} TWR(t_i)$ and $\sum_{t_i \in A_c} TWR(t_i)$ are the sums of the TWRs of the *Fixes* and *Authors* metrics for class $c \in C$. The quantities w_r, w_f, and w_a are weights that modify the TWR sum of each metric and their sum is equal to 1. The weighted sum, s_c, is called the score of class $c \in C$.

Finally, Schwa estimates the probability $p(c)$ of that a class c is defective as given in Equation. (5.3).

$$p(c) = 1 - \exp(-s_c) \tag{5.3}$$

We refer to this probability of defectiveness $p(c)$ as the defect score of class $c \in C$.

5.3.2 Budget Allocation Based on Defect Scores

BADS takes the defect scores ($\mathbf{s} = \{p(c)|c \in C\}$) as input and decides on how to allocate the available time budget to each class based on these scores, producing a vector $\mathbf{t}$ as output. Ideally, all the defective classes in the project should get more time budget while non-defective classes can be left out from test generation. However, the defect predictor only gives an estimation of the probability of defectiveness. Therefore, BADS allocates more time budget to the highly likely to be defective classes than to the less likely to be defective classes. This way we expect SBST to get higher time budget to extensively explore for test cases in defective classes rather than in non-defective ones.

5.3.2.1 Exponential Time Budget Allocation Based on Defect Scores

Algorithm 1 Exponential Time Budget Allocation Based on Defect Scores

Input: The set of all the classes C, where $N = |C|$
$\mathbf{s} = \{s_1, s_2, \ldots, s_N\}$
$T, t_{\min}, T_{DP}$
e_a, e_b, e_c
Output: $\mathbf{t} = \{t_1, t_2, \ldots, t_N\}$

1: **procedure** AllocateTimeBudget
2: $\mathbf{r} \leftarrow$ Assign-Rank($\mathbf{s}$)
3: $\mathbf{r}' \leftarrow$ Normalise-Rank($\mathbf{r}$)
4: $\mathbf{w}' \leftarrow \emptyset$
5: **for** ***all*** $c_i \in C$ **do**
6: $w'_i \leftarrow e_a + e_b * \exp(e_c * r'_i)$
7: $\mathbf{w} \leftarrow$ Normalise-Weight($\mathbf{w}'$)
8: $\mathbf{t} \leftarrow \emptyset$
9: **for** ***all*** $c_i \in C$ **do**
10: $t_i \leftarrow w_i * (T - N * t_{\min} - T_{DP}) + t_{\min}$
11: Return($\mathbf{t}$)

Algorithm 1 illustrates the proposed time budget allocation algorithm of BADS, where $\mathbf{s}$ is the set of defect scores of the classes, T is the total time budget for the project, $t_{\min}$ is the minimum time budget to be allocated for each class, T_{DP} is the time spent by the defect predictor module, and e_a, e_b, and e_c are parameters of the exponential function

that define the shape of the exponential curve. **t** is the set of time budgets allocated for the classes.

The defect scores assignment in Figure 5.6 is a good example of the usual defect score distribution by a defect predictor. Usually, there are only a few classes which are actually buggy. Allocating higher time budgets for these classes would help maximise the bug detection of the test generation tool. Following this observation and the results of our pilot runs, we use an exponential function (line 6 in Algorithm 1) to highly favour the budget allocation for the few highly likely to be defective classes.

Moreover, there is relatively higher number of classes which are moderately likely to be defective (e.g., 0.5 < defect score < 0.8). It is also important to ensure there is sufficient time budget allocated for these classes. Otherwise, neglecting test generation for these classes could negatively affect bug detection of the test generation tool. We introduce a minimum time budget, $t_{\min}$, to all the classes because we want to ensure that every class gets a budget allocated regardless of the defectiveness predicted by the defect predictor. The exponential function in Algorithm 1 together with $t_{\min}$ allow an adequate time budget allocation for the moderately likely to be defective classes.

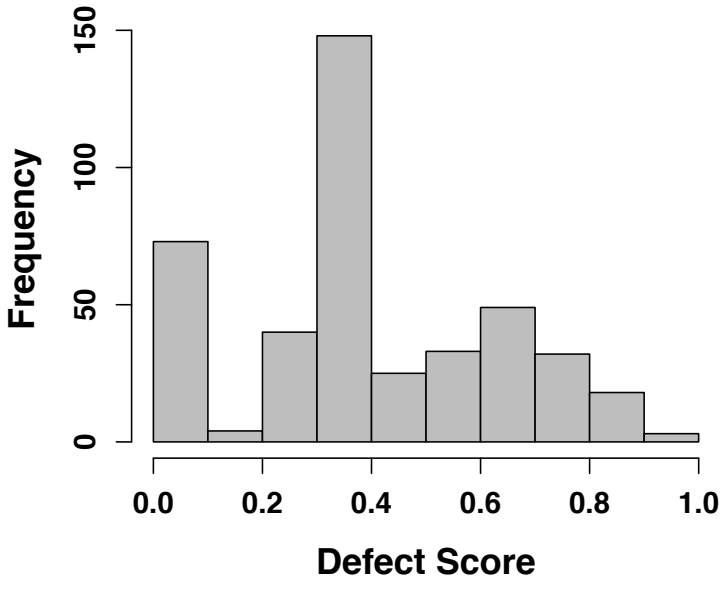

FIGURE 5.6: Distribution of the defect scores assigned by Schwa for the classes in Chart-9 bug from Defects4J.

Upon receiving the defect scores (**s**), BADS assigns ranks (**r**) for all the classes according to the defect scores. Next, the NORMALISE-RANK function normalises the ranks in the range [0,1], where the rank of the most likely to be defective class is 0 and the

least likely to be defective class is 1. Then, each class gets a weight ($w_i^{'}$) assigned based on its normalised rank by the exponential function. The amount of time budget allocated to class c_i is proportional to $w_i^{'}$. The parameters e_a, e_b, and e_c have to be carefully selected such that the weights are almost equal and significantly small for the lower-ranked classes, and the difference between the weights of adjacently ranked classes rapidly increases towards the highly-ranked classes. The NORMALISE-WEIGHT function normalises the weights to the range [0,1], ensuring the summation is equal to 1, and produces the normalised weights vector **w**. Finally, BADS allocates time budget for each class from the remaining available time budget, $T - N * t_{\min} - T_{DP}$, based on its normalised weight (line 10 in Algorithm 1) .

5.3.2.2 The 2-Tier Approach

According to the defect predictor outcome, almost all the classes in the project get non-zero defect scores attached to them. This gives the impression that all these classes can be defective with at least a slight probability. However, in reality, this does not hold true. For a given project version, there are only a few defective classes. A defect predictor is likely to predict that non-defective classes are also defective with a non-zero probability. While the exponential function disfavours the budget allocation for these less likely to be defective classes, $t_{\min}$ guarantees a minimum time budget allocated to them. If we decrease $t_{\min}$ in order to make the budget allocation negligible for the likely to be non-defective classes, then it would risk a sufficient time budget allocation for the moderately likely to be defective classes.

We propose the 2-Tier approach which divides the project into two tiers following the intuition that some of the classes are defective and the others are not. BADS sorts the classes into two tiers before the weights assignment, such that the highly likely to be defective classes are in the *first tier* and the less likely to be defective classes are in the *second tier*. This allows to further discriminate the less likely to be defective classes, and favour the highly likely to be defective classes by simply allocating only a smaller fraction of the total time budget to the *second tier* and allocating the rest to the *first tier*. Section 5.4.1.3 provides more details on the parameter selection of the 2-Tier approach.

5.3.3 Search-Based Software Testing

Given the maturity of EvoSuite (as discussed in Section 2.2.2), we use it as the SBST module in our defect prediction guided SBST approach. We use DynaMOSA as the SBST technique in EvoSuite (described in 2.2.4.3).

5.4 Experimental Evaluation

We evaluate our approach in terms of its efficiency in detecting bugs, and the effectiveness in detecting unique bugs, i.e., bugs that cannot be detected by the benchmark approach. Our first research question is:

RQ1: Is $SBST_{DPG}$ more efficient in detecting bugs compared to the state of the art?

To answer this research question, we run a set of experiments where we compare our approach against the baseline method discussed in Section 5.4.1.2. All methods are employed to generate test cases for Defects4J [136], which is a well-studied benchmark of buggy programs described in Section 4.1. Once the test suites are generated, we check if they detect the bugs in the programs, and report the results as the mean and median over 20 runs. To check for statistical significance of the differences and the effect size, we employ two-tailed non-parametric Mann-Whitney U-Test with the significance level (α) 0.05 [144] and Vargha and Delaney's $\widehat{A}_{12}$ statistic [145]. We also plot the results as boxplots to visualise their distribution.

In addition, to analyse the effectiveness of the proposed approach, we seek to answer the following research question:

RQ2: Does $SBST_{DPG}$ detect more unique bugs?

To answer this research question, we analyse the results from the experiments in more detail. While the first research question focuses on the overall efficiency, in the second research question we aim to understand if SBST_{DPG} is capable of detecting more unique bugs which can not be detected by the baseline method. Part of the efficiency of our proposed method, however, could be due to its robustness, which is measured by the success rate, hence we also report how often a bug is detected over 20 runs.

5.4.1 Experimental Settings

5.4.1.1 Time Budget

In real world scenarios, total time budget reserved for test generation for a project depends on how it is used in the industry. For example, a project having hundreds of classes and running SBST 1-2 minutes per class takes several hours to finish test generation. If an organisation wants to adapt SBST in their CI system [159], then it has to share the resources and schedules with the processes already in the system; regression testing, code quality checks, project builds etc. In such case, it is important that SBST uses minimal resources possible, such that it does not idle other jobs in the system due to resource limitations.

Panichella et al. [14] showed that DynaMOSA is capable of converging to the final branch coverage quickly, sometimes with a lower time budget like 20 seconds. This is particularly important since faster test generation allows more frequent runs and thereby it makes SBST suitable to fit into the CI/CD pipeline. Therefore, we decide that 30 seconds per class is an adequate time budget for test generation and 15 seconds per class is a tight time budget in a usual resource constrained environment. We conduct experiments for 2 cases of total time budgets (T); $15 * N$ and $30 * N$ seconds.

5.4.1.2 Baseline Selection

As discussed in Section 4.2, we use the current state-of-the-art SBST technique, DynaMOSA [14], with equal time budget allocation, SBST_{noDPG}, as our baseline for comparison. Previous work on bug detection capability of SBST allocated an equal time budget for all the classes [23, 26, 135]. Even though, Campos et al. [44] proposed a budget allocation approach targeting the maximum branch coverage, we do not consider this as a baseline in our work as we focus on bug detection instead. As we discussed in Chapter 4, our intended application scenario is generating tests to detect bugs not only limited to regressions, but also the bugs that are introduced to the system in different times. Hence, we consider generating tests to all of the classes in the project regardless of whether they have been changed or not. Therefore, in equal budget allocation, total time budget is equally allocated to all the classes in a project.

5.4.1.3 Parameter Settings

There are three modules in our approach. Each module has various parameters to be configured, and the following subsections outline the parameters and their chosen values in our experiments.

Schwa. Schwa has five parameters to be configured; w_r, w_f, w_a, TR, and n. We choose the default parameter values used in Schwa [158] as follows: $w_r = 0.25$, $w_f = 0.5$, $w_a = 0.25$, and $TR = 0.4$. Our preliminary experiments with $n = 50, 100, 500, 1000$ and *all commits* suggest that $n = 500$ gives most accurate predictions.

EvoSuite. Arcuri and Fraser [160] showed that parameter tuning of SBST techniques is an expensive and long process, and the default values give reasonable results when compared to tuned parameters. Therefore, we use the default parameter values used in EvoSuite in previous work [14, 89] except for the following parameters.

Coverage criteria: We use branch coverage as coverage criterion inline with the prior studies which investigated bug detection effectiveness of EvoSuite [23, 26]. EvoSuite with branch coverage was shown to be the most effective coverage criterion in terms of detecting bugs when compared with other criteria like line, output and weak mutation coverage [24, 43].

Assertion strategy: As Shamshiri et al. [23] mentioned, mutation-based assertion filtering can be computationally expensive and lead to timeouts sometimes. Therefore, we use all possible assertions as the assertion strategy.

Given a coverage criterion (e.g., branch coverage), DynaMOSA explores the search space of possible test inputs until it finds test cases that cover all of the targets (e.g., branches) or the time runs out (i.e., time budget). These are known as stopping criteria. This way, if the search achieves 100% coverage before the timeout, any remaining time budget will be wasted. At the same time, DynaMOSA aims at generating only one test case to cover each target in the class under test (CUT), since its objective is to maximise the coverage criterion given. This also helps in minimising the test suite produced. However, when it comes to detecting bugs in the CUT, just covering the bug does not necessarily imply that the particular test case can detect the bug. Hence, we find that using 100% coverage as a stopping criterion and aiming at finding only one test case for each target deteriorate the bug detection capability of DynaMOSA. Therefore, in our approach, we

configure DynaMOSA to generate more than one test case for each target in the CUT, retain all these test cases, disable test suite minimisation and remove 100% coverage from the stopping criteria. By doing this, we compromise the test suite size in order to increase the bug detection capability of SBST.

BADS. Following the results of our pilot runs, we use the default threshold of 0.5 to allocate the classes into the two tiers. In particular, the top half of the classes (ranked in descending order according to defect scores) are allocated in the *first tier* (N_1) and the rest are in the *second tier* (N_2). N_1 and N_2 are the number of classes in the *first* and *second tiers* respectively.

Our preliminary results also suggest that allocating 90% and 10% of the total time budget (T) to the *first tier* (T_1) and the *second tier* (T_2) sufficiently favours the highly likely to be defective classes, while not leaving out the less likely to be defective classes from test generation. In particular, we choose $T_1 = 27 * N_1$ and $T_2 = 3 * N_2$ seconds at $T = 15 * N$ and $T_1 = 54 * N_1$ and $T_2 = 6 * N_2$ seconds at $T = 30 * N$. We choose 15 and 30 seconds as t_{min} for the *first tier* ($t_{\min_1}$) at $T = 15 * N$ and $T = 30 * N$ respectively. The rationale behind choosing these values for $t_{\min_1}$ is that it guarantees the classes in the *first tier* at least get a time budget of the equal budget allocation (i.e., budget allocation without defect prediction guidance). For t_{min} of the *second tier* ($t_{\min_2}$), we assign 3 and 6 seconds at $T = 15 * N$ and $T = 30 * N$ because T_2 is not enough to go for an exponential allocation.

The parameters for the exponential function are as follows: $e_a = 0.02393705$, $e_b = 0.9731946$, and $e_c = -10.47408$. The rationale behind choosing the parameter values for the exponential function is as follows. The exponential curve is almost flat and equal to 0 for the values in the x axis from 0.5 to 1 (see Figure 5.7). Then, after $x = 0.5$, it starts increasing towards $x = 0$. Finally, at $x = 0$, the output is equal to 1.

5.4.1.4 Prototype

We implement the defect prediction guided SBST approach in a prototype tool in order to experimentally evaluate it. The prototyped tool is maintained and available to download from here: https://github.com/SBST-DPG/sbst-dpg.

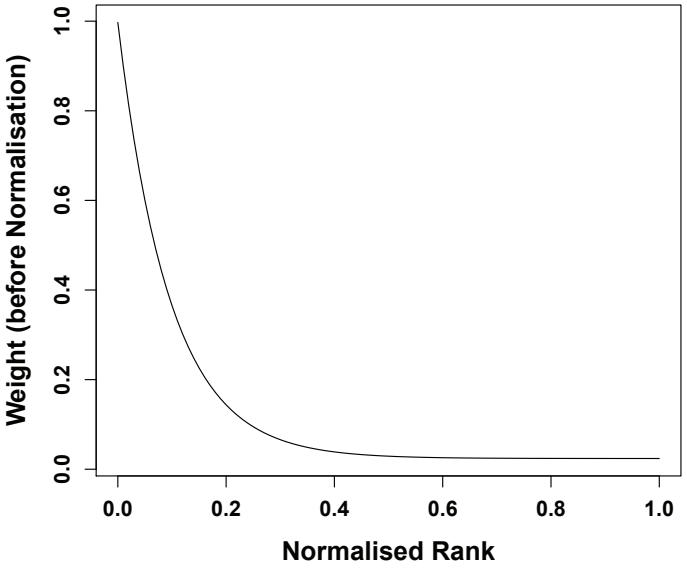

FIGURE 5.7: Exponential Function of BADS. $e_a = 0.02393705$, $e_b = 0.9731946$, and $e_c = -10.47408$

5.4.1.5 Experimental Protocol

As we mentioned earlier, to answer *RQ1* and *RQ2*, we conduct experiments for $T = 15{*}N$ and $30 * N$ seconds.

In SBST$_{DPG}$, Schwa uses current versions of the repositories of the projects. For each bug, Schwa predicts the defectiveness of the classes at the commit just before the bug fixing commit. For each bug in Defects4J, there is a buggy version and a fixed version of the project. We take each buggy version of the projects, and then generate test suites only for the buggy class(es) of that project version using the two approaches. To take the randomness of SBST into account, we repeat each test generation run 20 times, and carry out statistical tests when necessary. Consequently, we have to run a total of 2 (approaches) $*$ 511 (buggy classes) $*$ 20 (repetitions) $*$ 2 (time budgets) = 40,880 test generations. We collect the generated test suites after each test generation run. We determine if the 40,800 generated test suites detect the bug by using the method described in Section 4.4. Altogether, the experimental evaluation took roughly 34,600 CPU-hours.

5.4.2 Results

We present the results for each research question following the method described in Section 5.4. While the main aim is to evaluate if our approach is more efficient than the state of the art, we also focus on explaining its strengths and weaknesses.

TABLE 5.1: Mean and median number of bugs detected by the two approaches against different total time budgets.

T (s)	Mean		Median		p-value	$\widehat{A}_{12}$
	SBST_{DPG}	SBST_{noDPG}	SBST_{DPG}	SBST_{noDPG}		
$15 * N$	**151.45**	133.95	**150.5**	134.0	**<0.0001**	**0.94**
$30 * N$	**171.45**	166.9	**170**	167.5	0.0671	0.67

RQ1. Is SBST_{DPG} efficient in detecting bugs?

As described in Section 5.4, we perform 20 runs for each SBST approach and each buggy program in Defects4J and report the results as boxplots in Figure 5.8. As we can see, overall, our proposed method SBST_{DPG} detects more bugs than the baseline approach for both 15 and 30 seconds time budgets.

We also report the means, medians and the results from the statistical analysis in Table 5.1. SBST_{noDPG} detects 133.95 bugs on average at total time budget of 15 seconds per class. SBST_{DPG} outperforms SBST_{noDPG}, and detects 151.45 bugs on average, which is an average improvement of 17.5 (+13.1%) more bugs than SBST_{noDPG}. The difference of the number of bugs detected by SBST_{DPG} and SBST_{noDPG} is statistically significant according to the Mann-Whitney U-Test (p-value < 0.0001) with a large effect size ($\widehat{A}_{12} = 0.94$). Thus, we can conclude that SBST_{DPG} is more efficient than SBST_{noDPG}.

At total time budget of 30 seconds per class, SBST_{DPG} detects more bugs than the SBST_{noDPG}. According to the Mann-Whitney U-Test, the difference between SBST_{DPG} and SBST_{noDPG} is not statistically significant, with a p-value of 0.067. However the effect size of 0.67 suggests that SBST_{DPG} detects more bugs than SBST_{noDPG} 67% of the time, which is significant given how difficult it is to find bug detecting test cases [161].

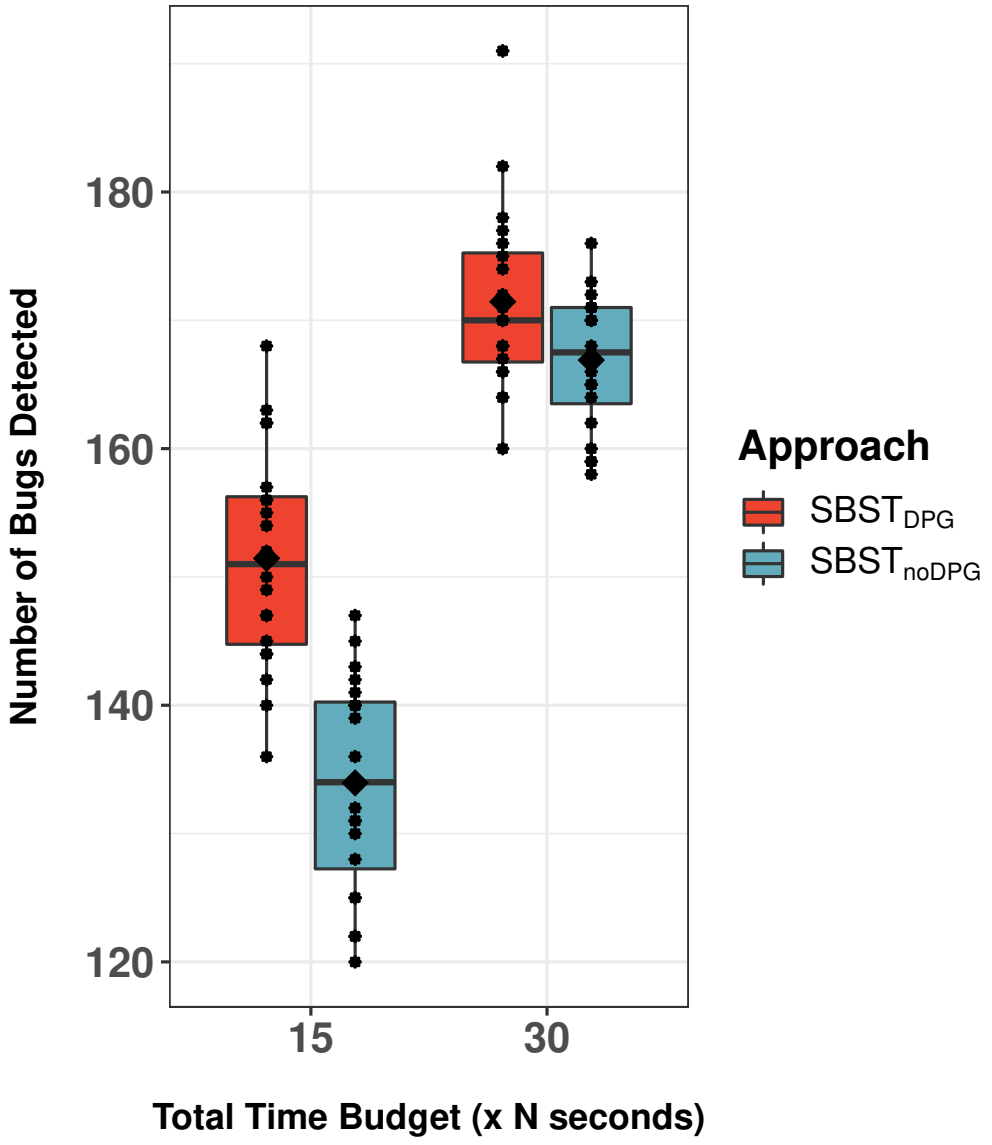

FIGURE 5.8: The number of bugs detected by the 2 approaches against different total time budgets

In summary, defect prediction guided SBST ($SBST_{DPG}$) is significantly more efficient than SBST without defect prediction guidance ($SBST_{noDPG}$) when they are given a tight time budget in a usual resource constrained scenario. When there is sufficient time budget $SBST_{DPG}$ is more effective than $SBST_{noDPG}$ 67% of the time.

To further analyse the differences between the two approaches, Figure 5.9 reports the distribution of the number of classes where a bug was detected across 20 runs for the 2 approaches grouped by the relative ranking position produced by Schwa at total time

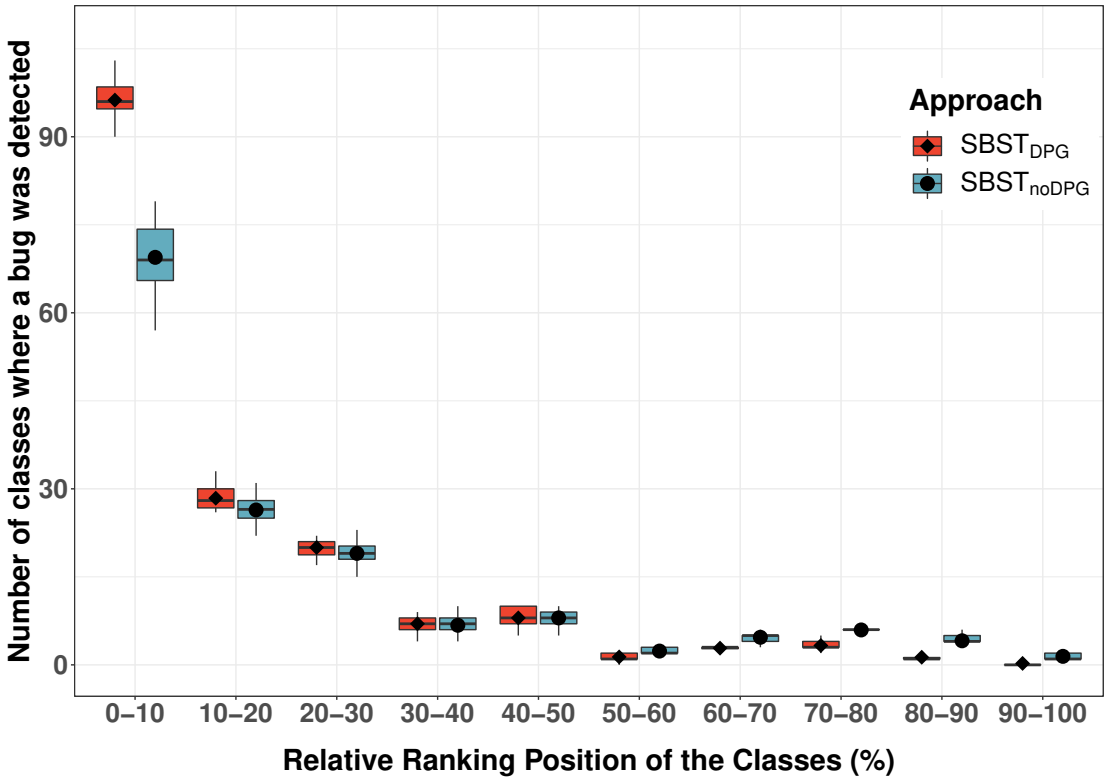

FIGURE 5.9: The number of classes where a bug was detected by the 2 approaches, grouped by the relative ranking positions (%) of the classes in the project at T = $15 * N$ seconds

TABLE 5.2: Summary of the bug detecting results grouped by the relative ranking position (%) of the classes in the project at T = $15 * N$ seconds.

Rank (%)	# Buggy Classes	Avg. Time Budget	Mean number of classes where a bug was found	
			SBST$_{DPG}$	SBST$_{noDPG}$
0 - 10	266	66.61	**96.25**	69.45
10 - 20	72	20.76	**28.40**	26.40
20 - 30	63	16.43	**20.00**	19.00
30 - 40	25	16.00	**7.00**	6.75
40 - 50	26	16.00	**8.00**	**8.00**
50 - 60	13	2.00	1.35	**2.35**
60 - 70	16	2.00	2.85	**4.70**
70 - 80	12	2.00	3.30	**5.95**
80 - 90	13	2.00	1.30	**4.10**
90 - 100	5	2.00	0.25	**1.45**

budget of 15 seconds per class. Relative ranking position is the normalised rank of the respective class as described in Algorithm 1.

We observe that when the buggy classes are correctly ranked at the top by Schwa, and allocated more time by BADS, the performance of SBST$_{DPG}$ is significantly better than

the baseline method. More than half of the buggy classes (52%) are ranked in the top 10% of the project by Schwa, as shown in Table 5.2, and allocated 66.61 seconds of time budget on average by BADS. Around 36% of the buggy classes are ranked in the 10-50% of the projects. BADS employs an exponential function to largely favour a smaller number of highly likely to be defective classes and allocates an adequate amount of time to the moderately defective classes.

Only 12% of the buggy classes are ranked below the first half of the project. BADS assumes not all classes in a project are defective and follows the 2-Tier approach to optimise the budget allocation for the project. Thus, all the classes in the *second tier* which contains the classes that are ranked as less likely to be buggy, get a very small time budget (2 seconds). Unsurprisingly, SBST_{noDPG} detected more bugs out of these 59 buggy classes than SBST_{DPG}. This indicates that the defect predictor's accuracy is key to the better performance of SBST_{DPG} and there is potential to improve our approach further.

For completeness, we also measure and present the number of true positives, false negatives, and recall of Schwa. Based on the 0.5 threshold, i.e., if the defect score is greater than or equal to 0.5 then the class is buggy and it is non-buggy if the defect score is less than 0.5, Schwa labels 436 buggy classes correctly (true positives) and mislabels 75 buggy classes (false negatives). Hence, Schwa achieves a recall of 85%.

The defect predictor (i.e., Schwa) and BADS modules add an overhead to SBST_{DPG}. While this overhead is accounted in the time budget allocation in SBST_{DPG}, we also report the time spent by the defect predictor and BADS modules together. Schwa and BADS spent 0.68 seconds per class on average (standard deviation = 0.4 seconds), which translates to a 4.53% and 2.27% overhead in 15 and 30 seconds per class time budgets respectively. Therefore, this shows the overhead introduced by Schwa and BADS in SBST_{DPG} is very small and negligible. The distribution of time spent by Schwa and BADS is reported in Appendix A as a histogram.

RQ2. Does SBST_{DPG} detect more unique bugs?

To investigate how our approach performs against each bug, we present an overview of the success rates for each SBST method at total time budget of 15 seconds per class in

Table 5.4. Success rate is the ratio of runs where the bug was detected. Due to space limitation, we omit the entries for bugs where none of the approaches were able to detect the bug. We also highlight the bugs that were detected by only one approach. As can be seen from Table 5.4, our approach outperforms the benchmark in terms of the success rates for most of the bugs.

TABLE 5.3: Summary of the bug detecting results at T = $15 * N$.

	Bugs detected	Unique bugs	Bugs detected in every run	Bugs detected more often
SBST$_{DPG}$	**236**	**35**	**84**	**127**
SBST$_{noDPG}$	215	14	76	47

This observation can be confirmed with the summary of the results which we report in Table 5.3. What is particularly interesting to observe from the more granular representation of the results in Table 5.4 is the high number of bugs where our approach has 100% success rate, which means that SBST$_{DPG}$ detects the respective bugs in all the runs. This is an indication of the robustness of our approach.

Certain bugs are harder to detect than others. Out of the 20 runs for each SBST approach, if a bug is only detected by one of the approaches, we call it a unique bug. The reason why we pay special attention to unique bugs is because they are an indication of the ability of the testing technique to detect what cannot be detected otherwise in the given time budget, which is an important strength [161]. SBST$_{DPG}$ detected 236 bugs altogether, which is 54.38% of the total bugs, whereas SBST$_{noDPG}$ detected only 215 (49.54%) bugs. SBST$_{DPG}$ detected 35 unique bugs that SBST$_{noDPG}$ could not detect in any of the runs. On the other hand, SBST$_{noDPG}$ detected only 14 such unique bugs. 30 out of these 35 bugs have buggy classes ranked in the top 10% of the project by Schwa, and the other 5 bugs in 10-50% of the project. We observe similar results at total time budget of 30 seconds per class as well, where SBST$_{DPG}$ detected 32 unique bugs, while SBST$_{noDPG}$ was only able to detect 13 unique bugs.

SBST$_{DPG}$ detected 127 bugs more times than SBST$_{noDPG}$, while for SBST$_{noDPG}$, this is only 47. 92 out of these 127 bugs have buggy classes ranked in the top 10% of the project and the other 35 bugs in 10-50% of the project.

If we consider a bug as detected only if all the runs by an approach detect the bug (success rate = 1.00), then the number of bugs detected by SBST$_{DPG}$ and SBST$_{noDPG}$

become 84 and 76. There are 27 bugs which only SBST$_{DPG}$ detected them in all of the runs.

> In summary, SBST$_{DPG}$ detects 35 more unique bugs compared to the benchmark approach. Furthermore, it detects a large number of bugs more frequently than the baseline. Thus, this suggests that the superior performance of SBST$_{DPG}$ is supported by both its capability of detecting new bugs which are not detected by the baseline and the robustness of the approach.

We pick the Math-94 bug from the motivating example in Section 5.2 and investigate the tests generated by the two approaches. SBST$_{noDPG}$ generated 30.75 test cases on average that cover the `true` branch of the `if` condition at line 412, yet it was not able to detect the bug in any of the runs. Schwa ranked Math-94 bug in the top 10% of the project and BADS allocated 37 seconds time budget to the search. Then, SBST$_{DPG}$ generated 49.8 test cases on average that cover the said branch. As a result, it was able to detect the bug in 7 runs out of 20. Allocating a higher time budget increases the likelihood of detecting the bug since it allows the search method to explore and exploit the search space extensively to find the test inputs that can detect the bug.

We also pick Time-8 bug and investigate the tests generated by the two approaches. Figure 5.10 shows the buggy code snippet and the applied patch for `DateTimeZone` class from Time-8. The `forOffsetHoursMinutes` method takes two integer inputs `hoursOffset` and `minutesOffset`, and returns the `DateTimeZone` object for the offset specified by the two inputs. If the method `forOffsetHoursMinutes` is called with the inputs `hoursOffset=0` and `minutesOffset=-15`, then it is expected to return a `DateTimeZone` object for the offset $-00:15$. However, the `if` condition at line 279 is evaluated to `true` and the method throws an `IllegalArgumentException` instead.

To detect this bug, a test case has to execute the `if` conditions at lines 273 and 276 to `false`; that is `hoursOffset` $\neq 0$ or `minutesOffset` $\neq 0$ and `hoursOffset` $\in [-23, 23]$, and then it has to execute the `if` condition at line 279 to `true` with a `minutesOffset` $\in [-59, -1]$. Moreover, there is a new condition introduced at line 282 in the fixed code to check if the `hoursOffset` is positive when the `minutesOffset` is negative (see Figure 5.10). Thus, this adds another constraint to the possible test inputs that can detect the bug, which is `hoursOffset` ≤ 0. Therefore, it is evident that it is hard not

only to find the right test inputs to detect the bug, but also to find test inputs to at least cover the buggy code.

As it was the case in Math-94, just covering the buggy code (`true` branch of the `if` condition at line 279) is not sufficient to detect the Time-8 bug. For an example, test inputs `hoursOffset=-4` and `minutesOffset=-150` cover the buggy code, however they cannot detect the bug. Therefore, the search method needs more resources to generate more test cases that cover the buggy code such that it eventually finds the right test cases that can detect the bug.

```
272 272     public static DateTimeZone forOffsetHoursMinutes(int hoursOffset,
                    int minutesOffset) throws IllegalArgumentException {
273 273         if (hoursOffset == 0 && minutesOffset == 0) {
274 274             return DateTimeZone.UTC;
275 275         }
276 276         if (hoursOffset < -23 || hoursOffset > 23) {
277 277             throw new IllegalArgumentException("Hours out of range: " +
                        hoursOffset);
278 278         }
279     -       if (minutesOffset < 0 || minutesOffset > 59) {
    279 +       if (minutesOffset < -59 || minutesOffset > 59) {
280 280             throw new IllegalArgumentException("Minutes out of range: " +
                        minutesOffset);
281 281         }
    282 +       if (hoursOffset > 0 && minutesOffset < 0) {
    283 +           throw new IllegalArgumentException("Positive hours must not
                        have negative minutes: " + minutesOffset);
    284 +       }
282 285         int offset = 0;
                ...
295 298     }
```

FIGURE 5.10: Buggy code and patch from Time-8 bug

Our investigation into the tests generated by the two approaches shows that the baseline, SBST_{noDPG}, covered the buggy code in 90% of the runs. SBST_{noDPG} generated 25.78 test cases on average that cover the buggy code and it was able to detect the bug in 14 runs out of 20. Whereas, SBST_{DPG} allocated 75 seconds time budget to the search as Schwa ranked the bug in the top 10% of the project and generated 109.8 test cases on average that cover the buggy code. As a result, it was able to detect the bug in all of the runs (success rate = 1.00). Therefore, this again confirms the importance of focusing the search more into the buggy classes to increase the likelihood of detecting the bug.

As outlined in Section 5.4.1.3, we configure DynaMOSA to generate more than one test case for each target in the CUT, retain all these test cases and disable test suite minimisation. By doing this, we expect to compromise the test suite size in order to maximise the bug detection of SBST. To investigate the benefit of configuring DynaMOSA in this way, we also run the same set of experiments using DynaMOSA with test suite minimisation and equal budget allocation, SBST_O. We compare its performance against SBST_{noDPG}. SBST_O detects 85.75 and 93.45 bugs on average at total time budget of 15 and 30 seconds per class. SBST_{noDPG} outperforms SBST_O with an average improvement of 48.2 (+56.2%) and 73.45 (+78.6%) more bugs in each case, which are statistically significant according to the Mann-Whitney U-Test (p-value < 0.0001) with a large effect size ($\widehat{A}_{12} = 1.00$). More details of the comparison are reported in Appendix A.

5.5 Threats to Validity

In addition to the validity threats discussed in Section 4.6, we discuss the following threat that is specific to this study.

Internal Validity. We employ an exponential function to allocate time budgets for classes based on the defect scores. As opposed to an exponential allocation, a direct mapping (i.e., linear budget allocation) would have been simple and straight-forward. However, as described in Section 5.3.2.1, there are only a few number of classes which are actually buggy (i.e., highly likely to be defective) and they need to be allocated more time budget to maximise the bug detection of the test generation tool. A linear allocation approach is not able to largely favour these small number of classes like the exponential allocation approach does.

5.6 Summary

We introduce defect prediction guided SBST (SBST_{DPG}) that combines class level defect prediction and search-based software testing to efficiently detect bugs in a resource constrained environment. SBST_{DPG} employs a budget allocation algorithm, budget allocation based on defect scores (BADS), to allocate time budgets for classes based on their likelihood of defectiveness. We validate our approach against 434 real bugs

from Defects4J dataset. Our experimental evaluation demonstrates that in a resource constrained environment, when given a tight time budget, SBST_{DPG} is significantly more efficient than the state-of-the-art approach with a large effect size. In particular, SBST_{DPG} detects 13.1% more bugs on average compared to the state-of-the-art SBST approach when they are given a tight time budget of 15 seconds per class. Further analysis of the results finds that the superior performance of SBST_{DPG} is supported by its ability to detect more unique bugs which otherwise remain undetected.

TABLE 5.4: Success rate for each method at $15 * N$ total time budget. Bug IDs that were detected by only one approach are highlighted with different colours; $SBST_{DPG}$ and $SBST_{noDPG}$.

Bug ID	$SBST_{DPG}$	$SBST_{noDPG}$	Bug ID	$SBST_{DPG}$	$SBST_{noDPG}$
Lang-1	1	0.45	Math-3	0.55	1
Lang-4	0.9	1	Math-4	1	1
Lang-5	0	0.2	Math-5	0.45	0.95
Lang-7	1	1	Math-6	1	1
Lang-8	0.1	0.1	Math-9	0.7	0.6
Lang-9	0.95	1	Math-10	0.1	0
Lang-10	0.95	0.8	Math-11	0.95	1
Lang-11	0.8	0.95	Math-14	1	1
Lang-12	0.2	0.8	Math-16	0	0.05
Lang-14	0.05	0	Math-21	0.05	0.45
Lang-17	0.05	0	Math-22	1	1
Lang-18	0.5	0.3	Math-23	0.95	0.8
Lang-19	0.05	0.7	Math-24	0.9	0.85
Lang-20	0.8	0.4	Math-25	0.1	0
Lang-21	0.1	0.1	Math-26	1	1
Lang-22	0.55	0.8	Math-27	0.6	0.65
Lang-23	1	0.95	Math-28	0.05	0
Lang-27	0.8	0.75	Math-29	0.9	1
Lang-28	0.05	0.05	Math-32	1	1
Lang-32	1	1	Math-33	0.45	0.35
Lang-33	1	1	Math-35	1	1
Lang-34	1	0.9	Math-36	0.2	0.1
Lang-35	1	0.3	Math-37	1	1
Lang-36	1	1	Math-40	1	0.95
Lang-37	0.65	0.2	Math-41	0.25	0.4
Lang-39	1	0.95	Math-42	0.95	0.95
Lang-41	0.7	1	Math-43	0.45	0.55
Lang-44	0.85	0.65	Math-45	0	0.3
Lang-45	1	1	Math-46	1	1
Lang-46	0.5	1	Math-47	1	0.95
Lang-47	0.95	0.9	Math-48	0.65	0.75
Lang-49	0.55	0.4	Math-49	0.8	0.75
Lang-50	0.3	0.3	Math-50	0.75	0.3
Lang-51	0.1	0.05	Math-51	0.35	0.25
Lang-52	1	1	Math-52	0.65	0.6
Lang-53	0.3	0.15	Math-53	1	1
Lang-54	0.05	0.05	Math-55	1	1
Lang-55	0.05	0	Math-56	1	0.9
Lang-57	1	1	Math-59	1	1
Lang-58	0	0.05	Math-60	0.95	0.95
Lang-59	1	0.95	Math-61	1	1
Lang-60	0.75	0.3	Math-63	1	0.4
Lang-61	1	0.25	Math-64	0.05	0
Lang-65	1	0.95	Math-65	0.25	0.25
Math-1	1	1	Math-66	1	1
Math-2	0	0.1	Math-67	1	1

TABLE 5.4: (continued)

Bug ID	$SBST_{DPG}$	$SBST_{noDPG}$	Bug ID	$SBST_{DPG}$	$SBST_{noDPG}$
Math-68	1	1	Time-15	0.4	0.3
Math-70	1	1	Time-16	0.15	0
Math-71	0.6	0.35	Time-17	1	0.55
Math-72	0.5	0.45	Time-22	0	0.25
Math-73	0.75	1	Time-23	0	0.2
Math-75	1	0.9	Time-24	0	0.45
Math-76	0.15	0.05	Time-26	0.1	0.05
Math-77	1	1	Time-27	0.15	0.5
Math-78	0.6	0.6	Chart-1	0.2	0.05
Math-79	0.15	0.05	Chart-2	0.05	0
Math-80	0.3	0	Chart-3	0.9	0.15
Math-81	0.15	0	Chart-4	0.85	0.3
Math-83	0.9	1	Chart-5	0.35	1
Math-84	0.15	0	Chart-6	0.8	1
Math-85	1	1	Chart-7	0.3	0.25
Math-86	0.95	0.85	Chart-8	1	1
Math-87	0.95	1	Chart-10	1	1
Math-88	0.75	0.7	Chart-11	0.2	1
Math-89	1	1	Chart-12	0.9	0.5
Math-90	1	1	Chart-13	0.9	0.2
Math-92	1	1	Chart-14	1	1
Math-93	0.35	0.25	Chart-15	1	0.9
Math-94	0.35	0	Chart-16	1	1
Math-95	1	1	Chart-17	1	1
Math-96	1	1	Chart-18	1	1
Math-97	1	1	Chart-19	1	0.15
Math-98	1	0.85	Chart-20	0.5	0.1
Math-100	1	1	Chart-21	0.55	0.05
Math-101	0.2	1	Chart-22	1	1
Math-102	0.75	0.5	Chart-23	1	1
Math-103	1	1	Chart-24	0	1
Math-104	0.5	0.4	Mockito-2	1	1
Math-105	1	1	Mockito-17	1	1
Math-106	0.15	0	Mockito-29	0.85	0.95
Time-1	1	1	Mockito-35	1	1
Time-2	0.85	1	Closure-6	0.05	0
Time-3	0.15	0.05	Closure-7	0.35	0.1
Time-4	0	0.3	Closure-9	0.6	0.15
Time-5	1	1	Closure-12	0.3	0.1
Time-6	1	0.8	Closure-19	0	0.1
Time-7	0.15	0	Closure-21	0.9	0.35
Time-8	1	0.7	Closure-22	0.5	0.5
Time-9	1	1	Closure-26	0.5	0.4
Time-10	0.1	0.1	Closure-27	0.25	0.1
Time-11	1	1	Closure-28	1	1
Time-12	1	0.55	Closure-30	1	0.95
Time-13	0.5	0.05	Closure-33	1	0.5
Time-14	0	0.95	Closure-34	0.05	0

TABLE 5.4: (continued)

Bug ID	$SBST_{DPG}$	$SBST_{noDPG}$	Bug ID	$SBST_{DPG}$	$SBST_{noDPG}$
Closure-39	1	0.6	Closure-116	0.2	0.1
Closure-41	0.1	0	Closure-117	0.4	0.05
Closure-43	0.05	0	Closure-119	0.25	0
Closure-46	1	1	Closure-120	0.2	0.1
Closure-48	0.1	0	Closure-121	0.55	0.2
Closure-49	0.45	0.5	Closure-122	0.05	0
Closure-52	0.4	0.1	Closure-123	0.15	0.1
Closure-54	1	0.8	Closure-125	0.45	0
Closure-56	0.95	1	Closure-128	0.15	0.1
Closure-60	0.1	0	Closure-129	0.2	0.05
Closure-65	0.9	0.45	Closure-131	0.15	0.9
Closure-72	0.2	0.3	Closure-137	0.95	1
Closure-73	1	1	Closure-139	0.15	0.05
Closure-77	0.7	0.25	Closure-140	0.85	0.25
Closure-78	0.05	0	Closure-141	0.3	0
Closure-79	1	0.85	Closure-144	0.3	0.1
Closure-80	0.2	0	Closure-146	0.15	0
Closure-81	0.35	0	Closure-150	0.45	0.1
Closure-82	1	1	Closure-151	1	1
Closure-86	0.15	0	Closure-160	0.55	0.05
Closure-89	0.05	0	Closure-164	0.35	0.45
Closure-91	0.15	0	Closure-165	0.95	0.8
Closure-94	0.25	0	Closure-167	0.35	0
Closure-104	0.95	0.5	Closure-169	0	0.05
Closure-106	1	0.95	Closure-170	0.2	0.2
Closure-108	0.8	0.2	Closure-171	0.9	0.05
Closure-110	0.95	1	Closure-172	0.65	0.15
Closure-112	0.1	0	Closure-173	1	0.5
Closure-113	0.25	0.05	Closure-174	1	1
Closure-114	0	0.1	Closure-175	0.75	0.15
Closure-115	0.3	0.25	Closure-176	0.1	0.1

Chapter 6

Impact of Defect Predictor Imprecision

6.1 Introduction

Often, the predictions produced by defect predictors are not perfectly accurate. In fact, the defect predictors proposed in previous work have wavering performance. For example, in their systematic literature review, Hall et al. [49] reported defect predictors having recall in the range 25% to 85% and precision in the range 5% to 95%. A lower recall and precision can significantly hamper the benefits of defect predictors for the developers who usually manually inspect or test the predicted buggy code to find bugs. Poor recall means there are higher false negatives in the predictions. This can lead the developers to completely miss bugs, since they are not likely to inspect the code with non-buggy label which usually constitute a large portion of the code base. Lower precision means there are higher false positives in the predictions. False positives cause developers to waste their precious time on inspecting non-buggy code, which eventually leads to losing trust on the defect predictor [39, 41]. In the context of developers using defect predictions, it is important the developers be informed of precise buggy locations (i.e., higher precision) at the expense of missing out bugs (i.e., lower recall).

The impact of false negatives and false positives on guiding search-based software testing (SBST) techniques has not been studied in previous studies. For example, in Chapter 5, we used a single defect predictor to guide a time budget allocation approach for SBST

and demonstrated the improved bug detection performance of the approach. The defect predictor used in the study have a relatively high performance, i.e., 85% recall. We cannot expect the same defect predictor to perform well when applied on another project with different characteristics and at different targeted prediction level (e.g., method level) [37]. A defect predictor with lower recall (i.e., higher false negatives) may cause the SBST technique to not generate tests for buggy areas in code. This could lead the SBST technique to miss bugs when it is guided by defect predictions. On the other hand, poor precision (i.e., higher false positives) may not be as important for the SBST technique because searching for tests to cover false positive areas in code may not be a significant burden to a machine compared to a human manually inspecting code with false alarms. Given such wavering performance of defect predictors and unanswered questions about their impact on guiding SBST, we aim at systematically investigating the impact of defect prediction imprecision on the bug detection performance of SBST. Therefore, in this chapter, we aim to achieve the following research objective;

> RO2: Understand the impact of imprecision in defect prediction for guiding search-based software testing.

To achieve this research objective, we design a study to systematically investigate the variations in bug detection effectiveness of SBST against imprecise defect predictions at various levels. Through experimental evaluation, we demonstrate that the recall of the defect predictor has a significant impact on the bug detection performance of SBST and the impact of precision is not practically significant. According to Zimmermann et al. [48], defect predictors with recall and precision higher than 75% are considered acceptable defect predictors. We simulate defect predictors for different configurations of recall and precision in the range 75% to 100%. In this study, we use fine-grained defect prediction, i.e., method level, hence the SBST technique receive further narrowed down locations of the bugs compared to coarse-grained defect predictions. We use the state-of-the-art SBST technique, DynaMOSA, to incorporate buggy methods predictions in the search process. DynaMOSA guided by defect prediction directs the search for tests towards the likely buggy methods in the code. We experimentally evaluate the bug detection effectiveness of SBST guided by defect prediction using 420 labelled bugs from Defects4J dataset as benchmark subjects.

6.2 Methodology

Our aim is to understand how the defect prediction imprecision impacts the bug detection performance of SBST. To this end, we design a study that addresses the following research question (RQ):

RQ: What is the impact of the imprecision of defect prediction on bug detection performance of SBST?

To address this research question, we measure the effectiveness of SBST in terms of detecting bugs when using defect predictors with different levels of imprecision. We use the state-of-the-art SBST technique, DynaMOSA [14], and incorporate predictions about buggy methods in order to guide the search for test cases towards likely buggy methods (see Section 6.2.2), which we refer as *SBST guided by DP* throughout the thesis. Fine-grained defect predictions such as method level is chosen so that the location of the bug is narrowed down better than coarse-grained defect predictions such as class level. Hence the defect predictors at method level provide additional information to the SBST technique such that it can further narrow down the search for test cases to likely buggy methods.

We measure defect predictor imprecision using recall and precision. Recall and precision have been widely used in previous work to measure the predictive power of defect predictors [49, 50]. We consider a defect predictor with either recall or precision less than 75% is not an acceptable defect predictor, as recommended by Zimmermann et al. [48]. Hence, we simulate defect predictors for varying levels of recall and precision in the range 75% to 100% (see Section 6.2.1) and measure the impact on the bug detection performance of SBST by the prediction imprecision.

6.2.1 Defect Prediction Simulation

To measure the bug detection performance of SBST against the imprecision of defect predictions, we simulate defect predictor outcomes at various levels of performance in the range 75% and 100% for both precision and recall. We do not use real defect predictors in our study because their performance cannot be controlled to systematically investigate

the impact of imprecision of defect prediction. Recall is the rate of the defect predictor identifying buggy methods. It is calculated as in Equation. (6.1), where tp is the number of true positives, i.e., number of buggy methods that are correctly classified, and fn is the number of false negatives, i.e., number of buggy methods that are incorrectly classified.

$$\text{recall} = \frac{tp}{tp + fn} \tag{6.1}$$

Precision is the rate of the correct buggy methods labelled by the defect predictor. It can be calculated as in Equation (6.2), where fp is the number of false positives, i.e., number of non-buggy methods that are incorrectly classified as buggy methods.

$$\text{precision} = \frac{tp}{tp + fp} \tag{6.2}$$

We simulate defect predictions from 75% to 100% recall in 5% steps, with 75% and 100% precision. Thus, there are altogether 12 defect predictor configurations, with the following values of (precision, recall): (75%, 75%), (75%, 80%), (75%, 85%), (75%, 90%), (75%, 95%), (75, 100%), (100%, 75%), (100%, 80%), (100%, 85%), (100%, 90%), (100%, 95%), (100, 100%). Our preliminary experiments suggest that the bug detection performance of SBST guided by DP changes by a small margin when the precision is changed from 100% to 75%, while keeping the recall unchanged. On the other hand, the bug detection performance of SBST guided by DP changes by a large margin when only the recall is changed from 100% to 75%. Hence, we decide to consider only the values of 75% and 100% for precision, while recall is sampled at 5% steps. For completeness, we report the Matthews correlation coefficient (MCC) of each defect prediction configuration in Appendix B.

The output of the simulated defect predictor is binary, i.e., method is buggy or not buggy, similar to most of the existing defect predictors. Some of the existing defect predictors output the likelihood of the components being buggy or the ranking of the components according to their likelihood of being buggy. Since we employ a theoretical defect predictor and not a specific one, we resort to the generic defect predictor, which is the one that gives a binary classification.

Algorithm 2 Defect Predictor Simulation

Input: r, p ▷ recall and precision
$M = \{m_1, \ldots, m_k\}$ ▷ ground truth

1: **procedure** SIMULATEDEFECTPREDICTOR
2: $d \leftarrow \text{COUNT}(m_i)$ for $m_i \in M$ s.t. $m_i = 1$
3: $nd \leftarrow |M| - d$
4: $M_b \leftarrow \{i \,|\, \forall i \in [1, k] \wedge m_i = 1\}$
5: $M_n \leftarrow \{i \,|\, \forall i \in [1, k] \wedge m_i = 0\}$
6: $tp \leftarrow d * r$
7: $fp \leftarrow tp * (1 - p)/p$
8: $C_b \leftarrow \text{RANDOMCHOICE}(M_b, tp) \cup \text{RANDOMCHOICE}(M_n, fp)$
9: $C \leftarrow \{c_i = 1 \,|\, \forall i \in [1, k] \wedge i \in C_b,\ c_i = 0 \,|\, \forall i \in [1, k] \wedge i \notin C_b\}$
10: $\text{RETURN}(C)$

Algorithm 2 illustrates the steps of simulating the defect predictor outputs for a given recall and precision combination. The procedure SIMULATEDEFECTPREDICTOR receives the set of methods in the project with the ground truth labels for their defectiveness, $M = \{m_1, \ldots, m_k\}$, where

$$m_i = \begin{cases} 1 & \text{if method with index } i \text{ is buggy} \\ 0 & \text{otherwise} \end{cases}$$

and outputs a set of labels for each method in the project, $C = \{c_1, \ldots, c_k\}$, where

$$c_i = \begin{cases} 1 & \text{if method with index } i \text{ is predicted buggy} \\ 0 & \text{otherwise} \end{cases}$$

First, it calculates the number of buggy (d) and non-buggy methods (nd) in the project (lines 2-3 in Algorithm 2). Next, it finds the set of indices of all the buggy (M_b) and non-buggy methods (M_n) in the project (lines 4-5). The true positives (tp) and false positives (fp) are then calculated for the given recall (r) and precision (p) (lines 6-7). The $\text{RANDOMCHOICE}(M_x, n)$ procedure returns n number of randomly selected methods from the set M_x, where $x \in \{b, n\}$. C_b is assigned a set of randomly picked tp number of buggy and fp number of non-buggy method indices (line 8). C_b is the set of buggy method indices as classified by the simulated defect predictor. The output is the set $C = \{c_1, \ldots, c_k\}$, where $c_i = 1$ if the method with index i is labelled as buggy and $c_i = 0$ if the method with index i is labelled as not buggy (line 9).

6.2.2 Search-Based Software Testing Guided By Defect Prediction

SBST techniques search for test cases to cover a given set of targets of a class by spending an allocated time budget. The set of coverage targets usually represents all of the code in the class under test (CUT). The buggy code can be just a few lines of code in the class and contained within a method. It is likely to be ineffective in terms detecting bugs to spend the allocated time budget searching for tests that exercise the non-buggy code which also constitutes a large part of the CUT. In this study, we simulate defect predictions targeted at a more granular level, i.e., method level. This further narrows down the location of the bug such that the SBST technique can differentiate methods in a class based on their defectiveness. This allows us to use these buggy method predictions to guide the search process in SBST techniques more towards the likely buggy areas within the CUT to increase the chances of detecting bugs.

We incorporate buggy method predictions in DynaMOSA [14], the state-of-the-art SBST technique, to guide the search for test cases towards likely buggy methods. As described in Section 2.2.4, DynaMOSA tackles the test generation problem as a many objective optimisation problem, where each coverage target in the program, e.g., branch and statement, is an objective to optimise. It is more effective at achieving high branch, statement and strong mutation coverage than previously proposed SBST techniques ([15, 51, 89]) [14]. In order to detect a bug, it is necessary to reach the buggy code according to the reachability, infection and propagation (RIP) model [162]. Therefore, the best choice among existing SBST techniques is the one that has higher code coverage. In the next sections, we refer to the DynaMOSA approach guided by the defect predictor as *SBST guided by DP*. SBST guided by DP is presented in Algorithm 3. It shares the same search steps and genetic operators as DynaMOSA, except for the updated steps shown in blue colour in Algorithm 3.

SBST guided by DP receives as input a class with methods labelled as buggy or non-buggy, which are labels that can be obtained using existing defect predictors [31, 32]. In our study, SBST guided by DP receives these labels from defect predictor simulations (Section 6.2.1).

SBST guided by DP devotes all the search resources to find tests that cover likely buggy methods, thereby increasing the chances of detecting bugs. Initially, SBST guided by DP filters out the coverage targets that are deemed to not contain buggy methods as

indicated by the defect prediction information, and keeps only targets that contain likely buggy methods (as shown in line 2 of Algorithm 3 and described in Section 6.2.2.1).

In Chapter 5, we showed that DynaMOSA detects significantly more bugs when it was configured to generate more than one test case to cover each of the coverage targets. Following this, SBST guided by DP also generates more than one test case for all the selected buggy targets, hence, further increases the chances of detecting bugs (lines 6, 7, 10 and 11 and described in Section 6.2.2.2).

To generate more than one test case for all the likely buggy targets, SBST guided by DP does not remove a target once it is covered during the search. This is likely to cause SBST guided by DP to miss nontrivial targets in the search and keep on generating tests to cover more trivial targets [51]. To address this, we use a method called balanced test coverage to dynamically disable targets from the search based on their current test coverage and number of independent paths (lines 3 and 13). This ensures that the nontrivial targets have an equal chance of being covered compared to the targets that are easier to cover. Use of this method in SBST guided by DP has a minimal impact on the conclusions of this study and it is more relevant in the proposed approach to achieve RO3 in Chapter 7. Hence, we describe this method in detail in Chapter 7 (see Section 7.3.3).

SBST guided by DP randomly generates a set of test cases that forms the initial population (line 5). Then, it evolves this initial population through creating new test cases via crossover and mutation (line 9), and selecting test cases to the next generation (line 14), until a termination criteria, such as maximum time budget, is met.

6.2.2.1 Filtering Targets with Defect Prediction

A defect predictor classifies the methods of the CUT as buggy or non-buggy, denoted as c_i, where

$$c_i = \begin{cases} 1 & \text{if method with index } i \text{ is predicted as buggy} \\ 0 & \text{otherwise} \end{cases}$$

This information is used to filter out the likely non-buggy targets from the set of all targets U using the classifications given (line 2). Spending the limited search resources

Algorithm 3 SBST Guided By Defect Prediction

Input: ▷
$U = \{u_1, \ldots, u_k\}$ ▷ the set of coverage targets of CUT
$G = \langle N, E \rangle$ ▷ control dependency graph of the CUT
$\phi : E \to U$ ▷ partial map between edges and targets
$C = \{c_1, \ldots, c_m\}$ ▷ the set of defectiveness classifications for methods in the CUT

1: **procedure** SBST
2: $U_B \leftarrow$ FilterTargets(U, C)
3: $L \leftarrow$ IndependentPaths(G) ▷ L is a vector of the number of independent paths for each edge
4: $U^* \leftarrow$ targets in U_B with no control dependencies
5: $P_0 \leftarrow$ RandomPopulation(M) ▷ M is the population size
6: $A \leftarrow$ UpdateArchive$(P_0, \emptyset, U_B)$ ▷ A is the archive
7: $U^* \leftarrow$ UpdateTargets(U^*, G, ϕ, U_B)
8: **for** $r \leftarrow 0$; !terminationCriteria; r++ **do**
9: $Q_r \leftarrow$ GenerateOffspring(P_r)
10: $A \leftarrow$ UpdateArchive(Q_r, A, U_B)
11: $U^* \leftarrow$ UpdateTargets(U^*, G, ϕ, U_B)
12: $R_r \leftarrow P_r \cup Q_r$
13: $U^* \leftarrow$ SwitchOffTargets(U^*, A, L, ϕ)
14: $P_{r+1} \leftarrow$ SelectPopulation(R_r, U^*, M)
15: $T \leftarrow A$ ▷ Update the final test suite T
16: Return(T)

on covering non-buggy targets is likely to be ineffective when it comes to detecting bugs. Filtering out targets that are unlikely to be buggy allows the search to focus on test cases that cover the likely buggy targets (i.e., $\forall u \in U_B$), hence, generating more effective test cases faster than other approaches which search for tests in all the targets in the CUT.

6.2.2.2 Dynamic Selection of Targets and Archiving Tests

At the start of the search, SBST guided by DP selects the set of targets $U^* \subseteq U_B$ that do not have control dependencies (line 4). These are the targets SBST guided by DP can cover without requiring to cover any other targets in the program (described in Section 2.2.4.3). At any given time in the search, it searches for test cases to cover only the targets in U^*.

Once a new population of test cases is generated (lines 5 and 9), the procedure UpdateTargets is executed to update U^* by adding new targets to the search. The procedure

UpdateTargets adds a target $u \in U_B$ to U^* only if the control dependent targets of u are covered as explained in Section 2.2.4.3.

SBST guided by DP maintains an archive of test cases found during the search which cover the selected targets. Once the search finishes, this archive forms the final test suite. Unlike in DynaMOSA, we configure the UpdateTargets procedure to not remove a covered target from U^* and the UpdateArchive procedure (lines 6 and 10) to archive all the test cases that cover the selected targets $u \in U_B$. This way, SBST guided by DP can generate more than one test case for each target $u \in U_B$, hence increasing the bug detection capability of the generated test suites. In Chapter 5, we show that DynaMOSA detects up to 79% more bugs when it was configured to not remove covered targets from the search and retain all the generated tests.

6.3 Analysis of Impact of Defect Prediction Imprecision

We design a set of experiments to evaluate the effectiveness of SBST guided by DP in terms of detecting bugs when using defect predictors with 12 different levels of imprecision as described in Section 6.2.1 (RQ). We use the bugs from the Defects4J dataset as the experimental subjects [136] (see Section 6.3.1.1).

To account for the randomness of the defect prediction simulation algorithm (Algorithm 2), we repeat the simulation runs 5 times for each defect predictor configuration (i.e., recall and precision pair). For each of these simulation runs, we repeat the test generation runs 5 times, to account for the randomness in SBST guided by DP.

Once tests are generated and evaluated for bug detection, we conduct two-way ANOVA test to statistically analyse the effects of recall and precision of the defect predictor on the bug detection effectiveness of SBST guided by DP.

6.3.1 Experimental Settings

6.3.1.1 Experimental Subjects

In our experiments, we further remove 14 bugs from the Defects4J dataset (version 1.5.0) described in Section 4.1. Those are 12 bugs that do not have buggy methods and 2 bugs

for which SBST guided by DP generated uncompilable tests (e.g., method signature is changed in the bug fix). This results in the following 14 bugs that are not part of the experiments: Lang-23, 25, 30, 56, 63, Math-12, 104, Time-11, Chart-23, Closure-15, 28, 83, 111 and Mockito-26 are removed. Thus, we evaluate SBST guided by DP on a total of 420 bugs. The number of bugs from each project is as follows; JFreeChart (25 bugs), Closure Compiler (170 bugs), Apache commons-lang (59 bugs), Apache commons-math (104 bugs), Mockito (37 bugs), and Joda-Time (25 bugs). The reasons to remove each bug are reported in Appendix B.

We calculate the adequate sample size [163] for two-way ANOVA test with power=0.80, alpha=0.05 and medium effect size (f=0.25). The required sample size with these parameters is 212, which is well below our sample size of 420 bugs.

The Defects4J benchmark gives a buggy version and a fixed version of the program for each bug in the dataset. The fixed version is different to the buggy version by the applied patch to fix the bug, which indicates the location of the bug. We label all the methods that are either modified or removed in the bug fix as buggy methods [132].

6.3.1.2 Prototype

DynaMOSA is implemented in the state-of-the-art SBST tool, EvoSuite [3]. For the experimental evaluation, we implement the changes described in Section 6.2.2 for SBST guided by DP. The changes are implemented within EvoSuite version 1.0.7, forked from the GitHub repository [98] on June 18^{th}, 2019. We also implement the defect predictor simulator as described in Section 6.2.1. The prototypes are available to download from here: `https://doi.org/10.6084/m9.figshare.16564146`

6.3.1.3 Parameter Settings

We use the default parameter settings of EvoSuite [89] and DynaMOSA [14] except for the parameters mentioned in the next part of the section. Parameter tuning of SBST techniques is a long and expensive process [160]. According to Arcuri and Fraser [160], EvoSuite with default parameter values performs on par compared to EvoSuite with tuned parameters.

Time Budget: We set 2 minutes as time budget per CUT for test generation. In practice, the time budget allocated for SBST tools depends on the size of the project, frequency of test generation runs and availability of computational resources in the organisation.

Real world projects are usually very large and can have thousands of classes [45]. If an SBST tool runs test generation for 2 minutes per class, then it will take at least 33 hours to finish the task for the whole project.

To address this issue, practitioners can adapt the SBST tools in their continuous integration (CI) systems [159]. However, the introduction of new SBST tools to the CI system should not make the existing processes in the system idle.

Thus, given the limited computational resources available in practice [44] and the expectation of faster feedback cycles from testing in agile development prompt the necessity of frequent test generation runs with limited testing budget. Therefore, we decide that 2 minutes per class is a reasonable time budget in a usual resource constrained environment.

Coverage criteria: Similar to RO1 (see Section 5.4.1.3), we use branch coverage as coverage criterion in SBST guided by DP as it was shown to be the most effective criterion in terms of detecting bugs [24, 43].

Termination criteria: We use only the maximum time budget as the termination criterion. Stopping the search after it covers all the targets is detrimental to bug detection. The search needs to utilise the full time budget to generate as many tests for each target in the CUT in order to increase the chances of detecting bugs. Therefore, we terminate the search for test cases only when the allocated time budget runs out.

Test suite minimisation: We disable test suite minimisation since all the test cases in the archive form the final test suite (see Section 6.2.2.2).

Assertion strategy: Similar to RO1 (see Section 5.4.1.3), we choose all possible assertions as the assertion strategy because the mutation-based assertion filtering can be computationally expensive and can lead to timeouts [23].

6.3.1.4 Experimental Protocol

We run experiments with SBST guided by DP using defect predictors with 12 different levels of imprecision as described in Section 6.2.1. For each bug in the Defects4J dataset, we checkout the buggy version of the project and collect the ground truth labels for the buggy and non-buggy methods. If a method is either modified or removed in the bug fix, we label that method as a buggy method, and non-buggy otherwise [132]. Then, for each of the six projects in the dataset, we combine the ground truth labels from all the bugs respective to each project. For example, for the Apache commons-math project, we combine the labels from all the 104 bugs from that project in the dataset. Then, we simulate defect prediction outcomes for each project using the defect prediction algorithm described in Section 6.2.1.

As described in Chapter 4, we assume an application scenario of generating tests to detect bugs not only limited to regressions, but also the bugs introduced to the code in various times in development. Therefore, we run test generation on the buggy version of the projects. We measure the bug detecting effectiveness of SBST guided by DP only on the Defects4J bugs. Thus, we only run test generation for buggy classes, i.e., classes that are modified in the bug fixes, in the projects.

For each level of defect predictor imprecision, we run test generation with SBST guided by DP 25 times for each bug in the dataset. Consequently, we have to run a total of 12 (levels of defect prediction imprecision) $*$ 25 (repetitions) $*$ 482 (buggy classes) = 144,600 test generations. We collect the test suites generated from each of the 144,600 test generation runs. We determine if the 144,600 generated test suites detect the bugs by using the method described in Section 4.4. Altogether, the experimental evaluation took roughly 180,750 CPU-hours.

6.3.2 Results

We present the results for our research question following the method described in Section 6.3. Our aim is to evaluate the effectiveness of bug detecting performance of SBST guided by DP when using imprecise defect predictors.

RQ. What is the impact of the imprecision of defect prediction on bug detection performance of SBST?

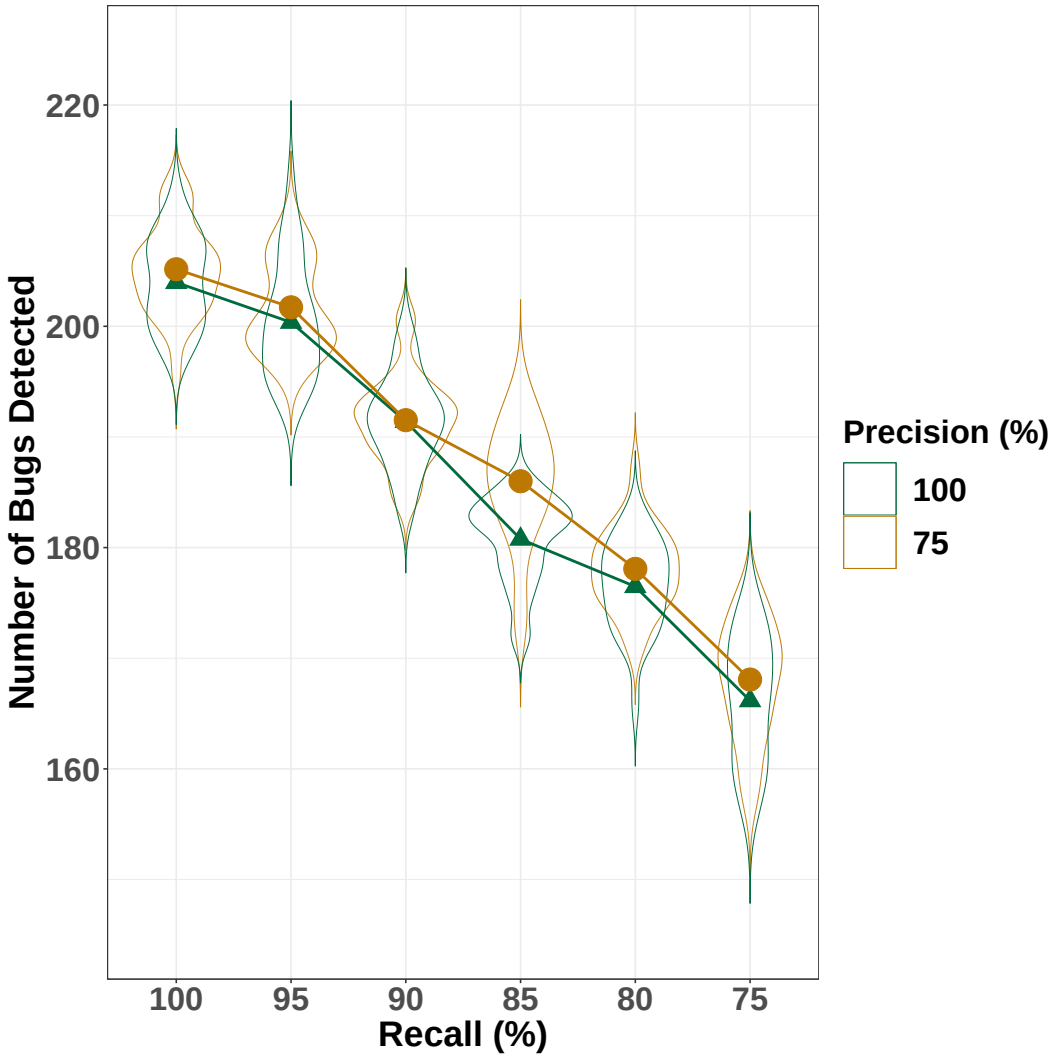

FIGURE 6.1: Distributions of the number of bugs detected by SBST guided by DP as violin plots together with the profile plot of mean number of bugs detected by SBST guided by DP for each combination of the groups of recall and precision.

Figure 6.1 shows the distributions of the number of bugs detected by SBST guided by DP as violin plots and the profile plot of the mean number of bugs detected by SBST guided by DP for each combination of the factors of six recalls and two precisions. The two lines in our profile plot run almost parallel to each other, i.e., the two lines do not cross each other at any point. This means that there is no observable interaction effect between recall and precision.

The two lines descent steeply from recall 100% to 75%. This shows that recall has an effect on number of bugs detected by SBST guided by DP. In particular, bug detection

effectiveness decreases as recall decreases.

The precision=75% line closely follows the precision=100% line while staying slightly above the latter, except at recall=85%, where there is a considerable gap between the two. We can soon see if this difference is significant from the two-way ANOVA test results. We report the mean and median number of bugs detected by SBST guided by DP in Appendix B.

To statistically test the effect of each of the metrics, recall and precision, and their interaction on the number of bugs detected by SBST guided by DP, we conduct the two-way ANOVA test. Prior to conducting two-way ANOVA test, we have to make sure that our data holds the following assumptions of the test.

1. The dependent variable should approximately follow a normal distribution for all the combinations of groups of the two independent variables.
2. Homogeneity of variances exists for all the combinations of groups of the two independent variables.

To check the first assumption, we conduct the Kolmogorov-Smirnov test [164] for normality of the distributions ($\alpha = 0.05$) of the number of bugs detected for each combination of the groups of recall and precision. Based on the results of the tests, we cannot reject our null hypothesis (p-values ≥ 0.131), i.e., H_0 = the number of bugs detected is normally distributed, hence we assume all the samples come from a normal distribution (i.e., H_0 is true). More details of the results of the normality tests are reported in Appendix B.

To check the second assumption, we conduct the Bartlett's test for homogeneity of variances ($\alpha = 0.05$) in each combination of the groups of recall and precision. Based on the results of the test, we cannot reject our null hypothesis (p-value = 0.305), i.e., H_0 = variances of the number of bugs detected are equal across all combinations of the groups, hence we assume the variances are equal across all samples (i.e., H_0 is true).

Table 6.1 shows the summary of the two-way ANOVA test results. According to the two-way ANOVA test, recall and precision explain a significant amount of variation in number of bugs detected by SBST guided by DP (p-values < 0.001). The test also indicates that we cannot reject the null hypothesis that there is no interaction effect between recall and precision on number of bugs detected (p-value = 0.105). That means

	Df	Sum Sq	Mean Sq	F value	p-value
Recall	5	51341	10268	497.42	<0.001
Precision	1	273	273	13.21	<0.001
Recall:Precision	5	190	38	1.84	0.105
Residuals	288	5945	21		

TABLE 6.1: Summary of the two-way ANOVA test results. Df = degrees of freedom, Sum Sq = sum of squares and Mean sq = mean sum of squares.

we can assume the effect of recall on number of bugs detected does not depend on the effect of precision, and vice versa.

To check if the observed differences among the groups are of practical significance, we measure the epsilon squared effect size ($\widehat{\epsilon^2}$) [147] of the variations in number of bugs detected with respect to recall and precision. We find that the effect of recall on bug detection effectiveness is large with an effect size of 0.89, while the effect of precision is very small ($\widehat{\epsilon^2} = 0.004$) [165], which can be seen from the overlapping distributions in the violin plots in Figure 6.1 as well.

To further analyse which groups are significantly different from each other, we conduct the Tukey's Honestly-Significant-Difference test [148]. The Tukey post-hoc test shows that the number bugs detected by SBST guided by DP is significantly different between each of the six levels of recall (p-values < 0.002). The Cohen's d effect sizes of the differences between the groups of recall range from medium ($d = 0.77$ for recall 95% and 100%) to large ($d \geq 1.33$ for all other pairs of groups). The Tukey post-hoc test results of all possible pairs of groups can be found in Appendix B.

In summary, the false negatives of the defect predictor has a significant impact on the bug detection performance of SBST. In particular, when the recall of the defect predictor decreases, the bug detection effectiveness significantly decreases with a large effect size. On the other hand, we conclude that there is no meaningful practical effect of precision on the bug detection performance of SBST, as indicated by a very small effect size.

6.3.2.1 Sensitivity to the Recall of the Defect Predictor

As shown in Figure 6.1, SBST guided by DP detects less number of bugs when using defect predictors with a lower recall compared to using one with a higher recall. In particular, SBST guided by DP detects 7.5 less bugs and misses test generation for 15 bugs on average (out of 420) when the recall decreases by 5% in our experiments. SBST guided by DP completely trusts the defect predictor and only generates tests for classes having at least one method predicted as buggy (e.g., true positive). The number of true positives by the defect predictor decreases when the recall decreases. This results in SBST guided by DP generating tests for a fewer number of classes as the recall decreases, hence detecting less number of bugs when recall drops from 100% to 75%.

We identify this as a weakness of SBST when using defect predictions. To mitigate this, SBST techniques have to take potential false negatives in the predictions into account. The current approach fully exploits the buggy methods predicted by the defect predictor. We recommend that SBST techniques require to explore the likely non-buggy methods while prioritising the exploitation of likely buggy methods. One way to do this is to always generate tests for methods that are predicted buggy, while also generating tests for predicted non-buggy methods at least with a minimum probability. This way the SBST technique gets a chance to search for tests in incorrectly classified buggy methods (when recall <100%), while also giving higher priority to methods that are predicted buggy by the defect predictor.

6.3.2.2 Number of Buggy Methods

As we discussed previously, when the recall of the defect predictor decreases, SBST guided by DP completely misses test generation for certain bugs, hence leads to poorer bug detection. In our experiments, SBST guided by DP misses test generation for 18.2% of the bugs on average when recall decreases from 100% to 75%. Further analysis of the results indicates that SBST guided by DP only misses test generation for 4.5% of the bugs on average for the bugs that spread across multiple methods, whereas it misses 24.7% of the bugs on average for the bugs that are concentrated into only one method. This suggests that the bugs that are found within only one method are more prone to the impact of recall compared to bugs that are spread across multiple methods.

To understand the effects of recall on detecting bugs which are found within only one method and spread across multiple methods, we conduct Welch ANOVA test [166] separately for the two subsets of our dataset, i.e., bugs having only one buggy method and bugs having more than one buggy method. The reason for carrying out Welch ANOVA test is because our data fails the assumption of homogeneity of variances for each combination of the groups of recall for bugs having only one buggy method. The results of the normality tests can be found in Appendix B.

	Num Df	Denom Df	F value	p-value
# buggy methods > 1	5.00	137.06	67.24	<0.001
# buggy methods $= 1$	5.00	136.68	395.91	<0.001

TABLE 6.2: Summary of the Welch ANOVA test results. Num Df = degrees of freedom of the numerator and Denom Df = degrees of freedom of the denominator.

The results of the Welch ANOVA test are shown in Table 6.2. There are 135 bugs which have more than one buggy method. The results for these bugs show that overall recall has a significant effect on number of bugs detected by SBST guided by DP (p-value <0.001) with a large effect size ($\widehat{\epsilon}^2 = 0.53$) [167]. However, the Games-Howell post-hoc test reveals that the bug detection effectiveness is not significantly different between recall 80%, 85% and 90%, and 95% and 100%. This can be seen in the violin plots in Figure 6.2 as well.

There are 285 bugs which have only one buggy method. The results of Welch ANOVA test for these bugs show that recall has a significant effect on number of bugs detected by SBST guided by DP (p-value <0.001) with a large effect size ($\widehat{\epsilon}^2 = 0.87$). The Games-Howell post-hoc test confirms that the number of bugs detected by SBST guided by DP is significantly different between each group of recall (p-values <0.001) with large effect sizes ($d \geq 0.98$) as can be seen in Figure 6.3. The Games-Howell post-hoc test results for all possible pairs of recalls can be found in Appendix B.

In summary, we find that recall has a significant effect on bug detection effectiveness of SBST guided by DP regardless of whether the bugs are found within one method or spread across multiple methods. However, for the bugs that are spread across multiple methods, the effect size of recall effect is smaller when compared to bugs that are found within one method ($0.53 < 0.87$). In contrast to bugs that are found within one method,

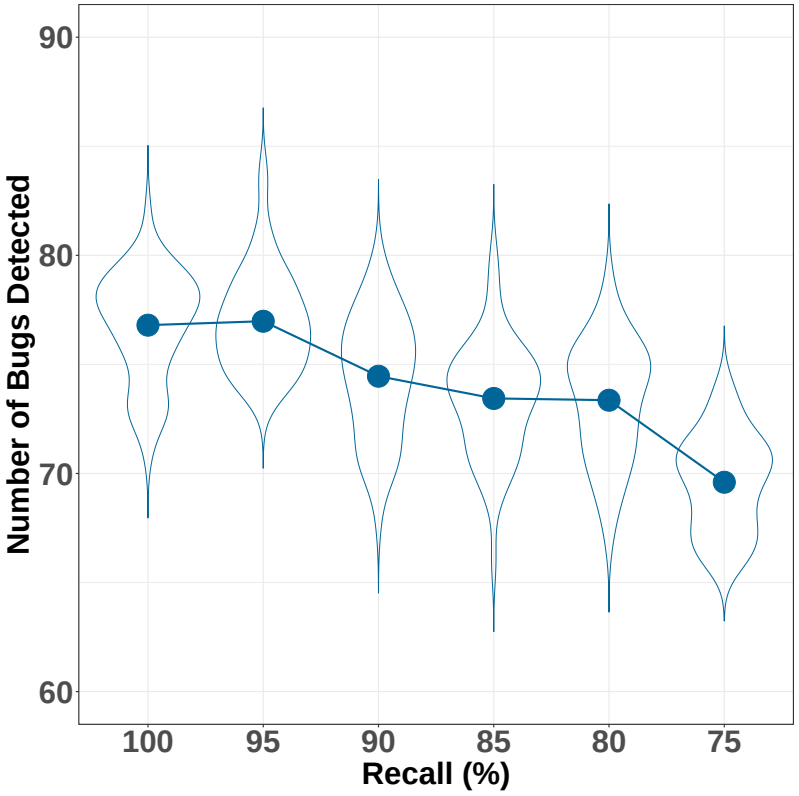

FIGURE 6.2: Distributions of the number of bugs detected by SBST guided by DP as violin plots together with the means plot of number of bugs detected by SBST guided by DP for the groups of recall. Only for the bugs that have more than one buggy method. Total number of bugs = 135.

the effect of recall is not significant between the groups of recall 80%, 85% and 90%, and 95% and 100% for the bugs that are spread across multiple methods.

6.3.2.3 Sensitivity to the Precision of the Defect Predictor

According to the two-way ANOVA test, the precision of the defect predictor has a statistically significant effect, however with a very small effect size, which suggests the effect is not of meaningful practical significance. Precision is associated with false positives (Equation. (6.2)), i.e., non-buggy methods predicted as buggy by the defect predictor. Change of precision from 100% to 75% means that there are false positives in the defect prediction results. We investigate the buggy method labels produced by the defect predictor and the bug detecting results of SBST guided by DP in our experiments to find out if false positives have actually helped SBST guided by DP to detect more bugs. We find that the false positives have not contributed to the bug detection performance

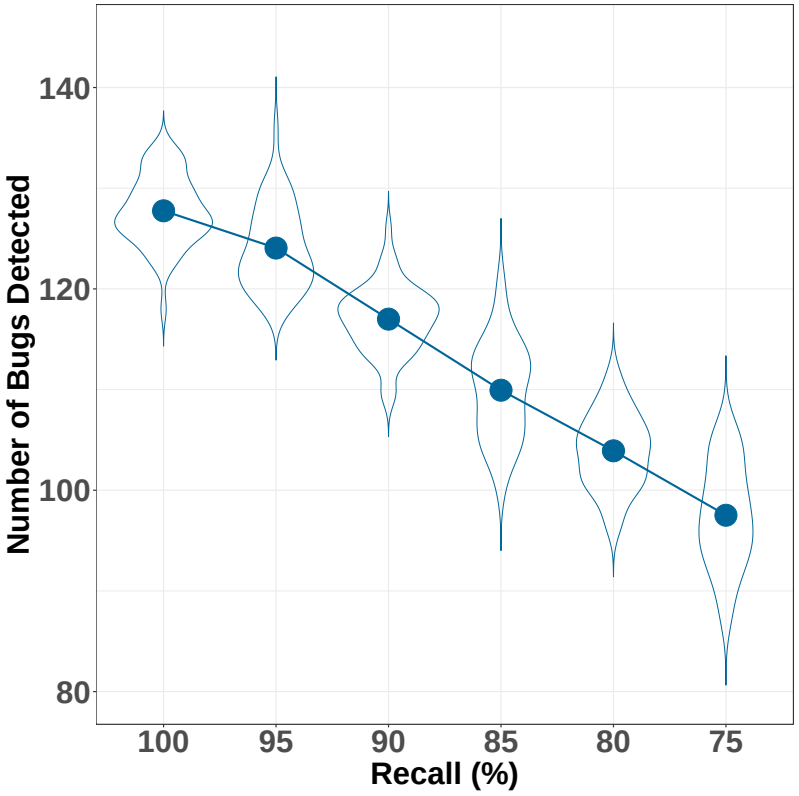

FIGURE 6.3: Distributions of the number of bugs detected by SBST guided by DP as violin plots together with the means plot of number of bugs detected by SBST guided by DP for the groups of recall. Only for the bugs that have one buggy method. Total number of bugs = 285.

of SBST guided by DP. We conclude that the impact of precision is not of practical significance to the bug detection performance of SBST.

Effects from false positives on SBST may be different when SBST guided by DP is given a small time budget. In particular, SBST guided by DP with a sufficient time budget like 120 seconds may have enough time to search for tests in actual buggy methods (i.e., true positives) despite searching for tests in false positives. Whereas the search for tests in actual buggy methods may be greatly impacted when SBST guided by DP is given a small time budget like 15 or 30 seconds. Hence, we further investigate the impact of the time budget on the conclusions about sensitivity to the defect prediction precision. We find that the conclusions remain the same at 5, 10, 15, 30, 60 and 120 seconds time budgets, i.e., there is no meaningful practical effect of precision on the bug detection performance of SBST. The results of the two-way ANOVA tests along with the violin and profile plots at each time budget are reported in the Appendix C.

6.3.3 Discussion

Defect predictors have mainly been used to provide a list of likely defective parts of a program (e.g., classes and methods) to programmers, who then manually inspect or test the likely defective parts to find the bugs [39, 41]. In this context, the precision of the defect predictor is very important [47]. Poor precision of the defect predictor means there are higher false positives. Higher false positives can waste developers' time and lead to losing their trust on the prediction results [39]. However, when the defect predictions are consumed by another automated testing technique such as SBST, this may not be the case. In the context of SBST, our study reveals contrasting findings. We find that the effect of precision on the bug detection performance of SBST is negligible, while the recall of the predictor has a significant impact with a large effect size.

We recommend that programmers improve the recall of the defect predictor at the cost of precision to achieve good performance in SBST guided by defect prediction. There is a trade-off between recall and precision of a defect predictor [168]. Defect predictors are usually good at detecting bugs (i.e., high recall) at the expense of false positives (i.e., low precision). Our study shows the bug detection effectiveness of SBST guided by DP is highly sensitive to recall, while the effect of precision is negligible. This means that most defect predictors proposed in the literature would be suitable for guiding SBST. As the scope of our study is to analyse the impact of precision in the range of an acceptable defect predictor, i.e., precision $\geq$ 75%, we cannot make any conclusions about defect predictors with precision below 75%. Therefore, we can conclude that it is beneficial to increase the recall of the defect predictor by sacrificing precision while maintaining it above an acceptable level, e.g., 75%.

The primary actionable conclusion from this study for the research community is to define recall-at-precision measure to be 75% for defect predictors in the context of combining defect prediction and SBST. Defect predictors having recall and precision greater than or equal to 75% are considered acceptable defect predictors [48]. Currently the defect prediction community is targeting to increase both precision and recall. We recommend the researchers to target higher recall while having a sufficiently high precision, which is 75%. This approach is widely used in training machine learning classifiers, where there is a particular precision level required to meet to avoid false positives, such that any classifier that fails to meet this criterion is considered unacceptable. When the

criterion for minimum precision level is met, increasing recall at the minimum precision or above becomes the goal.

6.4 Threats to Validity

In Section 4.6, we discussed the validity threats that are common to the three research studies conducted in this thesis. In addition, we discuss the following threats that are specific to this study.

Construct Validity. To systematically investigate the impact of defect prediction imprecision, we simulate the predictions by assuming a uniform distribution of defect prediction errors which is similar to previous work [129]. This means in our simulations, every method has an equal chance of being labelled incorrectly independent of each other. However, real defect predictors may have different distributions of their predictions depending on the underlying characteristics and nature of the prediction problem, which may impact the realism of a simulated defect predictor. Nevertheless, in the absence of prior knowledge about empirical or theoretical defect prediction distributions, it is reasonable to assume a uniform distribution of predictions in the defect prediction simulation.

SBST guided by DP generates more than one test case for each target in the CUT. This increases the chances of detecting bugs at the cost of larger test suites. Larger test suites are associated with a higher number of assertions in the tests generated by EvoSuite, which need to be manually adapted by developers in practice. This may increase the labelling cost for developers, the impact of which needs to be investigated in future work.

Internal Validity. To account for the randomness in the defect prediction simulation, we repeat the simulations 5 times for each combination of the groups of recall and precision. For each simulation, we repeat the test generation 5 times to account for the non-deterministic behaviour of SBST guided by DP. In total, we conduct 25 test generation runs for each bug and for each level of defect prediction imprecision.

External Validity. We investigate the impact of defect prediction imprecision only in the range of 75% to 100% for recall and precision. Therefore, our findings may not be generalised to the defect predictors which have recall or precision less than 75%. While

this choice of performance sampling in our simulation is a threat to external validity, it is also a threat to construct validity for lack of characterising all possible defect predictors. However, we opted to use this range with the justification that this is the range for an acceptable performance for a defect predictor as recommended by Zimmermann et al. [48].

6.5 Summary

We study the impact of imprecision in defect prediction on the bug detection performance of SBST. We use simulated defect predictors to systematically sample defect predictors in the range of 75% to 100% for recall and precision. We use the state-of-the-art SBST technique, DynaMOSA, and incorporate predictions about buggy methods as given by the simulated defect predictor to guide the search for test cases towards likely buggy methods. Through a comprehensive experimental evaluation on 420 bugs from the Defects4J dataset, we find that the recall of the defect predictor has a significant impact on the bug detection effectiveness of SBST with a large effect size. More specifically, SBST guided by DP finds 7.5 less bugs on average (out of 420 bugs) for every 5% decrements of recall. On the other hand, the impact of precision is not of practical significance as indicated by a very small effect size, hence we conclude that the precision of defect predictors has negligible impact on the bug detection effectiveness of SBST, as long as one uses a defect predictor with acceptable performance, i.e., with precision and recall greater than 75%. Further analysis of the results shows that the impact of the recall for the bugs that are spread across multiple methods is smaller compared to the bugs that are found within only one method.

Based on the results of our study, we make the following recommendations:

1. SBST techniques must take potential errors in the predictions into account, in particular the false negatives. Currently, the search for tests exploits the likely buggy targets, however, we recommend that SBST techniques also require to explore the likely non-buggy targets at least with a minimum probability. One possible solution is to prioritise predicted buggy parts of the program, while guiding the search with a certain probability towards locations that are predicted as not buggy.

2. In the context of combining defect prediction and SBST, it is beneficial to increase the recall of the defect predictor by sacrificing precision, while maintaining the precision above an acceptable level, e.g., 75%. When the predictions are used by SBST, an acceptable amount of false positives are not a problem. For SBST, it is important to be informed of most of the buggy targets even at the expense of acceptable level of false alarms. We recommend the researchers to target higher recall while having a sufficiently high precision, instead of trying to elevate both recall and precision.

Chapter 7

Guiding the Search Process with Defect Prediction

7.1 Introduction

Search-based software testing (SBST) techniques often aim at maximising code coverage, which is often used to define the objective functions used in them. For example, SBST techniques that formulate the test generation problem as a single objective formulation aggregate all the coverage targets of the class under test (CUT) into a single objective function (see Section 2.2.3). Many objective sorting algorithms simultaneously optimise test cases to meet multiple objectives where each of them represents each coverage target in the CUT (see Section 2.2.4). The existing SBST techniques treat all the coverage targets as equally important to cover, since covering any target increases the code coverage by an equal amount and rewards the search equally. In particular, for SBST techniques with a single objective function, covering any of the uncovered targets increases the fitness by an equal amount. For many objective sorting algorithms, covering any of the uncovered targets means the respective objective score is set to zero and the objective is removed from the set of uncovered objectives.

Only one or a few methods are buggy in a class. Hence, only a subset of all the coverage targets contains the buggy methods. It is likely to be ineffective in terms of bug detection to optimise test cases/suites to satisfy coverage targets that do not contain the buggy methods. In fact, the test suites generated by the existing SBST techniques have only

a few test cases that cover the buggy methods despite having high code coverage. We identify this as a main limitation in SBST techniques guided only by coverage. In the context of detecting bugs, the set of coverage targets that contain the buggy methods are likely to be more important to cover than the other targets.

We hypothesise that augmenting coverage information used by SBST techniques with defect prediction information in the search process improves the bug detection performance of SBST. We argue that SBST techniques should increase the test coverage towards buggy methods within the CUT to increase the chances of detecting bugs. We use defect prediction that works at method level to get information on which methods are likely to be buggy within a class. SBST technique can then identify the coverage targets that are likely to contain buggy methods to direct the test coverage. Our goal is to develop an SBST technique that guides the search for test cases towards likely buggy coverage targets in the class by using information from defect prediction. Therefore, in this chapter, we aim to achieve the following research objective;

> RO3: Develop an SBST technique that uses defect prediction to guide the search process to likely defective areas.

To achieve this research objective, we propose predictive many objective sorting algorithm (PreMOSA) and demonstrate its improved effectiveness and efficiency in terms of detecting bugs through an experimental assessment using theoretical defect predictors. PreMOSA is a many objective solver that uses information from a method level defect predictor and guides the search for tests towards the likely buggy methods. PreMOSA prioritises exploiting the coverage targets that contain likely buggy methods. It handles potential errors in the predictions by exploring the coverage targets that contain likely non-buggy methods with a lesser priority. In order to ensure trivial and nontrivial targets have a fair opportunity to be covered, PreMOSA employs a method to balance the test coverage among all the targets in the search. We use Defects4J dataset as benchmark subjects to experimentally evaluate PreMOSA against the state-of-the-art DynaMOSA in terms of bug detection effectiveness and efficiency. In the experimental assessment, we use theoretical (i.e., simulated) defect predictors that can be replaced with any real defect predictor in practice [31, 32]. We intentionally abstract the defect predictor component to avoid potential confounding effects that can be caused by using specific defect predictors.

7.2 Motivation

Figure 5.10 shows a buggy code snippet and the applied patch for `DateTimeZone` class from Time-8 bug in Defects4J [136]. The buggy method, `forOffsetHoursMinutes`, takes two integer inputs, `hoursOffset` and `minutesOffset`, and returns the `DateTimeZone` object for the offset specified by the two inputs. For example, if the method is called with the inputs `hoursOffset=0` and `minutesOffset=-30`, then it is expected to return a `DateTimeZone` object for the offset $-00:30$. However, such inputs execute the `true` branch of the `if` condition at line 279 and the method throws an `IllegalArgumentException` instead of the expected `DateTimeZone` object. This bug is fixed by modifying the `if` condition at line 279 and adding a new condition at line 282 as shown in the diff in Figure 5.10.

To detect this bug, test cases have to execute the `false` branches of the `if` conditions at line 273 and 276; that is `hoursOffset` $\neq 0$ or `minutesOffset` $\neq 0$ and `hoursOffset` $\in [-23, 23]$. They also have to execute the `true` branch at line 279 with an additional constraint; `minutesOffset` $\in [-59, -1]$. Furthermore, the newly added `if` condition at line 282 adds another constraint on the input `hoursOffset`; that is `hoursOffset` ≤ 0. In summary, only the test inputs sampled from the space where `hoursOffset` $\in [-23, 0]$ and `minutesOffset` $\in [-59, -1]$ can detect the bug.

It is evident that just covering the buggy code (i.e., the `true` branch of the `if` condition at line 279) is not sufficient to detect the bug. For example, the inputs `hoursOffset=12` and `minutesOffset=-60` cover the buggy code, however, they do not detect the bug. As shown in Figure 7.1, the space of all possible test inputs that cover the buggy code (i.e., `hoursOffset` $\in [-23, 23]$ and `minutesOffset` $\notin [0, 59]$) is larger than the space of test inputs that can detect the bug. The existing SBST techniques that aim at maximising code coverage, such as DynaMOSA are more likely to sample test inputs from the larger space of inputs that cover the buggy code without detecting the bug, and then terminate without actually detecting the bug.

The existing SBST approaches can be configured to generate many tests for each coverage target in the `DateTimeZone` class, and it will increase the chances of detecting the bug. However, there are 54 methods in the class and only one method is buggy. If we assume the test adequacy criterion to be branch coverage, then there are 201 coverage

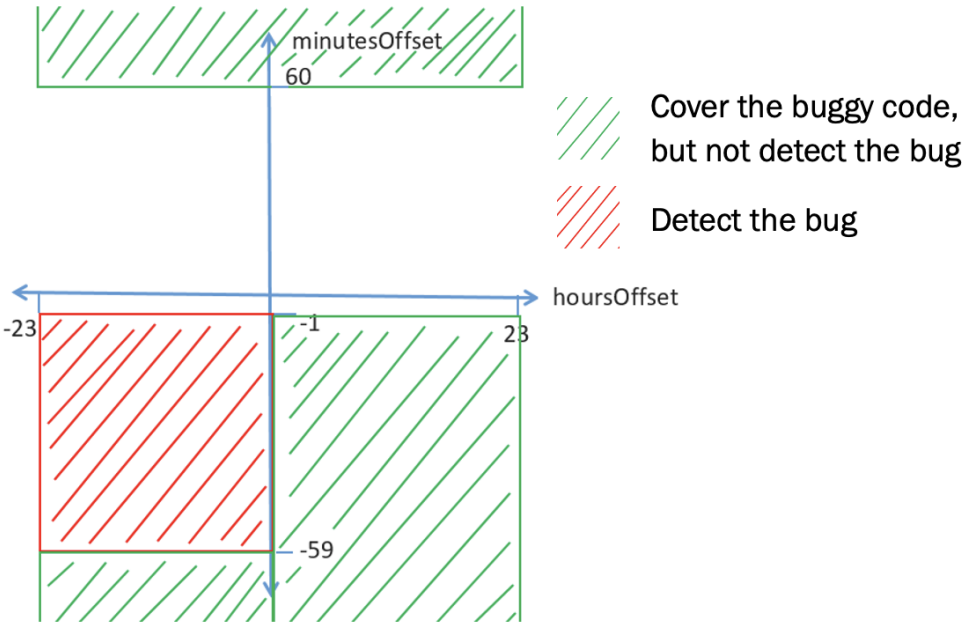

FIGURE 7.1: Search space of test inputs for covering the buggy code and detecting the bug for Time-8 bug

targets in total, while only 14 of them actually contain the buggy method. Thus, we find it is ineffective to spend all the critical search resources on covering all the 201 targets, when only a few of them leads to the buggy code. We propose to use buggy methods predictions from a defect predictor to decide where to increase the coverage within the class. Thus, our novel SBST approach concentrates the search for test cases more on the only buggy method in the project, `forOffsetHoursMinutes`.

7.3 Predictive Many-Objective Sorting Algorithm

Predictive many objective sorting algorithm (PreMOSA) is a novel search-based software testing approach that incorporates guidance from a defect predictor. PreMOSA receives as input a buggy program with methods labelled as buggy or non-buggy, which are labels that can be obtained using existing defect predictors [31, 32]. PreMOSA is not specific to a certain defect predictor. Hence, we use the most commonly used defect predictor output type in PreMOSA, which is binary classification [50]. PreMOSA uses this information to start searching for test inputs that cover targets that are deemed to contain buggy methods as indicated by the defect prediction information (see Section 7.3.1). This helps focus the search initially on covering the likely buggy targets rather than the likely non-buggy targets.

Most of the time, defect predictors are not 100% accurate, which means that they may label actual buggy methods as non-buggy. PreMOSA starts searching for test cases to cover the targets that contain predicted non-buggy methods, once the likely buggy targets coverage does not improve for a pre-defined number of consecutive iterations (see Section 7.3.1).

PreMOSA also generates more than one test case for all the selected targets, hence, increases the chances of detecting bugs (see Section 7.3.2).

Finally, to balance the test coverage among all the targets in the search, we introduce a method to dynamically disable coverage targets from the search based on their current test coverage and number of independent paths (see Section 7.3.3). This ensures that the non-trivial targets have an equal chance of being covered compared to the targets that are easier to cover.

PreMOSA is presented in Algorithm 4. It is based on a genetic algorithm (GA). PreMOSA creates an initial population of randomly generated test cases (line 9 in Algorithm 4). Then, it evolves this initial population through creating new test cases via crossover and mutation (line 13) and selecting test cases to the next generation (line 18), until a termination criterion, such as maximum time budget, is met.

7.3.1 Filtering Targets with Defect Prediction

A defect predictor classifies the methods of the CUT as buggy or non-buggy, denoted as c_i, where

$$c_i = \begin{cases} 1 & \text{if } m_i \text{ is predicted as buggy} \\ 0 & \text{otherwise} \end{cases}$$

where m_i denotes method with index i. PreMOSA starts with filtering the likely buggy and likely non-buggy targets, U_B and U_N respectively, from the set of all targets U using the classifications given (line 2 in Algorithm 4). The procedure FilterTargets labels targets as buggy if they belong to a likely buggy method and non-buggy otherwise. Initially, PreMOSA finds tests to cover only the likely buggy targets, hence, only the likely buggy targets are selected to be included in the search process in the beginning (line 5). This way, PreMOSA can extensively search for test cases that cover likely buggy

Algorithm 4 PreMOSA

Input:
$U = \{u_1, \ldots, u_k\}$ ▷ the set of coverage targets of CUT
$G = \langle N, E \rangle$ ▷ control dependency graph of the CUT
$\phi : E \to U$ ▷ partial map between edges and targets
$C = \{c_1, \ldots, c_m\}$ ▷ the set of defectiveness classifications for methods in the CUT

1: **procedure** PreMOSA
2: $U_B, U_N \leftarrow$ FilterTargets(U, C)
3: $L \leftarrow$ IndependentPaths(G) ▷ L is a vector of the number of independent paths for each edge
4: **if** U_B is not empty **then**
5: $U \leftarrow U_B$
6: **else**
7: $U \leftarrow U_N$
8: $U^* \leftarrow$ targets in U with no control dependencies
9: $P_0 \leftarrow$ RandomPopulation(M) ▷ M is the population size
10: $A \leftarrow$ UpdateArchive$(P_0, \emptyset)$ ▷ A is the archive
11: $U^* \leftarrow$ UpdateTargets(U^*, G, ϕ)
12: **for** $r \leftarrow 0$; !terminationCriteria; r++ **do**
13: $Q_r \leftarrow$ GenerateOffspring(P_r)
14: $A \leftarrow$ UpdateArchive(Q_r, A)
15: $U^* \leftarrow$ UpdateTargets(U^*, G, ϕ)
16: $R_r \leftarrow P_r \cup Q_r$
17: $U^* \leftarrow$ SwitchOffTargets(U^*, A, L, ϕ)
18: $P_{r+1} \leftarrow$ SelectPopulation(R_r, U^*, M)
19: $U^* \leftarrow$ AddNonBuggyTargets
20: $T \leftarrow A$ ▷ Update the final test suite T
21: Return(T)
22: **procedure** AddNonBuggyTargets
23: **if** trigger *not* fired to add non-buggy targets **then**
24: **if** # covered goals = prev. # covered goals **then**
25: w++
26: **else**
27: $w = 0$
28: **if** $w = I$ **then** ▷ I is max. # iterations without coverage improvement
29: $U \leftarrow U \cup U_N$
30: $U^* \leftarrow U^* \cup \{u \in U_N | u$ has no control dependencies$\}$
31: Return(U^*)

targets, which leads to generating more effective test cases faster than other approaches by increasing the chances of reaching the buggy code.

Defect predictors often are not 100% accurate, and it is likely that buggy methods may be labelled as non-buggy. To address this issue, PreMOSA considers targets that do not contain any methods that are predicted as buggy if it deems to have searched enough for tests that cover the likely buggy targets (line 19). If PreMOSA resorts to

searching for tests to cover only the likely buggy targets, then it will miss actual buggy targets that are incorrectly classified. Thus, PreMOSA starts finding tests to cover likely non-buggy targets once the coverage of likely buggy targets does not improve for a predefined number of consecutive iterations (I) in GA (line 28). This way, PreMOSA expects to account for the errors present in the predictions. Finally, if there are no likely buggy targets, either because the class is not buggy or the defect predictor is inaccurate, PreMOSA considers all targets from the start (line 7).

7.3.2 Updating Targets and Archiving Tests

PreMOSA generates more than one test case for all the selected targets in order to increase the chances of infection and propagation of the bugs. When updating targets in each iteration (lines 11 and 15), it does not remove covered targets from U^*, allowing it to keep generating more tests to cover those targets as well.

PreMOSA keeps an archive of all the test cases that cover the selected targets $u \in U$ during the search (lines 10 and 14). This archive of test cases form the final test suite. Thus, the final test suite is more likely to detect the bugs as it contains all the generated test cases which cover the potentially buggy targets.

Removing covered targets from U^* and archiving only the shortest test case for each covered target are beneficial for achieving high code coverage with a minimal test suite size [14]. However, just covering the buggy code is not sufficient to detect the bug. In Chapter 5, we showed that there was an average improvement of up to 79% in terms of detecting bugs when the state-of-the-art DynaMOSA was configured to not to remove covered targets from the search and retain all the generated tests. Therefore, we decide to archive all the test cases that cover the selected targets $u \in U$ and not to remove the covered targets from the search in PreMOSA.

7.3.3 Balanced Test Coverage of Targets

Figure 7.2 shows the control dependency graph (CDG) of the method `forOffsetHoursMinutes` from the motivating example in Section 7.2. Nodes denote the predicates and leaves denote the exit points of the program. For example, node 279-1 denotes the `minutesOffset`

`< 0` predicate at line 279 and leaf 280 denote the `return` statement at line 280. Lines between nodes denote the control dependency edges. For example, $b_{5,T}$ is the `true` branch of the `minutesOffset < 0` predicate. For simplicity, we do not include the nodes that are not predicates of the program.

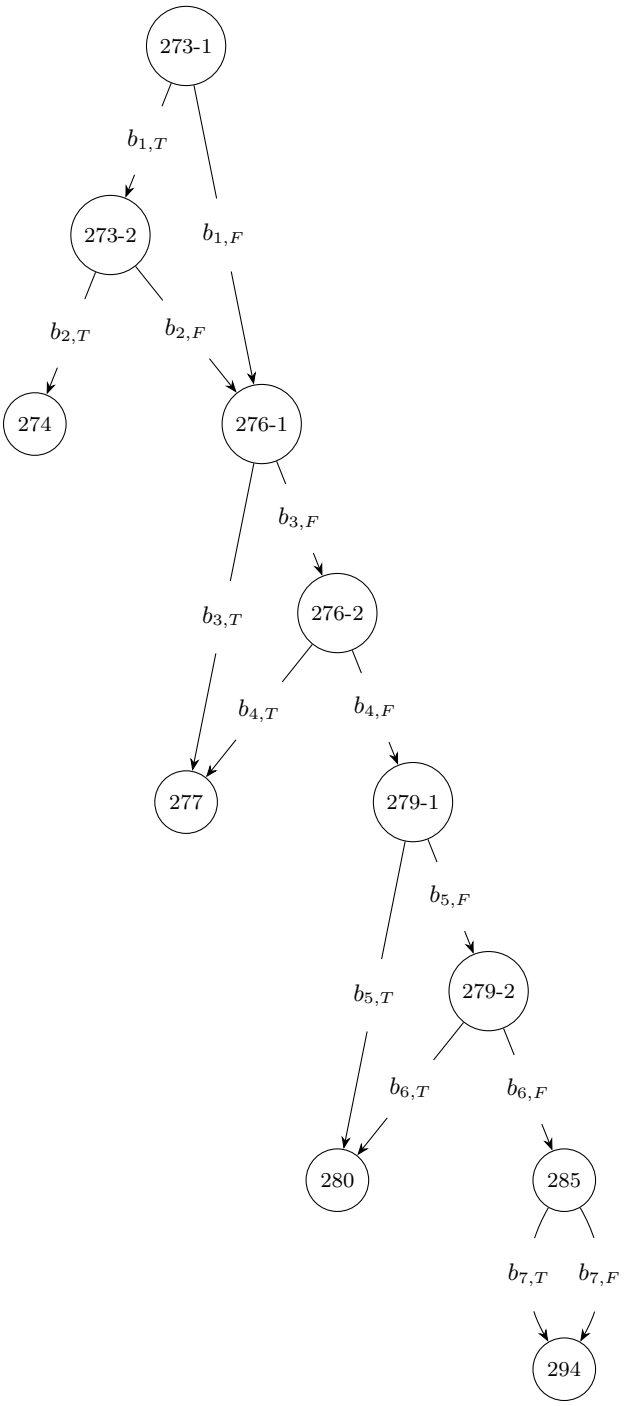

FIGURE 7.2: Control dependency graph of the method `forOffsetHoursMinutes` from Time-8 bug

In our running example, assume the branch coverage is the optimisation criterion. An

SBST technique that does not remove covered targets from the search is more likely to keep on generating test cases which cover more trivial branches like $b_{3,T}$ or $b_{4,T}$ rather than a less trivial branch $b_{5,T}$ (Figure 7.2). This is detrimental to the bug detection performance of SBST since it is necessary to find tests that exercise the branch $b_{5,T}$ in order to detect the bug, and also, to increase the chances of detecting the bug, SBST has to find as many tests as possible that cover $b_{5,T}$.

We introduce a new method to dynamically remove coverage targets from the search based on their current test coverage and number of independent paths, in order to balance the test coverage among all the targets. A balanced test coverage means that all the targets receive an equitable test coverage. This ensures that, in PreMOSA, the less trivial targets also get a good coverage in the presence of more trivial targets.

A balanced test coverage is achieved when the number of tests generated per an independent path of a target is equal for all of the targets. We measure the number of independent paths of a target by assuming the paths start at the control dependent edge of that target (line 3). An independent path is one that traverses one or more new edges in the control dependency graph.

In general, for each target $u \in U^*$, PreMOSA checks the current test coverage (i.e., number of tests in the archive that cover u), and then temporarily removes u from U^* in the current iteration, if the test coverage per an independent path from u is higher than the other targets (line 17).

7.3.3.1 Independent Paths

We use the number of independent paths of a target to determine how much of a test coverage a target should receive compared to other targets, in order to achieve a balanced test coverage for all targets. For a target $u \in U^*$, if there are many independent paths that start from u, then PreMOSA should generate more tests to cover u than the other targets which have few independent paths. In our running example, the target $b_{2,F}$ should receive more test coverage than $b_{2,T}$ because there are more independent paths leading up from $b_{2,F}$ (6) compared to $b_{2,T}$ (1).

In the beginning of the search, PreMOSA finds the number of independent paths of each edge in the control dependency graph G of the program (line 3). The control

dependency graph $G = \langle N, E \rangle$ consists of nodes $n \in N$ and edges $e \in E \subseteq N \times N$. The nodes represent statements in the program. The edges represent control dependencies between the statements. For each edge $e \in E$, the procedure IndependentPaths calculates the number of independent paths starting from e using the graph G. The actual executions of the paths start at the root node, however, in the calculation of number of independent paths of e, we assume the paths start at e. All the coverage targets that are directly control dependent by e have the same number of independent paths as that of e.

In the motivating example, the edges $b_{2,T}$, $b_{3,T}$, $b_{4,T}$, $b_{5,T}$, $b_{6,T}$, $b_{7,T}$ and $b_{7,F}$ have only one path each that start from those edges. There are 2 independent paths from the edge $b_{6,F}$, those are $b_{6,F} - b_{7,T}$ and $b_{6,F} - b_{7,F}$. Likewise, there are 7, 6, 6, 5, 4 and 3 independent paths that start from edges $b_{1,T}$, $b_{1,F}$, $b_{2,F}$, $b_{3,F}$, $b_{4,F}$ and $b_{5,F}$, respectively. If the optimisation problem is maximising the branch coverage, then these edges become the coverage targets in the search.

7.3.3.2 Temporarily Disabling Targets from the Search

In many objective optimisation, test cases are optimised simultaneously to satisfy all the coverage targets. Thus, the search resources (e.g., time budget) are not allocated to each coverage target individually. To focus the search differently on covering each target, PreMOSA dynamically switches off targets during the evolution. In every iteration in GA, for each target $u \in U^*$, the procedure SwitchOffTargets checks the current test coverage of u, and then removes u from U^*, if the test coverage per an independent path of u is higher than that of the other targets. Therefore, after calling the procedure SwitchOffTargets, only the targets which are having a low test coverage (per an independent path) remain in U^*. Then, the procedure SelectPopulation selects test cases to the next generation considering only these remaining targets in U^*. This paves the way for the search to find more test cases in the next generation that cover these targets, thereby guiding the search to a balanced test coverage for all the targets.

First, the procedure SwitchOffTargets finds the set of nodes with predicates N_P in G (line 2 in Algorithm 5). Next, for each node $n \in N_P$, it fetches the number of independent paths from the outgoing edges of n (lines 5-6). Then, it randomly selects a control dependent target from each outgoing edge of n (lines 7-8). We consider all

Algorithm 5 Temporarily Removal of Targets to Balance Test Coverage

1: **procedure** SwitchOffTargets(U^*, A, L, ϕ)
2: $N_P \leftarrow$ NodesWithPredicates(G)
3: **for** $n \in N_P$ **do**
4: $\{e_{n,T}, e_{n,F}\} \leftarrow$ outgoing edges in G from node n
5: $l_{n,T} \leftarrow$ GetIndependentPaths($L, e_{n,T}$)
6: $l_{n,F} \leftarrow$ GetIndependentPaths($L, e_{n,F}$)
7: $u_{n,T} \leftarrow$ RandomChoice($\{\phi(e_{n,T})\}$)
8: $u_{n,F} \leftarrow$ RandomChoice($\{\phi(e_{n,F})\}$)
9: $A_{n,T} \leftarrow$ GetTests($A, u_{n,T}$)
10: $A_{n,F} \leftarrow$ GetTests($A, u_{n,F}$)
11: **if** $\frac{|A_{n,T}|}{l_{n,T}} > \frac{|A_{n,F}|}{l_{n,F}}$ **then**
12: $U^* \leftarrow U^* - \{\phi(e_{n,T})\}$
13: **else if** $\frac{|A_{n,T}|}{l_{n,T}} < \frac{|A_{n,F}|}{l_{n,F}}$ **then**
14: $U^* \leftarrow U^* - \{\phi(e_{n,F})\}$
15: Return(U^*)

the control dependent targets of an edge receive the same test coverage. Hence, the test coverage of a randomly selected target of an edge is equal to the test coverage of that edge. Finally, it finds the edge which has the largest number of tests in the archive per an independent path, and removes all the control dependent targets of that edge from U^* (lines 9-14).

In the running example, if we consider the node 276-2, the outgoing edges are $b_{4,T}$ and $b_{4,F}$, and the number of independent paths from these edges are 1 and 4, respectively. Assume the coverage criterion is maximise branch coverage, hence $b_{4,T}$ and $b_{4,F}$ are also targets in the search, and there are currently 30 and 20 tests in the archive covering $b_{4,T}$ and $b_{4,F}$, respectively. Thus, $b_{4,T}$ has 30 (= 30/1) tests in the archive per an independent path, while $b_{4,F}$ has only 5 (= 20/4) tests per a path. Hence SwitchOffTargets temporarily removes the target $b_{4,T}$ from U^*, and paves way for the search to find more test cases that cover $b_{4,F}$. Overall, this encourages the search to have a balanced test coverage for all the targets rather an excessive coverage of more trivial targets like $b_{3,T}$ and $b_{4,T}$. As a result, a less trivial target like $b_{5,T}$, which contains the buggy code, receives a good coverage in the presence of other more trivial targets.

7.4 Experimental Evaluation

We design a set of experiments to evaluate PreMOSA in terms of its effectiveness and efficiency in detecting bugs compared to the state-of-the-art DynaMOSA. Through these experiments, we aim to investigate if augmenting coverage information with defect prediction information in the search process of SBST indeed helps to improve the bug detection performance of the generated test suites. Our first research question is:

RQ1: Is PreMOSA more effective in detecting bugs compared to the state-of-the-art DynaMOSA?

To answer this research question, we compare the number of bugs detected by PreMOSA against DynaMOSA, which we discuss in Section 7.4.1.3. We run test generation on Defects4J bugs [136] (discussed in Section 7.4.1.2) using both PreMOSA and the baseline. To account for randomness in PreMOSA and DynaMOSA, we repeat the test generation for 25 runs for each bug and testing approach. Once test cases are generated and evaluated for bug detection, we report the bug detection results as means and medians over 25 runs. To check if PreMOSA significantly detects more bugs than DynaMOSA and the effect size of the difference, we employ one-tailed non-parametric Mann-Whitney U-Test with the significance level (α) 0.05 [144] and Vargha and Delaney's $\widehat{A}_{12}$ statistic [145].

To analyse the efficiency of PreMOSA, we seek to answer the following research question:

RQ2: Is PreMOSA more efficient at generating test cases that can detect bugs compared to the state-of-the-art DynaMOSA?

To answer this research question, we measure the time to generate the first test case that can detect a bug by the two approaches over 25 runs. As we described in Section 4.3, a test case detects a bug if it satisfies all the three conditions of the reachability, infection and propagation (RIP) model, and we call such test cases *bug detecting tests* throughout the chapter. For each bug that is detected by both approaches, we calculate the difference of the mean time to generate the first test case that detects the particular bug by the two approaches. If the difference is positive, that means PreMOSA generates a test case to detect the bug in a shorter time. A negative difference means otherwise. To check

if PreMOSA generates a bug detecting test in a significantly shorter time, we employ one-tailed Wilcoxon signed-rank test [144] and its effect size, r [146]. We remind the reader that the time taken to generate the first bug detecting test is not equal to time taken to reveal the bug. The latter happens only after the test generation is completed.

Zimmermann et al. [48] argues that a defect predictor is strong if, and only if, all recall, precision and accuracy are greater than 75%. Therefore, we consider defect predictors having both recall and precision in the range 75% to 100% as acceptable defect predictors. In *RQ1* and *RQ2*, we simulate defect predictor outcomes for two levels of performance for PreMOSA; i) most conservative and acceptable defect predictor (recall=precision=75%) and ii) ideal defect predictor (recall=precision=100%). We will discuss this more in Section 7.4.1.1. We expect PreMOSA to perform best with the latter defect prediction simulation, and with the former simulation, we can see the most conservative performance of PreMOSA when using acceptable defect predictors.

In addition, to analyse the effects of the balanced test coverage of targets method, we seek to answer the following research question:

RQ3: How does the balanced test coverage of targets affect the coverage and bug detection of SBST?

To answer this research question, we extend the state-of-the-art SBST, DynaMOSA, to use the balanced test coverage of targets method described in Section 7.3.3 and refer to this as DynaMOSA+b throughout the chapter. We compare the number of bugs detected (effectiveness) and the time to generate bug detecting tests (efficiency) by DynaMOSA+b against DynaMOSA. We run test generation on Defects4J bugs using DynaMOSA+b and repeat the test generation for 25 runs. To measure and compare the effectiveness and the efficiency of detecting bugs of DynaMOSA+b against DynaMOSA, we follow the same procedure as described in *RQ1* and *RQ2*.

We expect DynaMOSA+b to cover more targets since it encourages the search method to find test cases that cover nontrivial targets by dynamically disabling trivial targets from the search. We compare branch coverage achieved by DynaMOSA+b against DynaMOSA for each bug. To check for statistical significance of the difference and the effect size, we employ two-tailed non-parametric Mann-Whitney U-Test with the significance level (α) 0.05 and Vargha and Delaney's $\widehat{A}_{12}$ statistic.

Ideally, balanced test coverage should make all the branches in the CUT have an equal number of tests per independent path, which means that the distribution of the number of tests per independent path of branches in a CUT should have a zero variance. Therefore, we compare the variation in the distributions of the number of tests per independent path of DynaMOSA+b and DynaMOSA. An approach with smaller variation indicates that it has achieved a better balance of test coverage compared to the approach that has larger variation. To do this, for each buggy class in Defects4J, we measure the number of tests that cover a branch at the end of the search for all the branches in the CUT by DynaMOSA+b and DynaMOSA. Then, we calculate the number of tests per an independent path for each branch in the CUT. Using this, we calculate the coefficient of variation (CV) of the number of tests per an independent path. CV is the ratio of the standard deviation to the mean of number of tests per an independent path. A smaller CV indicates a smaller variation of number of tests per an independent path, hence a better balance of test coverage. To check for statistical significance of the difference and the effect size of CV by DynaMOSA+b and DynaMOSA, we employ two-tailed non-parametric Mann-Whitney U-Test with the significance level (α) 0.05 and Vargha and Delaney's $\widehat{A}_{12}$ statistic.

7.4.1 Experimental Settings

7.4.1.1 Defect Prediction Simulation

We simulate defect predictor outcomes at two levels of recall and precision, which correspond to the theoretical upper bound and lower bound performance of an acceptable defect predictor. This would not be possible with real defect predictors since their performance cannot be controlled. Using a real defect predictor would have demonstrated the viability of PreMOSA in practice. However, it would then have the disadvantage of limiting the findings of our study to one single defect predictor, e.g., a specific defect predictor built with one learner and one set of metrics. Therefore, we abstract the defect predictor component in the experimental evaluation.

To simulate the defect predictions, we use the defect prediction simulation algorithm introduced in Chapter 6 (Section 6.2.1).

7.4.1.2 Experimental Subjects

We use the same set of experimental subjects used in the experimental evaluation in Chapter 6 (Section 6.3.1.1). That is a total of 420 manually validated real bugs from six real-world open source Java projects. Similar to Chapter 6, we label all the methods that are either modified or removed in the bug fix as buggy methods [132].

7.4.1.3 Baseline

As described in Section 4.2, we use the current state-of-the-art SBST technique, DynaMOSA [14], as the baseline. It is more effective at achieving high branch, statement and strong mutation coverage than previously proposed SBST techniques ([15, 51, 89]) [14].

We configure DynaMOSA to not remove the covered targets from the search, retain all the test cases generated, and continue the search until the full time budget is consumed in our experimental evaluation. DynaMOSA primarily focuses on achieving high code coverage with a minimal test suite size. Hence, it aims at generating only one short test case to cover each target in the program. However, just covering the buggy code is not sufficient to detect the bug. In Chapter 5, we showed that DynaMOSA detects 79% more bugs on average when it is configured to not remove covered targets from the search, use the full time budget, and retain all the generated tests in the final test suite (i.e., disable test suite minimisation).

7.4.1.4 Prototype

We implement PreMOSA in the state-of-the-art SBST tool, EvoSuite [3], within version 1.0.7, forked from the GitHub repository [98] on June 18^{th}, 2019. The prototype is available to download from here: `https://github.com/premosa-sbst`

7.4.1.5 Parameter Settings

There are several parameters that need to be configured in PreMOSA. Parameter tuning of search algorithms is a long and expensive process [160]. Arcuri and Fraser [160] showed

that the default parameter values in EvoSuite give reasonable results when compared to tuned parameters. Moreover, Panichella et al. [14] also used these default values in the state-of-the-art DynaMOSA. Therefore, we decide to use the default parameter values used in EvoSuite [89] and DynaMOSA [14] except for the following parameters.

Time Budget: Similar to Chapter 6, we set 2 minutes as the time budget for test generation per class as it is a reasonable time budget in a usual resource constrained environment (see Section 6.3.1.3).

Coverage criteria: Similar to Chapters 5 and 6 (see Sections 5.4.1.3 and 6.3.1.3), we use branch coverage as coverage criterion in PreMOSA as it was shown to be the most effective criterion in terms of detecting bugs when used in EvoSuite [24, 43].

Termination criteria: Similar to Chapter 6 (see Section 6.3.1.3), we use only the maximum time budget as the termination criterion since PreMOSA can increase the chances of detecting bugs by utilising the full time budget.

Test suite minimisation: We disable test suite minimisation since all the test cases in the archive form the final test suite (see Section 7.3.2).

Assertion strategy: Mutation-based assertion filtering can be computationally expensive and can lead to timeouts. Therefore, following a similar approach to previous work [23], we choose all possible assertions as the assertion strategy.

Similar to PreMOSA, we configure the baseline technique, DynaMOSA, as described above.

Finally, following the results of our pilot runs, we use 50 consecutive iterations for the parameter *maximum number of iterations without coverage improvement* (I) in PreMOSA. Furthermore, we configure PreMOSA to add non-buggy targets to the search if it cannot cover any buggy target in the first 25 iterations in the search. For some of the classes, PreMOSA cannot find a test that covers the buggy targets until the trigger is fired to add non-buggy targets to the search. Thus, all the search resources spent until this point are ineffective in terms of detecting bugs. Our preliminary results suggest that for a significant number of classes, PreMOSA covers the first buggy target within the first 25 iterations. Therefore, we decide to add non-buggy targets to the search if PreMOSA fails to cover at least one target after 25 iterations.

7.4.1.6 Experimental Protocol

We run experiments with PreMOSA using 2 instances of simulated defect predictors and DynaMOSA on 420 bugs. For each bug in the Defects4J dataset, we take the buggy version of the project and collect the ground truth labels for the buggy and non-buggy methods. Next, for each of the six projects in Defects4J, we combine all the ground truth labels from the bugs of those projects. For example, for Apache commons-lang project, we combine the labels from all the 59 bugs. Then, we simulate the defect predictor outcomes using the Algorithm 2 for each of the six projects in separate.

As described in Chapter 4, our intended application scenario is generating tests to detect bugs that already exist in the system. Hence, we run test generation on the buggy version of the projects. Since we are measuring the bug detection capability of both approaches only on the Defects4J bugs, we do not run test generation on the non-buggy classes, i.e., classes that are not modified in the bug fixes of the Defects4J bugs.

To take the randomness of SBST into account, we repeat each test generation run 25 times. Due to the randomness of the Defect Prediction Simulation Algorithm, we repeat the simulation runs 5 times for the recall=precision=75% experiments. For each of these simulated defect predictor instances, we repeat test generation runs 5 times. Consequently, we have to run a total of 3 (approaches) $*$ 25 (repetitions) $*$ 482 (buggy classes) = 36,150 test generations. We collect the generated test suites at the end of each test generation run. We evaluate if the 36,150 generated test suites detect the selected Defects4J bugs by following the method described in Section 4.4. Altogether, the experimental evaluation took roughly 48,800 CPU-hours.

The 'run bug detection' script from the test execution framework in Defects4J logs the test cases that produce different test execution results when run against the buggy and fixed versions. We configure both PreMOSA and DynaMOSA to log the time taken to generate each test case since the start of the search (in milliseconds). Figure 7.3 shows a sample test case with the time taken to generate it logged as a comment. Hence, we can find the time taken to generate test cases that detect the bugs by each approach, which will be used to evaluate the efficiency of the two approaches (*RQ2*).

```
@Test(timeout = 4000)
public void test006()  throws Throwable  {
    // time taken = 20971
    try {
      NumberUtils.createNumber("0x");
      fail("Expecting exception: NumberFormatException");

    } catch(NumberFormatException e) {
      //
      // For input string: \"\"
      //
      verifyException("java.lang.NumberFormatException", e);
    }
}
```

FIGURE 7.3: A sample test case with the time taken to generate.

7.4.2 Results

We present the results for our research questions following the method described in Section 7.4. Our main aim is to evaluate if PreMOSA is more effective and efficient than the state-of-the-art DynaMOSA.

RQ1: Is PreMOSA effective at detecting bugs?

As we described in Section 7.4, we perform 25 runs of PreMOSA using defect predictions at recall=precision=75% (PreMOSA-75) and recall=precision=100% (PreMOSA-100), and DynaMOSA against each buggy program in Defects4J (Section 7.4.1.2) and report the bug detection results as boxplots in Figure 7.4. As we can see, both PreMOSA-100 and PreMOSA-75 detect more bugs than DynaMOSA.

We report the means and medians of the number of bugs detected and the results from the statistical analysis in Table 7.1. DynaMOSA detects 197.16 bugs on average in 2 minutes. PreMOSA-100 and PreMOSA-75 outperform DynaMOSA, and detect 213.56 and 212.6 bugs on average, which are average improvements of 16.4 (+8.3%) and 15.44 (+7.8%) more bugs than DynaMOSA, respectively. The differences of the number of bugs detected by PreMOSA-100/PreMOSA-75 and DynaMOSA are statistically significant according to the Mann-Whitney U-Test (p-value <0.0001) with large effect sizes

($\widehat{A}_{12} \geq 0.98$). Thus, we conclude that PreMOSA is significantly more effective than DynaMOSA when using any acceptable defect predictor (i.e., recall, precision $\geq 75\%$).

PreMOSA-75 detects only 0.96 (-0.4%) less bugs on average than PreMOSA-100. According to the one-tailed Mann-Whitney U-Test, this difference is not statistically significant (p-value = 0.5512), and the $\widehat{A}_{12}$ statistic indicates a negligible effect size of 0.53. Therefore, we can confirm PreMOSA successfully accounts for errors in the predictions of defect predictors in the acceptable range.

Certain bugs are harder to detect than others. Similar to Chapter 5, we identify a bug as a unique bug if it is only detected by one approach, i.e., PreMOSA or DynaMOSA (see Section 5.4.2). The number of unique bugs detected by an approach is an indication of the ability of that approach to detect the bugs that are not detected otherwise in the given time budget, which is an important strength given how hard it is to detect a bug [161].

Table 7.2 shows a summary of the bug detection results of PreMOSA and DynaMOSA. PreMOSA-100 and PreMOSA-75 detect 287 and 292 bugs altogether, which is 68.3% and 69.5% of the total bugs respectively, whereas DynaMOSA detects only 280 (66.7%) bugs. PreMOSA-100 detects 17 unique bugs that DynaMOSA cannot detect in any of the runs, whereas DynaMOSA only detects 10 such unique bugs. Similarly, PreMOSA-75 detects 22 unique bugs that are not detected by DynaMOSA, whereas DynaMOSA only detects 10 unique bugs that PreMOSA-75 cannot detect in any of the runs. This shows that PreMOSA is capable of detecting more bugs that are not detected by DynaMOSA.

We find that PreMOSA-100 detects less bugs in total and less unique bugs than PreMOSA-75 when the bugs are isolated in buggy methods with private access modifier (i.e., private buggy methods). For example, PreMOSA-75 detects Closure-25, 50, 55, 57, 67, 68, 143, 154 bugs, which all have only private buggy methods, while PreMOSA-100 detects none of them. PreMOSA-100 starts the search for test cases to cover only the buggy targets. When all the buggy targets are in private methods, PreMOSA-100 has only limited guidance to cover these targets since it cannot directly call the private buggy methods. PreMOSA-100 will get further guidance to cover these targets only after the non-buggy targets are added to the search (line 19 in Algorithm 4). It will be able to indirectly call the buggy targets in private methods through non-buggy methods with non-private access modifier. In contrast, PreMOSA-75 is more likely to start with non-buggy targets

incorrectly predicted as buggy or all likely non-buggy targets (line 7 in Algorithm 4). This means PreMOSA-75 has a better chance of having more guidance to cover buggy targets in private methods from the beginning of the search compared to PreMOSA-100, and as a result, it is able to detect more bugs in total and more unique bugs than PreMOSA-100.

If we consider a bug as detected only if all the 25 runs by an approach detect that bug (i.e., success rate = 1.0), then the number of bugs detected by PreMOSA-100 and PreMOSA-75 becomes 140 and 127 respectively, whereas it is only 114 bugs for DynaMOSA. We further find that PreMOSA-100 detects 108 bugs more times than DynaMOSA, while for DynaMOSA, this is only 62 bugs. Similarly, PreMOSA-75 detects 124 bugs more times than DynaMOSA, whereas it is only 61 bugs for DynaMOSA. Altogether, this demonstrates that PreMOSA is also more robust in detecting bugs when compared to DynaMOSA. More details on the success rates by PreMOSA-100, PreMOSA-75 and DynaMOSA are reported in Appendix D.

TABLE 7.1: Mean and median number of bugs detected by PreMOSA and DynaMOSA in 2 minutes time budget.

	Mean	Median		p-value	$\widehat{A}_{12}$
PreMOSA-100	**213.56**	**213**	PreMOSA-100 vs. DynaMOSA	**<0.0001**	**0.99**
PreMOSA-75	**212.6**	**212**	PreMOSA-75 vs. DynaMOSA	**<0.0001**	**0.98**
DynaMOSA	197.16	197	PreMOSA-100 vs. PreMOSA-75	0.5512	0.53

TABLE 7.2: Summary of the bug detection results at 2 minutes.

	Bugs detected	Bugs detected in every run	Unique bugs
PreMOSA-100	**287**	**140**	**17**
PreMOSA-75	**292**	**127**	**22**
DynaMOSA	280	114	10

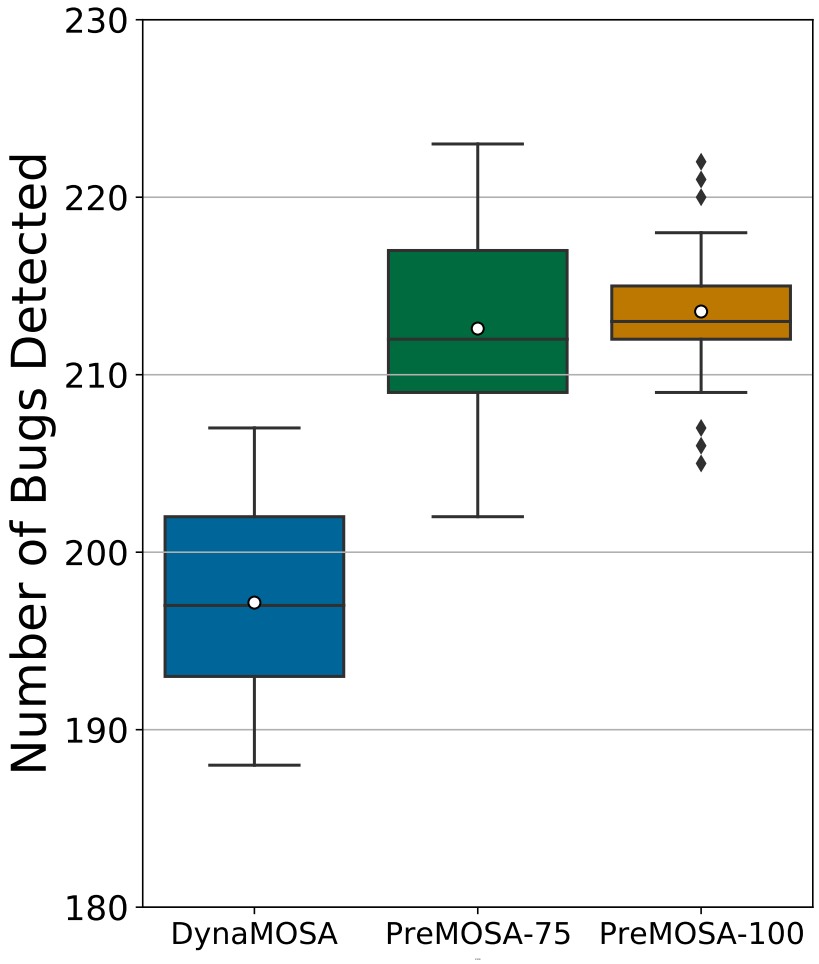

FIGURE 7.4: The number of bugs detected by PreMOSA and DynaMOSA in 2 minutes time budget

In summary, PreMOSA is significantly more effective than the state-of-the-art DynaMOSA with large effect sizes when using any acceptable defect predictor. The superior performance of PreMOSA is supported by both its capability to detect new bugs that are not detected by DynaMOSA and the robustness of the approach.

RQ2: Is PreMOSA efficient at generating test cases that can detect bugs?

As described in Section 7.4, for each approach, we calculate the mean time to generate the first test case that detects each bug. In the case of a bug that is detected by both PreMOSA-100 and DynaMOSA, we then calculate the difference of the mean times to

generate the first bug detecting test by the two approaches, i.e., mean time to generate the first bug detecting test by DynaMOSA - mean time to generate the first bug detecting test by PreMOSA-100. We repeat the same procedure for PreMOSA-75 and DynaMOSA as well. If the difference is positive, that means PreMOSA generates a bug detecting test in a shorter time on average. A negative difference means PreMOSA has a worst performance.

We report the means and medians of the differences of the time taken to generate bug detecting tests and the results from the statistical analysis in Table 7.3. The average difference of mean time to generate bug detecting tests between PreMOSA-100 and DynaMOSA is 2.59 seconds, and it is 2.02 seconds between PreMOSA-75 and DynaMOSA. According to the one-tailed Wilcoxon signed-rank test, these differences are statistically significant with p-values <0.05. However, we find that the effect sizes (i.e., r) estimated using the Wilcoxon signed-rank test are small. The effect size of the difference of mean time to generate bug detecting tests between PreMOSA-100 and DynaMOSA is 0.18, which translates to approximately 60% probability of PreMOSA-100 generating a bug detecting test faster than DynaMOSA [146]. The effect size of 0.11 between PreMOSA-75 and DynaMOSA suggests that PreMOSA-75 generates a bug detecting test faster than DynaMOSA approximately 56% of the time. Therefore, we can conclude PreMOSA is significantly faster than DynaMOSA to generate a bug detecting test when using any acceptable defect predictor.

TABLE 7.3: Mean and median difference of time taken to generate bug detecting tests by PreMOSA and DynaMOSA.

	Mean (s)	Median (s)	p-value	r
PreMOSA-100 vs. DynaMOSA	**2.59**	**0.22**	**0.0016**	0.18
PreMOSA-75 vs. DynaMOSA	**2.02**	**0.05**	**0.0347**	0.11

The above analysis is carried out with respect to the time to generate bug detecting test for each bug that is detected by all the approaches in the comparison. In addition, we also analyse the efficiency of PreMOSA and DynaMOSA with respect to the number of bugs detected over the time budget spent, which includes all the bugs in the dataset.

Figure 7.5 shows the median number of bugs detected by each approach over the time

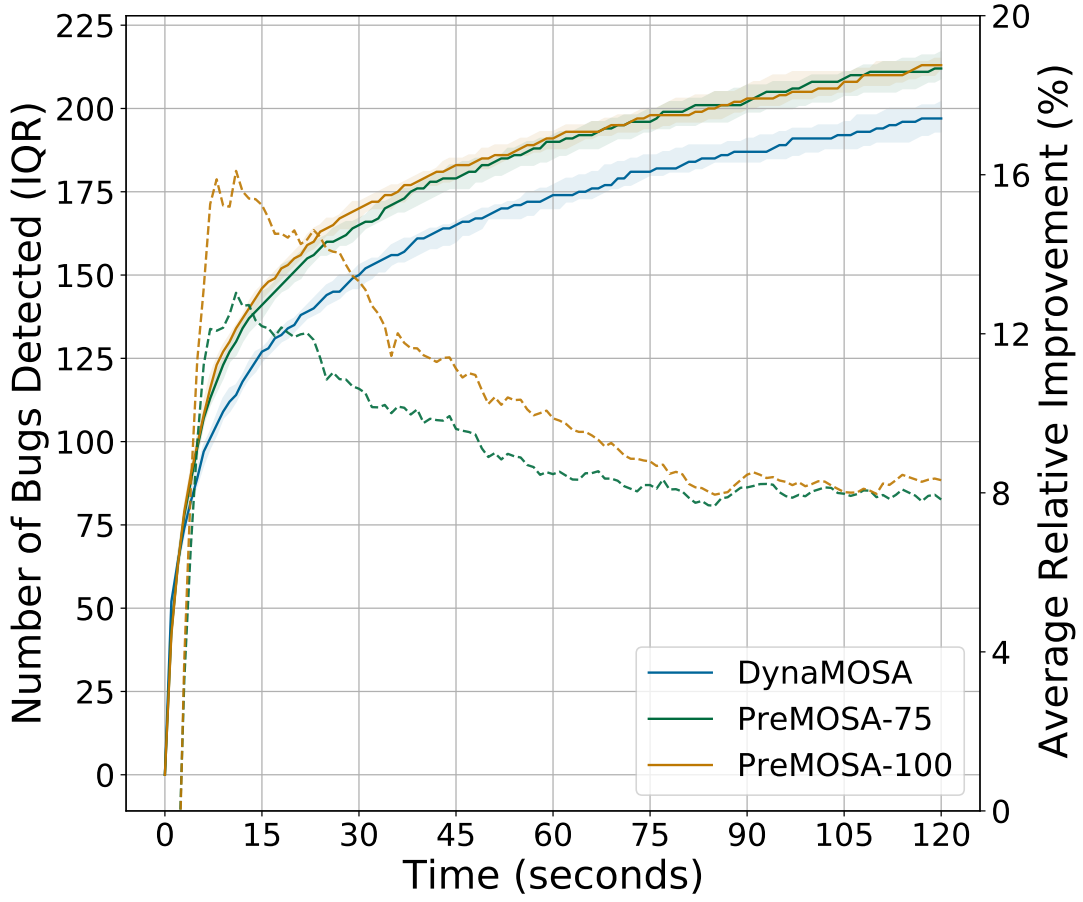

FIGURE 7.5: The number of bugs detected by PreMOSA and DynaMOSA over the time budget spent

budget spent. The number of bugs detected by an approach x ($x \in$ {PreMOSA, DynaMOSA}) at time t ($t \in [0, 120]$) is equal to the number of bugs that can be detected by the tests generated by x after t seconds of test generation. The shaded area around the curves depicts the interquartile range. The dashed lines depict the average improvements of PreMOSA-100 and PreMOSA-75 relative to the baseline DynaMOSA.

In the first 2 seconds, DynaMOSA has a head start, due to the slight additional overhead of PreMOSA in filtering targets and calculating number of independent paths (Sections 7.3.1 and 7.3.3). However, both PreMOSA-100 and PreMOSA-75 outperform DynaMOSA after 2 seconds.

According to the Mann-Whitney U-Test ($\alpha = 0.05$), PreMOSA-100 and PreMOSA-75 detect significantly more bugs than DynaMOSA with large effect sizes ($\widehat{A}_{12} \geq 0.87$) at any time after 3 seconds. This confirms that PreMOSA not only detects more bugs than DynaMOSA at the end of 120 seconds, but also is ahead of DynaMOSA from the

very beginning of the search (i.e., after 2 seconds). More details on the number of bugs detected by PreMOSA-100, PreMOSA-75 and DynaMOSA over the time budget spent are reported in Appendix D.

The relative improvement by PreMOSA using any acceptable defect predictor is much higher when it is given a tight time budget. In particular, the average relative improvements of PreMOSA-100 and PreMOSA-75 reach maximums of 16.1% and 13.0% at 11 seconds respectively. We also find that both PreMOSA-100 and PreMOSA-75 have an average improvement more than 10% in the interval of 6 and 38 seconds. This further demonstrates the increased efficiency of PreMOSA compared to DynaMOSA, such that the large improvements of PreMOSA occur when it is given tight time budgets like in a usual resource constrained scenario.

In summary, PreMOSA is significantly more efficient than the state-of-the-art DynaMOSA with small effect sizes when using any acceptable defect predictor. Overall, PreMOSA not only detects more bugs than DynaMOSA when they are given a reasonably large time budget, but also when they are given tight time budgets like in a resource constrained environment.

RQ3: How does the balanced test coverage affect the coverage and bug detection of SBST?

As we described in Section 7.4, we perform 25 runs of DynaMOSA+b against each buggy program in Defects4J. We report the bug detection results of DynaMOSA+b and DynaMOSA as boxplots in Figure 7.6. As we can see, DynaMOSA+b detects more bugs than DynaMOSA.

We report the means and medians of the number of bugs detected and the results from the statistical analysis in Table 7.4. DynaMOSA+b outperforms DynaMOSA, and detects 203.64 bugs on average, which is an average improvement of 6.48 (+3.3%) more bugs than DynaMOSA. The difference of the number of bugs detected by DynaMOSA+b and DynaMOSA is statistically significant according to the Mann-Whitney U-Test (p-value = 0.0004) with a large effect size ($\widehat{A}_{12} = 0.79$). Thus, we conclude that DynaMOSA+b is significantly more effective than DynaMOSA.

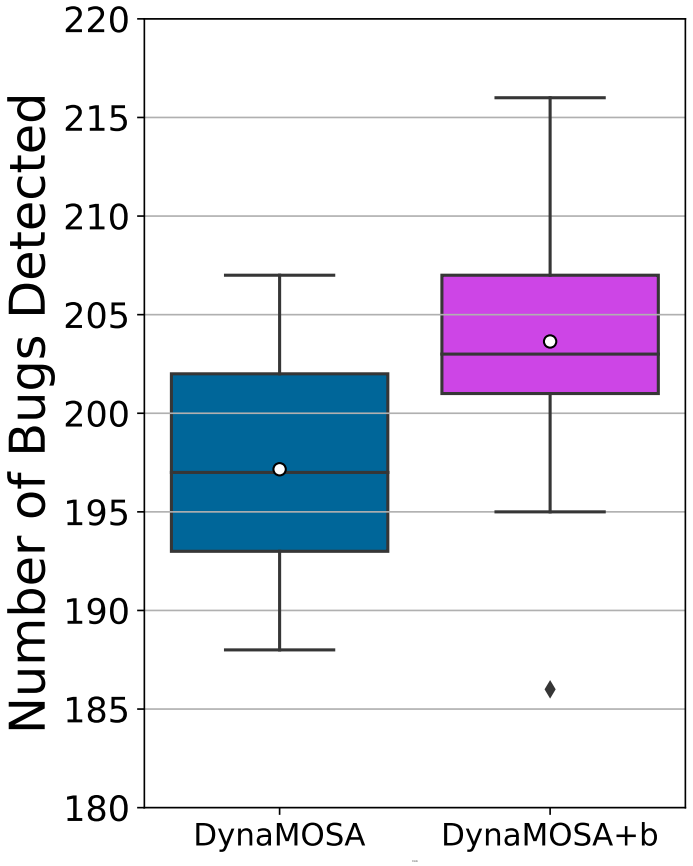

FIGURE 7.6: The number of bugs detected by DynaMOSA+b and DynaMOSA in 2 minutes time budget

TABLE 7.4: Mean and median number of bugs detected by DynaMOSA+b and DynaMOSA in 2 minutes time budget.

	Mean	Median		p-value	$\widehat{A}_{12}$
DynaMOSA+b	**203.64**	**203**	DynaMOSA+b vs. DynaMOSA	**0.0004**	**0.79**
DynaMOSA	197.16	197			

Table 7.5 shows a summary of the bug detection results of DynaMOSA+b and DynaMOSA. As we discussed in *RQ1*, a unique bug is a bug that is detected by only one approach and we consider such bugs are harder to detect. DynaMOSA+b detects 26 unique bugs that DynaMOSA cannot detect in any of the runs, whereas DynaMOSA only detects 14 such unique bugs. This shows that DynaMOSA+b is capable of detecting more bugs that are not detected by DynaMOSA.

If we consider a bug as detected only if all the 25 runs by an approach detect that bug (i.e., success rate = 1.0), then the number of bugs detected by DynaMOSA+b becomes 127, whereas it is only 114 bugs for DynaMOSA. DynaMOSA+b detects 114 bugs more

times than DynaMOSA, while for DynaMOSA, this is only 69 bugs. Altogether, this demonstrates that DynaMOSA+b is also more robust in detecting bugs when compared to DynaMOSA. More details on the success rates by DynaMOSA+b are reported in Appendix E.

TABLE 7.5: Summary of the bug detection results of DynaMOSA+b and DynaMOSA at 2 minutes.

	Unique bugs	Bugs detected in every run	Bugs detected more often
DynaMOSA+b	**26**	**127**	**114**
DynaMOSA	14	114	69

In summary, balanced test coverage significantly improves the bug detection effectiveness of SBST with a large effect size. The improved effectiveness is supported by the capability of SBST with balanced test coverage to detect new bugs that are not detected by SBST without balanced test coverage and the robustness of the approach.

Similar to *RQ2*, we evaluate the efficiency of DynaMOSA+b against DynaMOSA in two ways; i) with respect to the time to generate bug detecting test for each bug that is detected by both approaches, and ii) with respect to the number of bugs detected over the time budget spent, which includes all the bugs in the dataset.

In the first analysis, we calculate the mean time to generate the first bug detecting test by DynaMOSA+b for each bug. For each bug that is detected by both DynaMOSA+b and DynaMOSA, we then calculate the difference of the mean times to generate the first bug detecting test by the two approaches, i.e., mean time to generate the first bug detecting test by DynaMOSA - mean time to generate the first bug detecting test by DynaMOSA+b. If the difference is positive, that means DynaMOSA+b generates a bug detecting test in a shorter time on average. A negative difference means DynaMOSA+b has a worst performance.

We report the means and medians of the differences of the time taken to generate bug detecting tests and the results from the statistical analysis in Table 7.6. The average difference of mean time to generate bug detecting tests between DynaMOSA+b and DynaMOSA is only 0.69 seconds. According to the one-tailed Wilcoxon signed-rank

test, this difference is not statistically significant with a p-value = 0.4890. We cannot reject the null hypothesis, i.e., H_0 = mean time to generate bug detecting tests by DynaMOSA+b is greater than or equal to that of DynaMOSA. Therefore, we assume DynaMOSA+b is not significantly faster than DynaMOSA to generate a bug detecting test.

TABLE 7.6: Mean and median difference of time taken to generate bug detecting tests by DynaMOSA+b and DynaMOSA.

	Mean (s)	Median (s)	p-value	r
DynaMOSA+b vs. DynaMOSA	0.69	-0.02	0.4890	0.002

In the second analysis, we look at the number of bugs detected over the time budget spent considering all the bugs in the dataset. Figure 7.7 shows the median number of bugs detected by DynaMOSA+b and DynaMOSA over the time budget spent. The number of bugs detected by an approach x ($x \in$ {DynaMOSA+b, DynaMOSA}) at time t ($t \in [0, 120]$) is equal to the number of bugs that can be detected by the tests generated by x after t seconds of test generation. The shaded area around the curves depicts the interquartile range. The dashed lines depict the average improvements of DynaMOSA+b relative to DynaMOSA.

Only at the second and third seconds, DynaMOSA detects significantly more bugs than DynaMOSA+b. This is due to the slight additional overhead of DynaMOSA+b in calculating number of independent paths for the method balanced test coverage. From fourth to 25^{th} second, the number of bugs detected by DynaMOSA+b and DynaMOSA is not significantly different. According to the Mann-Whitney U-Test ($\alpha = 0.05$), DynaMOSA+b detects significantly more bugs than DynaMOSA with medium to large effect sizes after the 26^{th} second. This confirms that DynaMOSA+b not only detects more bugs than DynaMOSA at the end of 120 seconds, but also is ahead of DynaMOSA after 26 seconds from the start of the search. More details on the number of bugs detected by DynaMOSA+b over the time budget spent are reported in Appendix E.

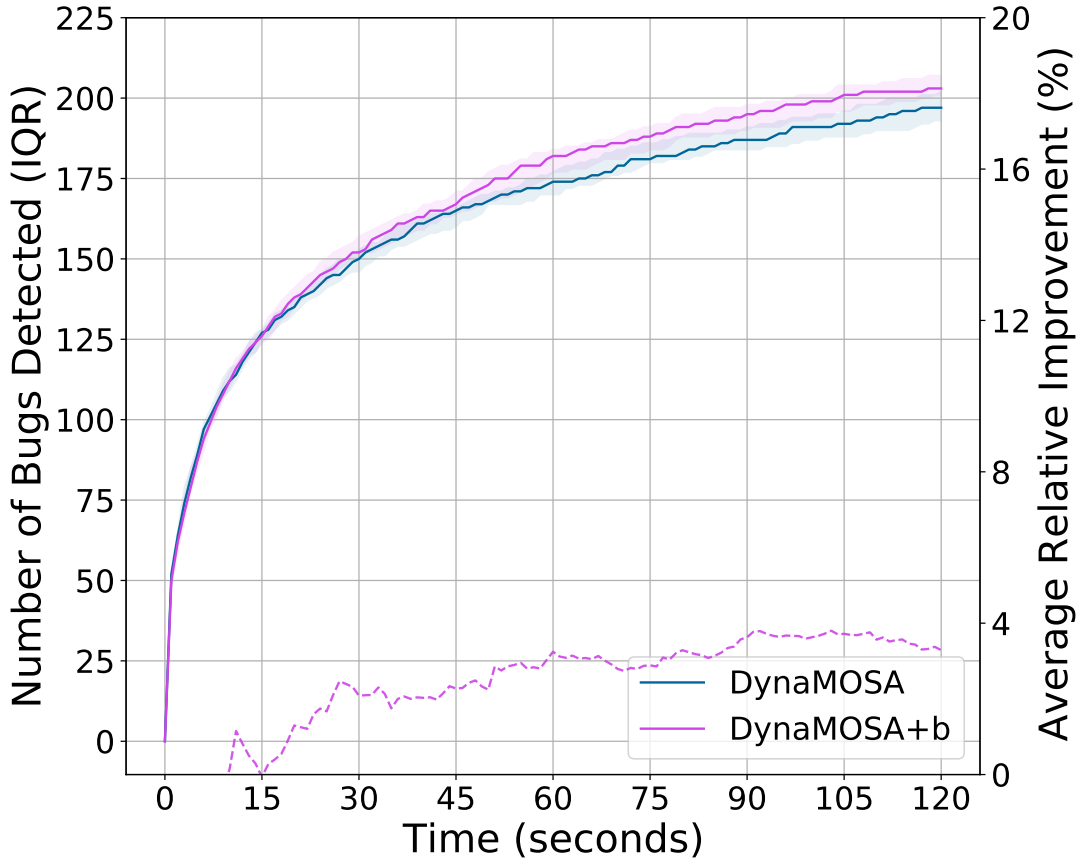

FIGURE 7.7: The number of bugs detected by DynaMOSA+b and DynaMOSA over the time budget spent

In summary, balanced test coverage does not significantly shorten the time taken to generate bug detecting tests. However, SBST with balanced test coverage detects significantly more bugs than SBST without balanced test coverage not only when they are given a reasonably large time budget, but also when they are given fairly tight time budgets like in a resource constrained environment.

As described in Section 7.4, we analyse the effect on coverage by the balanced test coverage method in two ways; i) with respect to branch coverage, and ii) with respect to coefficient of variation of number of tests per an independent path of the branches.

In the first analysis, we measure and compare the branch coverage by DynaMOSA+b and DynaMOSA for each buggy class in Defects4J. According to the two-tailed non-parametric Mann-Whitney U-Test, the branch coverage achieved by DynaMOSA+b and DynaMOSA are significantly different for 93 out of 482 buggy classes. Out of those 93 buggy classes, DynaMOSA+b achieves a significantly higher branch coverage for 65

buggy classes, while DynaMOSA achieves a significantly higher branch coverage for 28 buggy classes.

To check for the effect size of the difference of branch coverage, we calculate the Vargha and Delaney's $\widehat{A}_{12}$ statistic. We categorise the buggy classes, for which DynaMOSA+b and DynaMOSA achieved significantly different branch coverage, according to the size of the effect size, i.e., large, medium and small (described in Section 4.5). As seen in Table 7.7, DynaMOSA+b achieves significantly higher branch coverage with large and medium effect sizes for more buggy classes than DynaMOSA.

TABLE 7.7: Summary of the effect sizes of the differences of branch coverage by DynaMOSA+b and DynaMOSA.

	Large	Medium	Small
DynaMOSA+b	**19**	**41**	5
DynaMOSA	8	13	**7**

In summary, balanced test coverage significantly improves the branch coverage by SBST with large and medium effect sizes for higher number of classes.

In the second analysis, we calculate and compare the coefficient of variation of the number of tests per an independent path by DynaMOSA+b and DynaMOSA for each buggy class in Defects4J. According to the two-tailed non-parametric Mann-Whitney U-Test, the CV of the test suites generated by DynaMOSA+b and DynaMOSA are significantly different for 249 out of 482 buggy classes. Out of those 249 buggy classes, DynaMOSA+b has a significantly smaller CV of number of tests per an independent path for 148 buggy classes, while DynaMOSA has a significantly smaller CV for 101 buggy classes.

To check for the effect size of the difference of CV by DynaMOSA+b and DynaMOSA, we calculate the Vargha and Delaney's $\widehat{A}_{12}$ statistic. We categorise the buggy classes, for which DynaMOSA+b and DynaMOSA have significantly different CV of number of tests per an independent path, according to the size of the effect size, i.e., large, medium and small (described in Section 4.5). As seen in Table 7.8, DynaMOSA+b has a significantly smaller CV of number of tests per an independent path with large effect sizes for more buggy classes than DynaMOSA.

TABLE 7.8: Summary of the effect sizes of the differences of CV of number of tests per an independent path by DynaMOSA+b and DynaMOSA.

	Large	Medium	Small
DynaMOSA+b	**123**	25	0
DynaMOSA	76	25	0

In summary, SBST with balanced test coverage method achieves a significantly better balance of test coverage of targets with large effect sizes for higher number of classes.

7.4.3 Discussion

The execution time of PreMOSA is comprised of the time taken by the defect predictor and the execution time of the search process. With simulated defect predictors, it is not possible to know the execution time of an actual defect predictor. Also, the run-time of an actual defect predictor changes from one model to another model depending on several factors like the classifier used in the model. Therefore, in the experimental evaluation, we do not account for the time taken by the defect predictor, and allocate the full time budget of 2 minutes to the search process. However, we find that PreMOSA with an acceptable defect predictor reaches the final number of bugs detected by DynaMOSA in 79.2 seconds on average. This suggests that even if the defect predictor takes 40.8 seconds to run on average per CUT, PreMOSA will still perform on par with DynaMOSA. Furthermore, the defect predictor used in Chapter 5, Schwa, spent 0.68 seconds per class on average (with a standard deviation of 0.4 seconds). Therefore, the execution time of an actual defect predictor is not expected to affect the conclusions of this study.

We pick the Time-8 bug from the motivating example in Section 7.2 and investigate the bug detection results by PreMOSA and DynaMOSA. PreMOSA-100, PreMOSA-75 and DynaMOSA are able to detect the bug in all the 25 runs within the allocated two minutes time budget. However, DynaMOSA takes 18.6 seconds on average to generate the first bug detecting test, whereas PreMOSA-100 and PreMOSA-75 only take 7.1 and 10.2 seconds on average for the task. This means PreMOSA-100 and PreMOSA-75 reduce the time to generate bug detecting tests by 62% and 45% on average for Time-8 bug. This shows that in a resource constrained environment, when time budgets are

tight, PreMOSA is more likely to detect a bug compared to DynaMOSA which is not guided by defect prediction.

PreMOSA is guided by coverage and defect prediction information. It first attempts to cover the likely buggy targets and starts finding tests to cover likely non-buggy targets once it deems to have searched enough in likely buggy targets. On the other hand, DynaMOSA is only guided by coverage and aims at maximising code coverage. In our experiments, PreMOSA-100, PreMOSA-75 and DynaMOSA achieved 57.89%, 59.14% and 62.94% branch coverage of the classes under test on average, respectively.

In the experimental evaluation, we do not consider additional cost factors such as the effort required to insert test oracles manually or automatically and the execution time of test suites. PreMOSA generates more than one test case for each target in the CUT and retains all these test cases. DynaMOSA is also configured to do the same as described in Section 7.4.1.3. In our experiments, PreMOSA-100, PreMOSA-75 and DynaMOSA generate 12548, 13004 and 14344 test cases on average per test suite, respectively. Both PreMOSA and DynaMOSA are implemented in EvoSuite, which generates assertions in the tests assuming the program under test is correct. EvoSuite uses a mutation-based assertion filtering strategy to minimise the number of assertions in the generated test suites. However, we disable this in our experiments since it can be computationally expensive and can lead to timeouts. Therefore, in the experiments, there are 1,416,817, 1,462,391 and 1,277,024 assertions generated on average per test suite by PreMOSA-100, PreMOSA-75 and DynaMOSA, respectively. In practice, these assertions need to be updated manually or automatically for generated tests to reveal the bugs, which can be problematic when the test suites become large. Appropriate test suite minimisation techniques can be applied to the test suites generated by PreMOSA to mitigate this problem.

For completeness, we report the accuracy and Matthews correlation coefficient (MCC) [151] of the defect predictors used in PreMOSA. For recall=precision=100%, the accuracy of the defect predictor is 100%, and for recall=precision=75%, the accuracy is on average 99.97%. A high accuracy is observed for the defect predictor with recall=precision=75% because of the highly imbalanced nature of the Defects4J dataset, which we discussed in threats to construct validity in Section 4.6. MCC of the recall=precision=100% and recall=precision=75% predictors are 1.0 and 0.75 on average, respectively.

The baseline method, DynaMOSA, does not use a defect predictor and aims to cover all the targets in the CUT equally. This means that in the eyes of DynaMOSA, all the methods in a class are likely buggy, which translates to a 100% recall and precision per project as follows; Lang - 0.06%, Math - 0.03%, Time - 0.05%, Chart - 0.02%, Closure - 0.02% and Mockito - 0.15%.

7.5 Threats to Validity

In this section, we discuss the threats that are specific to this study, in addition to the validity threats discussed in Section 4.6.

Construct Validity. We use the defect prediction simulation algorithm introduced in Chapter 6 (see Section 7.4.1.1). There exists a construct threat to validity from the assumption of uniform distribution of predictions in the defect prediction simulation, which we discussed in Section 6.4.

Internal Validity. To account for the randomness of the simulated defect predictor, we repeat the simulations for 5 times for recall=precision=75% configuration. Then, for each simulation, we repeat the test generation for 5 times to account for the non-deterministic behaviour of PreMOSA. For PreMOSA-100 and DynaMOSA, we repeat the test generation runs for 25 times to account for the non-deterministic behaviour of the two techniques.

External Validity. Our findings may not be generalised to the defect predictors which have recall or precision less than 75%. We experimentally assess the bug detection performance of PreMOSA when using theoretically most conservative and acceptable defect predictors (recall=precision=75%) and ideal defect predictor (recall=precision=100%). The experimental results demonstrate the improved performance of PreMOSA when using either of the defect predictors, which suggest PreMOSA is significantly better at detecting bugs than DynaMOSA when using defect predictors having recall and precision greater than 75%. We choose 75% recall and precision as the lower bound for an acceptable defect predictor with the justification that Zimmermann et al. [48] recommended only the defect predictors having recall and precision more than 75% as acceptable defect predictors.

7.6 Summary

We show that augmenting coverage information with defect prediction information in the search process of SBST improves the bug detection performance of the generated test suites. We develop a many-objective optimisation approach for test generation called predictive many objective sorting algorithm (PreMOSA) that uses buggy methods predictions to decide where to increase the test coverage in the CUT. PreMOSA is equipped with a new method called balanced test coverage to allow the nontrivial targets to have an equal chance of being covered compared to the more trivial targets. We experimentally assess the performance of PreMOSA when using defect predictors having the theoretical upper and lower bound performance of acceptable defect predictors. We validate our technique against 420 labelled bugs from Defects4J dataset. Our experimental evaluation demonstrates that PreMOSA is significantly more effective than the state-of-the-art DynaMOSA with large effect sizes when using any acceptable defect predictor. In particular, it detects 8.3% and 7.8% more labelled bugs on average than DynaMOSA when using an ideal defect predictor and most conservative and acceptable defect predictor, respectively. We also find PreMOSA is significantly more efficient than DynaMOSA.

The performance of PreMOSA does not decrease significantly when replacing the ideal defect predictor (i.e., recall=precision=100%) with the most conservative defect predictor in the acceptable range (i.e., recall=precision=75%). On the other hand, if defect predictions with errors, i.e., false positives and false negatives, are directly used by developers, e.g., in code reviews and manual testing, it can lead to waste of developer time, miss important bugs, etc. [39]. Our results show that PreMOSA successfully accounts for errors in the predictions of defect predictors that are considered acceptable [34].

We find that after 60 seconds of time budget, there is no significant difference in the performances of PreMOSA with an ideal defect predictor and with the most conservative defect predictor. When using PreMOSA, we recommend practitioners to not focus on improving the defect predictor performance beyond 75% recall and precision if their testing resources allow reasonably large time budget for test generation. On the other hand, if there is a tight time budget for test generation, then improving the defect predictor performance would further improve the bug detection performance of PreMOSA. A statistical summary and a statistical comparison of the number of bugs

detected by PreMOSA-100 against PreMOSA-75 over the time budget spent is available in Appendix D.

Chapter 8

Conclusions

The main goal of this thesis is to improve the bug detection capability of search-based software testing (SBST) by incorporating defect prediction information. We devise three research objectives to achieve the main goal and conduct three studies to address them. The contributions made from the three studies demonstrate that defect prediction can successfully be used to guide SBST to effectively and efficiently detect bugs. While defect prediction guidance is beneficial for SBST, we find that the bugs missed by the defect predictor impacts the bug detection performance of SBST significantly. This can be mitigated by designing SBST techniques to handle the potential false negatives in the defect predictions. This chapter presents the findings from our studies along using defect prediction to improve the bug detection performance of SBST (Section 8.1) and impact of defect prediction imprecision on SBST and handling of them (Section 8.2). Figure 8.1, together with Table 8.1, show how the contributions, the findings of the studies, publications, the research objectives, and the main research objective are linked together.

8.1 Using Defect Prediction to Improve the Bug Detection Performance

The primary finding of this thesis is that defect prediction improves the effectiveness and efficiency of SBST in terms of detecting bugs. Modern fast-paced software development

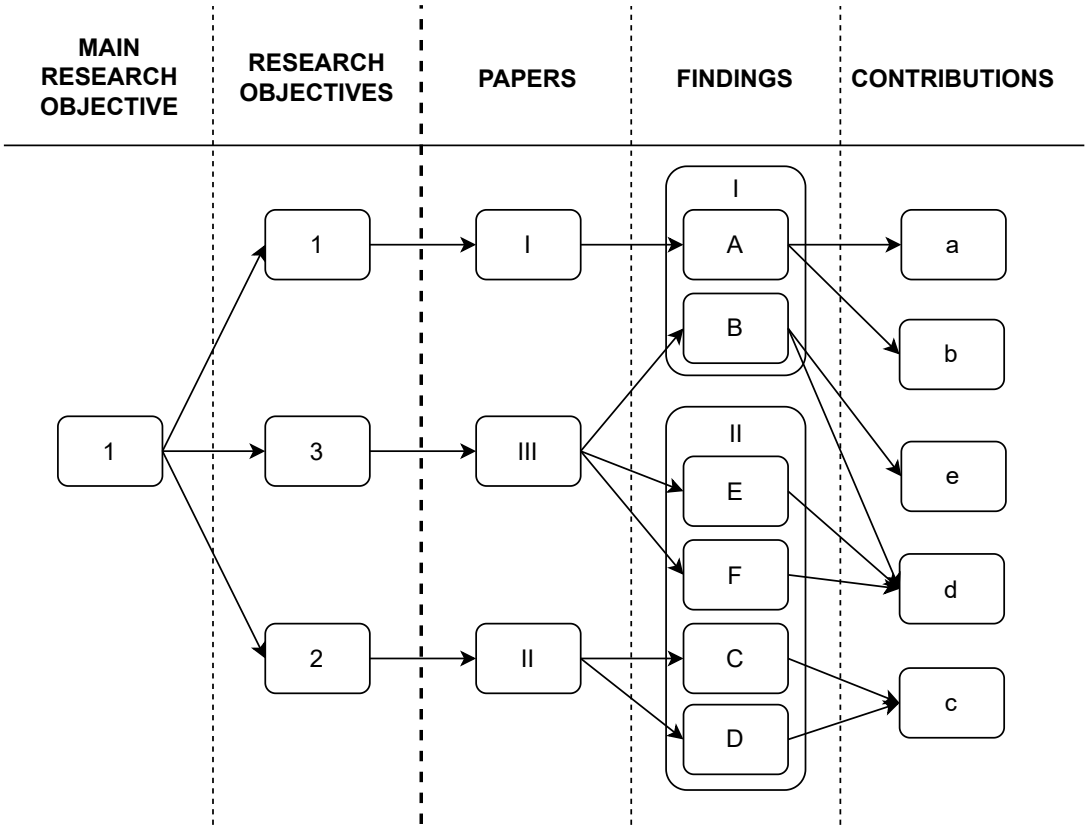

FIGURE 8.1: Overall mapping of the research

practices like agile require fast and frequent feedback from testing. With software systems becoming sophisticated and large, SBST techniques need to optimally utilise the available limited computational resources. When it comes to detecting bugs, the need arises for the SBST techniques to prioritise the search for tests towards the likely defective areas in software to increase the chances of detecting bugs. This thesis demonstrates that class level defect prediction can successfully be used to allocate time budgets to classes for SBST to run test generation with improved bug detection performance and method level defect prediction can successfully be used to guide the search process in SBST along with coverage information to increase the bug detection performance.

Running test generation for a project requires a large time budget since usually the industrial projects are very large containing thousands of classes. Given the usual resource constrained nature in development environments, generating tests for an entire project by allocating a large time budget for every class may not be feasible. For example, if a development team wants to run SBST in the continuous integration (CI) system, then the SBST tool needs to use as minimal resources as possible while maximising the chances of detecting bugs. This is because the CI systems already have high

demands from the existing processes in the system like code quality checks, dynamic analyses, project builds, integration testing and regression testing. In Chapter 5, we introduce defect prediction guided SBST (SBST_{DPG}) which allocates time budgets for classes based on their likelihood of defectiveness as given by a class level defect predictor and runs test generation with SBST. SBST_{DPG} ensures higher time budgets for highly likely to be defective classes at the expense of lower time budgets for the less likely to be defective classes. The results from the experimental evaluation demonstrate that it is beneficial to use class level defect prediction to guide time budget allocation for SBST (SBST_{DPG}) when there are limited resources available to run test generation, for example in CI systems or developer machines.

The existing SBST techniques aim at maximising code coverage and are guided by coverage only. For example, SBST_{DPG} uses the state-of-the-art SBST technique, DynaMOSA, as the SBST module and it treats all the coverage targets in the class are equally important to cover. This is detrimental to the bug detection since there are only a few targets that contain the buggy code and covering a large number of non-buggy targets in the class is likely to be ineffective in terms of detecting bugs. The allocated time budget for the SBST technique needs to be spent more to search for tests to cover buggy targets than the non-buggy ones. In Chapter 7, we introduce predictive many-objective sorting algorithm (PreMOSA) that augments coverage information with defect prediction information in the search process. PreMOSA uses buggy methods predictions to decide where to prioritise and increase the test coverage in the class. Based on the results of the experimental evaluation, we recommend practitioners that it is beneficial to use PreMOSA with a method level defect predictor having an acceptable performance (i.e., recall and precision $\geq 75\%$) in place of an SBST technique only guided by coverage. Our recommendation applies to scenarios where both tight and large time budgets are available in resource constrained environments.

PreMOSA and SBST_{DPG} are orthogonal to each other in terms of how they use defect prediction to guide the search for test cases. In particular, SBST_{DPG} uses class level defect prediction and PreMOSA uses method level defect prediction. Defect prediction is used outside of the search process in SBST_{DPG} and PreMOSA uses defect prediction information inside the search process. SBST_{DPG} allocates time budgets for classes and PreMOSA runs test generation for a class by spending a given time budget. Therefore,

PreMOSA and SBST$_{DPG}$ can be used together by simply replacing the SBST module in SBST$_{DPG}$ with PreMOSA to further improve the performance of them.

In practice, we recommend the practitioners to use PreMOSA and SBST$_{DPG}$ with an oracle automation strategy [143] as a post test generation step to reveal bugs. As we discussed in Section 4.3, the test suites can reveal the bugs once the assertions are checked by an oracle automation strategy. In the absence of such a strategy, practitioners can check the assertions manually, and to reduce the cost of manual work, they can adapt an appropriate test suite reduction technique such as a test suite minimisation or a prioritisation strategy with the number of test cases capped to an acceptable size.

8.2 Impact and Handling of Defect Prediction Imprecision

There is a plethora of defect predictors developed over the past 40 years and they have a wavering performance. A defect predictor that works well for a certain project may not have the same performance when it is applied to another project with different characteristics to the previous. Both SBST and defect prediction researchers need to know the impact of the variation of defect predictor performance on guiding SBST. For SBST researchers, it is important to know which types of errors in predictions need to be handled when using the predictions for guiding SBST. In the context of combining defect prediction and SBST, defect prediction researchers should target to minimise the significantly impactful errors in predictions rather than minimising them all.

In Chapter 6, we study the impact of imprecision in defect prediction on the bug detection performance of SBST and demonstrate that the recall of the defect predictor has a significant impact on the bug detection effectiveness of SBST while the impact of precision is not of practical significance. Based on the results of the study, we recommend SBST researchers to design SBST techniques that handle the potential false negatives in the predictions. One way to do this is to explore the likely non-buggy parts of the program at least with a minimum probability, while prioritising the exploitation of the likely buggy parts. SBST is able to afford a reasonable amount of false positives on its own. For SBST, it is important to be informed of the most of the buggy code even at the expense of an acceptable amount of false positives. If the SBST technique does not handle the potential false negatives, our recommendation to the defect prediction

researchers is to target higher recall while having a sufficiently high precision, instead of trying to elevate both recall and precision. This can be done by defining recall-at-precision measure to 75% and aiming to increase recall while maintaining precision at an acceptable level, i.e., 75%.

PreMOSA, which we introduced in Chapter 7, is designed to handle the false negatives in predictions, i.e., buggy methods that are incorrectly labelled as non-buggy methods. It prioritises exploiting the likely buggy targets and explores the likely non-buggy targets with a lesser priority. The bug detection performance of PreMOSA does not decrease significantly when an ideal defect predictor (i.e., recall=precision=100%) is replaced with the most conservative defect predictor in the acceptable range (i.e., recall=precision=75%), if there is a reasonably large time budget for test generation. This highlights the importance of accounting for errors in the defect predictions. We suggest that when the potential false negatives are handled by the SBST technique (e.g., PreMOSA), it is not necessary to further improve the defect predictor performance beyond an acceptable level, e.g., recall and precision $\geq$ 75%, if the testing resources allow reasonably large time budget for test generation. On the other hand, if there is a tight time budget for test generation, then it can be beneficial to further improve the defect predictor performance for the bug detection performance of the SBST technique.

8.3 Summary

This thesis demonstrates that it is beneficial in terms of bug detection for the SBST approaches to focus the test generation more on the buggy code as guided by defect prediction. The existing SBST approaches are not that effective in terms of bug detection and their bug detection capability has been studied by previous work along the lines of different coverage criteria and time budgets. We identify the gap that SBST performs rather poorly in terms of detecting bugs because there is no guidance for SBST in terms of where the buggy code is. In particular, coverage has always been used as the only guidance for SBST when searching for test cases and we argue coverage alone is not sufficient to guide SBST to detect bugs effectively and efficiently. This thesis investigates the idea of using defect prediction to inform SBST of the likely buggy areas in code in order to improve the bug detection capability of SBST. In conclusion, defect prediction

can be used successfully to guide SBST along with the coverage guidance to detect bugs effectively and efficiently.

Previous research shows that practitioners prefer precise defect predictions when they are used to assist manual tasks like code reviews or testing. Contrary to the existing knowledge, we find that SBST techniques can afford less precise defect predictors. For SBST, it is more important to be informed of the most of the buggy code, even at the expense of a reasonable amount of false alarms. Combining defect prediction and SBST is not straight-forward. Missing to correctly label buggy code is significantly detrimental to the bug detection of SBST. Unless there is a defect predictor that can label nearly all the buggy code correctly at most with an acceptable amount of false alarms, SBST must handle the cases where the defect predictor misses buggy code in order to effectively and efficiently detect bugs. Otherwise, using defect prediction can be a disadvantage for SBST rather than an advantage.

Table 8.1: Summary of the definitions of the main research objective, research objectives, papers, findings, and contributions.

Type	ID	Definition
Main Research Objective	1	Improve the bug detection capability of SBST by incorporating defect prediction information.
Research Objective	1	Develop an approach that allocates time budget to classes for test generation based on defect prediction.
	2	Understand the impact of imprecision in defect prediction for guiding search-based software testing.
	3	Develop an SBST technique that uses defect prediction to guide the search process to likely defective areas.
Paper	I	Defect prediction guided search-based software testing
	II	On the impact of imprecision in defect prediction for guiding search-based software testing
	III	An experimental assessment of using theoretical defect predictors to guide search-based software testing
Finding	I	Using defect prediction to improve the bug detection performance
	A	Use class level defect prediction to allocate time budgets for test generation with SBST such as SBST_{DPG}.
	B	Use method level defect prediction to guide the search process in SBST such as PreMOSA.
	II	Impact and handling of defect prediction imprecision
	C	SBST techniques must handle the potential false negatives in the predictions.
	D	In the context of combining defect prediction and SBST, increase recall while maintaining precision at an acceptable level, e.g., 75%, if the SBST technique does not handle the potential false negatives in the predictions.
	E	If the potential false negatives are handled by the SBST technique (e.g., PreMOSA), then it is beneficial to further improve the defect predictor performance only when there is a tight time budget for test generation.
	F	When there is a reasonably large time budget and SBST handles the potential false negatives (e.g., PreMOSA), do not focus on improving the defect predictor performance beyond an acceptable level, e.g., recall and precision $\geq 75\%$.

Table 8.1: (continued)

Type	ID	Definition
Contribution	a	Class level defect prediction can be used to guide SBST to efficiently and effectively detect bugs through time budget allocation in a resource constrained environment.
	b	A novel time budget allocation approach for SBST to run test generation with improved bug detection performance (SBST_{DPG}).
	c	The recall of the defect predictor has a significant impact on the bug detection performance of SBST and the impact of the precision is not practically significant.
	d	Method level defect prediction can be used to guide the search process in SBST along with coverage information to improve the bug detection performance of SBST.
	e	A novel SBST technique that augments defect prediction information with coverage information to guide the search process towards buggy areas in the class under test with improved bug detection performance of SBST (PreMOSA).

Chapter 9

Future Work

It would be interesting for the defect prediction research community to investigate improving the defect predictor performance by using the tests generated by SBST as feedback for the defect predictor. In particular, a feedback loop can be designed with defect prediction and SBST, so that the defect prediction model is reconstructed (e.g., retrained) with the outcomes (i.e., detects a bug or does not detect any bugs) of the tests generated by SBST until a stopping criteria is met (i.e., defect predictor performance saturates or reaches an expected level of performance).

We introduce a method called balanced test coverage to ensure all the coverage targets receive an equitable test coverage. Since there is no information to further differentiate targets within a method, the balanced test coverage method currently follows a binary approach. In particular, it switches-off targets if they have higher test coverage and switches-on otherwise. There is the potential to further guide the search for tests to interesting targets within a method by parameterising balanced test coverage. For example, it can switch-on and off targets with some probabilities rather than a binary approach. More recently, Wattanakriengkrai et al. [27] introduced a defect predictor that gives predictions at line level, LINE-DP, and it was shown to achieve a 61% recall with however a very low precision. A possible direction for future work is to guide the parameterised balanced test coverage method by using LINE-DP.

We use Defects4J dataset as the benchmark subjects to conduct the experimental evaluations in the thesis. It contains 438 bugs that are from manually validated bug fixes

from six real-world open source Java projects. We identify validating the proposed approaches in this thesis against other bugs datasets [139–141] as future work to support the external validity of our findings and increase the generalisability. The source code of the proposed approaches and replication packages are made available to enable other researchers to easily replicate our research work.

Generating more than one test case to cover each of the coverage targets is highly beneficial for bug detection. This can also increase the test suite size and as a result the cost of inserting oracles manually or automatically and the execution time of test suites can go high. To reduce this cost, appropriate test suite minimisation techniques can be adapted. A potential direction for future work could be to use defect prediction to complement existing test suite minimisation strategies, so that the test suite size is reduced without significantly losing the bug detection performance of the original test suites.

Defect prediction guided SBST approaches, e.g., SBST_{DPG} and PreMOSA, have the potential to provide assistance to developers when they manually inspect the defect predictions or to completely substitute the manual step. One of main limitations in defect prediction is that they do not explain why a certain entity is flagged as buggy, hence it can be a time consuming task for the developers to manually examine the likely buggy entity to find the bug. Another limitation is the false positives, which can cause the developers to waste their precious time. The test suites generated by SBST (containing bug detecting tests) can be handed down to developers with the predictions to make it easier for them to find and identify the root cause of the bugs. This may mitigate the explainability issue of defect predictions as the bug detecting tests can expose the buggy behaviour of the likely buggy entity. The developers are shielded from the effect of false positives since SBST uses the predictions directly and handles them. We identify investigating the potential benefits of defect prediction guided SBST approaches for the humans involved in defect prediction as future work to extend this research.

The concept of using defect prediction to guide SBST can be extended to other automated test generation techniques as well. For instance, the time budget allocation approach proposed in Chapter 5 can be used with random search [4] or dynamic symbolic execution [169] techniques instead of the SBST technique as they can benefit from

larger time budgets. We identify studying the bug detection improvements of other test generation techniques guided by defect prediction as future work.

Appendix A

Time Budget Allocation

A.1 Distribution of time spent by Schwa and BADS

Figure A.1 shows the distribution of the time spent per class by Schwa and BADS for the bugs in Defects4J. The mean time spent by Schwa and BADS per class is 0.68 seconds and the standard deviation is 0.4 seconds.

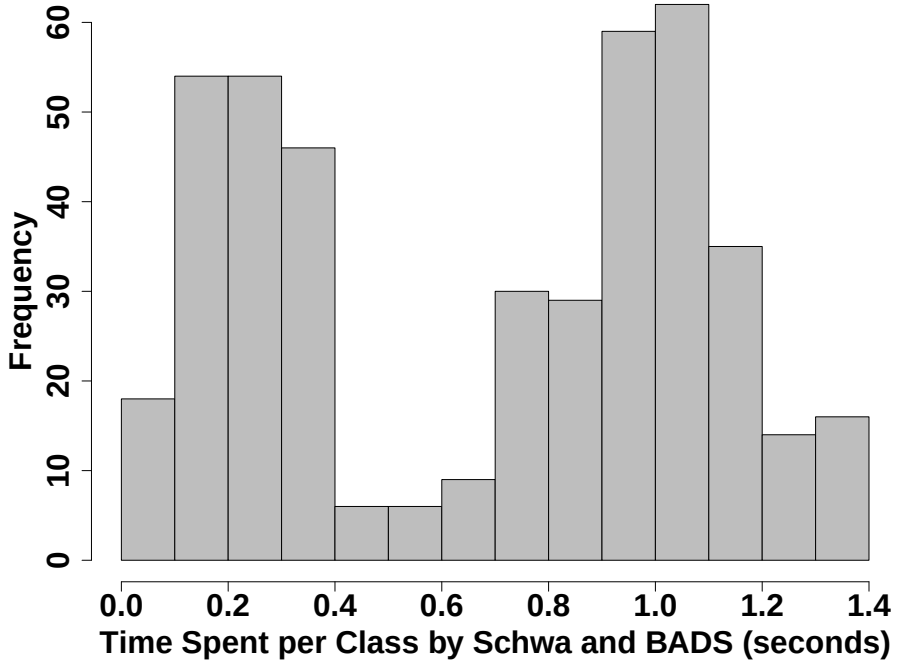

FIGURE A.1: Distribution of the time spent per class by Schwa and BADS for the bugs in Defects4J.

A.2 Bug detection performance comparison of SBST$_{noDPG}$ and SBST$_O$

In Chapter 5, DynaMOSA is configured to generate more than one test case for each target in the class under test (CUT), retain all these test cases and disable test suite minimisation (see Section 5.4.1.3). In Section 5.4.2, we investigate the benefit of configuring DynaMOSA this way by comparing the bug detection results of DynaMOSA with the configuration as outlined in Section 5.4.1.3 (SBST$_{noDPG}$) and DynaMOSA with test suite minimisation (SBST$_O$). In this appendix, we report the bug detection results of SBST$_{noDPG}$ and SBST$_O$ as boxplots (Figure A.2), the statistical summary and the results of the statistical tests (Table A.1), overview of the success rates (Table A.2) and summary of the bug detection results (Table A.3).

TABLE A.1: Mean and median number of bugs detected by SBST$_{noDPG}$ and SBST$_O$ against different total time budgets.

T (s)	Mean		Median		p-value	$\widehat{A}_{12}$
	SBST$_{noDPG}$	SBST$_O$	SBST$_{noDPG}$	SBST$_O$		
$15 * N$	**133.95**	85.75	**134.0**	85.0	**<0.0001**	**1.00**
$30 * N$	**166.9**	93.45	**167.5**	92.5	**<0.0001**	**1.00**

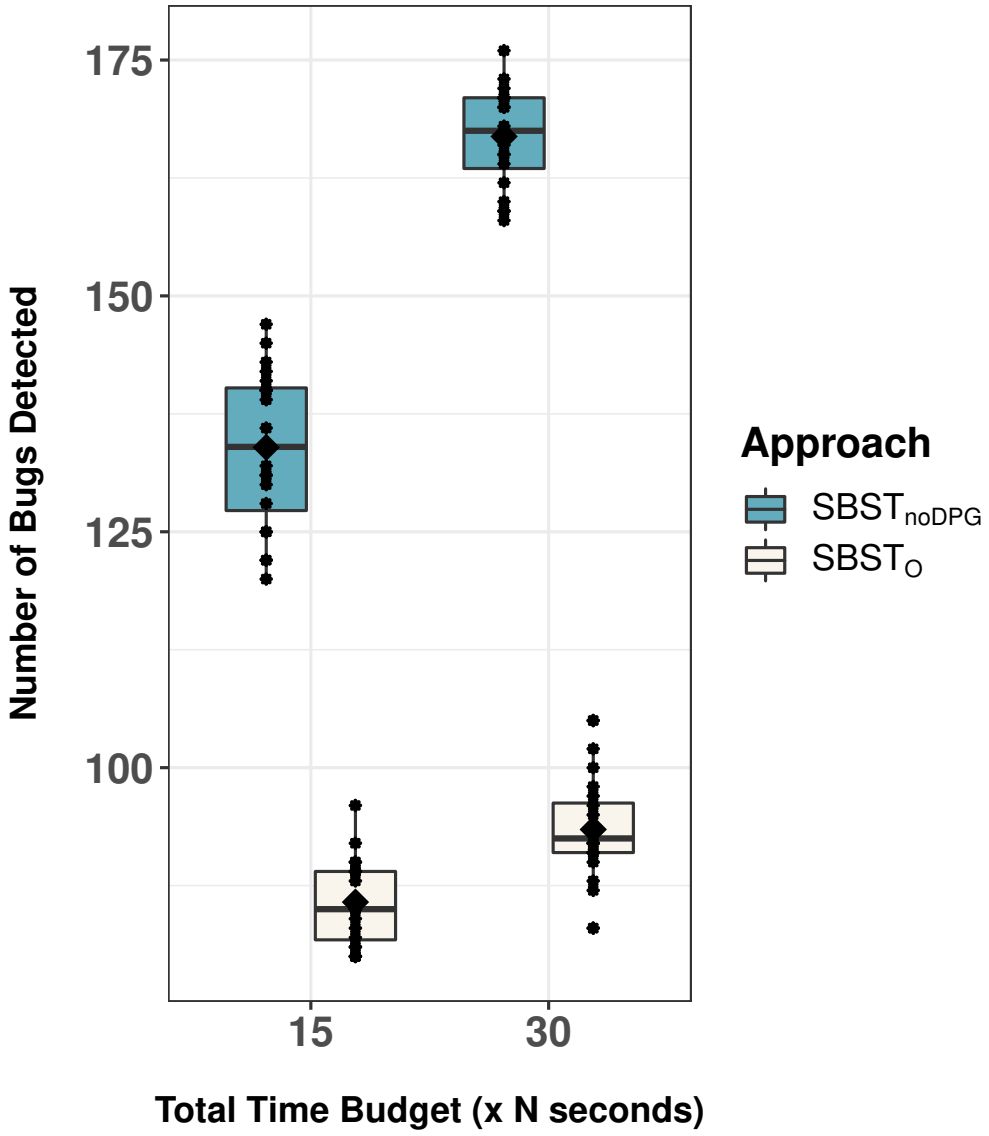

FIGURE A.2: The number of bugs detected by $SBST_{noDPG}$ and $SBST_O$ against different total time budgets

TABLE A.2: Success rate for $SBST_{noDPG}$ and $SBST_O$ at $15 * N$ total time budget. Bug IDs that were detected by only one approach are highlighted with different colours; $SBST_{noDPG}$ and $SBST_O$.

Bug ID	$SBST_{noDPG}$	$SBST_O$	Bug ID	$SBST_{noDPG}$	$SBST_O$
Lang-1	0.45	0.05	Math-5	0.95	0.95
Lang-4	1	0.3	Math-6	1	1
Lang-5	0.2	0	Math-9	0.6	0.05
Lang-7	1	1	Math-11	1	0.55
Lang-8	0.1	0	Math-14	1	1
Lang-9	1	1	Math-16	0.05	0
Lang-10	0.8	0.15	Math-21	0.45	0.1
Lang-11	0.95	0.75	Math-22	1	1
Lang-12	0.8	0.65	Math-23	0.8	0.6
Lang-18	0.3	0.5	Math-24	0.85	0.8
Lang-19	0.7	0.65	Math-26	1	0.05
Lang-20	0.4	0.1	Math-27	0.65	0.1
Lang-21	0.1	0	Math-29	1	0.45
Lang-22	0.8	0.5	Math-32	1	0.8
Lang-23	0.95	0.1	Math-33	0.35	0.1
Lang-27	0.75	0.05	Math-35	1	1
Lang-28	0.05	0	Math-36	0.1	0
Lang-32	1	1	Math-37	1	0.85
Lang-33	1	0.5	Math-40	0.95	0.8
Lang-34	0.9	0.4	Math-41	0.4	0
Lang-35	0.3	0.8	Math-42	0.95	0.95
Lang-36	1	0.6	Math-43	0.55	0
Lang-37	0.2	0.05	Math-45	0.3	0
Lang-39	0.95	0.65	Math-46	1	1
Lang-41	1	0.55	Math-47	0.95	1
Lang-44	0.65	0.2	Math-48	0.75	0.1
Lang-45	1	0.95	Math-49	0.75	0.3
Lang-46	1	0.8	Math-50	0.3	0.2
Lang-47	0.9	1	Math-51	0.25	0.1
Lang-49	0.4	0	Math-52	0.6	0
Lang-50	0.3	0	Math-53	1	0.85
Lang-51	0.05	0.05	Math-55	1	0.5
Lang-52	1	0.55	Math-56	0.9	0.85
Lang-53	0.15	0	Math-59	1	1
Lang-54	0.05	0.05	Math-60	0.95	0.15
Lang-57	1	0.15	Math-61	1	1
Lang-58	0.05	0.15	Math-63	0.4	0.4
Lang-59	0.95	0.55	Math-65	0.25	0.45
Lang-60	0.3	0.1	Math-66	1	1
Lang-61	0.25	0.1	Math-67	1	1
Lang-63	0	0.05	Math-68	1	0
Lang-65	0.95	0.75	Math-70	1	0.15
Math-1	1	0.85	Math-71	0.35	0.4
Math-2	0.1	0	Math-72	0.45	0.15
Math-3	1	0.2	Math-73	1	0.9
Math-4	1	0.6	Math-75	0.9	0

TABLE A.2: (continued)

Bug ID	$SBST_{noDPG}$	$SBST_{O}$	Bug ID	$SBST_{noDPG}$	$SBST_{O}$
Math-76	0.05	0.05	Chart-5	1	0.85
Math-77	1	1	Chart-6	1	0
Math-78	0.6	0.1	Chart-7	0.25	0
Math-79	0.05	0	Chart-8	1	0.3
Math-80	0	0.1	Chart-10	1	0.4
Math-81	0	0.2	Chart-11	1	0.25
Math-83	1	0.45	Chart-12	0.5	0
Math-85	1	0.75	Chart-13	0.2	0.1
Math-86	0.85	0.45	Chart-14	1	1
Math-87	1	0.9	Chart-15	0.9	0.95
Math-88	0.7	0.3	Chart-16	1	0.8
Math-89	1	0.85	Chart-17	1	1
Math-90	1	0.5	Chart-18	1	1
Math-92	1	1	Chart-19	0.15	0.45
Math-93	0.25	0.2	Chart-20	0.1	0
Math-95	1	0.65	Chart-21	0.05	0.05
Math-96	1	0.6	Chart-22	1	1
Math-97	1	0.8	Chart-23	1	0.3
Math-98	0.85	0.35	Chart-24	1	0.8
Math-100	1	0.45	Mockito-2	1	1
Math-101	1	0.15	Mockito-17	1	0.15
Math-102	0.5	0.6	Mockito-29	0.95	0.35
Math-103	1	0.85	Mockito-35	1	1
Math-104	0.4	0.25	Closure-7	0.1	0.35
Math-105	1	0.15	Closure-9	0.15	0
Time-1	1	1	Closure-12	0.1	0
Time-2	1	0.65	Closure-19	0.1	0
Time-3	0.05	0	Closure-21	0.35	0.2
Time-4	0.3	0.2	Closure-22	0.5	0.75
Time-5	1	0.6	Closure-26	0.4	0.7
Time-6	0.8	0.25	Closure-27	0.1	0.1
Time-8	0.7	0.4	Closure-28	1	0.75
Time-9	1	0.85	Closure-30	0.95	0.9
Time-10	0.1	0	Closure-33	0.5	0.4
Time-11	1	0.8	Closure-39	0.6	0.65
Time-12	0.55	0.2	Closure-46	1	1
Time-13	0.05	0.15	Closure-49	0.5	0.1
Time-14	0.95	0.25	Closure-52	0.1	0.1
Time-15	0.3	0	Closure-54	0.8	0.9
Time-17	0.55	0.05	Closure-56	1	1
Time-22	0.25	0	Closure-65	0.45	0
Time-23	0.2	0	Closure-72	0.3	0
Time-24	0.45	0.05	Closure-73	1	0.6
Time-26	0.05	0	Closure-77	0.25	0.05
Time-27	0.5	0.1	Closure-79	0.85	0.2
Chart-1	0.05	0.05	Closure-81	0	0.05
Chart-3	0.15	0.1	Closure-82	1	0.85
Chart-4	0.3	0.3	Closure-100	0	0.15

TABLE A.2: (continued)

Bug ID	$SBST_{noDPG}$	$SBST_O$	Bug ID	$SBST_{noDPG}$	$SBST_O$
Closure-104	0.5	1	Closure-140	0.25	0.35
Closure-106	0.95	0.9	Closure-141	0	0.05
Closure-108	0.2	0	Closure-144	0.1	0.05
Closure-110	1	1	Closure-148	0	0.35
Closure-113	0.05	0	Closure-150	0.1	0.1
Closure-114	0.1	0	Closure-151	1	0
Closure-115	0.25	0	Closure-160	0.05	0
Closure-116	0.1	0	Closure-164	0.45	0.3
Closure-117	0.05	0	Closure-165	0.8	1
Closure-119	0	0.05	Closure-167	0	0.05
Closure-120	0.1	0	Closure-169	0.05	0
Closure-121	0.2	0	Closure-170	0.2	0
Closure-123	0.1	0	Closure-171	0.05	0
Closure-128	0.1	0	Closure-172	0.15	0
Closure-129	0.05	0	Closure-173	0.5	0
Closure-131	0.9	0	Closure-174	1	0.9
Closure-137	1	0.3	Closure-175	0.15	0
Closure-139	0.05	0	Closure-176	0.1	0

TABLE A.3: Summary of the bug detecting results of $SBST_{noDPG}$ and $SBST_O$ at T = $15 * N$.

	Bugs detected	Unique bugs	Bugs detected in every run	Bugs detected more often
$SBST_{noDPG}$	**215**	**54**	**76**	**160**
$SBST_O$	170	9	28	28

Appendix B

Impact of Defect Predictor Imprecision

B.1 MCC of the defect prediction configurations

As we discussed in Section 4.6, for completeness, Table B.1 reports MCC (calculated as in Equation. (4.7)) of the 12 defect prediction configurations used in Chapter 6.

TABLE B.1: MCC of each defect prediction configuration.

Recall (%)	Precision (%)	MCC
100	100	1.0
	75	0.87
95	100	0.98
	75	0.85
90	100	0.95
	75	0.83
85	100	0.93
	75	0.80
80	100	0.90
	75	0.78
75	100	0.87
	75	0.75

B.2 Bugs excluded from Defects4J dataset

Table B.2 shows the bugs that are excluded from the Defects4J dataset in the experiments in Chapters 6 and 7 with the reasons to remove them.

TABLE B.2: Reasons for removing bugs from the dataset.

Bug ID	Reason
Lang-2	Deprecated
Lang-23	No buggy methods
Lang-25	No buggy methods
Lang-30	EvoSuite generates uncompilable tests
Lang-56	No buggy methods
Lang-63	EvoSuite generates uncompilable tests
Math-12	No buggy methods
Math-104	No buggy methods
Time-11	No buggy methods
Time-21	Deprecated
Chart-23	No buggy methods
Closure-15	No buggy methods
Closure-28	No buggy methods
Closure-63	Deprecated
Closure-93	Deprecated
Closure-111	No buggy methods
Mockito-26	No buggy methods

B.3 A statistical summary of the bug detection by SBST guided by DP

Table B.3 reports the number of bugs detected by SBST guided by DP when using the 12 defect predictor configurations studied in Chapter 6.

B.4 Results of the normality tests

Table B.4 reports the results of the Kolmogorov-Smirnov test for normality of the distributions of the number of bugs detected by SBST guided by DP when using 12 defect predictor configurations.

TABLE B.3: A statistical summary of the number of bugs detected by SBST guided by DP when using defect predictors with different recall and precision.

Recall (%)	Precision (%)	Mean	Median
100	100	203.96	204
	75	205.16	205
95	100	200.36	200
	75	201.72	200
90	100	191.4	192
	75	191.52	192
85	100	180.76	182
	75	186	187
80	100	176.48	177
	75	178.08	178
75	100	166.16	167
	75	168.08	169

TABLE B.4: The results of the Kolmogorov-Smirnov test for normality of the distributions ($\alpha = 0.05$) of the number of bugs detected for each combination of the groups of recall and precision.

Recall (%)	Precision (%)	Statistic	p-value	Conclusion
100	100	0.1328	0.7616	Do not reject H_0
	75	0.1192	0.8694	Do not reject H_0
95	100	0.1709	0.4153	Do not reject H_0
	75	0.1899	0.2908	Do not reject H_0
90	100	0.0881	0.9901	Do not reject H_0
	75	0.1147	0.8974	Do not reject H_0
85	100	0.2266	0.1307	Do not reject H_0
	75	0.1340	0.7484	Do not reject H_0
80	100	0.0972	0.9723	Do not reject H_0
	75	0.1189	0.8712	Do not reject H_0
75	100	0.1399	0.6881	Do not reject H_0
	75	0.1319	0.7705	Do not reject H_0

Figure B.1 shows the Q-Q plots of the distributions of the number of bugs detected by SBST guided by DP for 12 defect predictor configurations.

B.4.1 Bugs having only one buggy method

Table B.5 reports the results of the Kolmogorov-Smirnov test for normality of the distributions of the number of bugs detected by SBST guided by DP for the six groups of recall. Only the bugs that have one buggy method are considered.

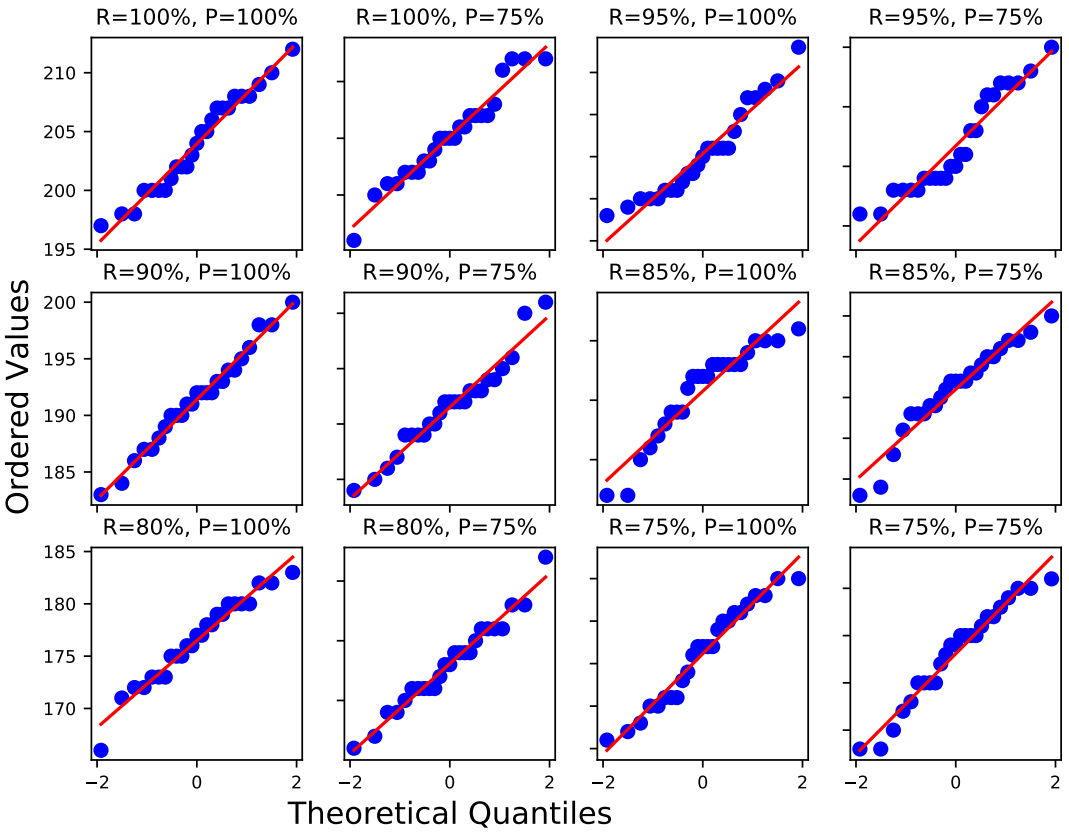

FIGURE B.1: Q-Q plots of the distributions of the number of bugs detected for each combination of the groups of recall and precision. R = Recall and P = Precision.

TABLE B.5: The results of the Kolmogorov-Smirnov test for normality of the distributions ($\alpha = 0.05$) of the number of bugs detected for the groups of recall. For the bugs that have one buggy method.

Recall (%)	Statistic	p-value	Conclusion
100	0.1090	0.5685	Do not reject H_0
95	0.1427	0.2372	Do not reject H_0
90	0.0985	0.7116	Do not reject H_0
85	0.1255	0.3812	Do not reject H_0
80	0.0835	0.8770	Do not reject H_0
75	0.0893	0.8206	Do not reject H_0

Figure B.2 shows the Q-Q plots of the distributions of the number of bugs detected by SBST guided by DP for the six groups of recall. Only the bugs that have one buggy method are considered.

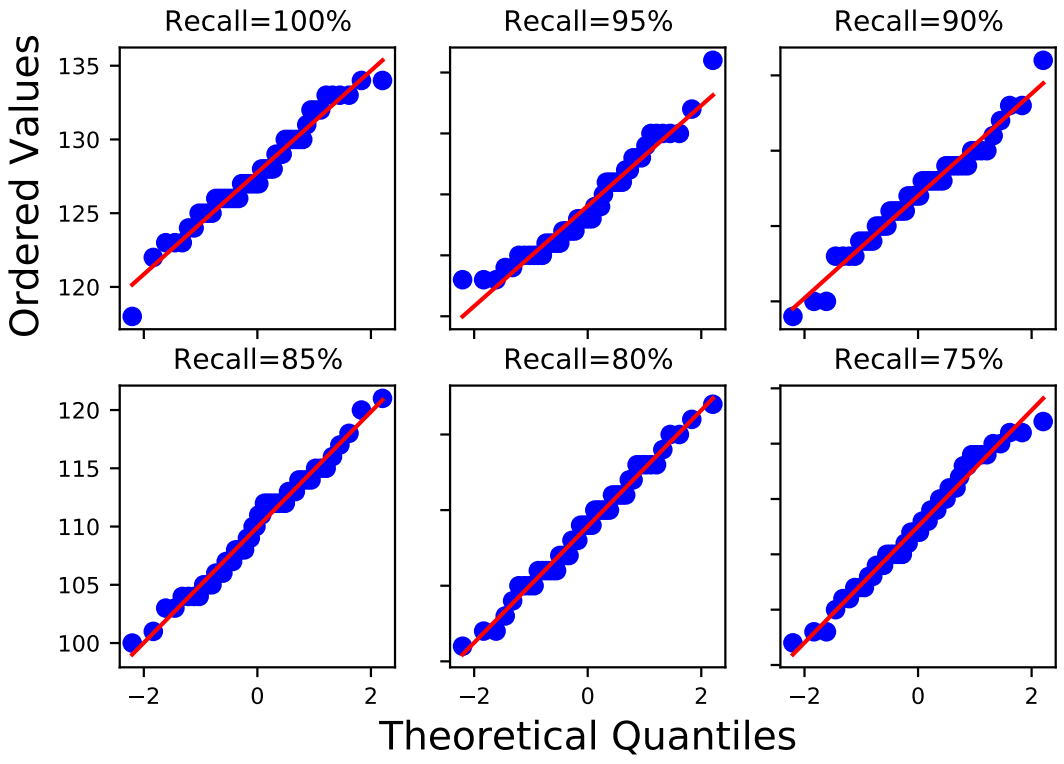

FIGURE B.2: Q-Q plots of the distributions of the number of bugs detected for the groups of recall. For the bugs that have one buggy method.

B.4.2 Bugs having more than one buggy method

Table B.6 reports the results of the Kolmogorov-Smirnov test for normality of the distributions of the number of bugs detected by SBST guided by DP for the six groups of recall. Only the bugs that have more than one buggy method are considered.

TABLE B.6: The results of the Kolmogorov-Smirnov test for normality of the distributions ($\alpha = 0.05$) of the number of bugs detected for the groups of recall. For the bugs that have more than one buggy method.

Recall (%)	Statistic	p-value	Conclusion
100	0.1885	0.0499	Reject H_0
95	0.1366	0.2826	Do not reject H_0
90	0.1166	0.4758	Do not reject H_0
85	0.1250	0.3864	Do not reject H_0
80	0.1685	0.1039	Do not reject H_0
75	0.1360	0.2871	Do not reject H_0

Figure B.3 shows the Q-Q plots of the distributions of the number of bugs detected by SBST guided by DP for the six groups of recall. Only the bugs that have more than one buggy method are considered.

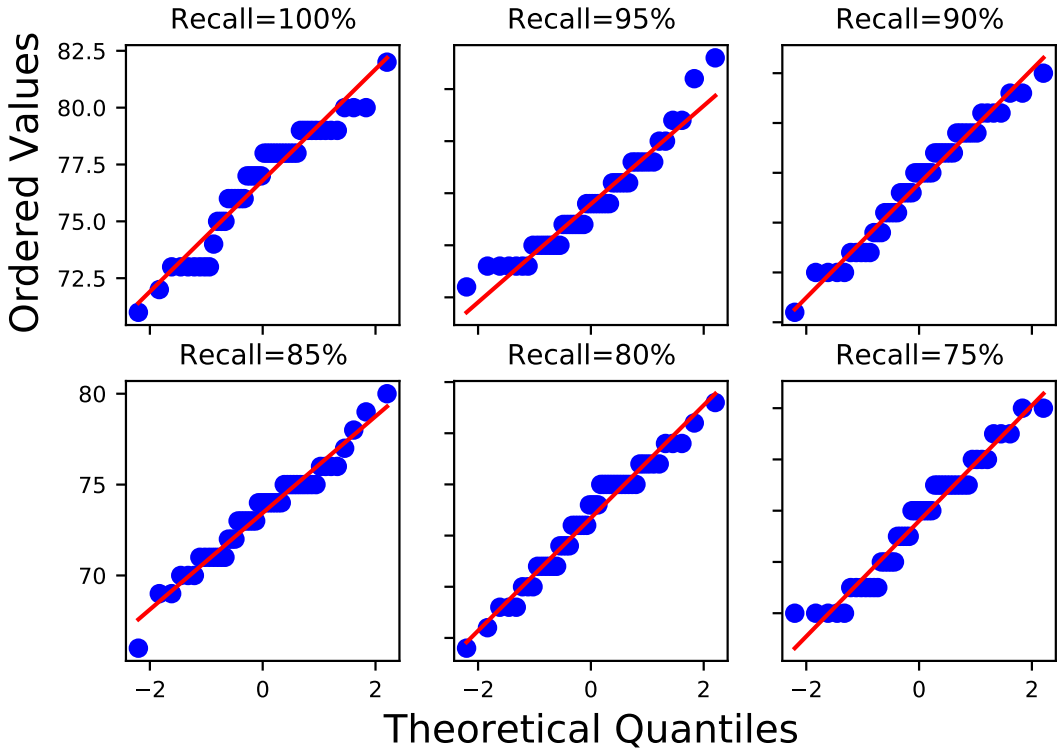

FIGURE B.3: Q-Q plots of the distributions of the number of bugs detected for the groups of recall. For the bugs that have more than one buggy method.

B.5 Results of the Tukey post-hoc test

Table B.7 reports the results of the Tukey's Honestly-Significant-Difference test with the Cohen's d effect sizes for all possible pairs of defect predictor configurations.

B.6 Results of the Games-Howell post-hoc test

Tables B.8 and B.9 report the results of the Games-Howell post-hoc test with the Cohen's d effect sizes for all possible pairs of recalls for the bugs having only one buggy method and the bugs having more than one buggy method, respectively.

TABLE B.7: The results of the Tukey's Honestly-Significant-Difference test with the Cohen's d effect sizes for all possible pairs. Diff is the difference in means. Lower and Upper denote the 95% family-wise confidence levels. R = Recall and P = Precision.

	Diff	Lower	Upper	p-value adj.	Cohen's d
Recall					
R=95%-R=100%	-3.52	-6.13	-0.91	0.0018	0.77
R=90%-R=100%	-13.10	-15.71	-10.49	<0.0001	2.89
R=85%-R=100%	-21.18	-23.79	-18.57	<0.0001	4.67
R=80%-R=100%	-27.28	-29.89	-24.67	<0.0001	6.01
R=75%-R=100%	-37.44	-40.05	-34.83	<0.0001	8.25
R=90%-R=95%	-9.58	-12.19	-6.97	<0.0001	2.11
R=85%-R=95%	-17.66	-20.27	-15.05	<0.0001	3.89
R=80%-R=95%	-23.76	-26.37	-21.15	<0.0001	5.23
R=75%-R=95%	-33.92	-36.53	-31.31	<0.0001	7.47
R=85%-R=90%	-8.08	-10.69	-5.47	<0.0001	1.78
R=80%-R=90%	-14.18	-16.79	-11.57	<0.0001	3.12
R=75%-R=90%	-24.34	-26.95	-21.73	<0.0001	5.36
R=80%-R=85%	-6.10	-8.71	-3.49	<0.0001	1.34
R=75%-R=85%	-16.26	-18.87	-13.65	<0.0001	3.58
R=75%-R=80%	-10.16	-12.77	-7.55	<0.0001	2.24
Precision					
P=75%-P=100%	1.91	0.87	2.94	0.0003	0.42
Recall:Precision					
R=95%:P=100%-R=100%:P=100%	-3.60	-7.83	0.63	0.1850	-
R=90%:P=100%-R=100%:P=100%	-12.56	-16.79	-8.33	<0.0001	2.77
R=85%:P=100%-R=100%:P=100%	-23.20	-27.43	-18.97	<0.0001	5.11
R=80%:P=100%-R=100%:P=100%	-27.48	-31.71	-23.25	<0.0001	6.05
R=75%:P=100%-R=100%:P=100%	-37.80	-42.03	-33.57	<0.0001	8.33
R=100%:P=75%-R=100%:P=100%	1.20	-3.03	5.43	0.9987	-
R=95%:P=75%-R=100%:P=100%	-2.24	-6.47	1.99	0.8465	-
R=90%:P=75%-R=100%:P=100%	-12.44	-16.67	-8.21	<0.0001	2.74
R=85%:P=75%-R=100%:P=100%	-17.96	-22.19	-13.73	<0.0001	3.96
R=80%:P=75%-R=100%:P=100%	-25.88	-30.11	-21.65	<0.0001	5.70
R=75%:P=75%-R=100%:P=100%	-35.88	-40.11	-31.65	<0.0001	7.90
R=90%:P=100%-R=95%:P=100%	-8.96	-13.19	-4.73	<0.0001	1.97
R=85%:P=100%-R=95%:P=100%	-19.60	-23.83	-15.37	<0.0001	4.32
R=80%:P=100%-R=95%:P=100%	-23.88	-28.11	-19.65	<0.0001	5.26
R=75%:P=100%-R=95%:P=100%	-34.20	-38.43	-29.97	<0.0001	7.53
R=100%:P=75%-R=95%:P=100%	4.80	0.57	9.03	0.0119	1.06
R=95%:P=75%-R=95%:P=100%	1.36	-2.87	5.59	0.9961	-
R=90%:P=75%-R=95%:P=100%	-8.84	-13.07	-4.61	<0.0001	1.95
R=85%:P=75%-R=95%:P=100%	-14.36	-18.59	-10.13	<0.0001	3.16
R=80%:P=75%-R=95%:P=100%	-22.28	-26.51	-18.05	<0.0001	4.91
R=75%:P=75%-R=95%:P=100%	-32.28	-36.51	-28.05	<0.0001	7.11

Table B.7: (continued)

	Diff	Lower	Upper	p-value adj.	Cohen's d
R=85%:P=100%-R=90%:P=100%	-10.64	-14.87	-6.41	<0.0001	2.34
R=80%:P=100%-R=90%:P=100%	-14.92	-19.15	-10.69	<0.0001	3.29
R=75%:P=100%-R=90%:P=100%	-25.24	-29.47	-21.01	<0.0001	5.56
R=100%:P=75%-R=90%:P=100%	13.76	9.53	17.99	<0.0001	3.03
R=95%:P=75%-R=90%:P=100%	10.32	6.09	14.55	<0.0001	2.27
R=90%:P=75%-R=90%:P=100%	0.12	-4.11	4.35	1.0000	-
R=85%:P=75%-R=90%:P=100%	-5.40	-9.63	-1.17	0.0021	1.19
R=80%:P=75%-R=90%:P=100%	-13.32	-17.55	-9.09	<0.0001	2.93
R=75%:P=75%-R=90%:P=100%	-23.32	-27.55	-19.09	<0.0001	5.14
R=80%:P=100%-R=85%:P=100%	-4.28	-8.51	-0.05	0.0450	0.94
R=75%:P=100%-R=85%:P=100%	-14.60	-18.83	-10.37	<0.0001	3.22
R=100%:P=75%-R=85%:P=100%	24.40	20.17	28.63	<0.0001	5.37
R=95%:P=75%-R=85%:P=100%	20.96	16.73	25.19	<0.0001	4.62
R=90%:P=75%-R=85%:P=100%	10.76	6.53	14.99	<0.0001	2.37
R=85%:P=75%-R=85%:P=100%	5.24	1.01	9.47	0.0034	1.15
R=80%:P=75%-R=85%:P=100%	-2.68	-6.91	1.55	0.6336	-
R=75%:P=75%-R=85%:P=100%	-12.68	-16.91	-8.45	<0.0001	2.79
R=75%:P=100%-R=80%:P=100%	-10.32	-14.55	-6.09	<0.0001	2.27
R=100%:P=75%-R=80%:P=100%	28.68	24.45	32.91	<0.0001	6.32
R=95%:P=75%-R=80%:P=100%	25.24	21.01	29.47	<0.0001	5.56
R=90%:P=75%-R=80%:P=100%	15.04	10.81	19.27	<0.0001	3.31
R=85%:P=75%-R=80%:P=100%	9.52	5.29	13.75	<0.0001	2.10
R=80%:P=75%-R=80%:P=100%	1.60	-2.63	5.83	0.9848	-
R=75%:P=75%-R=80%:P=100%	-8.40	-12.63	-4.17	<0.0001	1.85
R=100%:P=75%-R=75%:P=100%	39.00	34.77	43.23	<0.0001	8.59
R=95%:P=75%-R=75%:P=100%	35.56	31.33	39.79	<0.0001	7.83
R=90%:P=75%-R=75%:P=100%	25.36	21.12	29.59	<0.0001	5.59
R=85%:P=75%-R=75%:P=100%	19.84	15.61	24.07	<0.0001	4.37
R=80%:P=75%-R=75%:P=100%	11.92	7.69	16.15	<0.0001	2.63
R=75%:P=75%-R=75%:P=100%	1.92	-2.31	6.15	0.9413	-
R=95%:P=75%-R=100%:P=75%	-3.44	-7.67	0.79	0.2433	-
R=90%:P=75%-R=100%:P=75%	-13.64	-17.87	-9.41	<0.0001	3.00
R=85%:P=75%-R=100%:P=75%	-19.16	-23.39	-14.93	<0.0001	4.22
R=80%:P=75%-R=100%:P=75%	-27.08	-31.31	-22.85	<0.0001	5.96
R=75%:P=75%-R=100%:P=75%	-37.08	-41.31	-32.85	<0.0001	8.17
R=90%:P=75%-R=95%:P=75%	-10.20	-14.43	-5.97	<0.0001	2.25
R=85%:P=75%-R=95%:P=75%	-15.72	-19.95	-11.49	<0.0001	3.46
R=80%:P=75%-R=95%:P=75%	-23.64	-27.87	-19.41	<0.0001	5.21
R=75%:P=75%-R=95%:P=75%	-33.64	-37.87	-29.41	<0.0001	7.41
R=85%:P=75%-R=90%:P=75%	-5.52	-9.75	-1.29	0.0014	1.22
R=80%:P=75%-R=90%:P=75%	-13.44	-17.67	-9.21	<0.0001	2.96
R=75%:P=75%-R=90%:P=75%	-23.44	-27.67	-19.21	<0.0001	5.16

Table B.7: (continued)

	Diff	Lower	Upper	p-value adj.	Cohen's d
R=80%:P=75%-R=85%:P=75%	-7.92	-12.15	-3.69	<0.0001	1.74
R=75%:P=75%-R=85%:P=75%	-17.92	-22.15	-13.69	<0.0001	3.95
R=75%:P=75%-R=80%:P=75%	-10.00	-14.23	-5.77	<0.0001	2.20

Table B.8: The results of the Games-Howell post-hoc test with the Cohen's d effect sizes for all possible pairs. For the bugs that have one buggy method. Mean Diff is the difference in means. Df is the degree of freedom. Lower and Upper denote the 95% family-wise confidence levels. R = Recall.

Group	Mean Diff	Std. Error	t-value	Df	p-value	Upper	Lower	Cohen's d
R=100%-R=95%	-3.70	0.54	4.895	94.79	<0.001	-1.50	-5.90	0.98
R=100%-R=90%	-10.76	0.48	15.906	97.97	<0.001	-8.79	-12.73	3.18
R=100%-R=85%	-17.82	0.60	21.195	87.82	<0.001	-15.37	-20.27	4.24
R=100%-R=80%	-23.84	0.51	33.180	97.09	<0.001	-21.75	-25.93	6.64
R=100%-R=75%	-30.24	0.62	34.580	85.04	<0.001	-27.69	-32.79	6.92
R=95%-R=90%	-7.06	0.53	9.411	94.15	<0.001	-4.88	-9.24	1.88
R=95%-R=85%	-14.12	0.64	15.669	95.35	<0.001	-11.50	-16.74	3.13
R=95%-R=80%	-20.14	0.56	25.548	97.23	<0.001	-17.85	-22.43	5.11
R=95%-R=75%	-26.54	0.66	28.455	93.39	<0.001	-23.83	-29.25	5.69
R=90%-R=85%	-7.06	0.59	8.449	86.92	<0.001	-4.63	-9.50	1.69
R=90%-R=80%	-13.08	0.50	18.358	96.71	<0.001	-11.01	-15.15	3.67
R=90%-R=75%	-19.48	0.62	22.402	84.12	<0.001	-16.94	-22.02	4.48
R=85%-R=80%	-6.02	0.62	6.919	92.15	<0.001	-3.49	-8.55	1.38
R=85%-R=75%	-12.42	0.71	12.386	97.68	<0.001	-9.51	-15.34	2.48
R=80%-R=75%	-6.40	0.64	7.090	89.66	<0.001	-3.77	-9.03	1.42

TABLE B.9: The results of the Games-Howell post-hoc test with the Cohen's d effect sizes for all possible pairs. For the bugs that have more than one buggy method. Mean Diff is the difference in means. Df is the degree of freedom. Lower and Upper denote the 95% family-wise confidence levels. R = Recall.

Group	Mean Diff	Std. Error	t-value	Df	p-value	Upper	Lower	Cohen's d
R=100%-R=95%	0.18	0.34	0.373	97.84	0.999	1.59	-1.23	-
R=100%-R=90%	-2.34	0.38	4.414	96.27	<0.001	-0.80	-3.88	0.88
R=100%-R=85%	-3.36	0.36	6.583	97.57	<0.001	-1.88	-4.84	1.32
R=100%-R=80%	-3.44	0.37	6.620	97.05	<0.001	-1.93	-4.95	1.32
R=100%-R=75%	-7.20	0.33	15.280	97.11	<0.001	-5.83	-8.57	3.06
R=95%-R=90%	-2.52	0.37	4.836	95.14	<0.001	-1.00	-4.04	0.97
R=95%-R=85%	-3.54	0.35	7.064	96.91	<0.001	-2.08	-5.00	1.41
R=95%-R=80%	-3.62	0.36	7.091	96.15	<0.001	-2.14	-5.11	1.42
R=95%-R=75%	-7.38	0.33	16.004	97.70	<0.001	-6.04	-8.72	3.20
R=90%-R=85%	-1.02	0.39	1.868	97.54	0.568	0.43	-2.61	-
R=90%-R=80%	-1.10	0.39	1.983	97.88	0.360	0.51	-2.71	-
R=90%-R=75%	-4.86	0.36	9.536	93.20	<0.001	-3.38	-6.34	1.91
R=85%-R=80%	-0.08	0.38	0.149	97.89	1.000	1.48	-1.64	-
R=85%-R=75%	-3.84	0.35	7.850	95.53	<0.001	-2.42	-5.26	1.57
R=80%-R=75%	-3.76	0.35	7.538	94.49	<0.001	-2.31	-5.21	1.51

Appendix C

Impact of Precision for Different Time Budgets

In Chapter 6, we investigate the impact of the time budget on the conclusions about sensitivity to the defect prediction precision (see Section 6.3.2.3). In this appendix, we report the two-way ANOVA test results along with violin and profile plots at various time budgets from 5 to 60 seconds.

Tables C.1, C.2, C.3, C.4 and C.5 show the summary of the two-way ANOVA test results, and Figures C.1, C.2, C.3, C.4 and C.5 show the distributions of the number of bugs detected by SBST guided by DP as violin plots and the profile plot of the mean number of bugs detected by SBST guided by DP for each combination of the factors of six recalls and two precisions at time budgets of 5, 10, 15, 30 and 60 seconds, respectively.

C.1 Time Budget = 5 seconds

According to the two-way ANOVA test (Table C.1), recall explains a significant amount of variation in number of bugs detected by SBST (p-value < 0.001) at 5 seconds time budget. The effect of recall on bug detection effectiveness is large with an effect size ($\widehat{\epsilon}^2$) of 0.68. The test indicates that we cannot reject the null hypothesis that there is no effect from precision on number of bugs detected (p-value = 0.104). That means we can assume there is no effect from precision on number of bugs detected at 5 seconds time budget. These results are consistent with the results at 120 seconds time budget.

	Df	Sum Sq	Mean Sq	F value	p-value
Recall	5	15689	3138	133.76	<0.001
Precision	1	63	63	2.67	0.104
Recall:Precision	5	260	52	2.21	0.053
Residuals	288	6756	23.5		

TABLE C.1: Summary of the two-way ANOVA test results. Time Budget = 5 seconds. Df = degrees of freedom, Sum Sq = sum of squares and Mean sq = mean sum of squares.

C.2 Time Budget = 10 seconds

According to the two-way ANOVA test (Table C.2), recall explains a significant amount of variation in number of bugs detected by SBST (p-value < 0.001) at 10 seconds time budget. The effect of recall on bug detection effectiveness is large with an effect size ($\widehat{\epsilon}^2$) of 0.72. The test indicates that we cannot reject the null hypothesis that there is no effect from precision on number of bugs detected (p-value = 0.336). That means we can assume there is no effect from precision on number of bugs detected at 10 seconds time budget. There is an interaction effect between recall and precision on the number of bugs detected (p-value = 0.026), however this is not practically significant as explained by a very small effect size of 0.007 ($\widehat{\epsilon}^2$). These results are consistent with the results at 120 seconds time budget.

	Df	Sum Sq	Mean Sq	F value	p-value
Recall	5	21468	4294	157.94	<0.001
Precision	1	25	25	0.93	0.336
Recall:Precision	5	353	71	2.60	0.026
Residuals	288	7829	27		

TABLE C.2: Summary of the two-way ANOVA test results. Time Budget = 10 seconds. Df = degrees of freedom, Sum Sq = sum of squares and Mean sq = mean sum of squares.

C.3 Time Budget = 15 seconds

According to the two-way ANOVA test (Table C.3), recall (p-value < 0.001) and precision (p-value = 0.010) explain a significant amount of variation in number of bugs detected by SBST at 15 seconds time budget. The effect of recall on bug detection effectiveness is large with an effect size ($\widehat{\epsilon}^2$) of 0.75. The effect of precision at 15 seconds

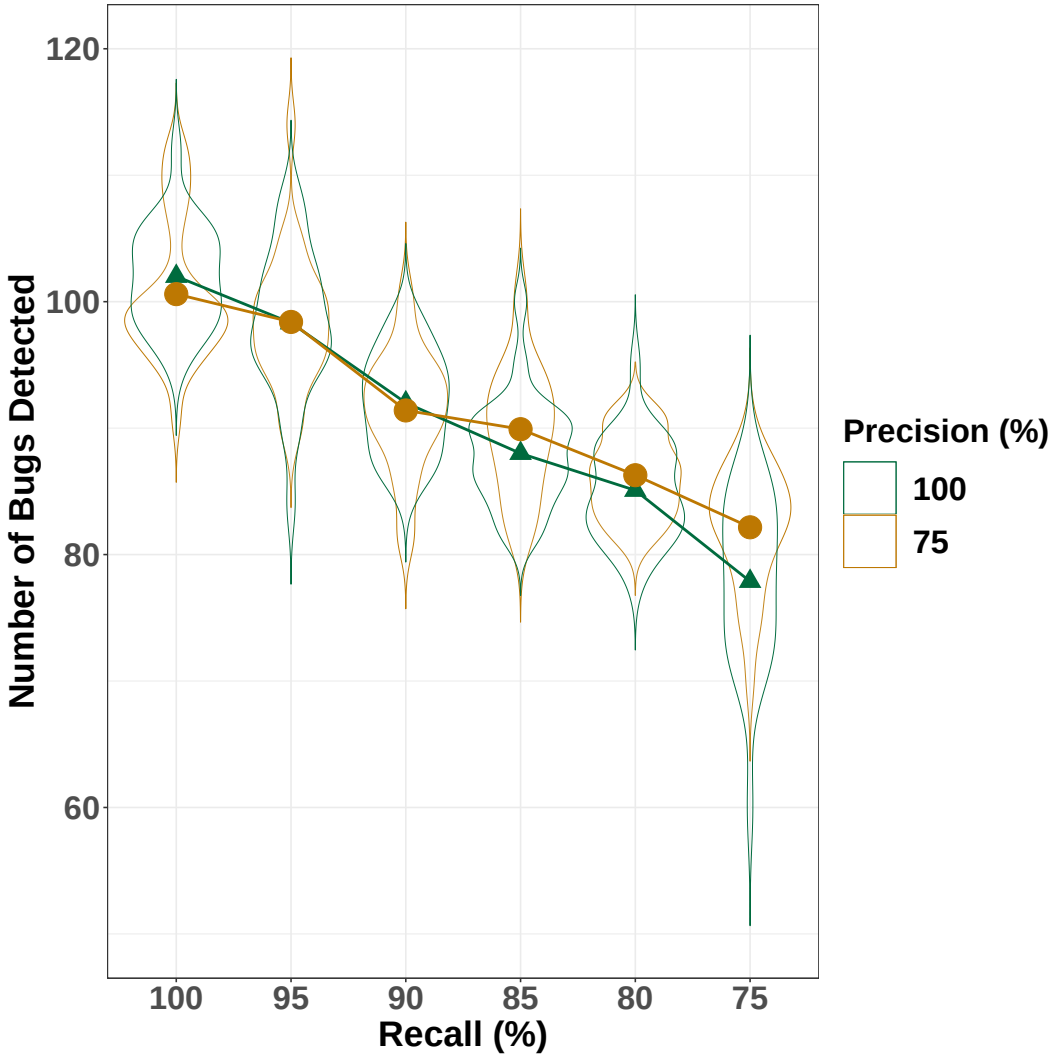

FIGURE C.1: Distributions of the number of bugs detected by SBST guided by DP as violin plots together with the profile plot of mean number of bugs detected by SBST guided by DP for each combination of the groups of recall and precision. Time Budget = 5 seconds.

time budget is not of practical significance as indicated by a very small effect size ($\widehat{\epsilon^2}$) of 0.005. There is an interaction effect between recall and precision on the number of bugs detected (p-value = 0.016), however this is not practically significant as explained by a very small effect size of 0.007 ($\widehat{\epsilon^2}$). These results are consistent with the results at 120 seconds time budget.

C.4 Time Budget = 30 seconds

According to the two-way ANOVA test (Table C.4), recall (p-value < 0.001) and precision (p-value = 0.006) explain a significant amount of variation in number of bugs

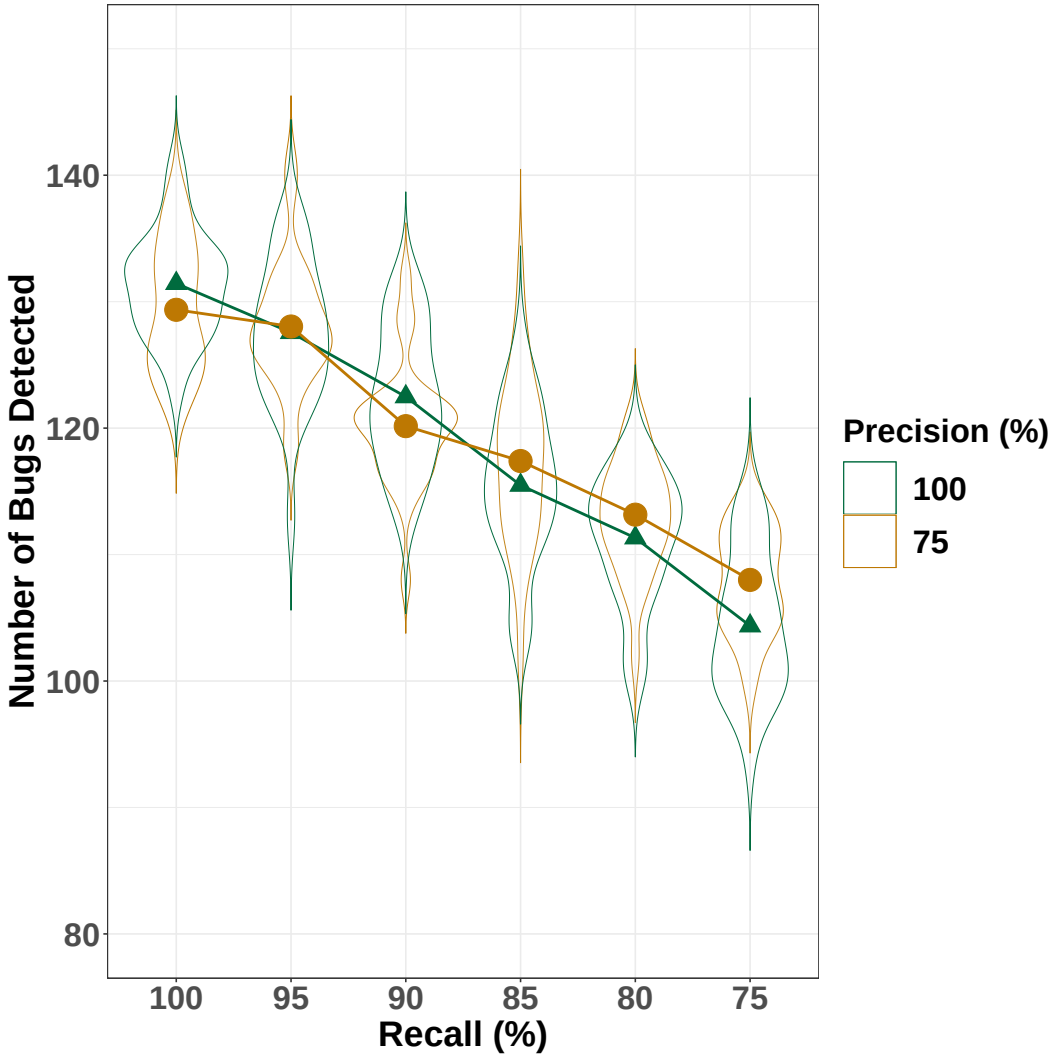

FIGURE C.2: Distributions of the number of bugs detected by SBST guided by DP as violin plots together with the profile plot of mean number of bugs detected by SBST guided by DP for each combination of the groups of recall and precision. Time Budget = 10 seconds.

detected by SBST at 30 seconds time budget. The effect of recall on bug detection effectiveness is large with an effect size ($\widehat{\epsilon^2}$) of 0.83. The effect of precision at 30 seconds time budget is not of practical significance as indicated by a very small effect size ($\widehat{\epsilon^2}$) of 0.004. There is an interaction effect between recall and precision on the number of bugs detected (p-value = 0.034), however this is not practically significant as explained by a very small effect size of 0.004 ($\widehat{\epsilon^2}$). These results are consistent with the results at 120 seconds time budget.

	Df	Sum Sq	Mean Sq	F value	p-value
Recall	5	26849	5370	189.64	<0.001
Precision	1	190	190	6.72	0.010
Recall:Precision	5	403	81	2.85	0.016
Residuals	288	8155	28		

TABLE C.3: Summary of the two-way ANOVA test results. Time Budget = 15 seconds. Df = degrees of freedom, Sum Sq = sum of squares and Mean sq = mean sum of squares.

	Df	Sum Sq	Mean Sq	F value	p-value
Recall	5	37820	7564	297.38	<0.001
Precision	1	197	197	7.74	0.006
Recall:Precision	5	312	62	2.45	0.034
Residuals	288	7325	25		

TABLE C.4: Summary of the two-way ANOVA test results. Time Budget = 30 seconds. Df = degrees of freedom, Sum Sq = sum of squares and Mean sq = mean sum of squares.

C.5 Time Budget = 60 seconds

According to the two-way ANOVA test (Table C.5), recall (p-value < 0.001) and precision (p-value $= 0.003$) explain a significant amount of variation in number of bugs detected by SBST at 60 seconds time budget. The effect of recall on bug detection effectiveness is large with an effect size ($\widehat{\epsilon}^2$) of 0.86. The effect of precision at 60 seconds time budget is not of practical significance as indicated by a very small effect size ($\widehat{\epsilon}^2$) of 0.004. There is an interaction effect between recall and precision on the number of bugs detected (p-value $= 0.041$), however this is not practically significant as explained by a very small effect size of 0.003 ($\widehat{\epsilon}^2$). These results are consistent with the results at 120 seconds time budget.

	Df	Sum Sq	Mean Sq	F value	p-value
Recall	5	43529	8706	385.22	<0.001
Precision	1	203	203	9.00	0.003
Recall:Precision	5	265	53	2.35	0.041
Residuals	288	6509	23		

TABLE C.5: Summary of the two-way ANOVA test results. Time Budget = 60 seconds. Df = degrees of freedom, Sum Sq = sum of squares and Mean sq = mean sum of squares.

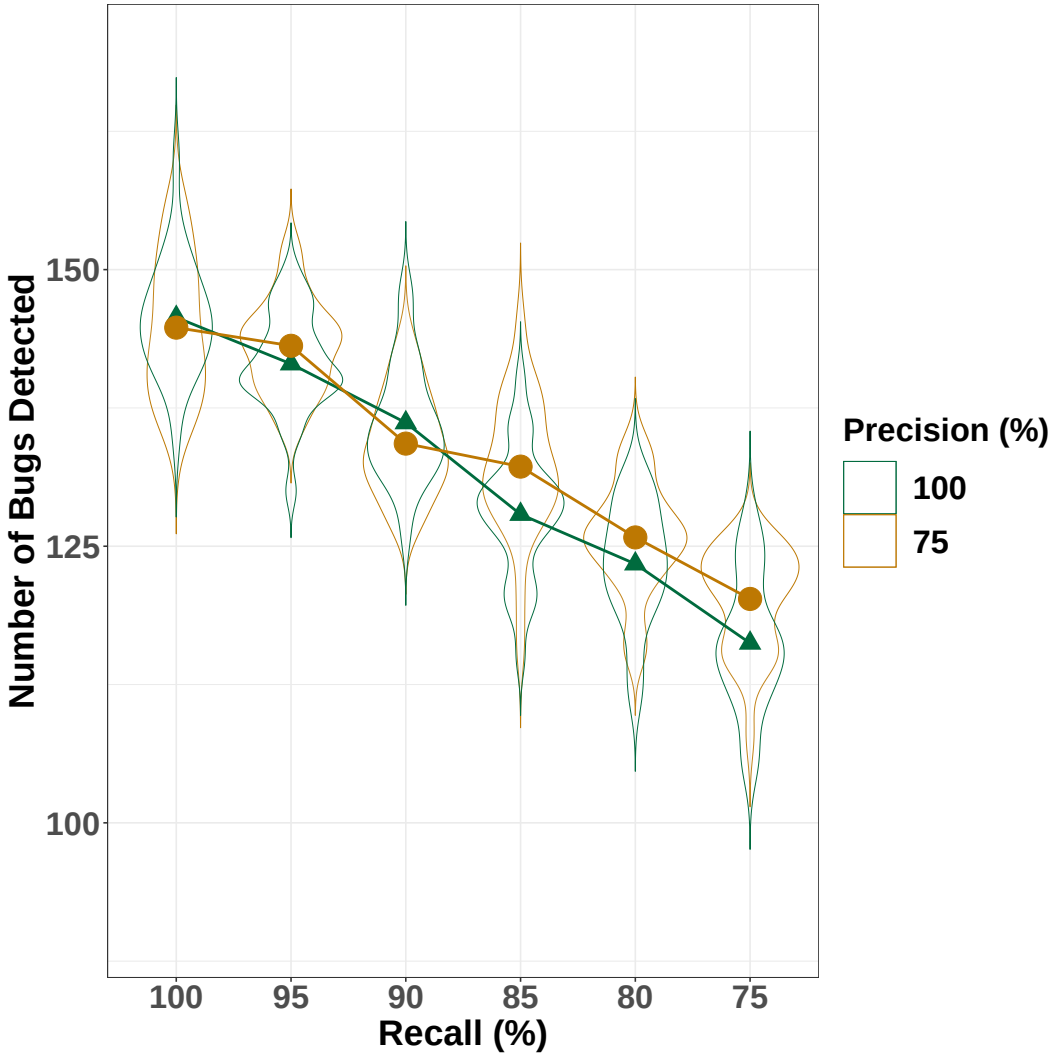

FIGURE C.3: Distributions of the number of bugs detected by SBST guided by DP as violin plots together with the profile plot of mean number of bugs detected by SBST guided by DP for each combination of the groups of recall and precision. Time Budget = 15 seconds.

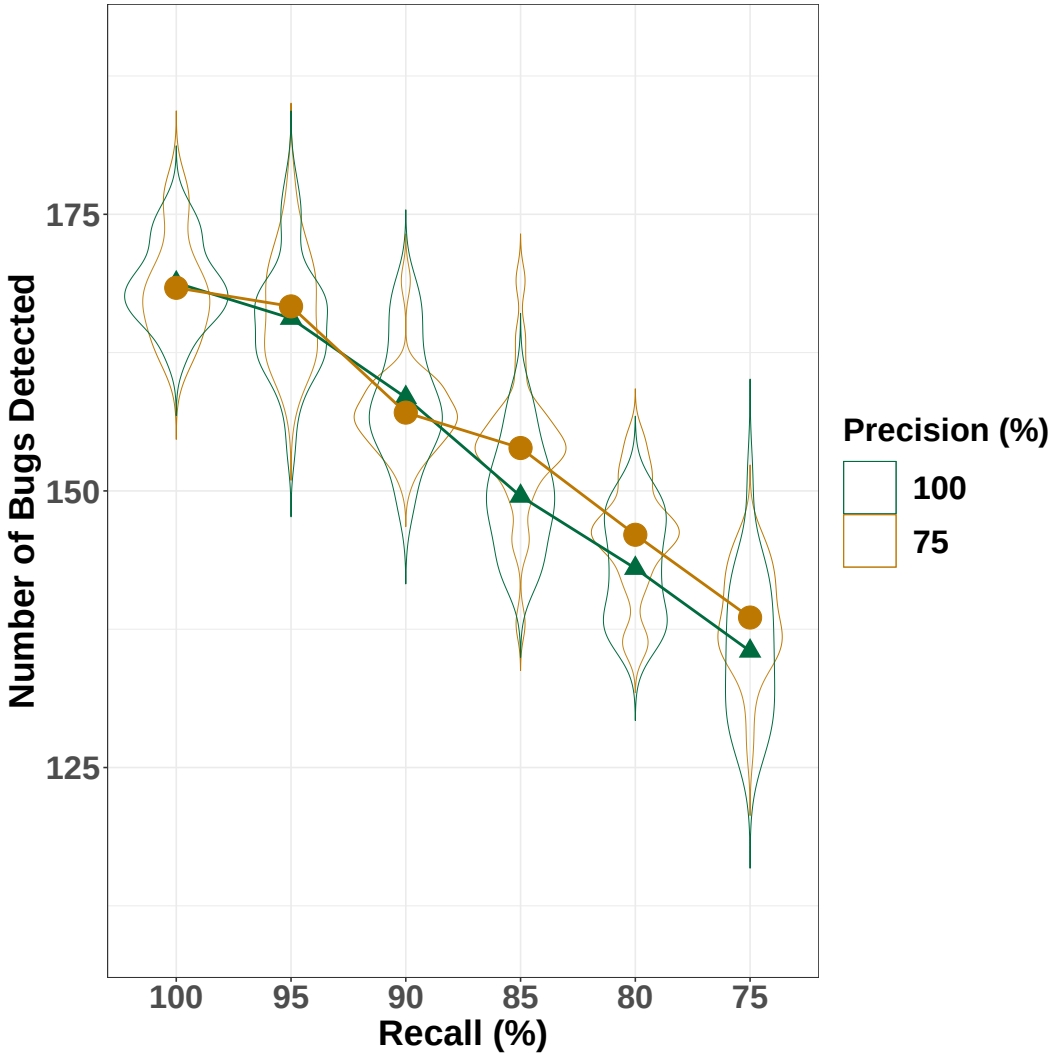

FIGURE C.4: Distributions of the number of bugs detected by SBST guided by DP as violin plots together with the profile plot of mean number of bugs detected by SBST guided by DP for each combination of the groups of recall and precision. Time Budget = 30 seconds.

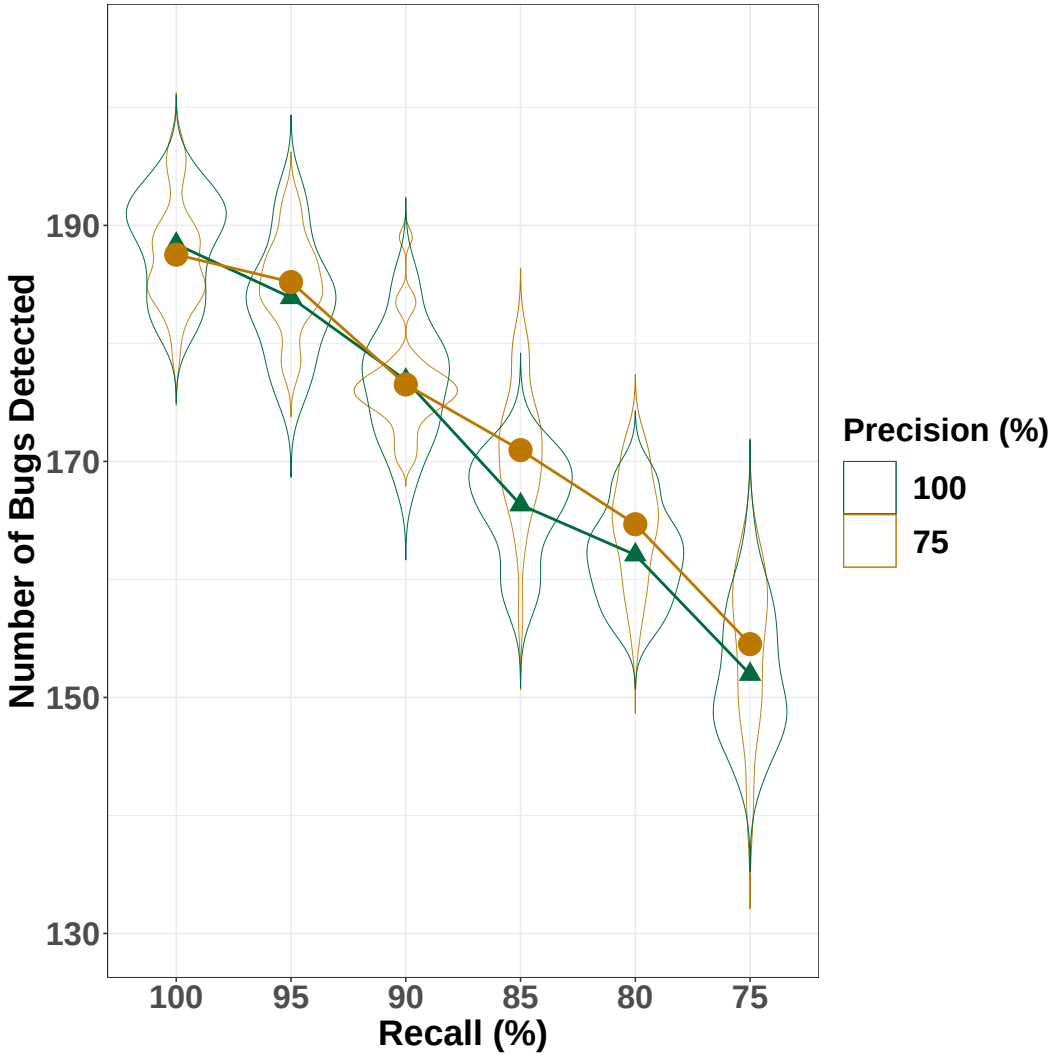

FIGURE C.5: Distributions of the number of bugs detected by SBST guided by DP as violin plots together with the profile plot of mean number of bugs detected by SBST guided by DP for each combination of the groups of recall and precision. Time Budget = 60 seconds.

Appendix D

Guiding the Search Process with Defect Prediction

D.1 Overview of the success rates of PreMOSA and DynaMOSA

Tables D.1 and D.2 show comparisons of the success rates for each bug by PreMOSA-100 against DynaMOSA and PreMOSA-75 against DynaMOSA, respectively.

D.2 Bug detection results of PreMOSA and DynaMOSA over the time budget spent

Tables D.3 and D.4 report statistical summary and the results of the statistical tests of the comparisons of the number of bugs detected by PreMOSA-100 against DynaMOSA and PreMOSA-75 against DynaMOSA over the time budget spent.

D.3 Bug detection results comparison of PreMOSA-100 against PreMOSA-75 over the time budget spent

Table D.5 reports a statistical summary and the results of the one-tailed Mann-Whitney U-Test of the comparison of the number of bugs detected by PreMOSA-100 against

PreMOSA-75 over the time budget spent.

TABLE D.1: Success rate for PreMOSA-100 and DynaMOSA at 2 minutes. Bug IDs that were detected by only one approach are highlighted with different colours; PreMOSA-100 and DynaMOSA.

Bug ID	PreMOSA-100	DynaMOSA
Lang-1	1	1
Lang-4	1	1
Lang-5	0.64	0.88
Lang-7	1	1
Lang-8	0.8	0.8
Lang-9	1	1
Lang-10	1	1
Lang-11	1	1
Lang-12	0.92	0.96
Lang-14	0.84	0.08
Lang-16	0.6	0.52
Lang-17	0	0.04
Lang-18	1	1
Lang-19	1	0.72
Lang-20	1	0.56
Lang-21	0.36	0.68
Lang-22	1	1
Lang-24	0.04	0.08
Lang-27	1	1
Lang-28	0.04	0
Lang-32	0.32	0.12
Lang-33	1	1
Lang-34	0.24	0.32
Lang-35	1	0.96
Lang-36	1	1
Lang-37	0.6	0.68
Lang-39	1	1
Lang-41	1	1
Lang-44	1	1
Lang-45	1	1
Lang-46	1	1
Lang-47	0.96	0.96
Lang-49	0.96	0.88
Lang-50	0.96	0.88
Lang-51	0.68	0.8
Lang-52	1	1
Lang-53	1	0.84
Lang-54	0.4	0.4
Lang-55	0.92	0.24
Lang-57	1	1
Lang-58	0.92	0.8
Lang-59	0.96	1
Lang-60	1	0.88
Lang-61	1	1
Lang-65	1	1
Math-1	1	1

Bug ID	PreMOSA-100	DynaMOSA
Math-2	0.36	0.4
Math-3	1	1
Math-4	1	1
Math-5	1	1
Math-6	1	1
Math-9	1	0.96
Math-10	0.08	0.04
Math-11	0.76	0.8
Math-14	1	1
Math-15	0.28	0.12
Math-16	0.52	0.24
Math-21	0.8	0.84
Math-22	1	1
Math-23	0.8	0.84
Math-24	0.88	0.88
Math-25	0.32	0.4
Math-26	1	1
Math-27	1	1
Math-28	0.04	0.04
Math-29	1	1
Math-32	1	1
Math-33	1	0.8
Math-35	1	1
Math-36	0.72	0.44
Math-37	1	1
Math-38	0.04	0.04
Math-40	1	1
Math-41	0.92	0.8
Math-42	1	0.96
Math-43	0.96	0.8
Math-45	0.88	0.8
Math-46	1	1
Math-47	1	1
Math-48	0.84	1
Math-49	1	0.84
Math-50	0.96	0.84
Math-51	0.68	0.76
Math-52	1	1
Math-53	1	1
Math-55	1	1
Math-56	1	1
Math-59	1	1
Math-60	1	0.96
Math-61	1	1
Math-63	1	0.96
Math-64	0.04	0

Table D.1: (continued)

Bug ID	PreMOSA-100	DynaMOSA
Math-65	1	0.84
Math-66	1	1
Math-67	1	1
Math-68	1	1
Math-69	0.16	0
Math-70	1	1
Math-71	0.92	0.96
Math-72	1	1
Math-73	1	1
Math-74	0.24	0.04
Math-75	1	1
Math-76	0.64	0.52
Math-77	1	1
Math-78	0.88	1
Math-79	0.96	0.28
Math-80	0.68	0.2
Math-81	0.52	0.2
Math-82	0.48	0.48
Math-83	1	0.96
Math-84	0.24	0.16
Math-85	1	1
Math-86	0.92	0.92
Math-87	1	1
Math-88	1	0.92
Math-89	1	1
Math-90	1	1
Math-92	1	1
Math-93	0.88	0.6
Math-94	0.84	0.72
Math-95	1	1
Math-96	1	1
Math-97	1	1
Math-98	1	1
Math-99	0.32	0.04
Math-100	1	1
Math-101	1	1
Math-102	1	1
Math-103	1	1
Math-105	1	1
Math-106	0.08	0.12
Time-1	1	1
Time-2	1	1
Time-3	0.88	0.32
Time-4	1	1
Time-5	1	1
Time-6	1	1
Time-7	0.32	0.2
Time-8	1	1

Bug ID	PreMOSA-100	DynaMOSA
Time-9	1	1
Time-10	1	1
Time-12	1	1
Time-13	0.72	0.84
Time-14	1	1
Time-15	1	0.96
Time-16	0.48	0.08
Time-17	1	1
Time-20	0.2	0
Time-23	1	0.2
Time-24	1	1
Time-25	0.48	0.08
Time-26	0.36	0.48
Time-27	1	1
Chart-1	1	0.44
Chart-2	0.32	0.16
Chart-3	0.88	1
Chart-4	1	0.96
Chart-5	1	1
Chart-6	1	1
Chart-7	1	0.88
Chart-8	1	1
Chart-9	0.24	0.28
Chart-10	1	1
Chart-11	1	1
Chart-12	1	0.92
Chart-13	0.92	1
Chart-14	1	1
Chart-15	1	1
Chart-16	1	1
Chart-17	1	1
Chart-18	1	1
Chart-19	0.96	0.96
Chart-20	0.88	0.72
Chart-21	1	0.84
Chart-22	1	1
Chart-24	1	1
Chart-25	0.12	0
Chart-26	0.64	0.2
Closure-4	0.04	0
Closure-6	0.12	0.56
Closure-7	0.88	0.6
Closure-9	0.48	0.48
Closure-12	0.84	0.92
Closure-13	0.08	0.04
Closure-18	0.32	0.04
Closure-19	0.92	0.36
Closure-21	1	0.92

TABLE D.1: (continued)

Bug ID	PreMOSA-100	DynaMOSA
Closure-22	1	0.92
Closure-26	1	0.96
Closure-27	1	1
Closure-30	1	1
Closure-33	0.96	1
Closure-34	0.24	0.08
Closure-39	1	1
Closure-40	0	0.08
Closure-41	0.76	0.8
Closure-42	0	0.08
Closure-43	0.12	0.12
Closure-45	0.04	0.16
Closure-46	1	1
Closure-48	0.04	0.04
Closure-49	0.8	0.92
Closure-52	1	0.92
Closure-54	1	1
Closure-55	0	0.04
Closure-56	1	1
Closure-57	0	0.04
Closure-58	0.04	0.04
Closure-60	0.84	0.12
Closure-65	0.92	0.96
Closure-68	0	0.04
Closure-69	0.04	0
Closure-70	0.04	0
Closure-71	0.04	0.04
Closure-72	1	1
Closure-73	0.92	0.96
Closure-75	0.64	0
Closure-76	0.04	0.12
Closure-77	0.96	0.88
Closure-79	1	1
Closure-80	1	0.52
Closure-81	0.6	0.36
Closure-82	1	0.96
Closure-85	0.08	0
Closure-86	1	0.04
Closure-89	0.04	0.04
Closure-91	0.92	0.76
Closure-92	0.12	0.04
Closure-94	0.88	0.36
Closure-95	0.12	0
Closure-99	0.2	0.2
Closure-100	1	0.96
Closure-101	0.04	0
Closure-102	0.36	0.2
Closure-103	0.12	0.04

Bug ID	PreMOSA-100	DynaMOSA
Closure-104	1	0.84
Closure-106	1	1
Closure-107	0	0.04
Closure-108	0.6	0.52
Closure-110	1	1
Closure-112	0.12	0.28
Closure-113	0.64	0.84
Closure-114	0.6	0.88
Closure-115	0.76	0.8
Closure-116	0.64	0.72
Closure-117	0.44	0.52
Closure-118	0.08	0.24
Closure-119	0.52	0.88
Closure-120	0.64	0.76
Closure-121	0.92	0.88
Closure-122	0.04	0.08
Closure-123	0.44	0.4
Closure-124	0.84	0.6
Closure-125	0.28	0.6
Closure-128	0.8	0.48
Closure-129	0.44	0.48
Closure-131	1	1
Closure-136	0.12	0.12
Closure-137	1	0.92
Closure-139	0.24	0.24
Closure-140	1	0.96
Closure-141	1	0.72
Closure-142	0	0.04
Closure-143	0	0.04
Closure-144	0.6	0.4
Closure-146	0.36	0.24
Closure-147	0.04	0.2
Closure-148	0.04	0.04
Closure-150	0.84	1
Closure-151	1	0.4
Closure-152	0.04	0.08
Closure-153	0.08	0
Closure-155	0.64	0.28
Closure-156	0.12	0.12
Closure-157	0.16	0.04
Closure-158	0.04	0
Closure-160	0.68	0.24
Closure-162	0.04	0.04
Closure-164	0.96	1
Closure-165	1	1
Closure-166	0.04	0
Closure-167	0.4	0.44
Closure-170	0.84	1

TABLE D.1: (continued)

Bug ID	PreMOSA-100	DynaMOSA
Closure-171	0.84	0.88
Closure-172	0.6	0.8
Closure-173	1	0.92
Closure-174	1	1
Closure-175	0.76	0.4
Closure-176	0.2	0.2
Mockito-2	1	1
Mockito-4	0.12	0
Mockito-9	0.4	0
Mockito-17	1	1
Mockito-23	0	0.04
Mockito-29	0.76	0.68
Mockito-35	1	1

TABLE D.2: Success rate for PreMOSA-75 and DynaMOSA at 2 minutes. Bug IDs that were detected by only one approach are highlighted with different colours; PreMOSA-75 and DynaMOSA.

Bug ID	PreMOSA-75	DynaMOSA
Lang-1	1	1
Lang-4	1	1
Lang-5	0.76	0.88
Lang-7	1	1
Lang-8	0.84	0.8
Lang-9	1	1
Lang-10	1	1
Lang-11	0.96	1
Lang-12	1	0.96
Lang-14	0.72	0.08
Lang-16	0.72	0.52
Lang-17	0.08	0.04
Lang-18	1	1
Lang-19	1	0.72
Lang-20	1	0.56
Lang-21	0.28	0.68
Lang-22	1	1
Lang-24	0	0.08
Lang-27	1	1
Lang-32	0.24	0.12
Lang-33	1	1
Lang-34	0.36	0.32
Lang-35	1	0.96
Lang-36	1	1
Lang-37	0.6	0.68
Lang-39	1	1
Lang-41	1	1
Lang-44	1	1
Lang-45	1	1
Lang-46	1	1
Lang-47	0.96	0.96
Lang-49	1	0.88
Lang-50	0.96	0.88
Lang-51	0.8	0.8
Lang-52	1	1
Lang-53	1	0.84
Lang-54	0.4	0.4
Lang-55	0.88	0.24
Lang-57	1	1
Lang-58	0.92	0.8
Lang-59	0.92	1
Lang-60	1	0.88
Lang-61	0.96	1
Lang-65	1	1
Math-1	1	1
Math-2	0.44	0.4

Bug ID	PreMOSA-75	DynaMOSA
Math-3	1	1
Math-4	1	1
Math-5	1	1
Math-6	1	1
Math-8	0.08	0
Math-9	1	0.96
Math-10	0.04	0.04
Math-11	0.84	0.8
Math-14	1	1
Math-15	0.16	0.12
Math-16	0.16	0.24
Math-21	0.8	0.84
Math-22	1	1
Math-23	0.72	0.84
Math-24	0.88	0.88
Math-25	0.28	0.4
Math-26	1	1
Math-27	1	1
Math-28	0.04	0.04
Math-29	1	1
Math-32	1	1
Math-33	0.92	0.8
Math-35	1	1
Math-36	0.72	0.44
Math-37	1	1
Math-38	0	0.04
Math-39	0.08	0
Math-40	1	1
Math-41	0.92	0.8
Math-42	1	0.96
Math-43	0.84	0.8
Math-45	0.92	0.8
Math-46	1	1
Math-47	1	1
Math-48	0.84	1
Math-49	1	0.84
Math-50	0.88	0.84
Math-51	0.68	0.76
Math-52	1	1
Math-53	1	1
Math-55	1	1
Math-56	1	1
Math-59	1	1
Math-60	1	0.96
Math-61	1	1
Math-63	0.96	0.96

TABLE D.2: (continued)

Bug ID	PreMOSA-75	DynaMOSA
Math-64	0.08	0
Math-65	0.88	0.84
Math-66	1	1
Math-67	1	1
Math-68	1	1
Math-69	0.12	0
Math-70	1	1
Math-71	0.92	0.96
Math-72	1	1
Math-73	1	1
Math-74	0.28	0.04
Math-75	1	1
Math-76	0.48	0.52
Math-77	1	1
Math-78	0.88	1
Math-79	0.92	0.28
Math-80	0.32	0.2
Math-81	0.44	0.2
Math-82	0.68	0.48
Math-83	1	0.96
Math-84	0.36	0.16
Math-85	1	1
Math-86	1	0.92
Math-87	1	1
Math-88	0.96	0.92
Math-89	1	1
Math-90	1	1
Math-92	1	1
Math-93	0.8	0.6
Math-94	0.92	0.72
Math-95	1	1
Math-96	1	1
Math-97	1	1
Math-98	1	1
Math-99	0.16	0.04
Math-100	1	1
Math-101	1	1
Math-102	1	1
Math-103	1	1
Math-105	1	1
Math-106	0.08	0.12
Time-1	1	1
Time-2	1	1
Time-3	0.92	0.32
Time-4	1	1
Time-5	1	1
Time-6	1	1
Time-7	0.16	0.2

Bug ID	PreMOSA-75	DynaMOSA
Time-8	1	1
Time-9	1	1
Time-10	0.88	1
Time-12	1	1
Time-13	0.72	0.84
Time-14	1	1
Time-15	1	0.96
Time-16	0.36	0.08
Time-17	1	1
Time-20	0.24	0
Time-23	0.96	0.2
Time-24	1	1
Time-25	0.2	0.08
Time-26	0.6	0.48
Time-27	1	1
Chart-1	0.92	0.44
Chart-2	0.68	0.16
Chart-3	0.96	1
Chart-4	1	0.96
Chart-5	1	1
Chart-6	1	1
Chart-7	1	0.88
Chart-8	1	1
Chart-9	0.16	0.28
Chart-10	1	1
Chart-11	0.96	1
Chart-12	0.96	0.92
Chart-13	0.68	1
Chart-14	1	1
Chart-15	1	1
Chart-16	1	1
Chart-17	1	1
Chart-18	1	1
Chart-19	1	0.96
Chart-20	0.76	0.72
Chart-21	0.96	0.84
Chart-22	1	1
Chart-24	1	1
Chart-25	0.12	0
Chart-26	0.28	0.2
Closure-4	0.12	0
Closure-6	0.2	0.56
Closure-7	0.72	0.6
Closure-9	0.68	0.48
Closure-10	0.04	0
Closure-12	0.84	0.92
Closure-13	0.16	0.04
Closure-18	0.12	0.04

TABLE D.2: (continued)

Bug ID	PreMOSA-75	DynaMOSA
Closure-19	0.72	0.36
Closure-21	1	0.92
Closure-22	1	0.92
Closure-25	0.04	0
Closure-26	1	0.96
Closure-27	0.84	1
Closure-30	1	1
Closure-33	0.96	1
Closure-34	0.4	0.08
Closure-39	1	1
Closure-40	0	0.08
Closure-41	0.72	0.8
Closure-42	0	0.08
Closure-43	0.08	0.12
Closure-45	0.04	0.16
Closure-46	1	1
Closure-48	0.08	0.04
Closure-49	0.8	0.92
Closure-50	0.04	0
Closure-52	1	0.92
Closure-54	1	1
Closure-55	0.08	0.04
Closure-56	1	1
Closure-57	0.04	0.04
Closure-58	0	0.04
Closure-60	0.8	0.12
Closure-65	1	0.96
Closure-66	0.04	0
Closure-67	0.04	0
Closure-68	0.04	0.04
Closure-70	0.16	0
Closure-71	0.12	0.04
Closure-72	1	1
Closure-73	1	0.96
Closure-75	0.72	0
Closure-76	0.04	0.12
Closure-77	0.88	0.88
Closure-79	1	1
Closure-80	1	0.52
Closure-81	0.4	0.36
Closure-82	1	0.96
Closure-85	0.04	0
Closure-86	0.6	0.04
Closure-89	0.08	0.04
Closure-90	0.04	0
Closure-91	0.72	0.76
Closure-92	0	0.04
Closure-94	0.84	0.36

Bug ID	PreMOSA-75	DynaMOSA
Closure-95	0.12	0
Closure-99	0.08	0.2
Closure-100	0.96	0.96
Closure-102	0.16	0.2
Closure-103	0.12	0.04
Closure-104	1	0.84
Closure-106	0.96	1
Closure-107	0	0.04
Closure-108	0.64	0.52
Closure-110	1	1
Closure-112	0.28	0.28
Closure-113	0.76	0.84
Closure-114	0.8	0.88
Closure-115	0.76	0.8
Closure-116	0.72	0.72
Closure-117	0.36	0.52
Closure-118	0.08	0.24
Closure-119	0.76	0.88
Closure-120	0.68	0.76
Closure-121	0.92	0.88
Closure-122	0.28	0.08
Closure-123	0.44	0.4
Closure-124	0.64	0.6
Closure-125	0.28	0.6
Closure-128	0.8	0.48
Closure-129	0.44	0.48
Closure-131	1	1
Closure-136	0.24	0.12
Closure-137	0.96	0.92
Closure-139	0.24	0.24
Closure-140	0.88	0.96
Closure-141	0.96	0.72
Closure-142	0	0.04
Closure-143	0.04	0.04
Closure-144	0.48	0.4
Closure-146	0.44	0.24
Closure-147	0.24	0.2
Closure-148	0.28	0.04
Closure-150	0.84	1
Closure-151	0.8	0.4
Closure-152	0.04	0.08
Closure-153	0.12	0
Closure-154	0.04	0
Closure-155	0.6	0.28
Closure-156	0.12	0.12
Closure-157	0.2	0.04
Closure-160	0.88	0.24
Closure-162	0	0.04

TABLE D.2: (continued)

Bug ID	PreMOSA-75	DynaMOSA
Closure-164	1	1
Closure-165	1	1
Closure-166	0.04	0
Closure-167	0.6	0.44
Closure-170	0.88	1
Closure-171	0.72	0.88
Closure-172	0.64	0.8
Closure-173	1	0.92
Closure-174	1	1

Bug ID	PreMOSA-75	DynaMOSA
Closure-175	0.88	0.4
Closure-176	0.2	0.2
Mockito-2	1	1
Mockito-4	0.16	0
Mockito-9	0.6	0
Mockito-17	1	1
Mockito-23	0	0.04
Mockito-29	0.84	0.68
Mockito-35	1	1

TABLE D.3: Mean and median number of bugs detected by PreMOSA-100 and DynaMOSA over the time budget spent.

Time (s)	Mean		Median		p-value	$\widehat{A}_{12}$
	PreMOSA -100	Dyna-MOSA	PreMOSA -100	Dyna-MOSA		
1	42.88	**51.60**	42	**52**	**<0.0001**	**0.03**
2	62.40	**64.28**	63	**64**	**0.0487**	0.34
3	**78.16**	75.28	**79**	74	**0.0239**	0.69
4	**89.52**	82.80	**90**	82	**<0.0001**	**0.88**
5	**99.48**	89.36	**99**	89	**<0.0001**	**0.96**
6	**108.12**	95.56	**108**	97	**<0.0001**	**0.97**
7	**116.00**	100.64	**116**	101	**<0.0001**	**0.99**
8	**121.68**	105.00	**123**	105	**<0.0001**	**1.00**
9	**125.92**	109.28	**127**	109	**<0.0001**	**1.00**
10	**129.48**	112.40	**130**	112	**<0.0001**	**1.00**
11	**133.28**	114.80	**134**	114	**<0.0001**	**1.00**
12	**136.48**	118.08	**137**	118	**<0.0001**	**1.00**
13	**139.68**	121.04	**140**	121	**<0.0001**	**1.00**
14	**142.48**	123.48	**143**	124	**<0.0001**	**1.00**
15	**145.20**	126.00	**146**	127	**<0.0001**	**1.00**
16	**147.32**	128.20	**148**	128	**<0.0001**	**1.00**
17	**149.28**	130.36	**149**	131	**<0.0001**	**1.00**
18	**151.44**	132.24	**152**	132	**<0.0001**	**1.00**
19	**153.32**	134.00	**153**	134	**<0.0001**	**1.00**
20	**155.40**	135.60	**155**	135	**<0.0001**	**1.00**
21	**157.08**	137.48	**156**	138	**<0.0001**	**1.00**
22	**159.16**	139.16	**159**	139	**<0.0001**	**1.00**
23	**161.00**	140.48	**160**	140	**<0.0001**	**1.00**
24	**162.68**	142.16	**163**	142	**<0.0001**	**1.00**
25	**164.12**	143.80	**164**	144	**<0.0001**	**1.00**
26	**165.12**	144.76	**165**	145	**<0.0001**	**1.00**
27	**166.36**	145.88	**167**	145	**<0.0001**	**1.00**
28	**167.32**	147.12	**168**	147	**<0.0001**	**1.00**
29	**168.28**	148.32	**169**	149	**<0.0001**	**1.00**
30	**169.48**	149.56	**170**	150	**<0.0001**	**1.00**
31	**170.52**	150.76	**171**	152	**<0.0001**	**1.00**
32	**171.36**	152.04	**172**	153	**<0.0001**	**1.00**
33	**172.04**	152.92	**172**	154	**<0.0001**	**1.00**
34	**172.72**	153.96	**174**	155	**<0.0001**	**1.00**
35	**173.36**	155.56	**174**	156	**<0.0001**	**1.00**
36	**174.92**	156.16	**175**	156	**<0.0001**	**1.00**
37	**175.48**	157.00	**177**	157	**<0.0001**	**1.00**
38	**176.48**	158.08	**177**	159	**<0.0001**	**1.00**
39	**177.32**	158.84	**178**	161	**<0.0001**	**1.00**
40	**178.20**	159.88	**179**	161	**<0.0001**	**1.00**

TABLE D.3: (continued)

Time (s)	Mean		Median		p-value	$\widehat{A}_{12}$
	PreMOSA -100	Dyna-MOSA	PreMOSA -100	Dyna-MOSA		
41	**178.88**	160.60	**180**	162	**<0.0001**	**1.00**
42	**179.80**	161.56	**181**	163	**<0.0001**	**1.00**
43	**180.68**	162.20	**181**	164	**<0.0001**	**1.00**
44	**181.28**	162.72	**182**	164	**<0.0001**	**1.00**
45	**181.84**	163.64	**183**	165	**<0.0001**	**1.00**
46	**182.32**	164.40	**183**	166	**<0.0001**	**1.00**
47	**183.12**	164.96	**183**	166	**<0.0001**	**1.00**
48	**183.80**	165.64	**184**	167	**<0.0001**	**1.00**
49	**184.36**	166.72	**185**	167	**<0.0001**	**1.00**
50	**184.80**	167.64	**185**	168	**<0.0001**	**1.00**
51	**185.48**	168.00	**186**	169	**<0.0001**	**1.00**
52	**186.04**	168.80	**186**	170	**<0.0001**	**1.00**
53	**186.84**	169.24	**186**	170	**<0.0001**	**1.00**
54	**187.52**	169.96	**187**	171	**<0.0001**	**1.00**
55	**188.20**	170.56	**188**	171	**<0.0001**	**1.00**
56	**188.80**	171.48	**189**	172	**<0.0001**	**1.00**
57	**189.20**	172.08	**189**	172	**<0.0001**	**1.00**
58	**190.00**	172.72	**190**	172	**<0.0001**	**1.00**
59	**190.68**	173.24	**191**	173	**<0.0001**	**1.00**
60	**190.96**	173.80	**191**	174	**<0.0001**	**1.00**
61	**191.52**	174.40	**192**	174	**<0.0001**	**1.00**
62	**192.08**	175.04	**193**	174	**<0.0001**	**1.00**
63	**192.48**	175.64	**193**	174	**<0.0001**	**1.00**
64	**193.04**	176.24	**193**	175	**<0.0001**	**1.00**
65	**193.48**	176.64	**193**	175	**<0.0001**	**1.00**
66	**193.76**	177.04	**193**	176	**<0.0001**	**1.00**
67	**194.00**	177.44	**193**	176	**<0.0001**	**1.00**
68	**194.36**	178.04	**194**	177	**<0.0001**	**1.00**
69	**195.04**	178.52	**195**	177	**<0.0001**	**1.00**
70	**195.52**	179.20	**195**	179	**<0.0001**	**1.00**
71	**195.84**	179.76	**195**	179	**<0.0001**	**1.00**
72	**196.16**	180.20	**196**	181	**<0.0001**	**1.00**
73	**196.68**	180.68	**197**	181	**<0.0001**	**1.00**
74	**197.04**	181.08	**197**	181	**<0.0001**	**1.00**
75	**197.52**	181.56	**198**	181	**<0.0001**	**1.00**
76	**197.92**	182.12	**198**	182	**<0.0001**	**0.99**
77	**198.24**	182.36	**198**	182	**<0.0001**	**0.99**
78	**198.60**	183.08	**198**	182	**<0.0001**	**1.00**
79	**199.04**	183.40	**198**	182	**<0.0001**	**1.00**
80	**199.36**	183.80	**198**	183	**<0.0001**	**0.99**
81	**199.52**	184.36	**198**	184	**<0.0001**	**0.99**

Table D.3: (continued)

Time (s)	Mean		Median		p-value	$\widehat{A}_{12}$
	PreMOSA -100	Dyna-MOSA	PreMOSA -100	Dyna-MOSA		
82	**199.84**	184.80	**199**	184	**<0.0001**	**0.99**
83	**200.28**	185.24	**199**	185	**<0.0001**	**0.98**
84	**200.68**	185.76	**200**	185	**<0.0001**	**0.98**
85	**200.88**	186.08	**200**	185	**<0.0001**	**0.98**
86	**201.16**	186.28	**201**	186	**<0.0001**	**0.98**
87	**201.56**	186.60	**201**	186	**<0.0001**	**0.99**
88	**202.08**	186.80	**202**	187	**<0.0001**	**0.99**
89	**202.56**	187.00	**202**	187	**<0.0001**	**0.99**
90	**203.16**	187.32	**203**	187	**<0.0001**	**0.99**
91	**203.40**	187.44	**203**	187	**<0.0001**	**0.99**
92	**203.72**	187.84	**203**	187	**<0.0001**	**0.99**
93	**204.00**	188.24	**203**	187	**<0.0001**	**0.99**
94	**204.52**	188.68	**203**	188	**<0.0001**	**0.99**
95	**204.88**	189.16	**204**	189	**<0.0001**	**0.99**
96	**205.16**	189.48	**204**	189	**<0.0001**	**0.99**
97	**205.48**	189.92	**205**	191	**<0.0001**	**0.99**
98	**205.92**	190.20	**205**	191	**<0.0001**	**0.99**
99	**206.12**	190.56	**205**	191	**<0.0001**	**0.99**
100	**206.44**	190.76	**205**	191	**<0.0001**	**0.99**
101	**206.88**	191.04	**206**	191	**<0.0001**	**0.99**
102	**207.08**	191.24	**206**	191	**<0.0001**	**0.99**
103	**207.20**	191.52	**206**	191	**<0.0001**	**0.99**
104	**207.64**	192.08	**206**	192	**<0.0001**	**0.99**
105	**208.00**	192.56	**208**	192	**<0.0001**	**0.99**
106	**208.32**	192.88	**208**	192	**<0.0001**	**0.99**
107	**208.76**	193.28	**208**	193	**<0.0001**	**0.99**
108	**209.20**	193.52	**210**	193	**<0.0001**	**0.99**
109	**209.32**	193.72	**210**	193	**<0.0001**	**0.99**
110	**209.64**	194.16	**210**	194	**<0.0001**	**0.99**
111	**210.32**	194.32	**210**	194	**<0.0001**	**0.99**
112	**210.64**	194.68	**210**	195	**<0.0001**	**1.00**
113	**210.96**	194.76	**210**	195	**<0.0001**	**1.00**
114	**211.44**	194.96	**210**	196	**<0.0001**	**0.99**
115	**211.76**	195.36	**211**	196	**<0.0001**	**0.99**
116	**212.04**	195.72	**212**	196	**<0.0001**	**0.99**
117	**212.44**	196.20	**213**	197	**<0.0001**	**0.99**
118	**212.76**	196.40	**213**	197	**<0.0001**	**0.99**
119	**213.12**	196.68	**213**	197	**<0.0001**	**1.00**
120	**213.56**	197.16	**213**	197	**<0.0001**	**0.99**

TABLE D.4: Mean and median number of bugs detected by PreMOSA-75 and DynaMOSA over the time budget spent.

Time (s)	Mean		Median		p-value	$\widehat{A}_{12}$
	PreMOSA -75	Dyna-MOSA	PreMOSA -75	Dyna-MOSA		
1	43.60	**51.60**	43	**52**	**<0.0001**	**0.08**
2	62.96	64.28	63	64	0.2792	0.41
3	**77.76**	75.28	**78**	74	**0.0336**	0.67
4	**88.28**	82.80	**88**	82	**<0.0001**	**0.87**
5	**97.64**	89.36	**98**	89	**<0.0001**	**0.94**
6	**106.28**	95.56	**107**	97	**<0.0001**	**0.94**
7	**112.84**	100.64	**113**	101	**<0.0001**	**0.97**
8	**117.68**	105.00	**118**	105	**<0.0001**	**0.97**
9	**122.56**	109.28	**123**	109	**<0.0001**	**0.99**
10	**126.44**	112.40	**127**	112	**<0.0001**	**0.98**
11	**129.76**	114.80	**130**	114	**<0.0001**	**0.99**
12	**133.08**	118.08	**134**	118	**<0.0001**	**0.99**
13	**136.44**	121.04	**137**	121	**<0.0001**	**0.99**
14	**138.72**	123.48	**139**	124	**<0.0001**	**0.99**
15	**141.36**	126.00	**141**	127	**<0.0001**	**1.00**
16	**143.76**	128.20	**143**	128	**<0.0001**	**1.00**
17	**145.92**	130.36	**145**	131	**<0.0001**	**1.00**
18	**148.32**	132.24	**147**	132	**<0.0001**	**1.00**
19	**150.12**	134.00	**149**	134	**<0.0001**	**1.00**
20	**151.76**	135.60	**151**	135	**<0.0001**	**1.00**
21	**153.96**	137.48	**153**	138	**<0.0001**	**1.00**
22	**155.88**	139.16	**155**	139	**<0.0001**	**1.00**
23	**157.12**	140.48	**156**	140	**<0.0001**	**1.00**
24	**158.36**	142.16	**158**	142	**<0.0001**	**1.00**
25	**159.40**	143.80	**160**	144	**<0.0001**	**1.00**
26	**160.72**	144.76	**160**	145	**<0.0001**	**1.00**
27	**161.72**	145.88	**161**	145	**<0.0001**	**1.00**
28	**163.08**	147.12	**162**	147	**<0.0001**	**1.00**
29	**164.16**	148.32	**164**	149	**<0.0001**	**1.00**
30	**165.44**	149.56	**165**	150	**<0.0001**	**1.00**
31	**166.60**	150.76	**166**	152	**<0.0001**	**1.00**
32	**167.48**	152.04	**166**	153	**<0.0001**	**1.00**
33	**168.44**	152.92	**167**	154	**<0.0001**	**1.00**
34	**169.68**	153.96	**170**	155	**<0.0001**	**1.00**
35	**171.12**	155.56	**171**	156	**<0.0001**	**1.00**
36	**172.04**	156.16	**172**	156	**<0.0001**	**1.00**
37	**172.92**	157.00	**173**	157	**<0.0001**	**1.00**
38	**173.84**	158.08	**175**	159	**<0.0001**	**1.00**
39	**174.88**	158.84	**176**	161	**<0.0001**	**1.00**
40	**175.48**	159.88	**176**	161	**<0.0001**	**1.00**

TABLE D.4: (continued)

Time (s)	Mean		Median		p-value	$\widehat{A}_{12}$
	PreMOSA -75	Dyna-MOSA	PreMOSA -75	Dyna-MOSA		
41	**176.44**	160.60	**178**	162	**<0.0001**	**1.00**
42	**177.44**	161.56	**178**	163	**<0.0001**	**1.00**
43	**178.12**	162.20	**179**	164	**<0.0001**	**1.00**
44	**178.88**	162.72	**179**	164	**<0.0001**	**1.00**
45	**179.36**	163.64	**179**	165	**<0.0001**	**1.00**
46	**180.12**	164.40	**180**	166	**<0.0001**	**1.00**
47	**180.68**	164.96	**181**	166	**<0.0001**	**1.00**
48	**181.32**	165.64	**181**	167	**<0.0001**	**1.00**
49	**181.92**	166.72	**183**	167	**<0.0001**	**1.00**
50	**182.56**	167.64	**183**	168	**<0.0001**	**1.00**
51	**183.12**	168.00	**184**	169	**<0.0001**	**1.00**
52	**183.72**	168.80	**185**	170	**<0.0001**	**0.99**
53	**184.44**	169.24	**185**	170	**<0.0001**	**1.00**
54	**185.12**	169.96	**186**	171	**<0.0001**	**0.99**
55	**185.72**	170.56	**186**	171	**<0.0001**	**0.99**
56	**186.40**	171.48	**187**	172	**<0.0001**	**0.99**
57	**186.96**	172.08	**188**	172	**<0.0001**	**0.99**
58	**187.32**	172.72	**188**	172	**<0.0001**	**0.99**
59	**188.00**	173.24	**190**	173	**<0.0001**	**0.99**
60	**188.52**	173.80	**190**	174	**<0.0001**	**0.99**
61	**189.28**	174.40	**190**	174	**<0.0001**	**0.99**
62	**189.80**	175.04	**191**	174	**<0.0001**	**0.99**
63	**190.28**	175.64	**191**	174	**<0.0001**	**0.98**
64	**190.92**	176.24	**192**	175	**<0.0001**	**0.98**
65	**191.64**	176.64	**192**	175	**<0.0001**	**0.98**
66	**192.08**	177.04	**192**	176	**<0.0001**	**0.98**
67	**192.60**	177.44	**193**	176	**<0.0001**	**0.99**
68	**192.92**	178.04	**194**	177	**<0.0001**	**0.99**
69	**193.44**	178.52	**194**	177	**<0.0001**	**0.98**
70	**194.08**	179.20	**195**	179	**<0.0001**	**0.97**
71	**194.44**	179.76	**195**	179	**<0.0001**	**0.98**
72	**194.80**	180.20	**196**	181	**<0.0001**	**0.98**
73	**195.20**	180.68	**196**	181	**<0.0001**	**0.97**
74	**195.92**	181.08	**196**	181	**<0.0001**	**0.97**
75	**196.44**	181.56	**196**	181	**<0.0001**	**0.97**
76	**196.88**	182.12	**197**	182	**<0.0001**	**0.97**
77	**197.56**	182.36	**199**	182	**<0.0001**	**0.97**
78	**197.88**	183.08	**199**	182	**<0.0001**	**0.96**
79	**198.24**	183.40	**199**	182	**<0.0001**	**0.96**
80	**198.56**	183.80	**199**	183	**<0.0001**	**0.96**
81	**198.88**	184.36	**200**	184	**<0.0001**	**0.96**

TABLE D.4: (continued)

Time (s)	Mean		Median		p-value	$\widehat{A}_{12}$
	PreMOSA -75	Dyna-MOSA	PreMOSA -75	Dyna-MOSA		
82	**199.12**	184.80	**201**	184	**<0.0001**	**0.95**
83	**199.68**	185.24	**201**	185	**<0.0001**	**0.95**
84	**200.04**	185.76	**201**	185	**<0.0001**	**0.95**
85	**200.36**	186.08	**201**	185	**<0.0001**	**0.95**
86	**200.92**	186.28	**201**	186	**<0.0001**	**0.96**
87	**201.32**	186.60	**201**	186	**<0.0001**	**0.96**
88	**201.76**	186.80	**201**	187	**<0.0001**	**0.97**
89	**202.20**	187.00	**201**	187	**<0.0001**	**0.97**
90	**202.56**	187.32	**202**	187	**<0.0001**	**0.97**
91	**202.76**	187.44	**203**	187	**<0.0001**	**0.97**
92	**203.28**	187.84	**204**	187	**<0.0001**	**0.97**
93	**203.72**	188.24	**205**	187	**<0.0001**	**0.97**
94	**204.16**	188.68	**205**	188	**<0.0001**	**0.97**
95	**204.32**	189.16	**205**	189	**<0.0001**	**0.97**
96	**204.48**	189.48	**205**	189	**<0.0001**	**0.97**
97	**204.88**	189.92	**206**	191	**<0.0001**	**0.97**
98	**205.32**	190.20	**206**	191	**<0.0001**	**0.98**
99	**205.64**	190.56	**207**	191	**<0.0001**	**0.98**
100	**206.08**	190.76	**208**	191	**<0.0001**	**0.98**
101	**206.48**	191.04	**208**	191	**<0.0001**	**0.98**
102	**206.80**	191.24	**208**	191	**<0.0001**	**0.98**
103	**207.08**	191.52	**208**	191	**<0.0001**	**0.98**
104	**207.44**	192.08	**208**	192	**<0.0001**	**0.98**
105	**207.92**	192.56	**209**	192	**<0.0001**	**0.98**
106	**208.16**	192.88	**210**	192	**<0.0001**	**0.98**
107	**208.68**	193.28	**210**	193	**<0.0001**	**0.98**
108	**209.12**	193.52	**210**	193	**<0.0001**	**0.99**
109	**209.32**	193.72	**211**	193	**<0.0001**	**0.99**
110	**209.48**	194.16	**211**	194	**<0.0001**	**0.98**
111	**209.76**	194.32	**211**	194	**<0.0001**	**0.98**
112	**209.96**	194.68	**211**	195	**<0.0001**	**0.98**
113	**210.24**	194.76	**211**	195	**<0.0001**	**0.98**
114	**210.72**	194.96	**211**	196	**<0.0001**	**0.99**
115	**211.00**	195.36	**211**	196	**<0.0001**	**0.99**
116	**211.24**	195.72	**211**	196	**<0.0001**	**0.99**
117	**211.48**	196.20	**211**	197	**<0.0001**	**0.98**
118	**211.96**	196.40	**211**	197	**<0.0001**	**0.98**
119	**212.32**	196.68	**212**	197	**<0.0001**	**0.98**
120	**212.60**	197.16	**212**	197	**<0.0001**	**0.98**

TABLE D.5: Mean and median number of bugs detected by PreMOSA-100 and PreMOSA-75 over the time budget spent.

Time (s)	Mean		Median		p-value	$\widehat{A}_{12}$
	PreMOSA -100	PreMOSA -75	PreMOSA -100	PreMOSA -75		
1	42.88	43.60	42	43	0.6554	0.47
2	62.40	62.96	63	63	0.6942	0.46
3	78.16	77.76	79	78	0.3665	0.53
4	89.52	88.28	90	88	0.1355	0.59
5	**99.48**	97.64	**99**	98	**0.0433**	0.64
6	108.12	106.28	108	107	0.1357	0.59
7	**116.00**	112.84	**116**	113	**0.0157**	0.68
8	**121.68**	117.68	**123**	118	**0.0030**	0.73
9	**125.92**	122.56	**127**	123	**0.0047**	0.71
10	**129.48**	126.44	**130**	127	**0.0045**	0.71
11	**133.28**	129.76	**134**	130	**0.0030**	0.73
12	**136.48**	133.08	**137**	134	**0.0029**	0.73
13	**139.68**	136.44	**140**	137	**0.0110**	0.69
14	**142.48**	138.72	**143**	139	**0.0048**	0.71
15	**145.20**	141.36	**146**	141	**0.0019**	0.74
16	**147.32**	143.76	**148**	143	**0.0039**	0.72
17	**149.28**	145.92	**149**	145	**0.0048**	0.71
18	**151.44**	148.32	**152**	147	**0.0100**	0.69
19	**153.32**	150.12	**153**	149	**0.0117**	0.69
20	**155.40**	151.76	**155**	151	**0.0036**	0.72
21	**157.08**	153.96	**156**	153	**0.0108**	0.69
22	**159.16**	155.88	**159**	155	**0.0066**	0.70
23	**161.00**	157.12	**160**	156	**0.0013**	0.75
24	**162.68**	158.36	**163**	158	**0.0008**	**0.76**
25	**164.12**	159.40	**164**	160	**0.0005**	**0.77**
26	**165.12**	160.72	**165**	160	**0.0005**	**0.77**
27	**166.36**	161.72	**167**	161	**0.0004**	**0.77**
28	**167.32**	163.08	**168**	162	**0.0012**	**0.75**
29	**168.28**	164.16	**169**	164	**0.0021**	0.74
30	**169.48**	165.44	**170**	165	**0.0015**	0.74
31	**170.52**	166.60	**171**	166	**0.0014**	0.75
32	**171.36**	167.48	**172**	166	**0.0014**	0.75
33	**172.04**	168.44	**172**	167	**0.0028**	0.73
34	**172.72**	169.68	**174**	170	**0.0062**	0.71
35	**173.36**	171.12	**174**	171	**0.0225**	0.66
36	**174.92**	172.04	**175**	172	**0.0120**	0.69
37	**175.48**	172.92	**177**	173	**0.0153**	0.68
38	**176.48**	173.84	**177**	175	**0.0103**	0.69
39	**177.32**	174.88	**178**	176	**0.0160**	0.68
40	**178.20**	175.48	**179**	176	**0.0132**	0.68

Table D.5: (continued)

Time (s)	Mean		Median		p-value	$\widehat{A}_{12}$
	PreMOSA -100	PreMOSA -75	PreMOSA -100	PreMOSA -75		
41	**178.88**	176.44	**180**	178	**0.0229**	0.66
42	**179.80**	177.44	**181**	178	**0.0283**	0.66
43	**180.68**	178.12	**181**	179	**0.0281**	0.66
44	**181.28**	178.88	**182**	179	**0.0391**	0.64
45	**181.84**	179.36	**183**	179	**0.0336**	0.65
46	**182.32**	180.12	**183**	180	**0.0358**	0.65
47	**183.12**	180.68	**183**	181	**0.0359**	0.65
48	**183.80**	181.32	**184**	181	**0.0329**	0.65
49	**184.36**	181.92	**185**	183	**0.0316**	0.65
50	**184.80**	182.56	**185**	183	**0.0407**	0.64
51	**185.48**	183.12	**186**	184	**0.0275**	0.66
52	**186.04**	183.72	**186**	185	**0.0375**	0.65
53	**186.84**	184.44	**186**	185	**0.0389**	0.64
54	**187.52**	185.12	**187**	186	**0.0390**	0.64
55	**188.20**	185.72	**188**	186	**0.0350**	0.65
56	**188.80**	186.40	**189**	187	**0.0408**	0.64
57	189.20	186.96	189	188	0.0520	0.63
58	**190.00**	187.32	**190**	188	**0.0351**	0.65
59	**190.68**	188.00	**191**	190	**0.0293**	0.66
60	**190.96**	188.52	**191**	190	**0.0458**	0.64
61	191.52	189.28	192	190	0.0732	0.62
62	192.08	189.80	193	191	0.0562	0.63
63	192.48	190.28	193	191	0.0693	0.62
64	193.04	190.92	193	192	0.0994	0.61
65	193.48	191.64	193	192	0.1603	0.58
66	193.76	192.08	193	192	0.1652	0.58
67	194.00	192.60	193	193	0.2265	0.56
68	194.36	192.92	194	194	0.2448	0.56
69	195.04	193.44	195	194	0.1752	0.58
70	195.52	194.08	195	195	0.2153	0.56
71	195.84	194.44	195	195	0.2268	0.56
72	196.16	194.80	196	196	0.2181	0.56
73	196.68	195.20	197	196	0.1959	0.57
74	197.04	195.92	197	196	0.2701	0.55
75	197.52	196.44	198	196	0.3272	0.54
76	197.92	196.88	198	197	0.3378	0.53
77	198.24	197.56	198	199	0.4305	0.51
78	198.60	197.88	198	199	0.4420	0.51
79	199.04	198.24	198	199	0.3889	0.52
80	199.36	198.56	198	199	0.3927	0.52
81	199.52	198.88	198	200	0.4497	0.51

Table D.5: (continued)

Time (s)	Mean		Median		p-value	$\widehat{A}_{12}$
	PreMOSA -100	PreMOSA -75	PreMOSA -100	PreMOSA -75		
82	199.84	199.12	199	201	0.3852	0.52
83	200.28	199.68	199	201	0.4152	0.52
84	200.68	200.04	200	201	0.3778	0.53
85	200.88	200.36	200	201	0.4115	0.52
86	201.16	200.92	201	201	0.4961	0.50
87	201.56	201.32	201	201	0.5116	0.50
88	202.08	201.76	202	201	0.4574	0.51
89	202.56	202.20	202	201	0.4267	0.52
90	203.16	202.56	203	202	0.3343	0.54
91	203.40	202.76	203	203	0.3414	0.53
92	203.72	203.28	203	204	0.4190	0.52
93	204.00	203.72	203	205	0.4729	0.51
94	204.52	204.16	203	205	0.4418	0.51
95	204.88	204.32	204	205	0.3852	0.52
96	205.16	204.48	204	205	0.3668	0.53
97	205.48	204.88	205	206	0.4002	0.52
98	205.92	205.32	205	206	0.3927	0.52
99	206.12	205.64	205	207	0.4229	0.52
100	206.44	206.08	205	208	0.4573	0.51
101	206.88	206.48	206	208	0.4039	0.52
102	207.08	206.80	206	208	0.4458	0.51
103	207.20	207.08	206	208	0.4922	0.50
104	207.64	207.44	206	208	0.4690	0.51
105	208.00	207.92	208	209	0.5078	0.50
106	208.32	208.16	208	210	0.5155	0.50
107	208.76	208.68	208	210	0.5388	0.49
108	209.20	209.12	210	210	0.5116	0.50
109	209.32	209.32	210	211	0.5155	0.50
110	209.64	209.48	210	211	0.4304	0.51
111	210.32	209.76	210	211	0.3521	0.53
112	210.64	209.96	210	211	0.3522	0.53
113	210.96	210.24	210	211	0.3486	0.53
114	211.44	210.72	210	211	0.3414	0.53
115	211.76	211.00	211	211	0.3100	0.54
116	212.04	211.24	212	211	0.2898	0.55
117	212.44	211.48	213	211	0.2732	0.55
118	212.76	211.96	213	211	0.3167	0.54
119	213.12	212.32	213	212	0.3559	0.53
120	213.56	212.60	213	212	0.2666	0.55

Appendix E

Balanced Test Coverage of Targets

E.1 Overview of the success rates of DynaMOSA+b and DynaMOSA

Table E.1 shows a comparison of the success rates for each bug by DynaMOSA+b against DynaMOSA.

E.2 Bug detection results of PreMOSA and DynaMOSA over the time budget spent

Table E.2 reports a statistical summary and the results of the statistical tests of the comparison of the number of bugs detected by DynaMOSA+b against DynaMOSA over the time budget spent.

TABLE E.1: Success rate for DynaMOSA+b and DynaMOSA at 2 minutes. Bug IDs that were detected by only one approach are highlighted with different colours; DynaMOSA+b and DynaMOSA.

Bug ID	DynaMOSA+b	DynaMOSA
Lang-1	1	1
Lang-4	1	1
Lang-5	0.72	0.88
Lang-7	1	1
Lang-8	0.88	0.8
Lang-9	1	1
Lang-10	1	1
Lang-11	1	1
Lang-12	1	0.96
Lang-14	0.16	0.08
Lang-16	0.6	0.52
Lang-17	0.08	0.04
Lang-18	0.96	1
Lang-19	1	0.72
Lang-20	0.8	0.56
Lang-21	0.32	0.68
Lang-22	1	1
Lang-24	0.16	0.08
Lang-27	1	1
Lang-28	0.12	0
Lang-32	0.4	0.12
Lang-33	1	1
Lang-34	0.28	0.32
Lang-35	0.96	0.96
Lang-36	1	1
Lang-37	0.88	0.68
Lang-39	1	1
Lang-40	0.04	0
Lang-41	1	1
Lang-44	1	1
Lang-45	1	1
Lang-46	1	1
Lang-47	1	0.96
Lang-49	0.8	0.88
Lang-50	0.56	0.88
Lang-51	0.6	0.8
Lang-52	1	1
Lang-53	1	0.84
Lang-54	0.28	0.4
Lang-55	0.8	0.24
Lang-57	1	1
Lang-58	0.88	0.8
Lang-59	1	1
Lang-60	0.84	0.88
Lang-61	1	1
Lang-65	1	1

Bug ID	DynaMOSA+b	DynaMOSA
Math-1	1	1
Math-2	0.48	0.4
Math-3	0.84	1
Math-4	1	1
Math-5	1	1
Math-6	1	1
Math-8	0.04	0
Math-9	1	0.96
Math-10	0.08	0.04
Math-11	0.88	0.8
Math-14	1	1
Math-15	0.08	0.12
Math-16	0.08	0.24
Math-21	0.88	0.84
Math-22	1	1
Math-23	1	0.84
Math-24	0.72	0.88
Math-25	0.44	0.4
Math-26	1	1
Math-27	1	1
Math-28	0	0.04
Math-29	0.96	1
Math-30	0.04	0
Math-32	1	1
Math-33	0.8	0.8
Math-35	1	1
Math-36	0.52	0.44
Math-37	1	1
Math-38	0.04	0.04
Math-39	0.04	0
Math-40	1	1
Math-41	0.68	0.8
Math-42	1	0.96
Math-43	0.92	0.8
Math-45	1	0.8
Math-46	1	1
Math-47	1	1
Math-48	0.68	1
Math-49	1	0.84
Math-50	0.8	0.84
Math-51	0.52	0.76
Math-52	1	1
Math-53	1	1
Math-55	1	1
Math-56	1	1
Math-59	1	1

Table E.1: (continued)

Bug ID	DynaMOSA+b	DynaMOSA
Math-60	1	0.96
Math-61	1	1
Math-63	1	0.96
Math-64	0.04	0
Math-65	0.88	0.84
Math-66	1	1
Math-67	1	1
Math-68	1	1
Math-70	1	1
Math-71	0.88	0.96
Math-72	0.88	1
Math-73	1	1
Math-74	0.12	0.04
Math-75	1	1
Math-76	0.36	0.52
Math-77	1	1
Math-78	1	1
Math-79	0.2	0.28
Math-80	0.32	0.2
Math-81	0.16	0.2
Math-82	0.56	0.48
Math-83	1	0.96
Math-84	0.24	0.16
Math-85	1	1
Math-86	0.92	0.92
Math-87	1	1
Math-88	0.96	0.92
Math-89	1	1
Math-90	1	1
Math-92	1	1
Math-93	0.48	0.6
Math-94	0.96	0.72
Math-95	1	1
Math-96	1	1
Math-97	1	1
Math-98	1	1
Math-99	0.52	0.04
Math-100	1	1
Math-101	1	1
Math-102	1	1
Math-103	1	1
Math-105	1	1
Math-106	0.04	0.12
Time-1	1	1
Time-2	1	1
Time-3	0.76	0.32
Time-4	1	1
Time-5	1	1

Bug ID	DynaMOSA+b	DynaMOSA
Time-6	1	1
Time-7	0.24	0.2
Time-8	1	1
Time-9	1	1
Time-10	1	1
Time-12	1	1
Time-13	0.56	0.84
Time-14	1	1
Time-15	1	0.96
Time-16	0.08	0.08
Time-17	1	1
Time-20	0.08	0
Time-23	0.96	0.2
Time-24	1	1
Time-25	0.2	0.08
Time-26	0.52	0.48
Time-27	0.96	1
Chart-1	0.48	0.44
Chart-2	0.12	0.16
Chart-3	0.96	1
Chart-4	0.96	0.96
Chart-5	1	1
Chart-6	1	1
Chart-7	0.56	0.88
Chart-8	1	1
Chart-9	0.12	0.28
Chart-10	1	1
Chart-11	1	1
Chart-12	0.92	0.92
Chart-13	0.72	1
Chart-14	1	1
Chart-15	1	1
Chart-16	1	1
Chart-17	1	1
Chart-18	1	1
Chart-19	0.92	0.96
Chart-20	0.8	0.72
Chart-21	0.88	0.84
Chart-22	1	1
Chart-24	1	1
Chart-25	0.04	0
Chart-26	0.36	0.2
Closure-1	0.04	0
Closure-6	0.44	0.56
Closure-7	0.84	0.6
Closure-9	0.32	0.48
Closure-12	0.92	0.92
Closure-13	0.16	0.04

TABLE E.1: (continued)

Bug ID	DynaMOSA+b	DynaMOSA
Closure-18	0	0.04
Closure-19	0.56	0.36
Closure-21	1	0.92
Closure-22	1	0.92
Closure-26	1	0.96
Closure-27	1	1
Closure-30	1	1
Closure-33	1	1
Closure-34	0.16	0.08
Closure-39	1	1
Closure-40	0.12	0.08
Closure-41	0.92	0.8
Closure-42	0	0.08
Closure-43	0	0.12
Closure-45	0.16	0.16
Closure-46	1	1
Closure-48	0.2	0.04
Closure-49	1	0.92
Closure-52	0.88	0.92
Closure-53	0.04	0
Closure-54	1	1
Closure-55	0.04	0.04
Closure-56	1	1
Closure-57	0	0.04
Closure-58	0	0.04
Closure-60	0	0.12
Closure-65	1	0.96
Closure-66	0.04	0
Closure-67	0.04	0
Closure-68	0	0.04
Closure-70	0.12	0
Closure-71	0.12	0.04
Closure-72	0.96	1
Closure-73	1	0.96
Closure-75	0.12	0
Closure-76	0.04	0.12
Closure-77	0.96	0.88
Closure-78	0.04	0
Closure-79	1	1
Closure-80	0.6	0.52
Closure-81	0.32	0.36
Closure-82	1	0.96
Closure-85	0.08	0
Closure-86	0.16	0.04
Closure-88	0.04	0
Closure-89	0.04	0.04
Closure-91	0.76	0.76
Closure-92	0	0.04

Bug ID	DynaMOSA+b	DynaMOSA
Closure-94	0.48	0.36
Closure-96	0.04	0
Closure-99	0.32	0.2
Closure-100	0.96	0.96
Closure-102	0.28	0.2
Closure-103	0.08	0.04
Closure-104	1	0.84
Closure-106	1	1
Closure-107	0	0.04
Closure-108	0.72	0.52
Closure-109	0.04	0
Closure-110	1	1
Closure-112	0.2	0.28
Closure-113	0.84	0.84
Closure-114	0.76	0.88
Closure-115	0.76	0.8
Closure-116	0.8	0.72
Closure-117	0.6	0.52
Closure-118	0.32	0.24
Closure-119	0.84	0.88
Closure-120	0.6	0.76
Closure-121	1	0.88
Closure-122	0.04	0.08
Closure-123	0.6	0.4
Closure-124	0.72	0.6
Closure-125	0.36	0.6
Closure-126	0.04	0
Closure-128	0.48	0.48
Closure-129	0.2	0.48
Closure-131	1	1
Closure-136	0.32	0.12
Closure-137	1	0.92
Closure-139	0.4	0.24
Closure-140	0.96	0.96
Closure-141	0.68	0.72
Closure-142	0.04	0.04
Closure-143	0.04	0.04
Closure-144	0.2	0.4
Closure-146	0.44	0.24
Closure-147	0.24	0.2
Closure-148	0	0.04
Closure-150	0.96	1
Closure-151	1	0.4
Closure-152	0	0.08
Closure-153	0.04	0
Closure-154	0.04	0
Closure-155	0.32	0.28
Closure-156	0.04	0.12

TABLE E.1: (continued)

Bug ID	DynaMOSA+b	DynaMOSA
Closure-157	0.2	0.04
Closure-158	0.08	0
Closure-160	0.68	0.24
Closure-162	0	0.04
Closure-163	0.04	0
Closure-164	1	1
Closure-165	1	1
Closure-167	0.56	0.44
Closure-170	0.92	1
Closure-171	0.84	0.88
Closure-172	0.76	0.8
Closure-173	0.92	0.92
Closure-174	1	1
Closure-175	0.96	0.4
Closure-176	0.2	0.2
Mockito-2	1	1
Mockito-9	0.48	0
Mockito-17	1	1
Mockito-23	0	0.04
Mockito-29	0.56	0.68
Mockito-35	1	1
Mockito-36	0.04	0

TABLE E.2: Mean and median number of bugs detected by DynaMOSA+b and DynaMOSA over the time budget spent.

Time (s)	Mean		Median		p-value	$\widehat{A}_{12}$
	Dyna-MOSA+b	Dyna-MOSA	Dyna-MOSA+b	Dyna-MOSA		
1	50.04	51.60	50	52	0.1234	0.37
2	61.40	**64.28**	62	**64**	**0.0079**	0.28
3	72.56	**75.28**	71	**74**	**0.0278**	0.32
4	81.00	82.80	79	82	0.0577	0.34
5	87.56	89.36	87	89	0.0922	0.36
6	94.12	95.56	94	97	0.2273	0.40
7	99.40	100.64	99	101	0.2755	0.41
8	104.52	105.00	104	105	0.7114	0.47
9	108.88	109.28	108	109	0.6265	0.46
10	112.56	112.40	112	112	0.9845	0.50
11	116.12	114.80	116	114	0.2624	0.59
12	119.04	118.08	119	118	0.2971	0.59
13	121.64	121.04	122	121	0.5520	0.55
14	123.84	123.48	124	124	0.6194	0.54
15	125.92	126.00	126	127	0.9922	0.50
16	128.56	128.20	129	128	0.6680	0.54
17	130.88	130.36	132	131	0.5207	0.55
18	132.96	132.24	133	132	0.4881	0.56
19	135.20	134.00	136	134	0.2758	0.59
20	137.36	135.60	138	135	0.0887	0.64
21	139.20	137.48	139	138	0.0574	0.66
22	140.84	139.16	141	139	0.0603	0.65
23	142.72	140.48	143	140	0.0615	0.65
24	144.64	142.16	145	142	0.0590	0.66
25	146.20	143.80	146	144	0.0780	0.64
26	**147.80**	144.76	**147**	145	**0.0145**	0.70
27	**149.48**	145.88	**149**	145	**0.0081**	0.72
28	**150.64**	147.12	**150**	147	**0.0088**	0.72
29	**151.76**	148.32	**152**	149	**0.0070**	0.72
30	**152.68**	149.56	**152**	150	**0.0199**	0.69
31	**153.92**	150.76	**153**	152	**0.0244**	0.68
32	**155.24**	152.04	**156**	153	**0.0160**	0.70
33	**156.44**	152.92	**157**	154	**0.0045**	0.73
34	**157.24**	153.96	**158**	155	**0.0085**	0.72
35	**158.28**	155.56	**159**	156	**0.0271**	0.68
36	**159.28**	156.16	**161**	156	**0.0154**	0.70
37	**160.24**	157.00	**161**	157	**0.0105**	0.71
38	**161.24**	158.08	**162**	159	**0.0162**	0.70
39	**162.08**	158.84	**163**	161	**0.0200**	0.69
40	**163.12**	159.88	**163**	161	**0.0189**	0.69

Table E.2: (continued)

Time (s)	Mean		Median		p-value	$\widehat{A}_{12}$
	Dyna-MOSA+b	Dyna-MOSA	Dyna-MOSA+b	Dyna-MOSA		
41	**163.88**	160.60	**165**	162	**0.0217**	0.69
42	**164.76**	161.56	**165**	163	**0.0293**	0.68
43	**165.68**	162.20	**165**	164	**0.0170**	0.70
44	**166.52**	162.72	**166**	164	**0.0108**	0.71
45	**167.36**	163.64	**167**	165	**0.0102**	0.71
46	**168.16**	164.40	**169**	166	**0.0067**	0.72
47	**168.96**	164.96	**170**	166	**0.0061**	0.73
48	**169.76**	165.64	**171**	167	**0.0075**	0.72
49	**170.60**	166.72	**172**	167	**0.0106**	0.71
50	**171.40**	167.64	**173**	168	**0.0149**	0.70
51	**172.80**	168.00	**175**	169	**0.0051**	0.73
52	**173.44**	168.80	**175**	170	**0.0073**	0.72
53	**174.08**	169.24	**175**	170	**0.0044**	0.73
54	**174.88**	169.96	**177**	171	**0.0031**	0.74
55	**175.60**	170.56	**179**	171	**0.0021**	**0.75**
56	**176.28**	171.48	**179**	172	**0.0031**	0.74
57	**176.96**	172.08	**179**	172	**0.0034**	0.74
58	**177.56**	172.72	**179**	172	**0.0027**	**0.75**
59	**178.48**	173.24	**181**	173	**0.0009**	**0.77**
60	**179.44**	173.80	**182**	174	**0.0006**	**0.78**
61	**179.84**	174.40	**182**	174	**0.0006**	**0.78**
62	**180.44**	175.04	**182**	174	**0.0007**	**0.78**
63	**181.16**	175.64	**183**	174	**0.0004**	**0.79**
64	**181.64**	176.24	**184**	175	**0.0006**	**0.78**
65	**182.08**	176.64	**184**	175	**0.0003**	**0.79**
66	**182.44**	177.04	**185**	176	**0.0004**	**0.79**
67	**183.00**	177.44	**185**	176	**0.0003**	**0.79**
68	**183.40**	178.04	**185**	177	**0.0004**	**0.79**
69	**183.72**	178.52	**186**	177	**0.0004**	**0.79**
70	**184.20**	179.20	**186**	179	**0.0006**	**0.78**
71	**184.68**	179.76	**186**	179	**0.0004**	**0.79**
72	**185.28**	180.20	**187**	181	**0.0006**	**0.78**
73	**185.72**	180.68	**187**	181	**0.0008**	**0.78**
74	**186.28**	181.08	**188**	181	**0.0005**	**0.79**
75	**186.80**	181.56	**188**	181	**0.0004**	**0.79**
76	**187.32**	182.12	**189**	182	**0.0009**	**0.77**
77	**188.00**	182.36	**189**	182	**0.0006**	**0.78**
78	**188.64**	183.08	**190**	182	**0.0012**	**0.77**
79	**189.28**	183.40	**191**	182	**0.0012**	**0.77**
80	**189.84**	183.80	**191**	183	**0.0007**	**0.78**
81	**190.32**	184.36	**191**	184	**0.0013**	**0.76**

Table E.2: (continued)

Time (s)	Mean		Median		p-value	$\widehat{A}_{12}$
	Dyna-MOSA+b	Dyna-MOSA	Dyna-MOSA+b	Dyna-MOSA		
82	**190.68**	184.80	**192**	184	**0.0019**	**0.76**
83	**191.08**	185.24	**192**	185	**0.0026**	**0.75**
84	**191.48**	185.76	**192**	185	**0.0036**	0.74
85	**191.92**	186.08	**193**	185	**0.0031**	0.74
86	**192.28**	186.28	**193**	186	**0.0020**	**0.75**
87	**192.84**	186.60	**193**	186	**0.0012**	**0.77**
88	**193.12**	186.80	**194**	187	**0.0008**	**0.78**
89	**193.68**	187.00	**194**	187	**0.0004**	**0.79**
90	**194.12**	187.32	**195**	187	**0.0004**	**0.79**
91	**194.52**	187.44	**195**	187	**0.0003**	**0.80**
92	**194.96**	187.84	**196**	187	**0.0003**	**0.80**
93	**195.24**	188.24	**196**	187	**0.0003**	**0.79**
94	**195.60**	188.68	**196**	188	**0.0004**	**0.79**
95	**196.04**	189.16	**197**	189	**0.0006**	**0.78**
96	**196.44**	189.48	**198**	189	**0.0004**	**0.79**
97	**196.88**	189.92	**198**	191	**0.0003**	**0.80**
98	**197.16**	190.20	**198**	191	**0.0004**	**0.79**
99	**197.40**	190.56	**198**	191	**0.0005**	**0.79**
100	**197.68**	190.76	**199**	191	**0.0003**	**0.79**
101	**198.04**	191.04	**199**	191	**0.0003**	**0.80**
102	**198.36**	191.24	**199**	191	**0.0002**	**0.80**
103	**198.80**	191.52	**199**	191	**0.0002**	**0.81**
104	**199.20**	192.08	**200**	192	**0.0002**	**0.81**
105	**199.72**	192.56	**201**	192	**0.0002**	**0.80**
106	**200.00**	192.88	**201**	192	**0.0002**	**0.81**
107	**200.40**	193.28	**201**	193	**0.0001**	**0.82**
108	**200.72**	193.52	**202**	193	**0.0001**	**0.82**
109	**201.00**	193.72	**202**	193	**0.0001**	**0.82**
110	**201.08**	194.16	**202**	194	**0.0002**	**0.81**
111	**201.36**	194.32	**202**	194	**0.0002**	**0.81**
112	**201.52**	194.68	**202**	195	**0.0003**	**0.80**
113	**201.68**	194.76	**202**	195	**0.0002**	**0.81**
114	**201.92**	194.96	**202**	196	**0.0002**	**0.81**
115	**202.12**	195.36	**202**	196	**0.0003**	**0.80**
116	**202.44**	195.72	**202**	196	**0.0002**	**0.80**
117	**202.68**	196.20	**202**	197	**0.0004**	**0.79**
118	**202.92**	196.40	**203**	197	**0.0003**	**0.79**
119	**203.32**	196.68	**203**	197	**0.0002**	**0.80**
120	**203.64**	197.16	**203**	197	**0.0004**	**0.79**

Bibliography

[1] Phillip Johnston and Rozi Harris. The boeing 737 max saga: Lessons for software organizations. *Software Quality Professional*, 21(3):4–12, 2019.

[2] Carol Clark. How to keep markets safe in the era of high-speed trading. *Chicago Fed Letter*, (303):1, 2012.

[3] Gordon Fraser and Andrea Arcuri. Evolutionary generation of whole test suites. In *2011 11th International Conference on Quality Software*, pages 31–40. IEEE, 2011.

[4] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *OOPSLA Companion*, pages 815–816, 2007.

[5] Corina S Păsăreanu and Neha Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 179–180. ACM, 2010.

[6] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and software Technology*, 43(14):833–839, 2001.

[7] Paul Baker, Mark Harman, Kathleen Steinhofel, and Alexandros Skaliotis. Search based approaches to component selection and prioritization for the next release problem. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 176–185. IEEE, 2006.

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

[9] Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International*

Conference on Software Testing, Verification and Validation (ICST), pages 1–12. IEEE, 2015.

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

[11] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 94–105, 2016.

[12] Aldeida Aleti and Lars Grunske. Test data generation with a kalman filter-based adaptive genetic algorithm. *Journal of Systems and Software*, 103:343–352, 2015.

[13] Carlos Oliveira, Aldeida Aleti, Yuan-Fang Li, and Mohamed Abdelrazek. Footprints of fitness functions in search-based software testing. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, page 1399–1407. Association for Computing Machinery, 2019. ISBN 9781450361118. doi: 10.1145/3321707.3321880.

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

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

[16] Carlos Oliveira, Aldeida Aleti, Lars Grunske, and Kate Smith-Miles. Mapping the effectiveness of automated test suite generation techniques. *IEEE Transactions on Reliability*, 67(3):771–785, 2018.

[17] Aldeida Aleti, Irene Moser, and Lars Grunske. Analysing the fitness landscape of search-based software testing problems. *Automated Software Engineering*, 24(3): 603–621, 2017.

[18] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated white-box test generation really help software testers? In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 291–301. ACM, 2013.

[19] Richard A DeMillo, A Jefferson Offutt, et al. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[20] Larry J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, 1990.

[21] Larry Joe Morell. A theory of error-based testing. Technical report, MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE, 1984.

[22] AJV Offutt. Automatic test data generation. 1989.

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

[24] Alireza Salahirad, Hussein Almulla, and Gregory Gay. Choosing the fitness function for the job: Automated generation of test suites that detect real faults. *Software Testing, Verification and Reliability*, 29(4-5):e1701, 2019.

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

[26] 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. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, pages 263–272. IEEE Press, 2017.

[27] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. Predicting defective lines

using a model-agnostic technique. *IEEE Transactions on Software Engineering*, pages 1–1, 2020. doi: 10.1109/TSE.2020.3023177.

[28] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. Predicting component failures at design time. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 18–27, 2006.

[29] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.

[30] Paulo André Faria de Freitas. Software repository mining analytics to estimate software component reliability. 2015.

[31] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Bug prediction based on fine-grained module histories. In *2012 34th international conference on software engineering (ICSE)*, pages 200–210. IEEE, 2012.

[32] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C Gall. Method-level bug prediction. In *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 171–180. IEEE, 2012.

[33] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering*, 33 (1):2–13, 2006.

[34] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, pages 9–9. IEEE, 2007.

[35] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 284–292. ACM, 2005.

[36] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 521–530. IEEE, 2008.

[37] Bora Caglayan, Burak Turhan, Ayse Bener, Mayy Habayeb, Andriy Miransky, and Enzo Cialini. Merits of organizational metrics in defect prediction: an industrial replication. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 89–98. IEEE Press, 2015.

[38] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. Change bursts as defect predictors. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 309–318. IEEE, 2010.

[39] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E James Whitehead Jr. Does bug prediction support human developers? findings from a google case study. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 372–381. IEEE Press, 2013.

[40] Chris Lewis and Rong Ou. Bug prediction at google, 2011. URL http://google-engtools.blogspot.com/2011/12/. Last accessed on: 16/09/2019.

[41] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. Lessons learned from using a deep tree-based model for software defect prediction in practice. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 46–57. IEEE Press, 2019.

[42] David Paterson, Jose Campos, Rui Abreu, Gregory M Kapfhammer, Gordon Fraser, and Phil McMinn. An empirical study on the use of defect prediction for test case prioritization. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 346–357. IEEE, 2019.

[43] Gregory Gay. The fitness function for the job: Search-based generation of test suites that detect real faults. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 345–355. IEEE, 2017.

[44] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. Continuous test generation: enhancing continuous integration with automated test generation. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 55–66. ACM, 2014.

[45] Manfred Broy, Ingolf H Kruger, Alexander Pretschner, and Christian Salzmann. Engineering automotive software. *Proceedings of the IEEE*, 95(2):356–373, 2007.

[46] The Apache Software Foundation. Apache commons math, 2019. URL https://github.com/apache/commons-math. Last accessed on: 19/09/2019.

[47] Zhiyuan Wan, Xin Xia, Ahmed E Hassan, David Lo, Jianwei Yin, and Xiaohu Yang. Perceptions, expectations, and challenges in defect prediction. *IEEE Transactions on Software Engineering*, 46(11):1241–1266, 2018.

[48] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100, 2009.

[49] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2011.

[50] Seyedrebvar Hosseini, Burak Turhan, and Dimuthu Gunarathna. A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Transactions on Software Engineering*, 45(2):111–147, 2017.

[51] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering*, 22(2):852–893, 2017.

[52] Gerardo Canfora, Michele Ceccarelli, Luigi Cerulo, and Massimiliano Di Penta. How long does a bug survive? an empirical study. In *2011 18th Working Conference on Reverse Engineering*, pages 191–200, 2011. doi: 10.1109/WCRE.2011.31.

[53] Kiran Lakhotia, Mark Harman, and Hamilton Gross. Austin: An open source tool for search based software testing of c programs. *Information and Software Technology*, 55(1):112–125, 2013.

[54] C Chao, J Komada, Q Liu, M Muteja, Y Alsalqan, and C Chang. An application of genetic algorithms to software project management. *Proceedings of the 9th international advanced science and technology*, pages 247–252, 1993.

[55] Enrique Alba and Francisco Chicano. Acohg: Dealing with huge graphs. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 10–17, 2007.

[56] Marco Ferreira, Francisco Chicano, Enrique Alba, and JA Gómez-Pulido. Detecting protocol errors using particle swarm optimization with java pathfinder. In *Proceedings of the High Performance Computing & Simulation Conference (HPCS'08)*, pages 319–325. Citeseer, 2008.

[57] Salah Bouktif, Houari Sahraoui, and Giuliano Antoniol. Simulated annealing for improving software quality prediction. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1893–1900, 2006.

[58] Yi Bian, Serkan Kirbas, Mark Harman, Yue Jia, and Zheng Li. Regression test case prioritisation for guava. In *International Symposium on Search Based Software Engineering*, pages 221–227. Springer, 2015.

[59] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. Combining multiple coverage criteria in search-based unit test generation. In *International Symposium on Search Based Software Engineering*, pages 93–108. Springer, 2015.

[60] Christopher L Simons and Ian C Parmee. Single and multi-objective genetic operators in object-oriented conceptual software design. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1957–1958, 2006.

[61] Mark O'Keeffe and Mel O Cinnéide. Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4):502–516, 2008.

[62] Giuliano Antoniol, Massimiliano Di Penta, and Mark Harman. Search-based techniques applied to optimization of project planning for a massive maintenance project. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 240–249. IEEE, 2005.

[63] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1):1–61, 2012.

[64] Joachim Wegener and Oliver Bühler. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *Genetic and Evolutionary Computation Conference*, pages 1400–1412. Springer, 2004.

[65] Wasif Afzal, Richard Torkar, Robert Feldt, and Greger Wikstrand. Search-based prediction of fault-slip-through in large software projects. In *2nd International Symposium on Search Based Software Engineering*, pages 79–88. IEEE, 2010.

[66] Seunghee Han, Jaeuk Kim, Geon Kim, Jaemin Cho, Jiin Kim, and Shin Yoo. Preliminary evaluation of path-aware crossover operators for search-based test data generation for autonomous driving. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 44–47. IEEE, 2021.

[67] Richard M Everson and Jonathan E Fieldsend. Multiobjective optimization of safety related systems: An application to short-term conflict alert. *IEEE Transactions on Evolutionary Computation*, 10(2):187–198, 2006.

[68] Julian Thomé, Alessandra Gorla, and Andreas Zeller. Search-based security testing of web applications. In *Proceedings of the 7th International Workshop on Search-Based Software Testing*, pages 5–14, 2014.

[69] Lionel C Briand, Yvan Labiche, and Yihong Wang. An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering*, 29(7):594–607, 2003.

[70] Daniel Di Nardo, Fabrizio Pastore, Andrea Arcuri, and Lionel Briand. Evolutionary robustness testing of data processing systems using models and data mutation (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 126–137. IEEE, 2015.

[71] Nigel Tracey, John Clark, John McDermid, and Keith Mander. A search-based automated test-data generation framework for safety-critical systems. In *Systems engineering for business process change: new directions*, pages 174–213. Springer, 2002.

[72] Gordon Fraser and Andrea Arcuri. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical Software Engineering*, 20(3):611–639, 2015.

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

[74] Anjana Perera, Aldeida Aleti, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Burak Turhan, Lisa Kuhn, and Katie Walker. Search-based fairness testing for regression-based machine learning systems. *Empirical Software Engineering*, 27 (3):1–36, 2022.

[75] Reyhaneh Jabbarvand, Jun-Wei Lin, and Sam Malek. Search-based energy testing of android. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1119–1130. IEEE, 2019.

[76] Aitor Arrieta, Shuai Wang, Goiuria Sagardui, and Leire Etxeberria. Search-based test case prioritization for simulation-based testing of cyber-physical system product lines. *Journal of Systems and Software*, 149:1–34, 2019.

[77] Alessio Gambi, Marc Mueller, and Gordon Fraser. Automatically testing self-driving cars with search-based procedural content generation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 318–328, 2019.

[78] Alessio Gambi, Marc Müller, and Gordon Fraser. Asfault: Testing self-driving car software using search-based procedural content generation. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 27–30. IEEE, 2019.

[79] Annibale Panichella. Beyond unit-testing in search-based test case generation: Challenges and opportunities. In *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*, pages 7–8. IEEE, 2019.

[80] 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):8, 2014.

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

[82] Andreas Windisch, Stefan Wappler, and Joachim Wegener. Applying particle swarm optimization to software testing. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1121–1128, 2007.

[83] Kamel Ayari, Salah Bouktif, and Giuliano Antoniol. Automatic mutation test input data generation via ant colony. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1074–1081, 2007.

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

[85] Mark Harman and Phil McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 73–83. ACM, 2007.

[86] Bogdan Korel. Automated software test data generation. *IEEE Transactions on software engineering*, 16(8):870–879, 1990.

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

[88] 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):226–247, 2009.

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

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

[91] Andrea Arcuri, José Campos, and Gordon Fraser. Unit test generation during software development: Evosuite plugins for maven, intellij and jenkins. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 401–408. IEEE, 2016.

[92] José Campos, Annibale Panichella, and Gordon Fraser. Evosuite at the sbst 2019 tool competition. In *Proceedings of the 12th International Workshop on Search-Based Software Testing*, pages 29–32. IEEE Press, 2019.

[93] Gordon Fraser and Andrea Arcuri. Evosuite at the sbst 2016 tool competition. In *2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*, pages 33–36. IEEE, 2016.

[94] G. Fraser and A. Arcuri. Evosuite at the sbst 2013 tool competition. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 406–409, March 2013. doi: 10.1109/ICSTW.2013.53.

[95] Gordon Fraser and Andrea Arcuri. Evosuite at the second unit testing tool competition. In Tanja E.J. Vos, Kiran Lakhotia, and Sebastian Bauersfeld, editors, *Future Internet Testing*, pages 95–100, Cham, 2014. Springer International Publishing. ISBN 978-3-319-07785-7.

[96] Gordon Fraser, José Miguel Rojas, and Andrea Arcuri. Evosuite at the sbst 2018 tool competition. In *Proceedings of the 11th International Workshop on Search-Based Software Testing*, SBST '18, pages 34–37, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5741-8. doi: 10.1145/3194718.3194729. URL http://doi.acm.org/10.1145/3194718.3194729.

[97] Gordon Fraser, José Miguel Rojas, José Campos, and Andrea Arcuri. Evosuite at the sbst 2017 tool competition. In *Proceedings of the 10th International Workshop on Search-Based Software Testing*, SBST '17, pages 39–41, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-2789-1. doi: 10.1109/SBST.2017..6. URL https://doi.org/10.1109/SBST.2017..6.

[98] EvoSuite. Evosuite - automated generation of junit test suites for java classes, 2019. URL https://github.com/EvoSuite/evosuite. Last accessed on: 29/11/2019.

[99] Gordon Fraser. Evosuite - automatic test suite generation for java, 2018. URL http://www.evosuite.org/. Last accessed on: 19/09/2019.

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

[101] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665, 2014.

[102] Kalyanmoy Deb. Multi-objective optimization. In *Search methodologies*, pages 403–449. Springer, 2014.

[103] Suraj Yatish, Jirayus Jiarpakdee, Patanamon Thongtanunam, and Chakkrit Tantithamthavorn. Mining software defects: Should we consider affected releases? In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 654–665. IEEE, 2019.

[104] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, 22(10):751–761, 1996.

[105] Hoa Khanh Dam, Truyen Tran, Trang Thi Minh Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering*, 2018.

[106] Atlassian. Jira, 2022. URL https://www.atlassian.com/software/jira. Last accessed on: 30/03/2022.

[107] Bugzilla. Bugzilla, 2022. URL https://www.bugzilla.org. Last accessed on: 30/03/2022.

[108] Git. Git, 2019. URL https://git-scm.com. Last accessed on: 19/09/2019.

[109] CVS. Cvs - open source version control, 2019. URL http://cvs.nongnu.org. Last accessed on: 30/03/2022.

[110] Subversion. Apache subversion, 2021. URL https://subversion.apache.org. Last accessed on: 30/03/2022.

[111] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

[112] Sunghun Kim, Thomas Zimmermann, Kai Pan, E James Jr, et al. Automatic identification of bug-introducing changes. In *21st IEEE/ACM international conference on automated software engineering (ASE'06)*, pages 81–90. IEEE, 2006.

[113] Maurício Aniche. *Java code metrics calculator (CK)*, 2015. Available in https://github.com/mauricioaniche/ck/.

[114] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[115] Tao Wang and Wei-hua Li. Naive bayes software defect prediction model. In *2010 International Conference on Computational Intelligence and Software Engineering*, pages 1–4. Ieee, 2010.

[116] Gerardo Canfora, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Multi-objective cross-project defect prediction. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 252–261. IEEE, 2013.

[117] Thilo Mende, Rainer Koschke, and Marek Leszak. Evaluating defect prediction models for a large evolving software system. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 247–250. IEEE, 2009.

[118] David Gray, David Bowes, Neil Davey, Yi Sun, and Bruce Christianson. Using the support vector machine as a classification method for software defect prediction with static code metrics. In *International Conference on Engineering Applications of Neural Networks*, pages 223–234. Springer, 2009.

[119] A Günes Koru and Hongfang Liu. An investigation of the effect of module size on defect prediction using static measures. In *Proceedings of the 2005 workshop on Predictor models in software engineering*, pages 1–5, 2005.

[120] Jun Wang, Beijun Shen, and Yuting Chen. Compressed c4. 5 models for software defect prediction. In *2012 12th International Conference on Quality Software*, pages 13–16. IEEE, 2012.

[121] Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, and Premkumar Devanbu. Bugcache for inspections: hit or miss? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 322–331. ACM, 2011.

[122] 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):340–355, 2005.

[123] Thelma Elita Colanzi, Wesley Klewerton Guez Assunção, Paulo Roberto Farah, Silvia Regina Vergilio, and Giovani Guizzo. A review of ten years of the symposium on search-based software engineering. In *International Symposium on Search Based Software Engineering*, pages 42–57. Springer, 2019.

[124] Xiaoxing Yang, Ke Tang, and Xin Yao. A learning-to-rank approach to software defect prediction. *IEEE Transactions on Reliability*, 64(1):234–246, 2015. doi: 10.1109/TR.2014.2370891.

[125] Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on software engineering*, 26(7):653–661, 2000.

[126] Thomas Shippey, Tracy Hall, Steve Counsell, and David Bowes. So you need more method level datasets for your software defect prediction? voilà! In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–6, 2016.

[127] Jirayus Jiarpakdee, Chakkrit Kla Tantithamthavorn, and John Grundy. Practitioners' perceptions of the goals and visual explanations of defect prediction models. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 432–443. IEEE, 2021.

[128] Jirayus Jiarpakdee, Chakkrit Kla Tantithamthavorn, Hoa Khanh Dam, and John Grundy. An empirical study of model-agnostic techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 48(1):166–185, 2022. doi: 10.1109/TSE.2020.2982385.

[129] Steffen Herbold. On the costs and profit of software defect prediction. *IEEE Transactions on Software Engineering*, 47(11):2617–2631, 2021. doi: 10.1109/TSE.2019.2957794.

[130] Eran Hershkovich, Roni Stern, Rui Abreu, and Amir Elmishali. Prioritized test generation guided by software fault prediction. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 218–225. IEEE, 2021.

[131] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F Bissyandé, Yves Le Traon, and Koushik Sen. Selecting fault revealing mutants. *Empirical Software Engineering*, 25(1):434–487, 2020.

[132] Joengju Sohn and Shin Yoo. Empirical evaluation of fault localisation using code and change metrics. *IEEE Transactions on Software Engineering*, 2019.

[133] Eran Hershkovich, Roni Stern, Rui Abreu, and Amir Elmishali. Prediction-guided software test generation. In *Proceedings of the 30th International Workshop on Principles of Diagnosis DX'19*, 2019.

[134] Amir Elmishali, Roni Stern, and Meir Kalech. Debguer: A tool for bug prediction and diagnosis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 9446–9451, 2019.

[135] Gregory Gay. Generating effective test suites by combining coverage criteria. In *International Symposium on Search Based Software Engineering*, pages 65–82. Springer, 2017.

[136] Rene Just. Defects4j - a database of real faults and an experimental infrastructure to enable controlled experiments in software engineering research, 2019. URL https://github.com/rjust/defects4j. Last accessed on: 02/10/2019.

[137] Aldeida Aleti and Matias Martinez. E-apr: Mapping the effectiveness of automated program repair. *arXiv preprint arXiv:2002.03968*, 2020.

[138] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, pages 609–620. IEEE Press, 2017.

[139] Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. Bugs. jar: a large-scale, diverse dataset of real-world java bugs. In *Proceedings of*

the 15th International Conference on Mining Software Repositories, pages 10–13, 2018.

[140] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*, 2019. URL https://arxiv.org/abs/1901.06024.

[141] Steffen Herbold, Alexander Trautsch, Benjamin Ledel, Alireza Aghamohammadi, Taher Ahmed Ghaleb, Kuljit Kaur Chahal, Tim Bossenmaier, Bhaveet Nagaria, Philip Makedonski, Matin Nili Ahmadabadi, Kristóf Szabados, Helge Spieker, Matej Madeja, Nathaniel Hoy, Valentina Lenarduzzi, Shangwen Wang, Gema Rodríguez-Pérez, Ricardo Colomo Palacios, Roberto Verdecchia, Paramvir Singh, Yihao Qin, Debasish Chakroborti, Willard Davis, Vijay Walunj, Hongjun Wu, Diego Marcilio, Omar Alam, Abdullah Aldaeej, Idan Amit, Burak Turhan, Simon Eismann, Anna-Katharina Wickert, Ivano Malavolta, Matús Sulír, Fatemeh Fard, Austin Z. Henley, Stratos Kourtzanidis, Eray Tuzun, Christoph Treude, Simin Maleki Shamasbi, Ivan Pashchenko, Marvin Wyrich, James Davis, Alexander Serebrenik, Ella Albrecht, Ethem Utku Aktas, Daniel Strüber, and Johannes Erbel. Large-scale manual validation of bug fixing commits: A fine-grained analysis of tangling. *CoRR*, abs/2011.06244, 2020. URL https://arxiv.org/abs/2011.06244.

[142] Nan Li and Jeff Offutt. Test oracle strategies for model-based testing. *IEEE Transactions on Software Engineering*, 43(4):372–395, 2016.

[143] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

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

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

[146] Catherine O Fritz, Peter E Morris, and Jennifer J Richler. Effect size estimates: current use, calculations, and interpretation. *Journal of experimental psychology: General*, 141(1):2, 2012.

[147] Soner Yigit and Mehmet Mendes. Which effect size measure is appropriate for one-way and two-way anova models? a monte carlo simulation study. *Revstat Statistical Journal*, 16(3):295–313, 2018.

[148] John W Tukey. Comparing individual means in the analysis of variance. *Biometrics*, pages 99–114, 1949.

[149] Ning Li, Martin Shepperd, and Yuchen Guo. A systematic review of unsupervised learning techniques for software defect prediction. *Information and Software Technology*, 122:106287, 2020.

[150] Martin Shepperd, David Bowes, and Tracy Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, 40(6):603–616, 2014.

[151] Jingxiu Yao and Martin Shepperd. Assessing software defection prediction performance: Why using the matthews correlation coefficient matters. In *Proceedings of the Evaluation and Assessment in Software Engineering*, pages 120–129. 2020.

[152] Jingxiu Yao and Martin Shepperd. The impact of using biased performance metrics on software defect prediction research. *Information and Software Technology*, 139: 106664, 2021.

[153] Claudia Ayala, Burak Turhan, Xavier Franch, and Natalia Juristo. Use and misuse of the term experiment in mining software repositories research. *IEEE Transactions on Software Engineering*, 2021.

[154] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 213–224. IEEE, 2016.

[155] Sebastiano Panichella, Alessio Gambi, Fiorella Zampetti, and Vincenzo Riccio. Sbst tool competition 2021. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 20–27, 2021. doi: 10.1109/SBST52555.2021.00011.

[156] Ignacio Manuel Lebrero Rial and Juan P. Galeotti. Evosuitedse at the sbst 2021 tool competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 30–31, 2021. doi: 10.1109/SBST52555.2021.00013.

[157] Andre Freitas. Schwa, 2015. URL https://pypi.org/project/Schwa. Last accessed on 16/09/2019.

[158] André Freitas. schwa, 2015. URL https://github.com/andrefreitas/schwa. Last accessed on 16/09/2019.

[159] Martin Fowler and Matthew Foemmel. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, 122:14, 2006.

[160] Andrea Arcuri and Gordon Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3):594–623, 2013.

[161] Andrew Habib and Michael Pradel. How many of all bugs do we find? a study of static bug detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM*, pages 317–328, 2018.

[162] Nan Li and Jeff Offutt. Test oracle strategies for model-based testing. *IEEE Transactions on Software Engineering*, 43(4):372–395, 2017. doi: 10.1109/TSE.2016.2597136.

[163] Franz Faul, Edgar Erdfelder, Albert-Georg Lang, and Axel Buchner. G* power 3: A flexible statistical power analysis program for the social, behavioral, and biomedical sciences. *Behavior research methods*, 39(2):175–191, 2007.

[164] Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951.

[165] Jacob Cohen. A power primer. *Psychological bulletin*, 112(1):155, 1992.

[166] Hangcheng Liu. *Comparing Welch ANOVA, a Kruskal-Wallis test, and traditional ANOVA in case of heterogeneity of variance*. Virginia Commonwealth University, 2015.

[167] Robert M Carroll and Lena A Nordholm. Sampling characteristics of kelley's ε and hays' ω. *Educational and Psychological Measurement*, 35(3):541–554, 1975.

[168] A Gunes Koru and Hongfang Liu. Building effective defect-prediction models in practice. *IEEE software*, 22(6):23–29, 2005.

[169] Yufeng Zhang, Zhenbang Chen, Ji Wang, Wei Dong, and Zhiming Liu. Regular property guided dynamic symbolic execution. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 643–653. IEEE, 2015.