Thesis: Mapping the Effectiveness of Automated Test Suite Generation Techniques

by

Carlos Oliveira, Master of Applied Computer Science

Thesis

Submitted by Carlos Oliveira

for fulfillment of the Requirements for the Degree of

Doctor of Philosophy (0190)

Supervisor: Dr. Aldeida Aleti

Associate Supervisor: Prof. Kate Smith-Miles, Dr. Yuan-Fang Li

Caulfield School of Information Technology
Monash University

September, 2019

Thesis: Mapping the Effectiveness of Automated Test Suite Generation Techniques

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Carlos Oliveira
September 22, 2019

Contents

List of Tables

List of Figures

Glossary

Automated Test Suite Generation [ATSG] is the process of generating test suites (a set of tests cases) automatically. 10

Automated Test Suite Generation Techniques [ATSGT] the method used to generate test suites automatically. 10

Class Under Test [CUT] a Java programming language class. 10

Features also referred as software features, is a distinguishing characteristic of a software item (Java classes). 10

Thesis: Mapping the Effectiveness of Automated Test Suite Generation Techniques

Carlos Oliveira, Master of Applied Computer Science
`carlos.guimaraes@monash.edu`
Monash University, 2019

Supervisor: Dr. Aldeida Aleti
`aldeida.aleti@monash.edu`
Associate Supervisor: Prof. Kate Smith-Miles, Dr. Yuan-Fang Li
`{kate.smith-miles@monash.edu, yuanfang.li}@monash.edu`

Abstract

Automated Test Suite Generation (ATSG) is an important topic in Software Engineering, with a wide range of techniques and tools being used in academia and industry. While their usefulness is widely recognized, due to the labor-intensive nature of the task, the effectiveness of the different techniques in automatically generating test cases for different software systems is not thoroughly understood. Despite many studies introducing various ATSG techniques (ATSGT), much remains to be learned, however, about what makes a particular technique work well (or not) for a specific software system.

In this thesis, we seek an answer to the question "*What features of a software system impact the effectiveness of ATSGTs?*" Once these features are identified, can they be used to select the most effective ATSGT for a particular software system? To this end, we have implemented the META tool (Mapping the Effectiveness of Test Automation), a new framework that identifies important software features that can be used to select suitable ATSGTs to apply to new software systems. META is an alternative to the current methodology used to assess ATSGT performance.

We evaluate the framework in two controlled experiments. In both experiments META successfully identified the software features associated with ATSGT performance indicating that ATSGTs are problem dependent. The area where the ATSGTs are expected to have a good performance has been successfully mapped. Additionally, decision trees have been generated by the META tool with an accuracy higher than 80%, as shown by n-fold cross validations.

Part I

Introduction and Background

Chapter 1

Introduction

Automated Test Suite Generation (ATSG) addresses the problem of automating the process of generating sets of test-cases given a particular testing goal, such as structural [54, 69, 70, 96], functional [107], non-functional [108], and state-based properties [34]. ATSG plays a critical role in the testing process due to the difficulty of manual test case generation. Nowadays, software systems are increasingly sophisticated and large, hence manually generating test cases that test all parts of the system is intellectually demanding and time-consuming. Furthermore, recent software development practices such as continuous delivery and integration demand for effective automated approaches to testing.

Due to its significant role in ensuring bug-free software, ATSG has been widely investigated, and a plethora of tools have been introduced. Investigations into ATSG techniques (ATSGT) started in the early 70s, and the number of papers in the area has increased since then, as shown by these comprehensive surveys [1, 4, 31, 7, 68]. While their usefulness is widely recognized, due to the labour-intensive nature of the software testing, the suitability of the different techniques in automatically generating test cases for different software systems is not thoroughly understood [7].

Research introducing new techniques or experimental studies investigating the performance of different ATSGTs usually is based on a small set of CUTs (Classes Under Test). These works present little information about the CUTs characteristics and how they are selected [47], offering little insight into the external validity of the findings. Interestingly, among 50 studies, more than one-third of the papers consider exclusively container classes (e.g., vectors and lists), which are not very common in industrial systems [47]. Furthermore, most papers are only describing the benefits of the newly introduced technique and the innovation carried out during development, while just a few mention the limitations or present negative results [7].

This raises questions in terms of how ATSGTs are evaluated, and whether the datasets used in experimental studies are diverse enough to rigorously measure the capabilities of the different techniques. Results claiming the superior performance of an ATSGT on a selected set of CUTs may not be generalizable to untested datasets. Ideally, an experimental study would present a description of the conditions under which an ATSG technique can be expected to succeed or fail, however, this is rarely included in published studies.

Furthermore, in the significant majority of research papers (if not all), the newly developed technique presents a superior performance in all considered CUTs. It seems unwise to believe that a specific ATSGT might always be superior in all possible scenarios. It is very realistic to expect that any technique has weaknesses and that some CUTs could be conceived where the technique would be less effective than others. It might also happen that in some specific cases their competitive advantage disappears. This fact can be associated with our current research culture, which leads researchers to think that negative results are somehow less of a contribution than positive ones. This mindset leads authors to only expose the strengths of a newly developed technique.

This behaviour might lead to a growth of papers showing ATSGTs reporting bad performance when tested in datasets that are different from the original study [62]. This type of study, however, are putting some light into the flaws of the current methodology used in performance assessment, raising questions like: What caused the techniques to have bad performance? Is the technique only effective in the original study dataset? Is the original dataset representative of real world classes? Is the ATSGT performance associated to any specific characteristics of the original dataset? Can we extract and measure these characteristics and associate them to the ATSGT performance? None of these question have been answered yet.

In this thesis, we propose an alternative methodology called META (Mapping the Effectiveness of Test Automation) to help answering all these questions. In essence, the META framework can be used to characterize the features of CUTs that have an impact on the ATSGT effectiveness. We show how such features can be measured for CUTs, and how the footprints of ATSGT (regions where ATSGT strengths are expected) can be visualized across the CUT space. The META framework can be used to make performance prediction, enabling the selection of the most suitable ATSGT according to the CUT features. The results can also lead to improvements since one of the aims of the META framework is to reveal ATSGT weaknesses.

1.1 Research Problems

Researchers are every day working on evermore sophisticated techniques for automated test suite generation. For each new technique developed, experimental studies are conducted to validate ATSGT performance. This experiments are usually based on publicly available collections of benchmark datasets. The conclusions from these experiments are usually not insightful and are limited by the scale of the studies that typically restrict the type or size of benchmark instances used.

We believe that performance improvement is generally the driving factor for the development of a new ATSGT. Unfortunately, we have reached a point that for every new paper proposing a new ATSGT, the general expectation of the average reviewer seems to be that the evaluation should show a result that surpasses the current state-of-art. The "ideal" proposal of a new ATSGT, of course, would demonstrate to the reviewer that it is better and faster. In most occasions, only with a clear win in both respects, the paper might have a chance to receive a clear acceptance by the reviewers [62].

This research culture has reflected negatively in the methodology currently in use for ATSGT evaluation. For example, some results cannot be reproduced due to the lack of details provided. This raises questions in terms of how ATSGTs are evaluated. One may claim, for example, that the datasets used in experimental studies are not diverse enough to rigorously measure the capabilities of the different techniques, compromising the external validity of the results.

Results claiming the superior performance of an ATSGT over other techniques on a selected set of CUTs may not be generalizable to untested datasets. This issue would be easily fixed if the experimental study had included a description of the conditions under which an ATSGT can be expected to succeed or fail. This information, however, is rarely seen in published studies and the reason for this might be that the current methodology does not require this type of analysis.

In conclusion, we believe that the current methodology used in ATSGT assessment is limiting the progress in the Software Testing Community by discouraging researches to publish bad ATSGT results. Papers with no clear superior performance over the current state of art ATSGT are highly likely to be rejected. We claim that there are always scenarios where a technique will fail or at least lose its competitive advantage. So, instead of rejecting the research, the community should require justification and explanation for positive and negative results.

1.2 Research Goal and Objectives

The main goal of this thesis is to introduce an alternative to the current methodology used in the assessment of ATSGTs. The alternative methodology aims to assess the effectiveness of ATSGTs by providing information about their suitability according to the features of the software systems, which as a consequence, enables the selection of the best technique for a particular software system. The alternative methodology, unlike the commonly used, aims to characterize both strengths and weaknesses of ATSGTs by using software features. The scope of this work is software systems developed using Object Oriented (OO) languages such as Java. However, the proposed framework can be, theoretically, applied to any software to which features are available.

In achieving this goal, we address six strategic objectives:

RO1 Propose an alternative methodology to assess ATSGTs performance using Software Features;

RO2 Define a suitable set of software features to characterize CUTs;

RO3 Create a benchmark set that enables ATSGTs assessment;

RO4 Identify the software features that are associated with ATSGT hardness;

RO5 Visualize performance using software features;

RO6 Predict ATSGTs performance using software features;

1.3 Thesis Organization

This thesis is structured as follows:

Part I Chapter 2 presents the Automated Software Testing Generation research area, introducing concepts required to a clear understanding of the ATSGTs being studied. Chapter 3 presents a comprehensive literature review regarding ATSGTs performance assessment.

Part II Chapter 4 presents the alternative framework to assess ATSGT performance [**RO1**]. Chapter 5 introduces a broad and complex set of software features to characterize software systems [**RO2**]. Additionally, a benchmark analysis is performed to help developing a benchmark set to truly challenge ATSGTs [**RO3**].

Part III Chapter 6 and 7 present two controlled experiments to validate the new framework to ATSGT assessment. In total, 6 ATSGTs are assessed, their performances are associated with software features [**RO4**] and visualized in a 2-D space using a dimensionality reduction technique [**RO5**]. Furthermore, decision trees are created using software features, allowing the ATSGT performance to be predicted [**RO6**].

Part IV presents the final considerations of this thesis and also plans for future works.

1.4 Publications

Publications arising from this thesis include:

Carlos Oliveira, Aldeida Aleti, Yuan-Fang Li, Mohamed Abdelrazek (2019), Footprints of Fitness Functions in Search-Based Software Testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'19)*. Accepted.

Carlos Oliveira, Aldeida Aleti, Kate Smiths-Miles, Lars Grunske (2018), Mapping the Effectiveness of Automated Test Suite Generation Techniques. In *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 771-785, Sept. 2018.

Carlos Oliveira (2017), Mapping Hardness of Automated Software Testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Doctoral Symposium, Santa Barbara, USA, pp. 440-443.

Chapter 2

Automated Software Testing

2.1 Introduction

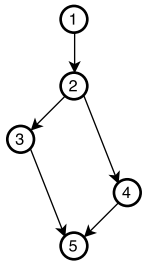
Automated Software Testing (AST) emerged from the attempt of organizations to do more with less. They wanted to test their software adequately but within a minimum schedule. A good definition of Automated Software Testing is: "The management and performance of test activities, to include the development and execution of test scripts to verify test requirements, using an automated test tool" [38].

Automated Test Suite Generation (ATSG) is one of the research areas inside AST. ATSGT refers to the generation of test suites (a test suite is a set of test cases) with no manual intervention. Once tests have been automated, they can be run quickly and repeatedly. A test case includes test inputs, execution conditions, and expected results developed for a particular objective, known as the oracle [29]. To illustrate what a test case looks like, consider the method `max` in Figure 2.1a. A potential test case for this method is `max(1,2)`, which executes the `if` part of the branch and expects as output the value 2. The test case `max(2,1)` on the other hand covers the `else` part of the branch. As a single test case may not cover all lines of code, many approaches to ATSG produce a suite of test cases.

```
int max(int a, int b){
1   int max=null;

2   if(a<=b){
3       max=b;
    }
    else{
4       max=a;
    }
5    return max;
}
```

(a) Method `max`.



(b) CFG of `max`.

Figure 2.1: Method `max` and its control flow graph (CFG).

The generation of a test case can be informed by the program structure and/or source code, the software specification and/or design models, information about the input/output data space, and information dynamically obtained from program execution [7]. A program structure can be represented as a Control Flow Graph (CFG) using graph notation. A CFG is a direct graph where the nodes correspond to the basic blocks (set of statements in a program) and the edges represent control flow paths [5]. The CFG is used by many ATSGTs to guide the generation of test suites.

ATSG plays a critical role in the testing process due to the complexity of the manual test case generation. In the two decades, software systems became increasingly complex with projects containing hundreds of thousands of lines of code [39]. Therefore, manually generating test cases for such projects is at least unreasonable. Moreover, currently methodologies such agile development requires efficiency, therefore, demanding for effective automated approaches to testing.

2.2 Automated Test Suite Generation - Techniques

The automation of the test suite generation has been researched for decades, since at least the 1970s. The approaches differ in many aspects, and have been extensively surveyed [1, 4, 31, 7, 68].

Random Testing (RT) is one of the earliest techniques, and one of the most fundamental testing methods [7]. The technique is easy to implement and can be combined with other automated testing techniques. Even if a software system has incomplete specifications, and the source code is unavailable, random testing remains a practically feasible technique [7]. RT is also one of the few testing techniques for which theoretical results exist in terms of fault detection capability [110].

Adaptive Random Testing (ART) is an enhancement to RT [17, 18], which is based on the assumption that failing test cases are usually grouped into contiguous regions. The method, however, was found to be highly inefficient in real software systems, even on trivial problems. For example, in the Triangle Classification program, Random Testing finds failures in few milliseconds, whereas ART's execution time is prohibitive [9]. The triangle classification program is a benchmark used in many testing papers. Assuming three non-zero, non-negative integer lengths for the sides of a triangle, the program decides if the triangle is isosceles, scalene, equilateral, or invalid [68]. For this reason, we choose RT as one of the techniques to investigate in this study.

Search-Based Software Testing (SBST) [54] uses search algorithms, such as genetic algorithms [58, 93, 2, 3]. These algorithms require a fitness function to guide the search towards high-quality solutions (a solution, in this case, is a test suite), which is based on the testing goals. SBST has been applied to a wide variety of testing goals including structural [54, 69, 70, 96], functional [107], non-functional [108], and state-based properties [34]. Different testing approaches have been developed with search techniques, such as integration testing [24, 15], mutation testing [52, 115], regression testing [102, 114], stress testing [33] and SBST approaches for web applications [6].

2.2.1 Random Testing (RT)

RT is generally regarded as not only a simple but also an intuitively appealing technique [19, 12]. In random testing, test cases are randomly generated based on a uniform distribution or according to the operational profile. A pool of randomly generated inputs is maintained, and a probability is used for either reusing a generated test case or creating a new one.

The main merits of random testing include the availability of efficient algorithms to generate test cases, and its ability to infer reliability and statistical estimates [19]. It has been shown that RT can be cost effective in many cases, providing a very high segment and branch coverage [37]. For more complex software or more sophisticated criteria, RT may fail to generate appropriate data in any reasonable time-frame [116]. Due to its simplicity and ease of implementation, RT is a popular technique for automated test case generation. There are no clear guidelines, however, to inform practitioners about its effectiveness on particular software systems with particular features. Furthermore, there is still a substantial lack of understanding of how exactly the performance of RT depends on software characteristics.

2.2.2 Whole Suite with Archive (WSA)

WSA [84] is a Search-Based Software Testing (SBST) method. SBST techniques employ search algorithms, such as a Hill Climbing [71], Simulated Annealing [98] and Genetic Algorithms [59], to automate or partially automate a testing task, e.g., the automatic generation of test data. The key to the optimization process is a problem-specific fitness function. The role of the fitness function is to guide the search to high-quality solutions from a potentially vast search space, within a practical time limit, which is why these methods are often known as fitness-guided automated testing. Work on SBST dates back to 1976 [71], with interest in the area beginning to gather pace in the 1990s.

WSA is implemented in EvoSuite [46] and uses Monotonic GA. To measure the quality of a solution, WSA uses *branch and method coverage* as a fitness function. The goal is to cover as many control structures, such as `if`, `while` statements, and methods as possible, by inspecting the logical predicates that result in both true and false. For all branches to be covered, each predicate must be executed at least twice, resulting in true and false values. In the considered SBST method, the branch distance $d(b, T)$ for each branch b in test suite T is defined as

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

The distance $d_{\min}^{b,T}$ is 0 if at least one of the branches has been covered, and > 0 otherwise. The fitness function that is minimized by WSA is:

$$f(T) = |M| - |M_T| + \sum_{b \in B} d(b,T), \tag{2.2}$$

where $|M|$ is the total number of methods, $|M_T|$ is the number of executed methods in test suite T and B is the set of branches in the program. WSA starts by generating a set of solutions, which are uniformly, randomly initialized. Formally, let t denote a test case, which consists of a sequence of statements $t = \langle s_1, s_2, ..., s_l \rangle$ of length l. A statement s_i can be a constructor, a field, a primitive, a method, or an assignment. A solution is defined as a test suite T, which is a collection $T = \{t_1, t_2, ..., t_n\}$ of test cases. An optimal solution T^* is a test suite that covers all possible branches and lines of code, i.e., 100% coverage.

Since the number of test cases in a test suite and the number of statements in a test case may vary, the solution representation is of variable size. The solutions are evolved in iterations until a stopping criterion is achieved, which usually is a predefined number of function evaluations. The Genetic Algorithm used by EvoSuite has four genetic operators that are applied to solutions at every iteration: crossover, mutation, selection, and replacement. Crossover creates two new solutions by combining test cases from two test suites in the population. The mutation operator is applied after the crossover operator, at a test suite level, and at a test case level. Test suites are mutated by changing each of the test cases with a probability $1/n$, where n is the number of test cases in the test suite. In addition, new test cases are added to the test suite at random. The mutation of test cases is performed by either adding, changing or removing statements from a test case with a probability $1/l$, where l is the number of statements in a test case.

A rank-based selection procedure is employed to select parent solutions that will undergo recombination and mutation procedures. Solutions are ranked based on the fitness function. When there is a tie between solutions, shorter test suites are assigned better ranks. As a result, solutions with better branch coverage and shorter length have a higher chance of projecting their 'genes' to the next generation. Similar to the original studies with EvoSuite, an elitist strategy is used as a replacement procedure [46]. The elitist strategy selects the best solutions to create the next generation. The elitist strategy consists of selecting the best individuals of the current generation to initialize the next generation [46]. The whole population is considered in the selection process.

2.2.3 Many-Objective Sorting Algorithm (MOSA)

MOSA [76] is a many-objective Genetic Algorithm used for automated test case generation. MOSA starts with an initial set of randomly generated test cases that evolve via crossover and mutation [76]. The selection scheme considers both the non-dominance relation and the proposed preference criterion, which determines the test case with the lowest objective score (branch distance + approach level) for each uncovered branch. Test cases are ranked according to the non-dominated sorting algorithm used by the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [32].

Next, the crowding distance is used to decide which test case to select: the test cases with a higher distance from the rest of the population have a higher probability of being selected. Test cases are added to the next generation based on their ranks. MOSA is implemented in EvoSuite [76], and the objective function $f(b, T)$ for each branch $b \in B$ in test suite T is defined as follows:

$$f(b, T) = al + \mathrm{norm}(d(b, T)) \tag{2.3}$$

The normalized branch distance $(d(b, T))$ is computed using the following equation:

$$d(b, T) = \begin{cases} 0 & \text{if a branch has been covered,} \\ \dfrac{D_{\min}(t \in T, b)}{D_{\min}(t \in T, b) + 1} & \text{if a predicate is executed at least twice,} \\ 1 & \text{otherwise.} \end{cases} \tag{2.4}$$

where $D_{\min}(t \in T, b)$ is the minimal non-normalized branch distance, computed according to any of the available branch distance computation schemes [76].

The distance $d_{\min}^{b,T}$ is 0 if at least one of the branches has been covered, and > 0 otherwise

The overall objective function of a test suite T with respect to all branches is computed as:

$$f(T) = |M| - |M_T| + \sum_{b \in B} f(b, T), \tag{2.5}$$

where M is the set of methods in T and M_T is the set of methods executed during the test.

2.3 Automated Test Suite Generation - Objective Functions

SBST approaches (WSA and MOSA) require the design of an appropriate objective function (Equations 2.2 and 2.3), which measures the quality of generated solutions, and guides the search process to promising areas of the search space. To this end, researchers have proposed different definitions of objective functions, which use different measures, such as structural coverage [85], approach level [78], distance calculation [46, 87], or the combination of more than one measure [99, 106]. It is not known, however, which approach would work best under what conditions. Given a software system with specific features, which SBST technique would be suitable?

To successfully adapt a search technique for generating test data, it is essential to reformulate the latter as a search problem, by defining the objective functions, representation, and move operators [2]. A candidate solution is a test case consisting of a sequence of input values, passed to the program upon execution to observe its behaviour. A set of test cases form a search space, and the representation of a candidate is usually a floating point number and binary code derived from the underlying data types of the programming language used. A neighbourhood structure in a candidate solution for local search constitutes a collection of solutions that are in some respects close to the current candidate solution. For numerical variables, such as real and integer variables, the neighbourhood is within a range of values that surround each value. The neighbours of Boolean variables are 'TRUE' or 'FALSE' values, while the neighbours of enumerated variables consist of any value from the enumeration.

A test adequacy criterion for structure testing is a testing aim that can be numerically measured and assessed, e.g. covered branches or statements. The test criterion is coded as an objective function, which is used to evaluate the performance of candidate test inputs. To assess the fitness of candidate solutions, the program is executed for the inputs generated. The objective function plays a vital role in the performance of search techniques. A well-defined objective function increases the likelihood of finding a solution and reaching higher overall coverage. Better guidance of the search process helps in consuming fewer system resources in the process [104, 13].

Miller and Spooner [71] were the first to suggest using search algorithms to automate test data generation. The tester selects a path through the program and then produces a straight-line version from the program that is equivalent to that path. The branch predicates in the path are extracted and rearranged into Boolean assignments as a 'path constraint' form. A real-valued function is built for the whole path, to estimate how close all the branch predicates on the path are to being satisfied. It is counted as positive when all the branch predicates are true, and negative if the opposite is the case. Test data are derived by choosing an initial set of data and then applying a numerical optimization algorithm. The process is terminated when the value of the objective function becomes positive. The function f is assigned by repeatedly executing the straight-line version and automatically collecting the path constraint values.

It was not until 1990 that Miller and Spooner's research directions were continued by Korel [60], which initially executes the program with some arbitrary inputs. If an undesired branch is taken, a local search algorithm is used to guide the program execution along the desired path, using a specific objective function derived from the predicate of the desired branch. This objective function has a real value, referred to as branch distance, which measures how close the predicate is to being executed.

The main weakness of Korel's method [60] is its limited ability to detect the infeasibility of the path. If an infeasible path is selected and the infeasibility is not detected, then a significant computational effort is spent before the search process terminates. The problem of a path's infeasibility means that Korel's approach [60] is best suited for software featuring a relatively small number of paths to reach the selected node [61].

This work focus on three main groups based on the definition of the objective function: the coverage-oriented approach, the distance-oriented approach and the structure-oriented approach. These three approaches were selected due to its differences in strategies to evaluate the branch predicates. Our goal is to understand how different predicates influence the performance of the selected fitness functions.

2.3.1 Coverage-oriented Function (CLF)

Various forms of coverage measures are used in coverage-oriented approaches: (i) statement coverage – estimates the percentage of program statements covered during testing, (ii) branch coverage – measures the extent to which branch statements in the code are covered during the test, (iii) path coverage – measures the number of feasible paths through the graph produced during the test.

Watkins [105] concentrates on full path coverage. Test data that follow previously uncovered paths are assigned higher fitness values than those that pass via paths that have already been covered. The penalization of executed paths, however, does not exploit the information in the branch predicates [99].

In this study, we modified the original version [85] to exercise both branch values (true and false), rather than arriving at just one value, as in the original definition of the function. An individual is rewarded on the basis of the number of branches executed. The individual covering the highest number of branches in the code gains the lowest fitness values. This function is primarily concerned with ensuring that the highest possible level of coverage is achieved.

Given a test suite T and a set of branches B of the program being tested, the coverage level $f_1(b, T)$ for each branch $b \in B$ on test suite T is defined as follows:

$$f_1(b, T) = \begin{cases} 0 & \text{if both branches are covered,} \\ 0.5 & \text{if the predicate is executed once,} \\ 1 & \text{otherwise,} \end{cases} \tag{2.6}$$

The overall objective function of a test suite is calculated using the following equation:

$$\text{CLF} = |M| - |M_T| + \sum_{b \in B} f_1(b, T), \tag{2.7}$$

where M is the set of methods in the object and M_T is the set of methods executed during the test. The difference $|M| - |M_T|$ is used to reward coverage of methods in the test objects which have no branch statements.

2.3.2 Distance-oriented Function (BDF)

Branch distance-oriented approaches exploit information from branch predicates, which evaluate how far a predicate is from obtaining its opposite value [71]. The work of Xanthakis et al. [112] was the first to apply GAs in the generation of structural test data. This method follows similar lines to earlier work by Miller and Spoone [71] and it, therefore, suffers from problems that resemble those discussed regarding this method, such as the limited ability to detect path infeasibility. A tester chooses a path, and then the branch predicates are extracted from it. A GA is employed to find test data that satisfies all branch predicates in the path at once. The objective function sums up the values of all branch distances.

Tracey et al. [98] employ simulated annealing to the generation of structural test data. The approach aims to search for test data that covers the program's statements in turn. Thus, the objective function indicates whether or not the target statement has been exercised. The objective function is the branch distance, which indicates how close the current execution is to adopting the desired branch according to the decision made. The objective function returns zero if the current execution leads to the target condition (branch or statement); otherwise, it returns positive values.

The search proceeds while looking for test data to cover each statement in turn. When the search process stagnates at one node, and no further progress can be made, the approach attempts to generate test data for the next target node. Unlike Korel's approach [60], the newly generated test data do not need to conform to an already successful sub-path. However, this leads to the search losing information about its progress [68]. The reason for this is because a solution that deviates from the desired path at an early stage of a search is assigned similar fitness values to those who deviate at an advanced stage of a search.

The main criticism of branch distance-oriented techniques is that control information about the target is not included in the objective function. This fact may cause the search to get stuck in local optima, thereby making it difficult to obtain full coverage [106, 68]. The control-oriented approaches discussed in the next section addresses this problem.

The objective function [46] uses a branch distance measurement, which reflects how close a branch's predicate is to being reached. First, given a set of branches B, the minimal branch distance for each branch $b \in B$ in test suite T is defined as follows:

$$f_2(b,T) = \begin{cases} 0 & \text{if the branch is covered,} \\ d(b,T) & \text{if the predicate is executed once,} \\ 1 & \text{otherwise,} \end{cases} \tag{2.8}$$

where $d(b,T)$ is 0 if at least one of the branch's values (true or false) has been covered, and > 0 otherwise. The objective function of a test suite T is:

$$\text{BDF} = |M| - |M_T| + \sum_{b \in B} f_2(b,T), \tag{2.9}$$

where M is the set of methods and M_T is the set of methods executed during the test.

2.3.3 Control-oriented Function (CFF)

Control-oriented approaches use control information for the objective function. This is achieved by using a control dependency graph to determine predicate paths for the intended node. Pargas et al. [78] apply a GA for statement and branch coverage, guided by the control dependencies in the program. For a goal node, a sequence of control-dependent nodes is specified, which should be exercised for the execution of the goal node. The objective function is equivalent to the number of successful control-dependent node executions.

It is worth noting that using only control structures in objective functions will form plateaux in the fitness landscape [68]. As no distance information can be exploited, this results in insufficient guidance towards unexplored structures. If the solutions fail to fulfill one of the branch predicates, no branch distance information is given on how to descend the landscape for the search process, as guidance for those individuals who are closer to exercising the desired node.

The objective function [78] uses a control dependency graph, which is equivalent to the test object's code, to compute an individual's fitness value. The fitness value is equivalent to the number of successful control dependent node executions towards the intended branch.

Let dn be the number of control dependent nodes for the current target branch, and en be the number of successfully executed control-dependent nodes; the objective function $f_3(b, T)$ for each branch $b \in B$ in test suite T is defined as follows:

$$f_3(b, T) = \text{norm}(dn - en) \tag{2.10}$$

where norm is a normalization function in the range [0, 1]. The fitness of a test suite T is calculated as:

$$\text{CFF} = |M| - |M_T| + \sum_{b \in B} f_3(b, T), \tag{2.11}$$

where M is the set of methods in T and M_T is the set of methods executed during the test.

2.4 Summary

In this chapter, we presented three ATSG techniques: Random Testing (RT), Whole Suite with Archive (WSA) and Many-Objective Sorting Algorithm (MOSA) and three objective functions: Coverage-oriented Function (CLF), Distance-oriented Function (BDF) and Control-oriented Function (CFF). The aforementioned techniques use different approaches to achieve a high testing coverage, representing different categories of the automated software testing generation research area.

Chapter 3

Literature Review

3.1 Introduction

In this chapter, we describe the current ATGST methodologies available in the literature for performance assessment. Moreover, we point out the current issues associated with these methodologies and how they may affect the credibility of the results.

3.2 Performance Assessment

As stated before, the assessment of newly developed techniques or experimental studies investigating the performance of different ATSGTs often is based on a small set of CUTs. Little information about the CUTs characteristics and how they are selected are presented [47], offering usually no insight into the external validity of the findings. Furthermore, the vast majority of the papers in the literature are only describing the benefits of the newly introduced technique and the innovation carried out during development, while just a small minority mention the limitations or present negative results [7]. In the following sections, we present the research studies using the most common methodology to assess the ATSGT performance (General Assessment) and others that tried to use software features in the assessment (Feature-based Assessment).

3.2.1 General Assessment

In this section, we present a set of papers using the general assessment methodology to evaluate the performance of newly developed techniques. The studies have been selected based on diversity, i.e., papers using a small benchmark set, large benchmark set, artificially generated instances and industrial code.

Ciupa [22] published ARTOO: Adaptive Random Testing for Object-Oriented Software. The tool is an extension of ART (Adaptive Random Testing) initially developed for numerical inputs. ARTOO implements the notion of distance between objects and also a new strategy to select inputs objects that have the highest average distance to those already used as test inputs.

The set of classes used to evaluate ARTOO performance was extracted from Eiffel-Base Library version 5.6. A total of 8 classes have been selected and the details are presented in Table 3.1. Analysis of the results shows improvements when compared to directed random strategy. The number of tests required to find the first fault is reduced by two orders of magnitude in some cases and also there was an increase in the number of faults detected.

However, no justification for the CUTs selection has been presented, and no additional CUT features are available. One might ask why, among almost 1000 classes, only 8 classes have been selected and why those 8. Moreover, considering only 8 classes reduces the external validity of the study considerably.

Table 3.1: Properties of the ARTOO CUTs

Class	Total lines of code	Lines of contract code	Routines	Attributes	Parent classes
ACTION SEQUENCE	2477	164	156	16	24
ARRAY	1208	98	86	4	11
ARRAYED LIST	2164	146	39	6	23
BOUNDED STACK	779	56	62	4	10
FIXED TREE	1623	82	125	6	4
HASH TABLE	1791	178	122	13	9
LINKED LIST	1893	92	106	6	19
STRING	2980	296	171	4	16

In [23], Ciupa evaluates the variance of the number of faults detected by random testing over time. An empirical test totalizing 1215 hours of randomly testing 27 Eiffel classes, each with 30 seeds of the random number generator. The AutoTest Framework [64] was used to execute the aforementioned tests.

The results showed that the relative number of faults detected by random testing over time can be predicted but that different faults are detected in each different execution. Moreover, it shows that random testing can quickly find faults: the first failure is likely to be detected within 30 seconds. **However, similarly to the previous approach, no information regarding the CUT selection criteria has been provided. Furthermore, no features have been presented to characterize the CUTs apart of their names.**

Csallner [27] presents an automatic error-detection approach that combines static checking and concrete test-case generation (CNC). The approach consists of taking the abstract error conditions inferred using theorem proving techniques by a static checker, deriving specific error conditions using a constraint solver, and producing concrete test cases using JCrasher tool [26] that are executed to determine whether an error exists.

Two Java projects have been used to evaluate CNC: the JABA bytecode analysis framework and the JMS module of the JBoss4 open source J2EE application server, which is an implementation of Sun's Java Message Service API. More specifically, they have run CnC on all the jaba.* packages of JABA, which consist of some 18 thousand non-comment source statements (NCSS), and on the JMS packages of JBoss 4.0 RC1, which consist of some five thousand non-comment source statements. **The number of CUTs and the selection criteria used is not available. Moreover, the only feature described is the NCSS.**

The results have demonstrated the usefulness of CnC in relation to ESC/Java [43] and JCrasher [26], as well as by finding bugs in real-world applications. The authors also stated that the best use of CnC is during development where the programmer can apply the tool to newly created code, inspecting reports of conditions indicating possible crashes.

In the area of regression testing, Taneja [94] presents DiffGen an automated regression unit-test generation and checking for Java programs. For two versions of a Java class, DiffGen instruments the code by adding new branches and using a testing generation tool to expose behavioral differences between the two class versions. DiffGen has been evaluated using JTopas (13 classes) previously used by [41] and [36]. Table 3.2 present the provided information regarding the CUTs.

The results showed that DiffGen can effectively detect regression faults that cannot be detected by the state-of-the-art techniques (5 of 7 faults that were not detected before). **However, none of the related papers [36, 41, 94] has presented a selection criteria for the CUTs. No additional CUT features are available.**

DSD-Crasher [28] is a bug finding tool that executes a three-step approach to program analysis. (a) Identify the program's intended execution behavior with dynamic

Table 3.2: Properties of the DiffGen CUTs

Version	Class	LOC
1	ExtIOException	78
1	AbstractTokenizer	1672
1	Token	159
1	Tokenizer	287
1	ExtIndexOutOfBoundsException	67
2	ExtIOException	89
2	ThrowableMessageFormatter	137
2	AbstractTokenizer	2966
2	Token	447
3	EnvironmentProvider	240
3	PluginTokenizer	407
3	StandardTokenizer	1992
3	StandardTokenizerProperties	2736

invariant detection; (b) Statically analyze the program within the restricted input domain to explore many paths; (c) Automatically generate test cases that focus on verifying the results of the static analysis. The tool has been accessed using JBoss JMS (Sun's Java Message Service API [51]) and Groovy (http://groovy.codehaus.org/).

The author's justification for selecting the projects are:

- **Groovy is a very representative test application for this kind of analysis: it is a medium-size, third-party application. Importantly, its test suite was developed completely independently of this evaluation by the application developers, for regression testing and not to yield good Daikon invariants.**

- **JBoss JMS is a good example of a third party application, especially appropriate for comparisons with CnC as it was a part of CnC's past**

evaluation [26]. Nevertheless, the existing test suite supplied by the original authors was insufficient, and we had to supplement it ourselves.

The results showed that DSD-Crasher is an improvement over using CnC alone [26]. The reduction in false positives enables DSD-Crasher (as opposed to CnC) to produce reasonable reports about NullPointerExceptions. **The number of CUTs used in the experiment is very low (34 classes in total). No additional CUT features are available.**

Thummalapenta addresses the issue of selecting relevant method-call sequences when automatically generating unit tests that achieve high structural code coverage with DyGen [95]. The novel approach generates tests by mining dynamic traces recorded during program executions. DyGen uses these traces to create parameterized unit tests (PUTs) and uses dynamic symbolic execution to generate new unit tests for the PUTs that can achieve high structural code coverage. The new tool is evaluated using 10 .NET libraries totalizing 5.757 classes (3.3).

Table 3.3: Properties of the DyGen CUTs

.NET Libraries	**KLOC**	**# Public classes**	**# Public methods**
mscorlib	179	1316	13199
System	149	947	8458
System.Windows.Forms	226	1403	17785
System.Drawing	24	223	2823
System.Xml	122	270	5426
System.Web.RegularExpressions	10	16	162
System.Configuration	17	105	773
System.Data	126	298	5464
System.Web	202	1140	11487
System.Transactions	9.5	39	405
TOTAL	**1063**	**5757**	**65982**

The new method generated regression tests covering 27,485 basic blocks, which is 24.3% higher than the number of blocks covered by recorded dynamic traces. **Even though a significant number of classes have been used in this experiment, the study does not present a selection criteria for the CUTs. One might think that the classes are too similar, risking the external validity. Moreover, the presented information regarding the CUTs is very minimal.**

In [74], Pacheco describes a new technique to select, from a large set of test inputs, a small subset that is likely to reveal faults in the CUT. The new approach takes a program or software component, a set of correct executions-say (extracted from observations of the software running properly or from an existing test suite that a user wishes to enhance) aiming to infer an operational model of the software's operation. After that, the execution operational inputs patterns that differ from the model in specific ways are suggestive of faults.

The new approach, called Eclat, has been evaluated using 6 libraries totalizing 631 CUTs and 75.000 non-comment non-blank lines of code. The chosen libraries are presented in Table 3.4.

Table 3.4: Properties of the Eclat CUTs

Program	versions	suites per version	public methods	NCNB LOC
BoundedStack	1	2	11	88
DSAA	1	1	110	640
JMLSamples	1	1	221	1392
utilMDE	1	2	69	1832
RatPoly	97	1	17	512
Directions	80	2	42	342

The results have shown that Eclat is effective in generating fault-revealing test inputs. The input generation technique generates legal, fault-revealing inputs for the methods in the test programs, and the input selection technique identifies inputs that have higher chances of revealing faults as the candidate inputs. Eclat reveals real, previously unknown errors in the test programs.

Similarly to the other approaches included in this section, a significant drawback of this work is that the authors do not present additional information regarding the CUTs. The author justifies the use of the CUTs mentioned above by saying: "all subject programs implement modestly-sized libraries designed to support larger programs; thus, unit testing is appropriate for them".

Ribeiro [83] tackles the problem of object reuse on standard typed Genetic Programming for representing and evolving test data. He proposes eCrash, a new methodology to overcome this limitation by reusing objects. Object Reuse aims to allow one instance to be passed to multiple methods as an argument, or several times to the same method as arguments. In the context of Object-Oriented Evolutionary Testing, it allows the generation of test programs that analyse structures of the SUTs that would not be reachable.

The new tool has been tested using two classes of the JDK 1.4.2. The results show improvements in the effectiveness of the search and also in the test case generation process. This result was obtained by yielding solutions with smaller overall size and lower structural complexity, and it is able to increase the feasibility of Test Programs. **In a similar fashion to the other approaches in this area, no additional information regarding the CUTs has been provided. Ribeiro justifies the selection of these 2 classes by saying: "Their selection is supported by the fact that they are container classes, which are a typical benchmark in software testing of OO programs".**

In [97], a genetic algorithm is used to automatically generate test cases for the unit testing of classes in a general usage scenario. Test cases are defined as chromosomes, which contain information on which objects to create, which methods to invoke and which values to use as inputs. The new algorithm mutates these structures to maximize a defined coverage measure. The new method has been implemented in the tool eToc (Evolutionary Testing of Classes) that has been tested against the CUT presented in Table 3.6.

Table 3.5: Properties of the eToc CUTs

Class	LOC	public methods
StringTokenizer	313	6
BitSet	1046	26
HashMap	982	13
LinkedList	704	23
Stack	118	5
TreeSet	482	15
Total	**3645**	**88**

The classes used to evaluate the proposed technique were taken randomly from the standard Java library distributed with the Software Development Kit (SDK) version 1.4.0. The results showed that using a genetic algorithm for the unit testing of classes can be very powerful. Optimal coverage of the public method branches was achieved within a reasonable computation time, and the generated test suites were generally small. **However, no additional information regarding the CUTs has been provided. The author does not provide a justification for the CUT selection.**

Inkumsah [56] developed EvaCon to overcome two main issues associated with high structural coverage. The first one is the need for in-depth knowledge of the program structure and semantics. The second one refers to specific method sequences required to lead the receiver object or non-primitive arguments to specific desirable states. EvaCon has been evaluated using 13 classes previously used in evaluating white-box test generation tools (Table 3.6).

Table 3.6: Properties of the EvaCon CUTs

Class	number of public methods	number of branches	LOC
BankAccount	6	6	60
BinarySearchTree	16	67	260
BinomialHeap	10	94	215
BitSet	25	130	638
DisjSet	6	44	140
FibonacciHeap	9	92	207
HashMap	10	89	374
LinkedList	29	105	738
ShoppingCart	6	13	117
Stack	5	16	160
StringTokenizer	5	47	222
TreeMap	47	252	1626
TreeSet	13	20	301

The experiments showed that the tests generated using EvaCon can achieve higher branch coverage than evolutionary testing or concolic testing alone, reaching a max improvement of 6%. **No additional information regarding the CUTs has been provided. The author does not justify the CUT selection.**

The aforementioned papers claim superiority of the newly developed techniques without considering the characteristics of the CUTs used in experiments. One of the risks involved is the possible similarity among the CUTs, compromising the external validity of the results. Ideally, the experiment should consider a diverse set of classes aiming to evaluate the techniques behaviour in CUTs with different characteristics. The use of features is essential to validate the results of the experiment by defining the scope of the benchmark set.

3.2.2 Feature-based Assessment

Analyzing the performance of different ATSG techniques (ATSGT) by using software features has been the focus of previous research. Cseppento et al. [29] evaluated Symbolic Execution-based Test Tools. The approach was based on collecting a representative set of programming language concepts (e.g., language constructors, operations and control flow statements) that are handled by the tools, map them to 300 code snippets that serve as inputs to the tools, create an automated framework to execute and evaluate these snippets, and perform experiments on four Java and one .NET test generator tool.

The results showed that the evaluation could identify both strengths and weaknesses in the tools by associating the code snippets with low coverage as weakness and the ones with high coverage as strengths. The downside of the research is the use of artificial code that might not reflect the characteristics of real software systems. No information about the complexity of the code snippets is provided, consequently, it is no clear how diverse they were.

Lammermann et al. [63] assess the difficulty of a test object for evolutionary testing using software measures for procedural code. The approach checks for a possible connection between different software measures and the execution of the evolutionary test. The results indicate that using source code or structured-based software features can only lead to mediocre predictions. Cyclomatic Complexity (CC) was able to generate the best forecasts.

Wang et al. [103] compared three well-known public-accessible unit test data generation tools: JCrasher [26], TestGen4J [66] and JUB [100]. Java classes were applied and evaluated based on their mutation scores. As a comparison, two additional sets of tests were created for each class. One test set contained random values and the other contained values to satisfy edge coverage. Results showed that the automatic test data generation tools generated tests with almost the same mutation scores as the random tests. The criteria used to select the set of classes in the experiment was not presented in the paper. The reason stated by the researchers is: "nobody has developed a general theory for how to choose representative classes for empirical studies, or how many classes we may need". Therefore, it could be the case that all the classes were elementary, explaining similar performances.

Chitirala [21] measured the effectiveness of two automated test generation tools, EvoSuite [44] and Tpalus [117], using metrics such as code coverage, mutation scores, and size of the test suite. The results showed that EvoSuite gains the upper hand when it comes to coverage scores and readability of the test cases, but Tpalus had success in terms of killing mutants. The difference was around 10% for the test subjects. The number of CUTs used in the experiment was deficient. Only 10 classes were used and, among them, 3 were container classes. Therefore, the results can not be generalized.

Shamshiri at al. [87] conducted a study on the effects of using evolutionary algorithms for test suite generation. They compared four ATSGTs: Genetic Algorithm (GA), Chemical Reaction Optimization Algorithm (CRO), Random+, and Pure Random. All the ATSGTs were implemented on EvoSuite [44]. 995 CUT were used in the experiments. These CUTs were extracted from SF110 corpus. The results showed that Random techniques are as effective as search-based ones in CUT containing only branchless methods. Random techniques presented better results in search spaces characterised with plateaus. However, when optimising problems characterised with gradients, search-based techniques outperformed random techniques.

Only recently, Panichella and Molina [77] considered an object-oriented complexity metric in the selection of benchmark subjects. However, the authors did not consider investigating the effect of this feature on the effectiveness of the techniques. In general, the literature investigating the effectiveness of different ATSGTs presents little or no information about the CUT characteristics used in the experiments, impacting the generalisability of the results.

Different from related work, this research aims to provide a better understanding of which CUT features are related to the performance of ATSGTs, and whether software features can be used to select an ATSGT for a particular software system. To this end, we employ well-known code-based metrics from the literature [20]. Furthermore, we take advantage of well-established metrics from graph theory and apply them to measure properties of the CFG of a software system.

Not all features, however, are indicative of how hard a particular CUT is for an ATSGT. So one of the main challenges is to identify which features should be selected to characterise CUT, such that they capture the difficulty of CUTs. Next, using these features we investigate whether it is possible to infer and visualise algorithm performance across instance space. Finally, we check whether features can be used to select a suitable ATSG approach.

3.3 Summary

In this chapter, we presented an overview of the literature regarding ATSGT assessment. The vast majority of papers use the general methodology to assess performance while only a minority tried to identify relationships between features and performances. Moreover, papers using general assessments do not present any justification for the selection of the CUTs used in the experiments. Additionally, only very few features are presented to characterize the CUTs.

Part II

Contributions

Chapter 4

META Framework

4.1 Introduction

In this chapter, the alternative method to ATSGT assessment is introduced. The concepts are described as a framework aiming to facilitate its understanding and usage. Section 1 presents the overall framework definition. Section 2 describes the main components of the framework: Classes Under Test, ATSGTs, Feature Selection, Performance Visualization and Performance Prediction.

4.2 Definition

The *Mapping the Effectiveness of Test Automation (META)* framework (Figure 4.1) is inspired from the Algorithm Selection Problem (ASP) [90] introduced in the area of optimisation and machine learning, and the *No Free Lunch theorem*, which tell us that there does not exist a single algorithm that can be expected to outperform all other algorithms on all problem instances [111]. Hence, if method A is superior over method B in solving a particular set of problems, then one may claim that there exist other untested problems where method B may outperform method A. Empirical studies in

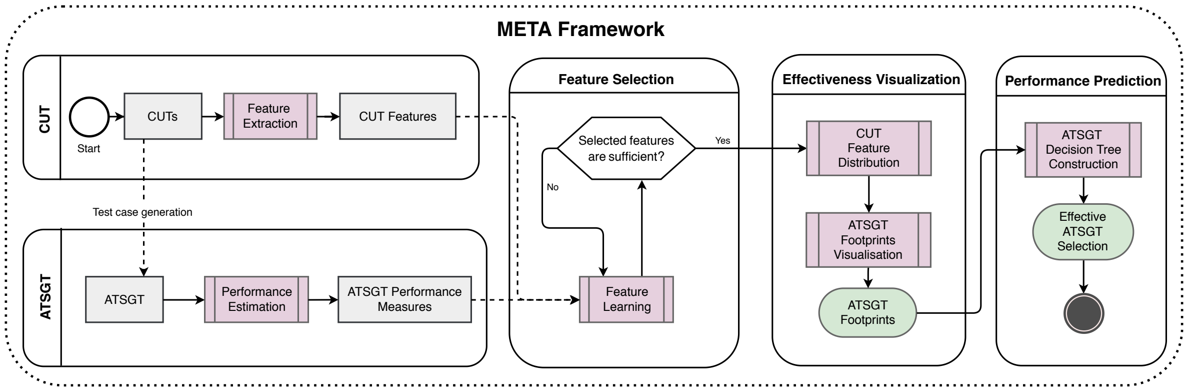


Figure 4.1: The main components of the Mapping the Effectiveness of Test Automation (META) Framework

the area of ATSG should focus on identifying conditions under which an algorithm is expected to succeed or fail instead of claiming superiority of a method over another.

In this thesis, these ideas were extended and adapted to assess the strengths and weaknesses of ATSGTs into an application called META tool. The proposed tool can be used to predict which ATSGT from a portfolio of different techniques is likely to perform best based on measurable features of a collection of CUTs. META is an alternative to the current ATSGT assessment methodology.

4.3 Main Components

The framework is composed of five main components: Classes Under Test, ATSGTs, Feature Selection, Performance Visualization and Performance Prediction. The components work together aiming to associate CUT features to ATSGT performance by using Evolutionary Algorithms, Machine Learning and Dimensionality Reduction techniques.

4.3.1 Classes Under Test

The first component of the META Framework contains the CUTs and their characteristics (features). The software systems are considered at a class level, and features are measured for each class. The features are defined and extracted from the CUTs to be subsequently analyzed by the feature selection method. The META framework does not impose restrictions on the programming language and can be applied to other programming languages for which automated test case generation approaches exist. For this thesis, however, we focus on JAVA programming language. However, further work is required to investigate META's effectiveness in other object oriented languages.

Features are problem dependent and must be chosen so that the varying complexities of the CUT instances are exposed, any known structural properties of the CUTs are captured, and any known advantages and limitations of the different ATSGTs are related to features. The most common measures and metrics used to characterise features of CUTs are extracted from code.

Other software features are extracted from the Control Flow Graph (CFG), which represents the structure of the code (see Figure 2.1). The most common metric that is based on the CFG is the Cyclomatic Complexity (CC) [67], which analyses the branching complexity of the CFG. Our approach uses well-established software measures, such as CC, and introduces new measures inspired by graph theory to analyze different aspects of the CFG. These metrics can provide valuable information about graph complexity. In essence, the software features considered in this work can be classified into code-based and graph-based features.

A good selection of CUTs and features is crucial to a proper ATSGT assessment using META. In order to define hardness, CUTs that can challenge all ATSGTs in the portfolio are required, and also features that can explain it. Therefore, Chapter 5 presents a comprehensive analysis of the features and benchmarks available in the literature to later present the ones selected to be part of this study [**RO2 and RO3**].

4.3.2 Automated Software Test Generation Techniques

In this component, the ATSGT portfolio is defined. Moreover, the metric used to assess their performance (evaluate the quality of test suites) is also defined. The most common is the code coverage, i.e., the degree to which a particular test suite tests a software system. Mutation testing is another technique commonly used to evaluate test suites. It involves changing a program slightly with mutation operators. Each new version generated is called a mutant and tests detect and reject mutants (kill mutants) by causing the behavior of the original code to differ from the mutant. Test suites are measured by the percentage of mutants that they kill.

For this study, branch coverage is used, also known as decision coverage, which quantifies the percentage of branches/decisions that have been executed by the test suite. The term branch/decision means that two or more exits are possible from statements like an IF or a CASE, always generating two outcomes, either true or false. Branch coverage is widely used as a performance metric in the literature [75, 47, 44]. One of the main benefits of branch coverage is its low overhead on the execution of the program under test [113]. The European Cooperation for Space Standardization (ECSS) gives 100% branch coverage as one of the measures to assure the quality of a critical software [109].

4.3.3 Feature Selection

The feature learning component of the META framework identifies the most significant features of CUTs that impact the effectiveness of the ATSGTs [**RO4**]. The input to this step is the collection of CUT features (Section 4.3.1) and the performance of the ATSGTs (Section 4.3.2). The output is the set of features that best describe the reasons why ATSGTs are effective or not in generating test suites.

Initially, the features are analysed, normalized if needed, and the ones presenting high correlation with any other feature were removed. This process is highly recommended when using dimensionality reduction techniques, such as Principal Component Analysis (PCA), as including nearly-redundant variables, can cause these methods to overemphasize their contributions.

The next step consists of selecting the best set of features to highlight the strengths and weaknesses of the ATSGTs on the CUT space. A genetic algorithm is used to search for the best features to explain good and bad performances. The GA has the following characteristics:

Chromosome: the solution is a set of features, with a minimum of three and a maximum of ten. These limits were defined based on experiments conducted in the initial stage of the research. We detected that solutions containing more than ten features had none to insignificant variation gain. In other words, in most of the cases the GA best solution would contain less than 10 features without any limit set. In cases where the solution size was greater than 10, the difference between the amount of data retained after the dimension reduction (PCA variation) was negligible when compared to the best solution containing 10 or fewer features.

Selection Strategy: Roulette Wheel Selection was chosen. RWS allocates probabilities of selection based on the fitness of solutions, allowing poor solutions to be selected with a very small probability. This strategy is effective in preventing premature convergence [118].

Crossover Operator: Uniform Crossover has been chosen due to its efficiency when applied to GA with small to medium populations [91].

The main steps of the method are presented in Algorithm 1.

Algorithm 1 Feature Selection with a Genetic Algorithm

```
1: procedure FeatureLearning(features, ATSGTP)
2:     Input: $n, m_r, c_r$    ▷ $n$ is population size, $m_r$ is mutation rate, $c_r$ is crossover rate, ATSGTP is ATSGT performance.
3:     $P \leftarrow$ RandomSolutions(n)    ▷ P is the population
4:     for $i \leftarrow 1$ to $n$ do
5:     $Q \leftarrow \emptyset$    ▷ Q is the Auxiliary Population
6:     while !TerminationCodition do
7:         for $i \leftarrow 1$ to $n$ do
8:             $p_1, p_2 \leftarrow$ RouletteWheelSelection(P,F)    ▷ $p_1, p_2$ are parent solutions
9:             $o_1, o_2 \leftarrow$ UniformCrossover($p_1, p_2, c_r$)
10:            $o_1' \leftarrow$ Mutation($o_1, m_r$)
11:            $o_2' \leftarrow$ Mutation($o_2, m_r$)
12:            $f(o_1') \leftarrow$ EvaluateFitness($o_1'$,ATSGP)
13:            $f(o_2') \leftarrow$ EvaluateFitness($o_2'$,ATSGP)
14:            Insert($o_1', o_2', Q$)
15:        $R \leftarrow P \cup Q$
16:        RankSolutions($R$)
17:        $P \leftarrow$ SelectBestSolutions($R$)
18:    Return($P$)
19: procedure EvaluateFitness($s$, ATSGP)
20:    Input: $s$    ▷ set of features to be evaluated
21:    2D_coordinates $\leftarrow$ PCA($s$, ATSGP)
22:    $f(s) \leftarrow$ SVM(2D_coordinates, ATSGP)
23:    Return($f(s)$)    ▷ Return the fitness of the set of features
```

An individual solution contains the set of features that are used to characterise the CUTs. The crossover operator (line 9) takes two sets of features and combines them at different random positions. Then both solutions are mutated. The mutation operator (Lines 10 and 11) replaces a random feature from the set of features with a new feature. Solutions are evaluated based on how well the set of features can be used to separate the CUT space into ATSGT footprints.

The first step of the evaluation process consists of using Principal Component Analysis (PCA) to reduce the number of dimensions to 2. PCA is a statistical procedure that applies an orthogonal transformation to change a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components. The number of principal components is less than or equal to the number of original variables [57]. PCA is one of the most widely used statistical tools for data analysis and dimensionality reduction [16].

The second step consists of applying Support Vector Machines (SVM) in the 2D coordinates generated by the PCA method. A SVM is a discriminative classifier formally defined by a separating hyperplane [42]. In two dimensional space this hyperplane is a line dividing a plane in two parts where in each class lay in either side. The goal of SVM is to verify the quality of the selected features in explaining the ATSGT performances. The fitness function is described by the second procedure in the algorithm 1.

Figure 4.2 is an example showing a clear separation between ATSGT 1 and ATSGT 2. In this example, SVM would produce a highly accurate prediction score, and consequently, the set of features used to create this space would receive a high fitness score. A high fitness means that the selected features were able to identify accurately the characteristics associated with good and bad performance.

In summary, the feature selection process aims to find the best set of features to explain the ATGST performance in the two dimensional space. The 2D space is created using PCA, reducing the number of dimensions from n to 2. In the 2D space, each dot is a CUT labeled as easy or hard according to the performance results. Finally, SVM is applied to find how well the selected features can explain the performance results. The SVM accuracy is the solution fitness.

4.3.4 Effectiveness Visualisation

Once the most significant features are identified (4.3.3), they are used to visualise the footprints of the ATSGTs, as shown in Figure 4.2. PCA is used as a dimensionality

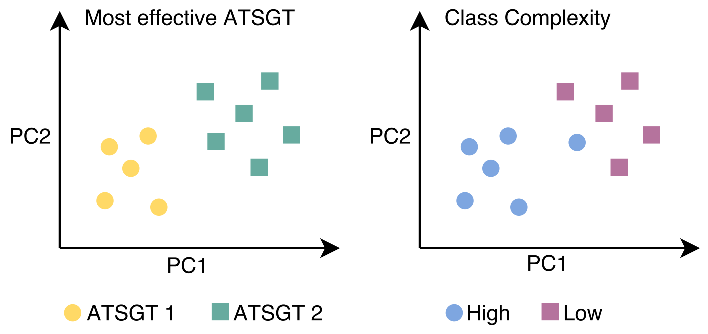


Figure 4.2: Visualising the strengths and weaknesses of ATSGs.

reduction technique on the optimal subset of features. The aim is to plot the performance of the two ATSGTs across the CUT space in 2D, which is likely to reveal where the methods are performing well, and where they are suffering.

Using the two principal components from the PCA analysis, the effectiveness of the ATSGTs is visualised in a 2D map, as illustrated in Figure 4.2. The plot on the left visualises the effectiveness of two ATSG techniques, ATSGT1 and ATSGT2 over all CUTs. The yellow circles are the CUTs where ATSGT1 was the most effective, whereas the green squares represent the CUTs where the second technique was the most effective.

Next, the same CUTs are projected in a 2-dimensional map based on how they score in terms of the most significant features identified in the feature learning step. The plot on the right in Figure 4.2 illustrates this point. The blue circles indicate that the respective CUTs have high-class complexity, while the CUT with low-class complexity is represented as purple squares. Looking at both plots, we notice that ATSGT1 is effective when class complexity is high, whereas ATSGT2 works best when class complexity is low.

4.3.5 Performance Prediction

A decision tree (DT) is produced using the most significant CUT features (4.3.3). The DT can be used to select the most effective ATSGT for new software systems based on their features. The clustering algorithm DBSCAN [14] is used to identify the areas in the CUT space where the different techniques in the portfolio are effective or not. DBSCAN is a high performance clustering algorithm that can find clusters of different shapes and manages the noise points effectively [14]. The method consists of identifying the clusters by acknowledging the fact that within each cluster the typical density of points is higher than outside of the cluster.

In the next step, the C4.5 [82] algorithm is used to generate the decision tree with the features identified in the feature learning phase described in Section 4.3.3).The C4.5 was developed by Ross Quinlan and it is often referred to as statistical classifier [49]. C4.5 performs by default a tree pruning process. This leads to the formation of smaller trees, more simple rules and produce more intuitive interpretations [49]. In [65], Lim et al. show that the c4.5 algorithm can provide good classification accuracy, being the fastest among the decision tree algorithms with univariate splits.

Other advantages of c4.5 are [49]:

- C4.5 builds models that can be easily interpreted.
- It can handle both categorical and continuous values.
- It can deal with noise and deal with missing value attributes.

4.4 Summary

This chapter presented the META Framework, an alternative methodology to asses ATSGT performance [**RO1**]. The framework is divided into 5 main parts. The **CUT** contains the Features and CUTs selected to be part of the performance assessment [**RO2 and RO3**]. The **ATSGT** contains the ATSGTs to be assessed and the performance measures. The **Feature Selection** section manages the interaction between the CUT and ATSGT. This component is the part of the META framework that enables the identification of features that influence more significantly the ATSGT performance [**RO4**].

Once the critical features have been identified through the feature learning procedure, they are used to analyse and visualise the footprints of the ATSG techniques, which help understand the strengths and weaknesses of algorithms. The feature learning procedure identifies features that capture the difficulty of CUTs. A training set of CUTs is used to learn critical features.

This process helps to identify which ATSGTs in a portfolio is likely to be best for a relevant set of CUTs, and which ones are likely to produce suboptimal results. The footprints are generated in the **Effectiveness Visualization** step and show the performance of the different ATSGTs across the CUT space using dimensionality reduction techniques [**RO5**].

Finally, in the **Performance Prediction** section, a decision tree is created based on the most significant CUT features, which can be used to predict and select the most effective ATSGT for generating test cases for new software projects. Its accuracy is evaluated in a validation set of CUTs [**RO6**].

Chapter 5

Feature Set and Benchmark Analysis

5.1 Introduction

The META Framework relies on a good selection of CUTs to be effective, i.e., the selected classes need to be diverse enough to challenge the ATSGTs in the portfolio. Additionally, a broad and robust set of features is required to enable performance characterization. Without a good selection of CUTs and features, the process might fail due to the inability to differentiate performances. This chapter describes how the CUTs have been selected and also presents the set of features used to characterize ATSGT performance [**RO2 and RO3**].

5.2 Benchmarks

Benchmarks have been used in many areas of computer science to compare the performance of different systems such as databases and information retrieval algorithms. The development and use of a benchmark within a research area are usually accompanied by rapid technical progress and community building [89]. This fact has led researches to formulate a theory of benchmarking within scientific disciplines [89]. The theory states that software engineering research should become more experimental and cohesive by working as a community to define benchmarks[89].

The development of a benchmark requires a community to verify their understanding of the field by coming to an agreement on what are the key problems, and encapsulating the knowledge in an evaluation. The use of benchmark results in a more severe examination of research contributions, and a general improvement both in the tools and techniques being created. During the benchmarking process, there is greater communication and collaboration among different research groups, leading to a solid consensus on the community's research objectives. The technical advancement and increased cohesiveness in the community have been viewed as positive side-effects of benchmarking. Therefore, benchmarking is strongly recommended to communities aiming to achieve these positive effects [89].

Any research community that is enough well-established and practice collaboration can benefit from benchmarking. Multi-author publications, multi-site research projects, and standards for reporting, are examples for the latter.

5.2.1 Examples

TPC-A is the Transaction Processing Council (TPC) Benchmark A for Online Transaction Processing including a LAN or WAN network. It emerged from the DebitCredit test initially issued in 1984. This effort was led by Jim Gray [35] but had so many contributors from industry and academia that the author on the paper was given as "Anon et al." This paper arouses the database community. Researchers began to publish improvements and variants of the test, and vendors used the version and interpretation that made their product appear most efficient, further fuelling "benchmarketing" wars. Eventually, several representatives from industry and academia formed the TPC to standardise and manage benchmarks for Database Management Systems (DBMS). Developing TPC-A took more than two years and required almost 1200 person-days of effort contributed by researchers and 35 vendors who were members of the consortium. The process required many meetings as well as laboratory work by the participants. The TPC-A specification is more than 40 pages long with 11 different clauses covering issues such as transaction and terminal profiles, scaling rules, response time, and rules for full disclosure [89].

SPEC (Standard Performance Evaluation Corporation) is a consortium with committees that create a variety of benchmarks. CPU2002 is the one responsible for evaluating computer systems. It replaces CPU95 and is programmed to be replaced in 2004. Members of the committees include hardware and software vendors, universities and customers. Requirements, test cases, and votes on benchmark composition are solicited from committee members and the general public through SPEC's web site. CPU2000 consists of 26 programs (12 with only integer arithmetic and 14 with floating point) to be compiled and run on a computer system [89].

SF100 [45] is a benchmark containing a large set of projects that were extracted from the SourceForge open-source development platform. The selection was targeting a dataset of all projects written in the JAVA programming language, collected using a Web crawler. The final project list contains 100 projects randomly selected from a final filtered list of 321 projects.

SF110 [47] is a combination of SF100 and the ten most popular open source projects according to the SourceForge website.

According to the author, SF100 and SF110 do not only represent the largest case studies in the literature of test data generation for object-oriented software to date but most importantly they are the only ones that are not negatively affected by threats to external validity [45].

5.2.2 Issues

While it is very useful to speed up development in the research area, complains regarding benchmarks are not uncommon. Most of the questions are related to the lack of a methodology when creating benchmarks. In the worst cases, no details regarding the content are presented. One good example comes from the Operational Research (OR) Community. Hooker, in his paper "Testing heuristics: we have all it wrong" [55], expressed his concerns about the lack of diversity and usefulness of commonly studied benchmarks and their inherent bias. He claims that these instances are typically well suited to the first study that chooses to report on them but not necessarily diverse or challenging.

Back to Software Engineering Community, Fraser in [45] questions, "How are case studies chosen in the literature?". He states that in most cases, this choice is not made systematically, i.e., researchers choose software artifacts without providing any specific and unbiased motivation. Notice that, in many software testing contexts, this is the only viable option. This is a typical example in the context of testing techniques targeted for industrial systems. Obtaining real data from industry is challenging and time-consuming activity, and so case studies tend to be either "small" or biased toward a specific kind of software.

The situation gets even worse when in a group of 50 papers, more than one third of them are considering container classes (e.g., vectors and lists) exclusively. Targeting only these particular types of classes might create a strong bias in the results due to the not common practice of writing new container classes. For example, when an industrial set of classes containing 4,208 classes is considered, not even a single class is a container class [47]. Thus there is a high chance that the results are misleading, since the datasets on which the tools are evaluated may not be diverse enough to highlight their true capabilities. Therefore, most of the results claiming the superior performance of an

ATCG technique over other methods on a selected set of CUTs cannot be generalized to untested datasets.

Panichella et al. [75] have shown that 56% of the classes in SF110 are trivial, i.e., with Cyclomatic Complexity (CC) equal to 1. This means that the classes have no branches and can be covered with a simple method call. These types of classes present no challenge to even the most straightforward tool.

Table 5.1 presents a sample of 50 studies where only a single paper [73] explained why a particular set of classes was selected and how this selection was done.

Table 5.1: Evaluation Setting in the Literature [47]

Tool	Reference	Projects	Classes	Source
APex	[Jamrozik et al. 2012]	9	9	Open Source
Artoo	[Ciupa et al. 2008a]	1	8	Open Source
AutoTest	[Ciupa et al. 2008b]	1	27	Open Source
Ballerina	[Nistor et al. 2012]	6	14	Open Source
CarFast	[Park et al. 2012]	12	1,500	Generated
Check'n'Crash	[Csallner et. al. 2005]	2	-	OS / Literature
Covana	[Xiao et al. 2011]	2	388	Open Source
CSBT	[Sakti et al. 2012]	2	3	Open Source
DiffGen	[Taneja and Xie 2008]	1	21	Literature
DSDCrasher	[Csallner et al. 2008]	2	24	Open Source
DyGen	[Thummalapenta et al. 2010]	10	5,757	Industrial
Eclat	[Pacheco and Ernst 2005]	7	631	OS/Lit./Constr.
eCrash	[Ribeiro et al. 2010]	1	2	Open Source
eCrash	[Ribeiro et al. 2009]	1	2	Open Source
eToc	[Tonella 2004]	1	6	Open Source

continued

continued

Tool	Reference	Projects	Classes	Source
eToc	[McMinn et al. 2012]	10	20	Open Source
EvaCon	[Inkumsah and Xie 2008]	1	6	Open Source
EvoSuite	[Fraser and Arcuri 2011a]	6	727	OS + Industrial
EvoSuite	[Fraser and Arcuri 2013c]	20	1,741	OS + Industrial
Jartege	[Oriat 2005]	1	1	Constructed
JAUT	[Charreteur et. al. 2010]	3	7	Constructed
JCrasher	[Csallner et. al. 2004]	1	8	Literature
JCute	[Sen and Agha 2006]	1	6	Open Source
jFuzz	[Jayaraman et. al. 2009]	1	-	Open Source
JPF	[Visser et al. 2004]	1	1	Open Source
JPF	[Visser et al. 2006]	1	4	Constructed
JTest+Daikon	[Xie and Notkin 2006]	1	9	Constructed / Lit.
JWalk	[Simons 2007]	6	13	Constructed
Korat	[Boyapati et al. 2002]	1	6	Literature
MSeqGen	[Thummalapenta et al. 2009]	2	450	Open Source
MuTest	[Fraser and Zeller 2012]	10	952	Open Source
NightHawk	[Andrews et al. 2007]	2	20	Literature
NightHawk	[Andrews et al. 2011]	1	34	Open Source
NightHawk	[Beyene et. al. 2012]	2	-	Open Source
OCAT	[Jaygarl et al. 2010]	3	529	Open Source
Palus	[Zhang et al. 2011]	6	4,664	OS + Industrial
Pex	[Tillmann and Schulte 2005]	2	8	Constructed
PexMutator	[Zhang et al. 2010]	1	5	Open Source

continued

continued

Tool	Reference	Projects	Classes	Source
Randoop	[Pacheco et al. 2007]	14	4,576	OS / Industrial
Rostra	[Xie et al. 2004]	1	11	Constructed / Lit.
RuteJ	[Andrews et al. 2006]	1	1	Open Source
Symclat	[d'Amorim et al. 2006]	5	16	Constructed / Lit.
Symstra	[Xie et al. 2005]	1	7	Literature
Symbolic JPF	[Pasareanu et al. 2008]	1	1	Industrial
Symbolic JPF	[Staats and Pasareanu 2010]	6	6	Industrial/OS
TACO	[Galeotti et al. 2010]	6	6	OS/Lit.
Testera	[Marinov and Khurshid 2001] 4		4	Open Source
TestFul	[Baresi et al. 2010]	4	15	OS + Literature
YETI	[Oriol 2012]	100	6,410	Open Source
N/A	[Arcuri and Yao 2008]	1	7	Open Source
N/A	[Wappler and Wegener 2006]	2	4	Open Source
N/A	[Andrews et al. 2008]	2	2	Open Source

5.3 Features

Features are problem dependent and must be chosen so that the varying complexities of the CUT instances are exposed, any known structural properties of the CUTs are captured, and any known advantages and limitations of the different ATSGTs are related to features. The most common measures and metrics used to characterise features of CUTs are extracted from code.

Other software features are extracted from the Control Flow Graph (CFG) which represents the structure of the code (see Figure 2.1). The most common metric that is based on the CFG is the Cyclomatic Complexity (CC) [67], which analyses the branching complexity of the CFG. Our approach uses well-established software measures, such as CC, and introduces new measures inspired by graph theory to analyze different aspects of the CFG. These metrics can provide valuable information about graph complexity. In essence, the software features considered in this work can be classified into code-based and graph-based features.

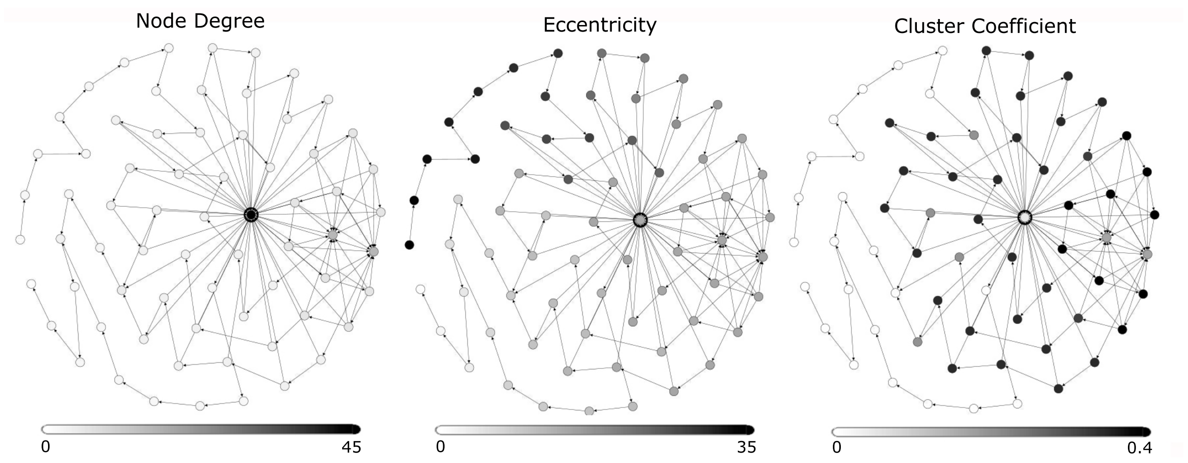


Figure 5.1: Example of Graph Features

5.3.1 Code-based Features

Useful features of CUTs are measurable properties that (a) can be computed in polynomial time and (b) are expected to expose what makes a ATSGT hard for a given CUT. At the same time, features must correlate to ATSGT performance, measure diverse aspects of the CUTs, and be uncorrelated with one another. The feature set should be small in size, yet it should comprehensively measure aspects of the CUTs that either challenge ATSGTs or make their task easy.

The first group of objected oriented code metrics was proposed based on measurement theory and expertise of experienced software developers [20]. The set includes simple features that count the number of methods or lines of code to more elaborated features that measure the interaction between methods and the depth of the inheritance tree. In total, 25 code-based metrics are used (Table 10.1).

The second group has been explicitly defined to capture predicate hardness. For each CUT, features of the predicates are measured, such as the number of variables in a predicate, number of inequalities, and type of variables. The second set of features is present in Table 10.2.

We postulate that it is characteristics of branches in a CUT that make it harder or easier for a search-based software testing method to cover. The META framework will identify the specific characteristics that have an impact.

5.3.2 CFG-based Features

The selected set includes features used in graph theory and also features created specifically for software systems like CC [67]. The CFG is a directed graph, however, undirected graph features are also used by removing the direction restriction. The CFG is generated per method, therefore, to generate values for the whole CUT, metrics are reported as average, standard deviation, sum, maximum, and minimum values. The full list of CFG-based features is presented in Table 10.3.

5.3.3 Feature Extraction

Software systems are considered at a class level, and features are measured for each class. The **feature extraction** procedure shown in Figure 4.1 can be instantiated with existing tools for feature extraction, such as jCT [30] and CKJM [92]. In this work, we introduce new metrics that are based on control flow graph (CFG) features and inspired from graph theory (Table 10.3). The feature extraction procedure for these metrics can be performed using the software package NetworkX [86].

5.4 Benchmark Assessment

In this section we will use META tool to analyze the biggest benchmark set publicly available. The SF110 Corpus of Classes [47] is a statistically representative sample of 110 Java projects from SourceForge. The project details are presented in Table 5.2

Table 5.2: Summarized statistics of the whole SF110 corpus [47]

Property	Value
Number of Projects	110
Number of Testable Classes	23,886
Number of Target Branches	808,056
Number of Java Files	27,997
Lines of Code	6,628,619
Non-Commenting Source Statements	2,340,843
Average Cyclomatic Complexity Number	2.63
Number of Jar File Libraries	1,939

The analysis consists in extracting the most significant features available in the literature associated with ATSGT hardness and visualizing it in a 2-dimensional space. The selected features are: Average Cyclomatic Complexity, Average Method Complexity and Coupling between Object Classes.

In order to visualize the SF110 classes in a meaningful way, we apply the META Tool visualization component. The SF110 classes are projected in a 2-dimensional space using the three features mentioned above, revealing classes considered easy and hard according to the literature. Three new axes were created, which are linear combinations of the three selected features. **Projecting it using the two principal components**

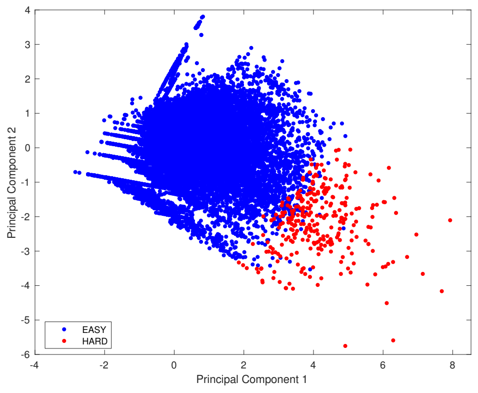
holds 89% of the variation in the data. The coordinate system that defines the new instance space is defined as:

$$\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} 0.7 & 0.75 & 0.92 \\ -0.57 & 0.61 & -0.02 \end{bmatrix} \begin{bmatrix} \text{Average Cyclomatic Complexity} \\ \text{Average Method Complexity} \\ \text{Coupling between Object Classes} \end{bmatrix} \tag{5.1}$$

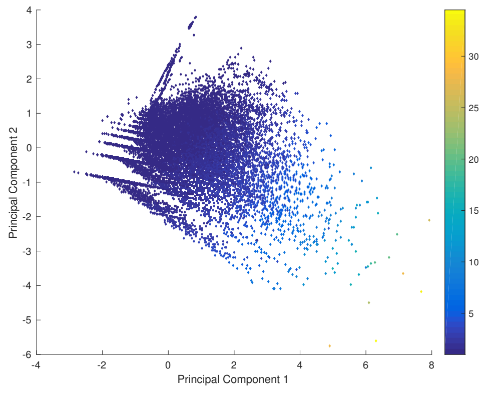
Applying this new coordinate system on all SF110 classes and plotting, we create a new 2-dimensional space with the two main principal components p_1 and p_2. The centre of this new space corresponds to classes with average values of the features Average Cyclomatic Complexity, Average Method Complexity and Coupling between Object Classes. The classes that are near to each other in the new space have similar values of these three features. Figure 5.2a presents the 2-dimensional image of SF110. Figures 5.2b, 5.2c, 5.2d present the feature distribution in the SF110 class space.

The literature states CUTs with Cyclomatic Complexity lower than five are extremely easy to be solved due to the reduced number of branches [75]. Therefore, we consider a CUT hard if its Average Cyclomatic Complexity is equal to or higher than 5, meaning that the CUT contains at least two conditional statements. Table 10.5 presents the selected classes, and Table 5.3 shows a summary of the selected projects. Figure 5.2a presents the 2-dimensional image of SF110 complexity.

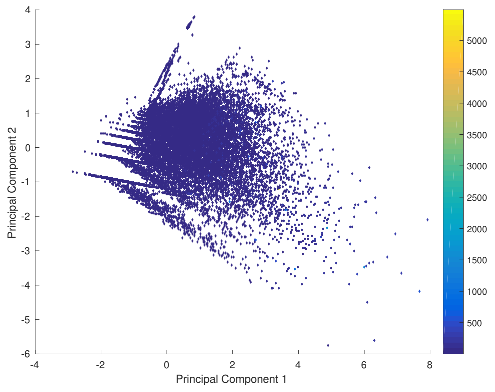
Using these CUTs, we were able to successfully associate ATGST performance to features in the first controlled experiment (Chapter 6). However, these CUTs were unable to challenge the ATSGTs in the second experiment portfolio. Therefore, a different approach was required to define the second experiment dataset. The alternative approach is detailed in the next section.



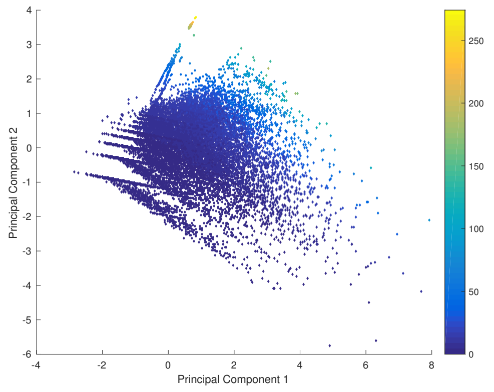
(a) SF110 CUT Space.



(b) Average Cyclomatic Complexity



(c) Average Method Complexity



(d) Coupling between Object Classes

Figure 5.2: SF110 Benchmark in a 2D representation

5.5 Benchmark Generation

As stated previously, a critical step of META framework is the dataset that is used to train the META model. In the previous section, we examined the diversity of a common dataset used in SBST (SF110 [47]). We observed that this benchmark instances are not diverse enough, as one of the fitness functions had superior performance in all instances, suggesting one of the following: (i) the instances are not revealing the unique strengths and weaknesses of each fitness function as much as is desired, or (ii) the features are not discriminant enough. Therefore, we propose a method to generate new CUTs, in order to enrich the repository's diversity.

Table 5.3: Summary of SF110 classes with Ciclomatic Complexity equal to or higher than five

Project	# Classes	Average CC	Average Method Complexity	Coupling between Object Classes
squirrel-sql	17	7.88	86.77	4.12
sweethome3d	5	6.79	143.95	18.60
vuze	78	7.56	104.92	23.86
freemind	10	8.66	72.73	11.00
checkstyle	4	5.50	42.13	8.00
weka	24	7.76	133.61	9.42
liferay	30	6.68	89.44	11.90
pdfsam	16	8.07	99.60	9.06
firebird	1	7.20	107.00	36.00
dsachat	2	6.00	122.38	17.50
beanbin	1	5.50	61.00	0.00
jsecurity	1	10.22	80.00	2.00
jmca	3	7.60	98.94	4.00
tullibee	2	20.50	144.13	1.50
byuic	2	8.27	123.77	8.50
saxpath	1	5.67	37.67	0.00
gangup	4	7.99	122.11	11.00
apbsmem	1	5.00	253.00	6.00
a4j	1	5.50	114.50	8.00
httpanalyzer	1	10.67	229.67	2.00
javaviewcontrol	3	7.70	85.89	1.67
corina	7	6.60	107.15	5.00
schemaspy	1	5.43	97.86	7.00
javabullboard	1	5.39	75.36	0.00
lilith	22	7.34	70.38	8.82
summa	5	6.64	94.33	4.20
nutzenportfolio	1	6.40	80.00	13.00
dvd-homevideo	2	5.51	186.70	3.00
diebierse	1	5.33	79.67	5.00
biff	1	10.39	183.68	4.00
jiprof	7	7.18	93.36	6.86
lagoon	1	6.80	108.00	6.00
db-everywhere	2	5.62	105.61	12.50
jhandballmoves	3	5.29	41.88	8.00
hft-bomberman	1	7.67	94.11	6.00
templateit	1	5.00	65.33	15.00
noen	1	5.15	81.85	10.00
openjms	6	6.83	90.00	1.17
echodep	3	16.41	227.30	48.00
battlecry	2	8.91	127.89	4.00
fixsuite	1	10.50	172.00	8.00
openhre	1	8.00	97.71	3.00
wheelwebtool	6	7.37	131.77	7.17
javathena	4	10.33	72.63	5.00
ipcalculator	2	6.08	130.42	1.00
xbus	2	6.33	84.00	3.00
ifx-framework	1	6.33	73.67	1.00
shop	2	7.38	84.30	8.50
jaw-br	1	18.00	236.00	9.00
jopenchart	3	5.11	71.72	4.33
jiggler	11	8.11	291.63	4.36
gfarcegestionfa	5	7.06	103.56	3.80
dcparseargs	1	5.00	61.56	3.00
celwars2009	1	7.60	160.25	3.00
heal	4	8.36	179.82	14.50
feudalismgame	2	19.53	293.42	8.50
newzgrabber	4	5.78	127.73	6.00
TOTAL	326			

While there is no doubt that these problem repositories have had a tremendous impact on SBST studies, and have improved research practice by ensuring comparability of performance evaluations, there is also concern that these repositories may not be a representative sample of the larger population of software testing problems. It is important to challenge whether existing datasets are enabling us to evaluate fitness function performance in an unbiased manner, and therefore we seek new tools and methodologies that enable us to generate new problem instances that drive an improved understanding of the strengths and weaknesses of different approaches. The development of such methodologies to support the objective assessment of different fitness functions is at the core of the META framework. This is achieved by generating artificial CUTs.

The generated CUTs must be diverse and large enough to cover a wide degree of problem difficulty uniformly, that is for all fitness functions there must exist both easy and hard instances, and the transition from easy to hard should be densely covered [72]. The most obvious way to artificially generate CUTs is to select and sample an arbitrary probability distribution. However, this approach lacks control, as there is no guarantee that the resulting dataset will have specific features. Hence, we employ a different method, initially introduced for machine learning problems [72], where datasets are evolved using a genetic algorithm to be present at target locations in the instance space.

In this work, we use a Genetic Programming (GP) algorithm to evolve branch predicates that are easy and hard for different fitness functions. In GP the individuals in the population are computer programs represented as trees. New programs are produced by removing branches from one tree and inserting them into another. This simple process ensures that the new program is also a tree and thus it is also syntactically valid.

The GP uses a variable length (n) solution representation:

$[(vt_0, ct_0, cv_0), (vt_1, ct_1, cv_1), ..., (vt_n, ct_n, cv_n)]$, where each gene (vt_i, ct_i, cv_i) has three components: variable type vt, comparator type ct, and variable value vv. The GP algorithm was set only to generate classes that are similar to real-world CUTs (unrealistic CUTs were discarded). The CUTs have one method and one nested set of conditional statements up to 4 levels and 14 branches. The possible values for variable types, comparator types and

- Variable types: double, long, integer and boolean,
- Comparator types: $=, \neq, >, <, \leq, \geq$
- Variable value ranges: Boolean: 0, 1, Double: $[-1.7 * 10^{308}, +1.7 * 10^{308}]$, Integer: $[-2^{31}, +2^{31}]$, Long: $[-2^{63}, +2^{63}]$

The GP uses mutation and crossover operators to evolve classes of various characteristics and levels of difficulty for our fitness functions. The mutation operator modifies the variable type, the comparator type or the comparator value. For example, given a parent solution $s = [(vt_0, ct_0, vv_0), (vt_1, ct_1, vv_1), ..., (vt_n, ct_n, vv_n)]$, the mutation operator generates a new candidate solution $s' = [(vt_0, ct_0, vv_0), (vt'_1, ct_1, vv_1), ..., (vt_n, ct_n, vv_n)]$ by mutating the variable type vt_1 into vt'_1.

The crossover consists of exchanging predicates of two parent solutions to generate two new solutions. For example, given two parent solutions

$$s^1 = [(vt^1_0, ct^1_0, vv^1_0), (vt^1_1, ct^1_1, vv^1_1), ..., (vt^1_n, ct^1_n, vv^1_n)]$$

and

$$s^2 = [(vt^2_0, ct^2_0, vv^2_0), (vt^2_1, ct^2_1, vv^2_1), ..., (vt^2_n, ct^2_n, vv^2_n)]$$

the crossover operator generates the following two new candidate solutions

$$s'^1 = [(vt_0^2, ct_0^2, vv_0^2), (vt_1^1, ct_1^1, vv_1^1), ..., (vt_n^1, ct_n^1, vv_n^1)]$$

and

$$s'^2 = [(vt_0^1, ct_0^1, vv_0^1), (vt_1^2, ct_1^2, vv_1^2), ..., (vt_n^2, ct_n^2, vv_n^2)]$$

.

The main steps of the method are presented in Algorithm 2. The following parameter values were used: (a) Population size: 500, (b) Mutation rate: 0.1, (c) Crossover rate: 0.85, (d) Termination condition: 1000 iterations.

Algorithm 2 Generating CUTs with Genetic Programming.

```
 1: procedure EvolveHardCUTs(dataSet, OF1, OF2)
 2:     Input: $n, m_r, c_r$ ▷ $n$ is population size, $m_r$ is mutation rate, $c_r$ is crossover rate, FF1
    and FF2 are the fitness functions.
 3:     $P \leftarrow$ RandomSolutions(n)                          ▷ P is the population
 4:     for $i \leftarrow 1$ to $n$ do
 5:     $Q \leftarrow \emptyset$                                   ▷ Q is the Auxiliary Population
 6:     while !TerminationCodition do
 7:         for $i \leftarrow 1$ to $n$ do
 8:             $p_1, p_2 \leftarrow$ RouletteWheelSelection(P)
 9:             $o_1, o_2 \leftarrow$ UniformCrossover($p_1, p_2, c_r$)
10:             $o'_1 \leftarrow$ Mutation($o_1, m_r$)
11:             $o'_2 \leftarrow$ Mutation($o_2, m_r$)
12:             $f(o'_1) \leftarrow$ EvaluateFitness($o'_1$,FF1, FF2)
13:             $f(o'_2) \leftarrow$ EvaluateFitness($o'_2$,FF1, FF2)
14:             Insert($o'_1, o'_2, Q$)
15:         $R \leftarrow P \cup Q$
16:         RankSolutions($R$)
17:         $P \leftarrow$ SelectBestSolutions($R$)
18:     Return($P$)
19: procedure EvaluateFitness($c$, FF1, FF2)
20:     $bc_1 \leftarrow$ execute($c$, FF1)
21:     $bc_2 \leftarrow$ execute($c$, FF2)              ▷ $bc_1, bc_2$ are the branch coverage performance
22:     Return($bc_1/bc_2$)
```

The GP algorithm evolved 202 classes. CFF did not outperform the other objective functions in any of these classes. It is possible that CFF is superior in CUTs with unrealistic features, such as nested if conditions with a tree size larger than 10. However, our CUT generator was set up in a way that unrealistic CUTs were discarded. An example of a CUT is shown in Figure 5.3.

```java
public class Evolved_CUT_1 {
        public void method1 (Double number00,Long number10,Integer number20,Integer
            number30,Long number40,Boolean number50,Boolean number60){
                if(( number00 <= 2.6234427914696525E307D)){
                        System.out.println("b1");
                        if(( number10 < -4282759360621669546L)){
                                System.out.println("c1");
                                if(( number30 == 1640314761)){
                  System.out.println("d1");
                                } else { System.out.println("d2"); }
                        } else { System.out.println("c2");
                                if(( number40 > 550514008264203262L)){
                  System.out.println("d3");
                                } else { System.out.println("d4"); }
                        }
                } else {
                System.out.println("b2");
                        if(( number20 > 1937584)) {
                        System.out.println("c3");
                                if(( number50 == false)) {
                        System.out.println("d5");
                                } else { System.out.println("d6"); }
                        } else { System.out.println("c4");
                                if(( number60 == true)){
                        System.out.println("d7");
                                } else {
                        System.out.println("d8");
                                }
                        }
                }
        }
}
```

Figure 5.3: An example of an evolved CUT.

The generated classes contain one method with six parameters that have been typed randomly during the generation process aiming to insert hardness into the classes. The method contains nested "if / else" operators that are used by the genetic programming algorithm to create conditions that will increase the class difficulty to each ATSGT in the portfolio. The depth of the "if / else" tree is limited to 3.

5.6 Summary

In this chapter, we presented the dataset and the features that will be used in the two controlled experiments that aim to validate the alternative method to ATSGT assessment called META. We described two types of features extracted either from the code or from the class control-flow graph. We used some of these features to assess the quality (hardness) of a publicly available benchmark SF110 using META tool 2-D visualization component. Additionally, we presented an alternative method to generate (evolve) easy and hard CUTs that can be used when the current benchmark is not challenging enough.

The dataset selection **[RQ2]** and feature set definition **[RQ3]** presented in this chapter are the most important step of the META. In order to detect the techniques' footprint, META requires all the techniques to be pushed to its limits. This is done by defining a dataset that contains CUTs with different characteristics that can be described as features. In this way, the techniques are evaluated in a variety of scenarios and, most of the times, they lose their competitive advantage according to the variation of a specific feature. META has been built to capture these variations by using machine learning algorithms.

Part III

Practical Validation

Chapter 6

META Evaluation 1

6.1 Introduction

This chapter describes the first controlled experiment designed to evaluate the META Framework capabilities. The experiment consists in use META Tool to identify the characteristics of the CUTs that cause each ATSGT in the portfolio to have a competitive advantage over each other, using these characteristics to visualize the ATSGTs footprints and predict their performance. The experimental results indicate that the ATSGTs tested are problem dependent and its performance can be predicted with high accuracy.

6.2 Experiments Setup

Three techniques were selected to be part of the study. The first technique is called WSA (2.2.2) and uses an SBST approach that is considered one of the most promising and has been exhaustively investigated by the community [7]. The second method is also part of the SBST category and is known as MOSA [76] (2.2.3). It introduces a new preference criterion based on the Pareto optimality algorithm. The third method is

RT (2.2.1), which is recommended as a comparison baseline to evaluate new techniques [53].

The selected tool used in the simulations was EvoSuite [46], known as the state-of-the-art in search-based software testing generation [48]. The tool implements all three techniques (WSA, MOSA and RT). Each ATSGT configuration was executed 10 times on each CUT to account for randomness inherent in the ATSGTs. The number 10 has been derived from SF110 first case study [47]. The same number of runs has also been applied in [75], using a similar dataset extracted from SF110. The experiments took more than 4.200 minutes. The tool was executed using different random seeds (10 different random seeds), and the average of the ten runs was considered. The time-out was two minutes per class, stated as the best trade-off between time and branch coverage in [47]. EvoSuite was executed using its default parameter settings [46]. Recent research has shown that the default values, commonly used in the literature, give reasonably acceptable results [11]. Table 10.4 presents the complete set of default parameters. The employed machines were hosted in a cloud cluster of 10 instances with two CPUs and 8 GB of RAM each were used.

The CUTs considered in this study are part of the SF110 Corpus of Classes [47], which is a statistically representative sample of 110 Java projects from SourceForge. Classes with branchless methods, i.e., methods that can be completely covered with a simple call, have been excluded and only classes with CC equal or higher than five have been selected. A CC higher than five means that the methods have at least two conditional statements. From 326 classes, 115 have been excluded due to execution errors. In total, 211 classes have been considered. An overview of the selected classes is presented in Table 6.1.

Table 6.1: Overview of the selected projects.

Number of Classes	211
Average Cyclomatic Complexity:	7.57
Standard Deviation of Cyclomatic Complexity	4.23
Minimum Cyclomatic Complexity	3.00
Maximum Cyclomatic Complexity	171

Branch coverage was used as the ATSGT performance measure. An ATSGT is considered superior if its branch coverage is at least 1% higher than the other techniques, otherwise, the label "Equal" is used.

6.3 Results

The statistics from the results of the experiments are presented in Table 6.2 for both class and project level.

Table 6.2: Branch coverage statistics. Confidence Intervals (CI) were calculated using bootstrapping [40] at 95%.

ATSGT	Group	Med	CI	Mean	CI	Std
RT	Class	0.50	[0.40 - 0.60]	0.51	[0.46 - 0.56]	0.37
	Project	0.41	[0.28 - 0.54]	0.44	[0.37 - 0.52]	0.31
MOSA	Class	0.47	[0.40 - 0.55]	0.50	[0.45 - 0.55]	0.37
	Project	0.44	[0.27 - 0.54]	0.43	[0.35 - 0.52]	0.31
WSA	Class	0.48	[0.40 - 0.56]	0.50	[0.45 - 0.55]	0.37
	Project	0.40	[0.27 - 0.54]	0.44	[0.36 - 0.51]	0.31

As can be seen, the overall performances of the three ATSGTs are very similar, with a median coverage between 0.41 and 0.5. The standard deviation of the different techniques is also very similar. These statistics indicate that, overall, there is little difference between the effectiveness of the three techniques in generating high-quality test suites. The question, however, remains whether for particular CUTs a certain ATSGT is better than the others.

6.3.1 Feature Selection

As described in Section 4.3.3, the META framework performed feature learning on the 56 features that were extracted from the 211 CUTs. The aim is to select the best set of features that highlight the strengths and weaknesses of the ATSGTs. The feature learning process searches for groups containing between 3 and 10 features. To account for the randomness in the results, each trial was run 10 times on each CUT for each approach, using different random seeds, and the mean was considered. The Support Vector Machine (SVM) identified high-density areas in the 2D CUT space where WSA, MOSA and RT perform well with **88% accuracy**. The META framework identified the following optimal features which best capture the difficulty in generating test cases for the CUTs:

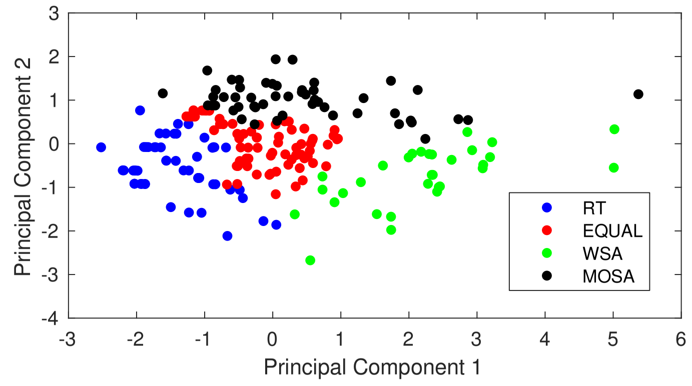
i) *Number of Methods*;

ii) *Coupling between Object Classes*;

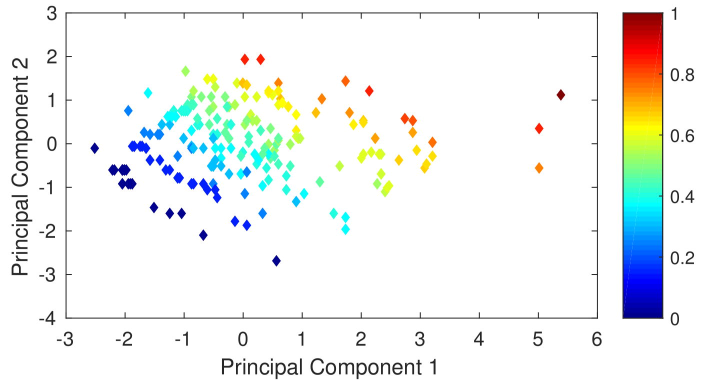
iii) *Response for a Class.*

Using these three features, we were able to characterise the CUT Space and define the footprints of the techniques. In essence, the answer to the fourth research object is

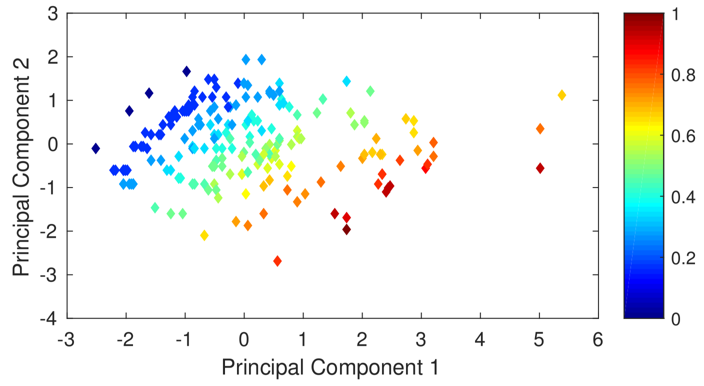
RO4: The most significant features that have an impact on the effectiveness of WSA, MOSA and RT are **Number of Methods**, **Coupling between Object Classes**, and **Response for a Class**.



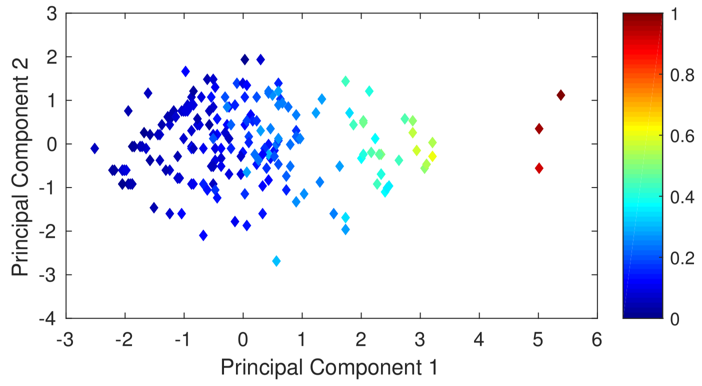
(a) Effectiveness Map.



(b) Coupling Between Object Classes



(c) Number of Methods.



(d) Response for a Class.

Figure 6.1: Visualisation of the effectiveness footprints of Automated Test Suite Generation Techniques. The principal components are defined in Equation 7.1.

6.3.2 Effectiveness Visualization

To visualize the results in a meaningful way, we apply PCA as a dimensionality reduction technique on the optimal subset of features presented in the previous section. The aim is to plot the performance of the two ATSGTs across the CUT space in 2D, which is likely to reveal where the methods are performing well, and where they are suffering. Three new axes were created, which are linear combinations of the selected set of features. **Projecting it using the two principal components holds 86% of the variation in the data.** The coordinate system that defines the new instance space is defined as:

$$\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} 0.7 & 0.75 & 0.92 \\ -0.57 & 0.61 & -0.02 \end{bmatrix} \begin{bmatrix} \text{Number of Methods} \\ \text{Coupling between Object Classes} \\ \text{Response for a Class} \end{bmatrix} \tag{6.1}$$

Applying this new coordinate system on all CUTs and plotting, we create a new CUT sub-space with the two main principal components p_1 and p_2. The centre of this new space corresponds to a CUT with average values of the features Number of Methods, Coupling between Object Classes and Response for a Class. CUTs that are near to each other in the CUT sub-space have similar values of these three features.

Using a DBSCAN to cluster the CUTs, we identified the area in the new space where the different techniques perform well. We also determined the area where the features were unable to identify the best technique, which was labeled as *Equal*. Figure 7.1 is analysed in the same way as in Figure 4.2.

Figure 6.1a presents the footprints of the ATSG techniques in the CUT-space. The most important feature that impacts ATSGT performance is the Coupling between Object Classes. Figure 6.1b shows the distribution of the Coupling between Object

Classes feature in the CUT-space. The feature has the highest contribution to the first principal component, providing a visual performance boundary. The higher the feature value the harder it is for RT, while the SBST techniques (WSA and MOSA) have better performance with higher values of the feature. In essence, we can conclude that

RO5: Using the two principal PCA components (features **number of methods**, **the coupling between object classes**, and **the response for a class**), we can visualize the ATSGT's footprints in 2D with a variation of 86% (very little data lost).

The distribution of the other two features is presented in Figures 6.1c and 6.1d. The SBST techniques are more effective in generating test cases for CUT with a high Number of Methods and high values of the feature Response for a Class.

Table 6.3 summarises the results for all CUTs. The CUTs are grouped according to the ATSGT Footprint, indicating which technique is the most effective (RT, WSA, MOSA, or Equal). It can be noticed that the overall performances of the three techniques are similar, as shown in the last three rows of Table 6.3. However, analysing the performance within the footprint area, it is possible to see a significant diïňĂerence between the means of the different techniques, ranging from 1% to more than 10%. For example, in the WSA footprint area, the mean of WSA is 4.42% better than RT and 10.53% better than MOSA.

The nonparametric Wilcoxon test [25] with a p-value threshold of 0.05 has been applied to check whether the different performances are statistically significant or not. Table 6.4 presents the results inside each footprint area. RT is statistically superior to MOSA and WSA in 36% and 39% of the cases. MOSA is statistically superior to RT

Table 6.3: The number of classes and average branch coverage reported as percentage (%) grouped by project and footprint area. Effect Size Statistic calculated using Vargha-Delaney ($\hat{A}_{12}$)

Project	Equal Footprint				RT Footprint				MOSA Footprint				WSA Footprint			
	Classes	RT	WSA	MOSA	Classes	RT	WSA	MOSA	Classes	RT	WSA	MOSA	Classes	RT	WSA	MOSA
squirrel-sql	5	82.80	82.80	82.80	6 | 3*	88.80	79.22	84.85	2 | 1*	69.43	70.80	65.20	2 | 2*	68.35	49.39	71.25
sweethome3d	5	14.00	13.92	14.00	-	-	-	-	-	-	-	-	-	-	-	-
vuze	4	91.70	91.85	91.95	1 | 1***	93.30	92.80	92.90	3 | 2*	72.26	75.07	71.40	1 | 1*	8.00	15.00	35.30
checkstyle	-	-	-	-	2 | 1**	48.20	38.00	36.65	2 | 1*	12.60	16.05	12.35	-	-	-	-
freemind	6	24.50	24.50	24.50	3 | 1*	47.27	45.47	44.53	-	-	-	-	1 | 1*	97.00	95.70	99.20
weka	7	57.43	53.92	57.71	2 | 1*	91.85	85.55	86.25	3 | 2*	22.18	23.97	23.03	5 | 3*	43.82	42.52	47.02
liferay	12	75.83	73.46	75.86	1 | 1*	97.20	95.60	95.60	7 | 6**	47.45	55.32	48.75	3 | 2***	61.47	60.07	64.90
jmca	-	-	-	-	1 | 1*	86.60	85.20	85.40	1 | 1***	59.10	71.70	50.20	1 | 1*	37.80	45.70	46.40
jsecurity	-	-	-	-	-	-	-	-	1 | 1***	87.70	89.60	87.90	-	-	-	-
pdfsam	15	19.50	19.50	19.15	-	-	-	-	-	-	-	-	-	-	-	-
firebird	1	96.00	96.00	96.00	-	-	-	-	-	-	-	-	-	-	-	-
dschat	2	1.00	1.00	1.00	-	-	-	-	-	-	-	-	-	-	-	-
beanbin	1	100.00	100.00	100.00	-	-	-	-	-	-	-	-	-	-	-	-
tullibee	2	100.00	100.00	100.00	-	-	-	-	-	-	-	-	-	-	-	-
saxpah	1	100.00	100.00	100.00	-	-	-	-	-	-	-	-	-	-	-	-
gangup	2	2.50	2.50	2.50	1 | 1***	12.40	5.00	5.00	-	-	-	-	-	-	-	-
apbsmem	1	22.00	22.00	22.00	-	-	-	-	-	-	-	-	-	-	-	-
a4j	1	10.00	10.00	10.00	-	-	-	-	-	-	-	-	-	-	-	-
httpanalyzer	1	5.00	5.00	5.00	-	-	-	-	-	-	-	-	-	-	-	-
javaviewcontrol	1	90.00	90.00	90.00	1 | 1***	87.00	86.50	85.50	-	-	-	-	1 | 1**	27.60	31.70	35.40
corina	4	50.60	50.60	49.90	3 | 2*	54.70	44.93	50.60	-	-	-	-	-	-	-	-
schemaspy	1	7.00	7.00	7.00	-	-	-	-	-	-	-	-	-	-	-	-
lilith	3	39.00	37.67	39.00	1 | 1*	76.80	75.60	74.60	-	-	-	-	1 | 1***	47.50	47.30	47.60
summa	1	52.00	52.00	52.00	1 | 1*	42.10	35.10	40.90	1 | 1***	25.30	47.67	12.00	2 | 2*	78.45	64.65	79.30
dvd-homevideo	2	9.00	9.00	7.71	-	-	-	-	-	-	-	-	-	-	-	-
diebierse	1	33.00	33.00	33.00	-	-	-	-	-	-	-	-	-	-	-	-
biff	1	16.40	16.80	16.80	-	-	-	-	-	-	-	-	-	-	-	-
jiprof	2	100.00	100.00	100.00	-	-	-	-	1 | 1 **	67.50	72.50	69.90	2 | 1*	45.20	36.36	48.00
jhandballmoves	1	98.00	98.00	98.00	-	-	-	-	1 | 1***	70.00	71.20	70.80	1 | 1*	39.80	39.00	41.80
openjms	6	43.40	42.44	42.84	-	-	-	-	-	-	-	-	-	-	-	-
echodep	2	7.00	7.00	7.00	-	-	-	-	1 | 1*	2.80	4.00	1.00	-	-	-	-
battlecry	2	80.00	80.00	80.00	-	-	-	-	-	-	-	-	-	-	-	-
fixsuite	1	9.00	9.00	9.00	-	-	-	-	-	-	-	-	-	-	-	-
wheelwebtool	2	51.50	44.36	51.50	1 | 1**	54.00	51.70	52.40	-	-	-	-	3 | 2*	50.53	54.27	56.17
javathena	2	39.25	39.25	37.00	-	-	-	-	-	-	-	-	1 | 1*	92.50	72.00	92.90
ipcalculator	1	33.00	33.00	33.00	-	-	-	-	1 | 1**	94.40	95.50	64.67	-	-	-	-
ifx-framework	1	43.00	43.00	43.00	-	-	-	-	-	-	-	-	-	-	-	-
jaw-br	1	0.00	0.00	0.00	-	-	-	-	-	-	-	-	-	-	-	-
jopenchart	2	82.50	82.50	77.43	1 | 1*	73.50	71.00	71.00	-	-	-	-	-	-	-	-
jiggler	1	86.00	86.00	86.00	3 | 2**	57.60	36.50	56.50	-	-	-	-	2 | 1**	64.60	65.10	67.30
gfarcegestionfa	2	4.50	4.50	4.50	1 | 1**	88.60	88.57	88.20	-	-	-	-	-	-	-	-
dcparseargs	1	98.00	98.00	98.00	-	-	-	-	-	-	-	-	-	-	-	-
celwars2009	1	8.67	13.00	13.00	-	-	-	-	-	-	-	-	-	-	-	-
feudalismgame	2	2.50	2.50	1.93	-	-	-	-	-	-	-	-	-	-	-	-
newzgrabber	2	5.50	4.90	5.50	-	-	-	-	1 | 1****	54.50	59.67	36.63	1 | 1****	8.80	8.90	9.40
lagoon	-	-	-	-	1 | 1*	34.60	34.00	34.40	-	-	-	-	-	-	-	-
hft-bomberman	-	-	-	-	1 | 1*	9.00	7.50	5.00	-	-	-	-	-	-	-	-
xbus	-	-	-	-	1 | 1 ***	89.80	88.00	88.00	-	-	-	-	1 | 1*	98.40	76.60	100.00
shop	-	-	-	-	1 | 1*	22.30	13.00	9.75	-	-	-	-	1 | 1*	55.90	55.40	62.40
byuic	-	-	-	-	-	-	-	-	1 | 1*	48.60	50.80	43.40	1 | 1*	25.30	23.80	25.40
db-everywhere	-	-	-	-	-	-	-	-	1 | 1*	57.10	59.40	52.30	1 | 1*	40.29	33.40	50.10
openhre	-	-	-	-	-	-	-	-	1 | 1***	6.90	7.00	6.50	-	-	-	-
heal	-	-	-	-	-	-	-	-	2 | 1*	81.50	84.19	59.28	2 | 2*	67.80	66.65	71.50
nutzenportfolio	-	-	-	-	-	-	-	-	-	-	-	-	1 | 1**	8.20	6.42	8.40
javabullboard	-	-	-	-	-	-	-	-	-	-	-	-	1 | 1*	28.60	25.30	29.30
templateit	-	-	-	-	-	-	-	-	-	-	-	-	1 | 1*	27.86	30.20	36.40
Total	112	45.03	44.76	44.89	33	62.78	57.96	59.40	30	51.72	56.14	45.61	36	48.86	45.45	53.28

RT Overall Performance	**50.49**	* Large Effect Size	** Medium Effect Size
WSA Overall Performance	**49.40**	*** Small Effect Size	**** Negligible Effect Size
MOSA Overall Performance	**49.75**		

Table 6.4: The number of classes per footprint area divided in three p-values categories. P-values were calculated using the Wilcox nonparametric test.

P-values	Footprint					
	RT		MOSA		WSA	
	WSA	MOSA	RT	WSA	RT	MOSA
<=0.05	13	12	10	11	13	18
>0.05 and <= 0.1	5	8	3	4	4	5
>0.1	15	13	17	15	19	13
Total	33	33	30	30	36	36

and WSA in 33% and 37% of the cases. Lastly, WSA is statistically superior to RT and MOSA in 36% and 50% of the cases.

In order to evaluate the magnitude of the performance difference, the Vargha-Delaney Effect Size Statistic ($\hat{A}_{12}$) [101] is applied. The $\hat{A}_{12}$ statistic is equal to 0.5 if two ATSGTs have equivalent performances and different otherwise. The closer to 0.5, the more similar the performances are. Suppose we are comparing ATSGT A x ATSGT B. The $\hat{A}_{12}$ statistic will be higher than 0.5 if ATSGT A has better performance than ATSGT B and lower than 0.5 if ATSGT B has better performance than A.

According to Table 6.3, the performances in the Equal footprint area are very similar (all classes present a small or negligible effect size). However, the other footprint areas present a significant difference in the coverages. Most of the CUTs in RT, WSA and MOSA footprint areas present a large effect size. In case the most suitable technique is selected (as suggested by the META framework) the coverage can increase by 8%. In our experiments (15,167 branches in total), this means that an additional 1,213 lines will be executed, which is quite significant.

6.3.3 Performance Prediction

ATSGT Decision Tree (DT) was constructed using the c4.5 algorithm implemented in WEKA [50] as J48 algorithm. The following parameters were used: (a) pruning confidence: 0.25, (b) minimum number of instances: 2, (c) number of folds: 3, (d) seed: 1. We use the training data to build a predictive model that is mapped to a tree structure. The goal is to achieve a perfect classification of the ATSGTs with the minimal number of decisions. The rules defining the DT for ATSGT selection are shown in Figure 6.2.

The decision tree can be used to select the appropriate method (WSA, MOSA or RT) when a new software project requires the generation of test cases. The results were validated for consistency and accuracy using a 10-fold cross-validation technique. Results from the classifier are presented in Table 6.5 and 6.6.

Table 6.5: Results from the 10-fold cross validation of the decision tree. The model took 0.1 seconds to build.

Number of leaves in the decision tree	16	
Size of the tree	31	
Kappa statistic	0.84	
Mean absolute error	0.07	
Root mean squared error	0.23	
Correctly Classified Instances	186	**88.15%**
Incorrectly Classified Instances	25	**11.85%**

Table 6.6: Results from the 10-fold cross-validation of the decision tree. Detailed Accuracy By Class.

ATSGT	**Precision**	**Recall**	**F-Score**
RT	91.2%	98.1%	94.5%
EQUAL	87.8%	85.5%	86.7%
WSA	89.3%	80.6%	84.7%
MOSA	84.6%	86.3%	85.4%
Average	88.1%	88.2%	88.1%

Similar accuracy was found with 2 and 3-fold cross validation (85.31% and 88.15% respectively). Precision denotes the proportion of predicted positive cases that are

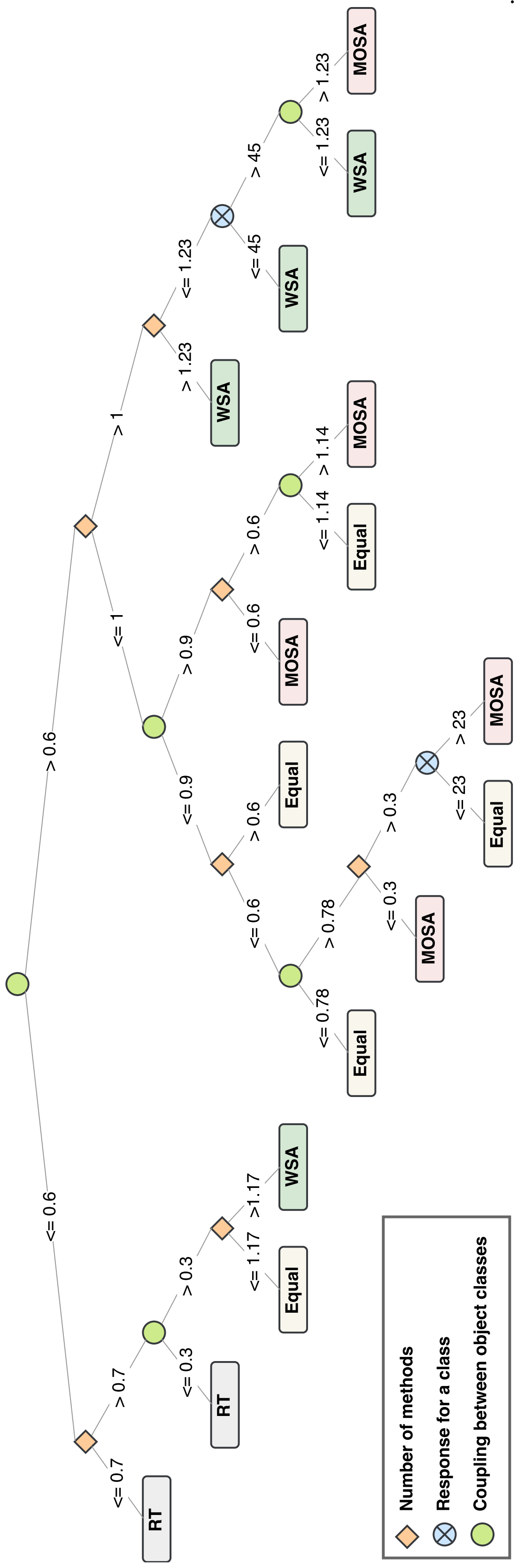


Figure 6.2: ATSGT decision tree.

correctly real positives [81]. The recall is the proportion of real positive cases that are correctly predicted positive [81]. As there is always a quality compromise between Precision and Recall, being desirable but different features, the F-Measure is used as a harmonic mean to counter this problem. It references the true positives to the arithmetic mean of predicted positives and real positives, being a constructed rate normalized to an idealized value [81]. The Kappa statistic is a metric that compares an observed accuracy with an expected accuracy (random chance) [88]. A value greater than zero means that the classifier is doing better than chance.

In summary, the ATSGT decision tree selects the most effective ATSGT technique with high accuracy (88.1%) and F-Score (88.1%), hence we conclude that the answer to the sixth research objective is as follows:

RO6: Using the features **number of methods**, **the coupling between object classes**, and **the response for a class**, it is possible to predict the most suitable ATSGT accurately..

What becomes apparent in the ATSGT decision tree is that RT most of the time outperforms MOSA and WSA in CUTs with low coupling between object classes (≤ 0.6). However, in CUT instances with more than one method (> 1.17) WSA is the most effective technique.

The tree contains leaves marked as Equal, where all ATSGTs perform the same. As RT is a much more effortless technique than WSA and MOSA, in terms of implementation and application, it is reasonable to suggest that one should use RT in cases where the decision tree cannot distinguish between the different techniques. The DT would be useful in making such predictions and guide practitioners in selecting the right method.

In most cases, when there is a high number of methods WSA outperforms the other techniques. Furthermore, in CUTs with a high response for a class (>23) MOSA is the most effective technique, unless the response for a class is greater than 45 and the coupling between objects is low (≤ 1.23), in which case the most effective technique is WSA. The response for a class measures the number of different methods that are executed when an object of that class receives a message (when a method is invoked for that object). The coupling between object classes, on the other hand, represents the number of classes coupled to a given class.

When the coverage achieved by each technique for different levels of coupling between object classes are plotted, as shown in Figure 6.3, it is clear that MOSA is the best technique in CUTs with high values of this feature. In CUTS with low coupling between object classes (<3), RT becomes the best performer.

To conclude, MOSA is the winner in complex CUTs as defined by coupling between object classes, which was the most significant feature in terms of describing CUT hardness, followed by WSA. As expected, RT outperforms the other techniques only in easy CUTs.

6.4 Threats to Validity

Threats to Internal Validity. In this study, we focus on 3 ATSGTs, WSA, MOSA and RT. Although these selected techniques may influence the set of features that have been identified as important, the META framework is conceived in a way that the model of ATSGT effectiveness can continuously improve in accuracy with more techniques. In other words, the more information is provided to the META framework, the more accurate the results will be. The selected dataset, features and performance metric have

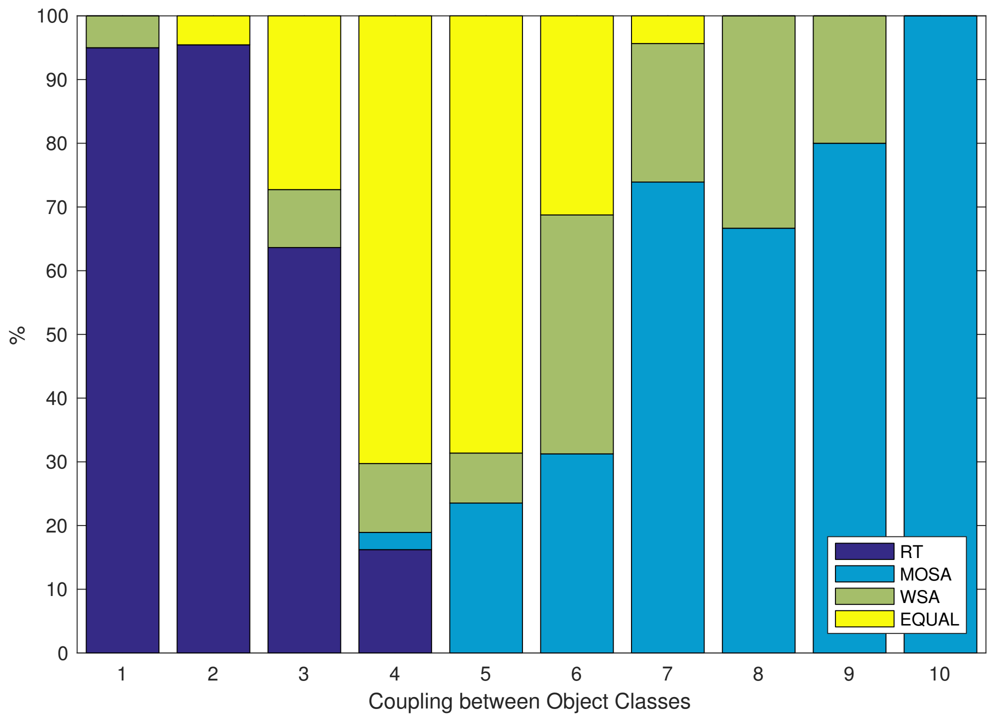


Figure 6.3: Coupling between Object Classes [0,1]. Results have been discretized in 10 bins, eg., 1 represents the range [0,0.1).

a critical influence on the final results. Changing any of these might lead to different outcomes. The META framework is a continuously evolving tool.

The validity of the presented experiments can be questioned on the grounds that the stochastic behaviour of the ATSGTs may lead to different results for the same problem instance. This threat was reduced by conducting multiple runs for each ATSGT and CUT instance. The Wilcox nonparametric test was applied to check for statistical significance in the results. Moreover, Vargha-Delaney ($\hat{A}_{12}$) was used to measure the magnitude of the difference between the performance achieved by two different algorithms. These two statistical tests have been recommended by Arcuri et al. [10].

We also presented results using bootstrapping [40] to create confidence intervals for some of the statistics. These results help the reader to analyse how reliable the presented results are. The experimental results may be affected by the implementation of the META framework. To make sure that the code does not contain errors or bias towards certain methods, we have followed regular code-review and testing sessions.

Threats to External Validity. Threats to external validity relate to the generalizability of the experimental results and are specifically related to the SF110 Corpus if it correctly represents common software systems and the selected ATSGTs. The SF110 Corpus has been specifically assembled with a focus on having a representative benchmark and is accepted by the community. However, there may be bias towards SBST (WSA and MOSA). In the future, we will include other benchmark sets into our investigation.

Chapter 7

META Evaluation 2

7.1 Introduction

This chapter describes the second controlled experiment designed to validate the META Framework capabilities. Differently from the controlled experiment 1, we assessed the performance of the objective function part of the ATSGT. META Tool was used to identify the CUTs features responsible for providing a competitive advantage to each ATSGT in the portfolio. Moreover, the identified features were used to visualize the ATSGTs footprints and predict their performance. The results indicate that the ATSGTs in the portfolio are problem dependent, and its performance can be forecasted with high accuracy.

7.2 Experiment Setup

Three fitness functions were selected to be part of the study. The first fitness function is called CLF (Section 2.3.1) and it is a control flow based approach. As stated in Section 2.3.1, CLF fitness function was modified to exercise both branch values (true and false), rather than arriving at just one value. The second fitness function is a branch distance approach and it is known as BDF (Section 2.3.2). A function that calculates how far a test it is from executing a branch is used to guide the search. The third method is CFF (2.3.3), a control flow based approach.

In the same way as the first experiment, the selected tool used in the simulations was EvoSuite [46]. The three fitness functions in the portfolio (CLF, BDF and CFF) were implemented and executed using the WSA method. Each ATSGT configuration was executed ten times on each CUT to account for randomness inherent in the ATSGTs. The tool was executed using different random seeds (10 different random seeds), and the average of the ten runs was considered. The time-out was two minutes per class, stated as the best trade-off between time and branch coverage in [47].

In all experiments, EvoSuite was executed with its default parameter setting (Table 10.4). The employed machines were hosted in a cloud cluster of 10 instances with two CPUs and 8 GB of RAM each were used. Similarly to the first experiment, the software systems are considered at a class level, and features are measured for each class.

The CUTs considered in this study were evolved using a Genetic Programming Algorithm (5.5). Since the main goal of the fitness function is to guide the SBST method to execute as many branches as possible, we developed an algorithm to evolve hard predicates to each fitness function in the portfolio. The classes are identical, the only exception is the branch predicates.

Genetic Programming (GP) is an evolutionary computation technique that aims to automatically solve problems without a clear description of the solution structure [80]. In GP, a population of computer programs is evolved, and in every generation, the GP transforms the population into a new population that might possibly be better. The evolution is guided by a fitness function that is nothing more than the problem trying to be solved.

In every iteration, the population is transformed by two main operations: the Crossover and the mutation. Crossover is the creation of a new program by combining randomly selected parts of two selected programs in the population. The mutation is the creation of a new program by randomly modifying a randomly selected part of a program in the population.

The algorithm was created to evolve hard CUTs for each fitness function in the portfolio. In other words, the defined fitness function compares the performance of the fitness function A with the other two, aiming to make the performance difference as big as possible. The crossover and mutation only are performed over the branch predicate. The following variable types and operators are used:

- Variable Types: Boolean, Integer, Long, Double;
- Comparison Operators: Greater than (>), less than (<), equal to (==), not equal to (!=), greater than or equal to (>=), less than or equal to (<=).

The CUTs have the same number of methods (1) and the same number of branches (7). In every iteration new programs are generated aiming to create hard and more complex CUTs to each fitness function in the portfolio by changing the variable values, variable types and the comparison operator via mutation or crossover.

As we stated in chapter 6, features are problem dependent and must be chosen so that the varying complexities of the CUT instances are exposed, any known structural properties of the CUTs are captured, and any known advantages and limitations of the different ATSGTs are related to features. Therefore, in order to capture the characteristics that influence the performance of the fitness functions, we need to extract features direct from the branch predicate structure.

Unfortunately, the code-based and graph-based features used in Chapter 6 were not useful due to the high similarities of the CUTs. Therefore, a new set of features were designed specifically to tackle the fitness function problem. Table 10.2 presents the features extracted from the branch predicates. All the features are code-based.

Branch coverage was used as the ATSGT performance measure. An ATSGT is considered superior if its branch coverage is at least 1% higher than the other techniques, otherwise, we use the label 'Equal'.

7.3 Results

The statistics from the results of the experiments are presented in Table 7.1 for the class level. BDF is overall the best fitness function covering approximately 10% more branches than CLF and 34% more than CFF. The question, however, remains whether for particular CUTs a specific fitness function is better than the others.

Table 7.1: Branch coverage statistics. Confidence Intervals (CI) were calculated using bootstrapping at 95%.

Fitness Function	Median	CI	Mean	CI	Std
BDF	0.93	[0.92 - 0.95]	0.89	[0.88 - 0.91]	0.10
CLF	0.84	[0.81 - 0.87]	0.78	[0.75 - 0.81]	0.19
CFF	0.58	[0.48 - 0.69]	0.55	[0.52 - 0.58]	0.20

7.3.1 Feature Selection

As described in Section 4.3.3, the META framework performed feature learning on the 16 predicate features that were extracted from the 202 CUTs. The aim is to select the best set of features that highlights the strengths and weaknesses of the ATSGTs. The feature learning process searches for groups containing between 3 and 10 features. The Support Vector Machine (SVM) identified high-density areas in the 2D CUT space where BDF and CLF perform well with **75% accuracy**. The META framework identified the following optimal features which best capture the difficulty in generating test cases for the CUTs:

i) *Integer Variables with* $<=$ *and* $>=$ *comparator*;

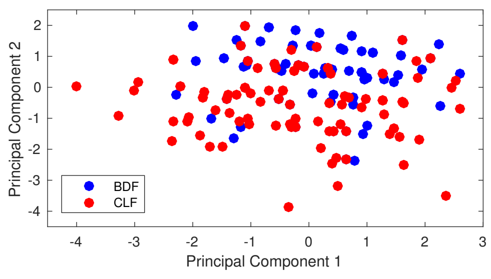
ii) *Equalities with Double Variables*;

iii) *Double Variables with* $<=$ *and* $>=$ *comparator.*

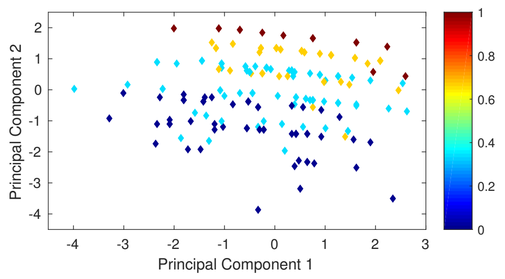
iv) *Equalities with Long Variables*;

v) *Inequalities with Long Variables*;

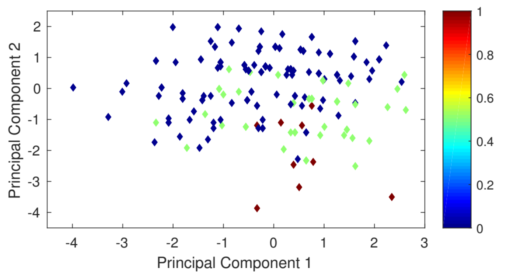
vi) *Long Variables with* $<=$ *and* $>=$ *comparator.*



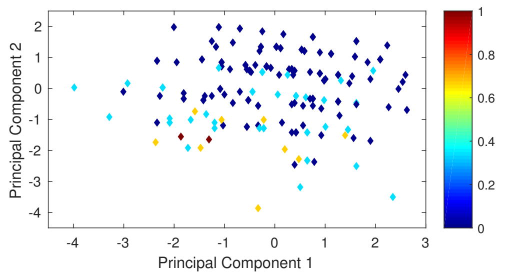
(a) Effectiveness Map



(b) Integer Variables with $<=$ and $>=$ comparator.



(c) Number of Equalities with Long type.



(d) Number of equalities with Double type.

Figure 7.1: Visualisation of the effectiveness footprints of SBST Fitness Functions. The principal components are defined in Equation 7.1.

Using these six features, we were able to characterise the CUT Space and define the footprints of the techniques. In essence, the answer to the fourth research object is

RO4: The most significant features that have an impact on the effectiveness of BDF and CLF are **Integer Variables with <= and >= comparator**, **Equalities with Double Variables**, **Double Variables with <= and >= comparator**, **Equalities with Long Variables**, **Inequalities with Long Variables** and **Long Variables with <= and >= comparator**.

7.3.2 Performance Visualization

Using these six features the META framework created the CUT Space and identified the footprints of the two objective functions. Using the footprint visualisation method, a 2d effectiveness map is created, as shown in Figure 7.1a. Each point is a CUT, which is colored red if CLF is the most effective objective function, and blue if BDF is the winner. **The first two components used to visualise the CUT space explain 47.5% of the variation in the data.** The values of these two components are as follows:

$$\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} -.28 & .21 & -.75 & -.25 & .50 & .76 \\ -.77 & .55 & .11 & .59 & .01 & -.13 \end{bmatrix} \begin{bmatrix} \text{i} \\ \text{ii} \\ \text{iii} \\ \text{iv} \\ \text{v} \\ \text{vi} \end{bmatrix} \tag{7.1}$$

These are the footprints of the two objective functions. The best separation is provided by the second principal component (PC2) axis. Therefore, the features that contribute the most to the PC2 are the most important ones. This answers the first research challenge:

Figures 7.1b, 7.1c and 7.1d show the same principal components, however, the CUTs are now colored according to the most significant features. Figure 7.1b shows how the CUTs score according to the Integer Variables with <= and >= comparator. When both Figures 7.1a and 7.1b are considered side by side, it becomes clear that BDF is effective in generating test suite for classes that have a high number of Integer Variables with <= and >= comparator, whereas CLF is more effective in CUTs that score low according to this feature. Similarly, Figures 7.1c and 7.1d show that BDF is more effective when both the number of equalities with long type and the number of equalities with double type is low, while CLF is more effective when the CUTs score low in these two features.

CLF presents better or equal performance in all cases inside the CLF footprint area, having cases where the performance difference reaches 11%. In most of the cases, however, the difference ranges between 2 and 5%. On the other hand, BDF presents a significantly superior performance inside BDF footprint area. In the best case, the average branch coverage reaches a difference of 55%. This indicates that BDF is better than CLF on average, but there are problem instances where CLF outperforms BDF.

Therefore, we can conclude that the answer to the fifth research object is

RO5: Using the two principal PCA components (features **Integer Variables with $\leq$ and $\geq$ comparator**, **Equalities with Double Variables**, **Double Variables with $\leq$ and $\geq$ comparator**, **Equalities with Long Variables**, **Inequalities with Long Variables** and **Long Variables with $\leq$ and $\geq$ comparator**), we can visualize the ATSGT footprints in 2D with a variation of 47.5%.

7.3.3 Performance Prediction

Next, a decision tree is constructed using the same procedure applied in 6.3.3. The generated tree can be used for objective function selection when solving new SBST problems. The goal is to achieve an optimal classification of the Objective Functions with a minimal number of decisions. The rules defining the decision tree for Objective Function selection are shown in Figure 7.2.

The decision tree can be used to select the appropriate objective function when a new software project requires the generation of test cases. The results were validated for consistency and accuracy using 10-fold cross-validation technique. Results from the classifier are presented in Table 7.2 and 7.3.

Table 7.2: Result from the 10-fold cross-validation of the decision tree. The model took 0.1 seconds to build.

Kappa statistic	0.48	
Mean absolute error	0.3	
Root mean squared error	0.4	
Correctly Classified Instances	150	**74.26**%
Incorrectly Classified Instances	52	**25.74**%

Table 7.3: Result from the 10-fold cross-validation of the decision tree.

Objective function	**Precision**	**Recall**	**F-Score**
CLF	75.8%	71.3%	73.5%
BDF	74.3%	77.2%	75.0%
Average	74.3%	74.3%	74.2%

Precision denotes the proportion of predicted positive cases that are correctly real positives [81]. The recall is the proportion of real positive cases that are correctly predicted positive [81]. As there is always a quality compromise between Precision and Recall, being desirable but different features, the F-Measure is used as a harmonic

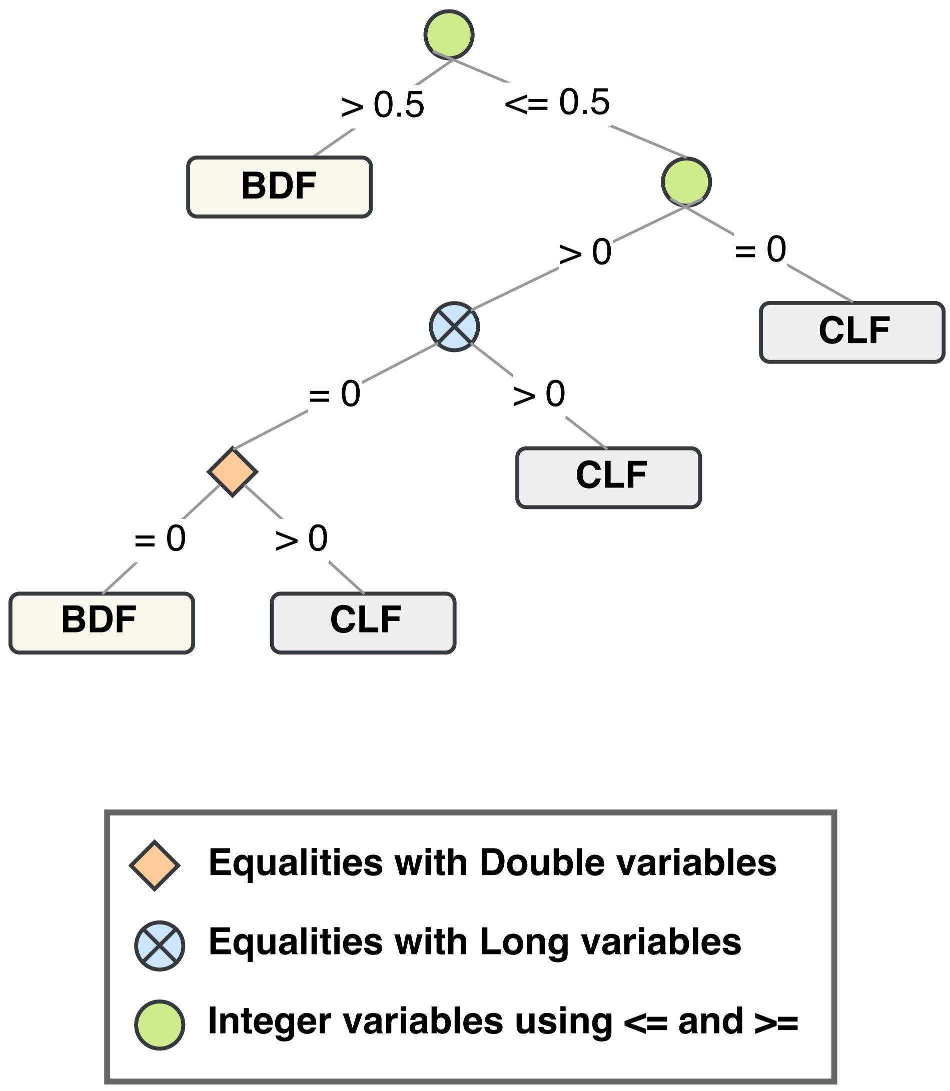

> 0.5
<= 0.5
BDF
> 0
= 0
CLF
= 0
> 0
CLF
= 0
> 0
BDF
CLF
Equalities with Double variables
Equalities with Long variables
Integer variables using <= and >=

Figure 7.2: Objective Function decision tree.

mean to counter this problem. It references the true positives to the arithmetic mean of predicted positives and real positives, being a constructed rate normalized to an idealized value [81].

In summary, the decision tree selects the most effective objective function with high accuracy. Hence, we conclude that the answer to the sixth research objective is as follows:

RO6: Using the features**Integer Variables with <= and >= comparator**, **Equalities with Double Variables**, **Double Variables with <= and >= comparator**, **Equalities with Long Variables**, **Inequalities with Long Variables** and **Long Variables with <= and >= comparator** we can accurately predict the most suitable ATSGT.

Analyzing Figure 7.2, we observe that BDF has issues when dealing with equalities using Long and Double variables. CLF is superior in most of the cases where the branch predicate present equalities with Double or Long. While BDF has superior performance when the number of Integer variables using <= or >= is higher than 0.5.

7.4 Threats to Validity

Threats to Internal Validity. This study focus on 3 SBST fitness functions: BDF, CLF and CFF. These selected techniques may affect the set of features that we have identified as important. However, as mentioned in the previous chapter, the META framework can continuously improve in accuracy with more techniques. When new ATSGTs, features, or dataset are included, the framework is rerun and the decision tree is updated, which may result in different footprints and predictions. It is expected that the accuracy of the results will improve with more ATSGTs, features, and dataset.

Threats to External Validity. Threats to external validity relate to the generalizability of the experimental results and are specifically related to the artificially generated CUTs if it correctly represents common software systems and the selected

ATSGTs. The artificial instances have been specifically generated to challenge the fitness functions in the portfolio and contain similarities to some private benchmark set [79]. While the META framework is not ATSGT dependent, the identified features may evolve when new ATSGTs are included. In the future, we intend to make the process of adding new techniques and new benchmark sets easier, such that the META framework can continuously evolve.

Chapter 8

Discussion

This chapter discusses the strengths, limitations, and future research directions for META Framework, drawing from the experimental results that were performed to assess its effectiveness. The META Framework has been developed to address the limitations of the current methods to assess the performance of ATSGTs. The greatest strength of this new methodology is the focus in generating insights about the ATSGT performance and not only to report which method is the best. META, backed by the No-free-lunch theorem, assumes that no ATSGT will always be the best in all possible scenarios. Therefore, it requires the CUTs to be diverse enough to challenge all the ATSGTs in the portfolio, highlighting its strengths and weaknesses. META provides as outcomes the software features that challenge the ATSGTs (Feature Selection), allowing the definition of the ATSGT footprints in 2d (Performance Visualization) and a decision tree (Performance Prediction). The downside of the technique mainly stems from the fact that it is time consuming due to the extensive analysis required for the selection of the software systems (CUTs) and software features. Usually, when META is unable to identify hardness, it means that either the CUTs are not diverse enough to challenge the ATSGTs or the features cannot capture the hardness of the problems, requiring additional analysis to extend both sets. Sometimes, it might happen that the CUTs that challenge a specific ATSGT in the portfolio are not realistic. Therefore, no benchmark will contain this type of CUTs. However, this can be addressed by having an online repository of features and benchmark sets.

8.1 Strengths of META Framework

The META framework has as its backbone the *No Free Lunch theorem* [111], which tells us that no single algorithm can outperform all other algorithms in all problem instances. In other words, if method A is superior over method B in solving a particular set of problems, then one may claim that there exist other untested problems where method B may outperform method A.

META provides an unbiased environment by claiming that all ATSGTs have strengths and weaknesses. If META did not detect any weaknesses in a specific ATSGT, the features and benchmark sets need to be re-evaluated. For example, imagine that Formula 1 is composed of two cars: Ferrari and Mercedes. Ferrari is better in straight lines while Mercedes is better in curves. If all the circuits are 100% straight lines, Ferrari will always win. On the other hand, if the track contains only curves, Mercedes will always win. META requires both curves and straight lines to assess the cars correctly. If one of them is missing, META will fail.

META requires a shift in the current mindset created by the current methodology that almost always requires new developed ATSGTs performance to be superior to the current state-of-art. As META will always highlight weakness along with the strengths of the tested techniques, there will be no 100% superior ATSGT. What we have now are the conditions that make ATSGT A better than ATSGT B and vice-versa.

8.1.1 Performance Assessment with Insights

One of the main qualities of META is the generation of insights as an outcome of the performance assessment. The features are used to characterize performance, highlighting both strengths and weaknesses of all ATSGTs in the portfolio. These insights can guide the development of new ATSGTs and also the improvement of the existing ones by describing its problems.

The need for insights leads to the development of new features that aims to highlight the hidden characteristics of the CUTs that are crucial to the understanding of ATSGT performance. This might help to accelerate the development of new ATSGTs ready to tackle these hard problems.

8.1.2 Performance Visualization

The generation of insights by using features to analyze ATSGT performance allowed the characterization of good and bad performance. However, an easy way to visualize these different performances was still missing. META presents an innovative way of describing performance in a 2-Dimensional space using Principal Component Analysis. The new method makes easier the visualization of the ATSGT footprints, providing an easy understanding of ATSGT performance by showing its variation according to the feature values.

META provides a 2-Dimensional scatter graph with dots representing the CUTs used in the experiment (CUT Space). The ATSGTs are described by different colors representing its footprint, i.e., area where its performance is superior. Auxiliary graphs describing the same CUT Space are used to represent each feature variation aiming to provide the reader with associations between ATSGT footprint and feature variations.

8.1.3 Performance Prediction

The definition of the association between ATSGT performance and features also enables META to predict performance. This is done by generating a decision tree using the J48 Algorithm. META uses both the features with more significance and the ATSGT footprints as inputs. The result is a set of rules that can be used to predict which ATSGT is likely to perform better according to the values of the features extracted from the CUTs. A decision tree is a handy tool for practitioners who would like to apply automated software testing in industry or academia. It provides them with the most effective method, given the specific features of the software system being tested.

Part IV

Conclusion

Chapter 9

Conclusion

Given a portfolio of ATSGTs, the META framework can be used to select the technique that is likely to be best for a relevant set of CUT instances. Using the framework, relationships between features of CUTs and the effectiveness of ATSGTs were uncovered. Three significant features that impact ATSGT effectiveness were identified. An ATSGT decision tree was built using these three significant features, which identified the correct technique in more than 88% of the cases.

The effectiveness of three objective functions widely used in Search-Based Software Testing was also investigated. Objective functions are crucial in guiding the search algorithm and finding high-quality test cases. META identified the CUT features that impact the effectiveness of different objective functions and also generated a decision tree based on the most significant features, which can be used for objective function selection. Using these significant features, we constructed a decision tree, which identified the correct technique in more than 76% of the cases.

Beyond the challenge of accurately predicting which ATSGT / Objective Function is likely to perform best for a given CUT, based on the relationship between CUT features and objective function performance, the META framework also explains why. Insights can be drawn from the most significant features that can lead to better guidance on improvements and in the development of new techniques.

References

[1] W. Afzal, R. Torkar, and R. Feldt. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, 51(6):957–976, 2009.

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

[3] A. Aleti, I. Moser, and L. Grunske. Analysing the fitness landscape of search-based software testing problems. *Automated Software Engineering*, pages 1–19, 2016.

[4] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36(6):742–762, 2010.

[5] F. E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[6] N. Alshahwan and M. Harman. Automated web application testing using search based software engineering. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 3–12. IEEE Computer Society, 2011.

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

[8] W. N. Anderson Jr and T. D. Morley. Eigenvalues of the laplacian of a graph. *Linear and multilinear algebra*, 18(2):141–145, 1985.

[9] A. Arcuri and L. Briand. Adaptive random testing: An illusion of effectiveness? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 265–275. ACM, 2011.

[10] A. Arcuri and L. 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.

[11] A. Arcuri and G. Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3):594–623, 2013.

[12] A. Arcuri, M. Z. Iqbal, and L. Briand. Formal analysis of the effectiveness and predictability of random testing. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 219–230. ACM, 2010.

[13] A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In *Genetic and Evolutionary Computation Conference*, volume 2, pages 1329–1336, 2002.

[14] B. Borah and D. Bhattacharyya. An improved sampling-based dbscan for large spatial databases. In *Intelligent Sensing and Information Processing, 2004. Proceedings of International Conference on*, pages 92–96. IEEE, 2004.

[15] L. C. Briand, J. Feng, and Y. Labiche. Using genetic algorithms and coupling measures to devise optimal integration test orders. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 43–50. ACM, 2002.

[16] E. J. Candès, X. Li, Y. Ma, and J. Wright. Robust principal component analysis? *Journal of the ACM (JACM)*, 58(3):11, 2011.

[17] T. Y. Chen, F.-C. Kuo, and H. Liu. Adaptive random testing based on distribution metrics. *Journal of Systems and Software*, 82(9):1419–1433, 2009.

[18] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

[19] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In *Advances in Computer Science - ASIAN 2004, Higher-Level Decision Making, 9th Asian Computing Science Conference*, volume 3321 of *Lecture Notes in Computer Science*, pages 320–329. Springer, 2004.

[20] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, 1994.

[21] S. C. R. Chitirala. *Comparing the effectiveness of automated test generation tools "EVOSUITE" and "Tpalus"*. PhD thesis, UNIVERSITY OF MINNESOTA, 2015.

[22] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Artoo: Adaptive random testing for object-oriented software. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 71–80, New York, NY, USA, 2008. ACM.

[23] I. Ciupa, A. Pretschner, A. Leitner, M. Oriol, and B. Meyer. On the predictability of random tests for object-oriented software. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 72–81. IEEE, 2008.

[24] T. E. Colanzi, W. K. G. Assunção, S. R. Vergilio, and A. Pozo. Integration test of classes and aspects with a multi-evolutionary and coupling-based approach. In *International Symposium on Search Based Software Engineering*, pages 188–203. Springer, 2011.

[25] W. J. Conover and W. J. Conover. Practical nonparametric statistics. 1980.

[26] C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[27] C. Csallner and Y. Smaragdakis. Check'n'crash: combining static checking and testing. In *Proceedings of the 27th international conference on Software engineering*, pages 422–431. ACM, 2005.

[28] C. Csallner, Y. Smaragdakis, and T. Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):8, 2008.

[29] L. Cseppento and Z. Micskei. Evaluating symbolic execution-based test tools. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–10. IEEE Computer Society, 2015.

[30] A. Dautovic, A. Gonzalez-Sanchez, R. Abreu, H.-G. Gross, and A. J. van Gemund. 2011 26th ieee/acm international conference on automated software engineering (ase).

[31] M. Dave and R. Agrawal. Search based techniques and mutation analysis in automatic test case generation: a survey. In *Advance Computing Conference (IACC), 2015 IEEE International*, pages 795–799. IEEE, 2015.

[32] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. In *International conference on parallel problem solving from nature*, pages 849–858. Springer, 2000.

[33] C. Del Grosso, G. Antoniol, M. Di Penta, P. Galinier, and E. Merlo. Improving network applications security: a new heuristic to generate stress testing data. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1037–1043. ACM, 2005.

[34] K. Derderian, R. M. Hierons, M. Harman, and Q. Guo. Automated unique input output sequence generation for conformance testing of fsms. *The Computer Journal*, 49(3):331–344, 2006.

[35] D. J. DeWitt and C. Levine. Not just correct, but correct and fast: A look at one of jim gray's contributions to database system performance. *SIGMOD Rec.*, 37(2):45–49, June 2008.

[36] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[37] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Trans. Software Eng.*, 10(4):438–444, 1984.

[38] E. Dustin, J. Rashka, and J. Paul. *Automated software testing: introduction, management, and performance.* Addison-Wesley Professional, 1999.

[39] D. Dvorak. Nasa study on flight software complexity. In *AIAA Infotech@ Aerospace Conference and AIAA Unmanned... Unlimited Conference*, page 1882, 2009.

[40] B. Efron. Bootstrap methods: another look at the jackknife. In *Breakthroughs in statistics*, pages 569–593. Springer, 1992.

[41] R. B. Evans and A. Savoia. Differential testing: a new approach to change detection. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, pages 549–552. ACM, 2007.

[42] T. Evgeniou and M. Pontil. Support vector machines: Theory and applications, 01 2001.

[43] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[44] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM, 2011.

[45] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 178–188. IEEE, 2012.

[46] G. Fraser and A. Arcuri. Whole test suite generation. *Software Engineering, IEEE Transactions on*, 39(2):276–291, 2013.

[47] G. Fraser and A. Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Trans. Softw. Eng. Methodol.*, 24(2):8:1–8:42, Dec. 2014.

[48] G. Fraser and A. Arcuri. Evosuite at the sbst 2016 tool competition. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*, SBST '16, pages 33–36, New York, NY, USA, 2016. ACM.

[49] B. Gupta, A. Rawat, A. Jain, A. Arora, and N. Dhami. Analysis of various decision tree algorithms for classification in data mining. *International Journal of Computer Applications*, 163(8):15–19, 2017.

[50] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[51] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. Java message service. *Sun Microsystems Inc., Santa Clara, CA*, 9, 2002.

[52] M. Harman, Y. Jia, and W. B. Langdon. Strong higher order mutation-based test data generation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 212–222. ACM, 2011.

[53] M. Harman, S. A. Mansouri, and Y. Zhang. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. *Department of Computer Science, King's College London, Tech. Rep. TR-09-03*, 2009.

[54] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010.

[55] J. N. Hooker. Testing heuristics: We have it all wrong. *J. Heuristics*, 1(1):33–42, 1995.

[56] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 297–306. IEEE Computer Society, 2008.

[57] I. Jolliffe. *Principal component analysis*. Springer, 2011.

[58] B. F. Jones, D. E. Eyres, and H.-H. Sthamer. A strategy for using genetic algorithms to automate branch and fault-based testing. *The Computer Journal*, 41(2):98–107, 1998.

[59] B. F. Jones, H.-H. Sthamer, and D. E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.

[60] B. Korel. Automated software test data generation. *Software Engineering, IEEE Transactions on*, 16(8):870–879, 1990.

[61] B. Korel. Dynamic method for software test data generation. *Software Testing, Verification and Reliability*, 2(4):203–213, 1992.

[62] H.-P. Kriegel, E. Schubert, and A. Zimek. The (black) art of runtime evaluation: Are we comparing algorithms or implementations? *Knowl. Inf. Syst.*, 52(2):341–378, Aug. 2017.

[63] F. Lammermann, A. Baresel, and J. Wegener. Evaluating evolutionary testability for structure-oriented testing with software measurements. *Applied Soft Computing*, 8(2):1018 – 1028, 2008.

[64] A. Leitner, I. Ciupa, B. Meyer, and M. Howard. Reconciling manual and automated testing: the autotest experience. In *Proceedings of the 40th Hawaii International Conference on System Sciences - 2007, Software Technology*, January 3-6 2007.

[65] T.-S. Lim, W.-Y. Loh, and Y.-S. Shih. A comparison of prediction accuracy, complexity, and training time of thirty-three old and new classification algorithms. *Machine learning*, 40(3):203–228, 2000.

[66] M. Maratmu. Testgen4j. developer.spikesource.com/wiki/index.php/Projects:testgen4j, last access December 2008.

[67] T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, July 1976.

[68] P. McMinn. Search-based software test data generation: A survey. *Software Testing Verification and Reliability*, 14(2):105–156, 2004.

[69] P. McMinn, M. Harman, K. Lakhotia, Y. Hassoun, and J. Wegener. Input domain reduction through irrelevant variable removal and its effect on local, global, and hybrid search-based structural test data generation. *IEEE Transactions on Software Engineering*, 38(2):453–477, 2012.

[70] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *IEEE transactions on software engineering*, 27(12):1085–1110, 2001.

[71] W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223, 1976.

[72] M. A. MuÃśoz Acosta, L. Villanova, D. Baatar, and K. Smith-Miles. Instance spaces for machine learning classification. *Machine Learning*, 107:109–147, 01 2018.

[73] M. Oriol. Random testing: Evaluation of a law describing the number of faults found. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 201–210, April 2012.

[74] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *European Conference on Object-Oriented Programming*, pages 504–527. Springer, 2005.

[75] A. Panichella, F. Kifetew, and P. Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2017.

[76] A. Panichella, F. M. Kifetew, and P. 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, April 2015.

[77] A. Panichella and U. R. Molina. Java unit testing tool competition - fifth round. In *2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*, pages 32–38, May 2017.

[78] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *Software Testing Verification and Reliability*, 9(4):263–282, 1999.

[79] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, 2013.

[80] R. Poli, W. B. Langdon, and N. F. McPhee. *A Field Guide to Genetic Programming.* lulu.com, 2008.

[81] D. M. W. Powers. Evaluation: From precision, recall and f-measure to roc., informedness, markedness & correlation. *Journal of Machine Learning Technologies*, 2(1):37–63, 2011.

[82] R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[83] J. C. B. Ribeiro, M. A. Zenha-Rela, and F. F. De Vega. Enabling object reuse on genetic programming-based approaches to object-oriented evolutionary testing. In *European Conference on Genetic Programming*, pages 220–231. Springer, 2010.

[84] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering*, 22(2):852–893, Apr 2017.

[85] M. Roper. Computer aided software testing using genetic algorithms, 1997.

[86] D. A. Schult and P. Swart. Exploring network structure, dynamics, and function using networkx. In *Proceedings of the 7th Python in Science Conferences (SciPy 2008)*, volume 2008, pages 11–16, 2008.

[87] S. Shamshiri, J. M. Rojas, G. Fraser, and P. McMinn. Random or genetic algorithm search for object-oriented test suite generation? In *Proceedings of the 2015*

Annual Conference on Genetic and Evolutionary Computation, pages 1367–1374. ACM, 2015.

[88] J. Sim and C. C. Wright. The kappa statistic in reliability studies: use, interpretation, and sample size requirements. *Physical therapy*, 85 3:257–68, 2005.

[89] S. E. Sim, S. Easterbrook, and R. C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 74–83, Washington, DC, USA, 2003. IEEE Computer Society.

[90] K. A. Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Comput. Surv.*, 41(1):6:1–6:25, Jan. 2009.

[91] W. M. Spears. Adapting crossover in evolutionary algorithms. In *Evolutionary programming*, pages 367–384, 1995.

[92] D. Spinellis and M. Jureczko. Ckjm extended - an extended version of tool for calculating chidamber and kemerer java metrics. Online, 2011. http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/, last change: May 06, 2011 5:05 pm.

[93] P. R. Srivastava and T.-h. Kim. Application of genetic algorithm in software testing. *International Journal of software Engineering and its Applications*, 3(4):87–96, 2009.

[94] K. Taneja and T. Xie. Diffgen: Automated regression unit-test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 407–410. IEEE Computer Society, 2008.

[95] S. Thummalapenta, J. De Halleux, N. Tillmann, and S. Wadsworth. Dygen: automatic generation of high-coverage tests via mining gigabytes of dynamic traces. In *International Conference on Tests and Proofs*, pages 77–93. Springer, 2010.

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

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

[98] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on*, pages 285–288. IEEE, 1998.

[99] N. J. Tracey. *A search-based automated test-data generation framework for safety-critical software.* PhD thesis, Citeseer, 2000.

[100] M. Tyborowsky. Jub (junit test case builder). Online, 2002. http://jub.sourceforge.net/, last access December 2008.

[101] A. Vargha and H. 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.

[102] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Timeaware test suite prioritization. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 1–12. ACM, 2006.

[103] S. Wang and J. Offutt. Comparison of unit-level automated test generation tools. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, pages 210–219. IEEE, 2009.

[104] A. Watkins and E. M. Hufnagel. Evolutionary test data generation: a comparison of fitness functions. *Software: Practice and Experience*, 36(1):95–116, 2006.

[105] A. L. Watkins. The automatic generation of test data using genetic algorithms. In *Software Quality Conference*, volume 2, pages 300–309, 1995.

[106] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[107] J. Wegener and O. 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.

[108] J. Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, 15(3):275–298, 1998.

[109] Y. Wei, B. Meyer, and M. Oriol. Is branch coverage a good measure of testing effectiveness? In *Empirical Software Engineering and Verification*, pages 194–212. Springer, 2010.

[110] L. J. White and E. I. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, (3):247–257, 1980.

[111] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[112] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios. Application of genetic algorithms to software testing. In *International Conference on Software Engineering and its Applications*, pages 625–636, 1992.

[113] Q. Yang, J. J. Li, and D. M. Weiss. A survey of coverage-based testing tools. *The Computer Journal*, 52(5):589–597, 2009.

[114] S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 201–212. ACM, 2009.

[115] Y. Zhan and J. A. Clark. Search-based mutation testing for simulink models. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1061–1068. ACM, 2005.

[116] Y. Zhan and J. A. Clark. The state problem for test generation in simulink. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1941–1948. ACM, 2006.

[117] S. Zhang, D. Saff, Y. Bu, , and M. D. Ernst. Combined static and dynamic automated test generation. In *Proc. 11th International Symposium on Software Testing and Analysis (ISSTA 2011)*, 2011.

[118] J. Zhong, X. Hu, J. Zhang, and M. Gu. Comparison of performance between different selection strategies on simple genetic algorithms. In *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)*, volume 2, pages 1115–1121. IEEE, 2005.

Chapter 10

Appendix

Table 10.1: Code-based features.

	Feature ▷ Definition
1	**Number of Static Methods** ▷ The total number of methods that are static in a class.
2	**Number of Public Static Methods** ▷ The total number of methods that are public and static at the same time.
3	**Number of Protected Static Methods** ▷ The total number of methods that are protected and static at the same time.
4	**Number of Private Static Methods** ▷ The total number of methods that are private and static at the same time.
5	**Number of Default Static Methods** ▷ The total number of methods that are static and have default access at the same time.
6	**Number of Methods** ▷ The total number of methods in a class.
7	**Number of Public Methods** ▷ The total number of methods with public access in a class.

8 **Number of Protected Methods** ▷ The total number of methods with protected access.

9 **Number of Private Methods** ▷ The total number of methods with private access.

10 **Number of Default Methods** ▷ The total number of methods with default access.

11 **Depth of Inheritance Tree** ▷ The depth of the inheritance tree (DIT) metric provides for each class a measure of the inheritance levels from the object hierarchy top. In Java where all classes inherit Object the minimum value of DIT is 1 [20].

12 **Number of Children** ▷ A class's number of children (NOC) metric measures the number of immediate descendants of the class [20].

13 **Coupling between object classes** ▷ The coupling between object classes (CBO) metric represents the number of classes coupled to a given class (efferent couplings). This coupling can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions [20].

14 **Response for a Class** ▷ The metric called the response for a class (RFC) measures the number of different methods that can be executed when an object of that class receives a message (when a method is invoked for that object) [20].

15 **Lack of cohesion in methods** ▷ A class's lack of cohesion in methods (LCOM) metric counts the sets of methods in a class that are not related through the sharing of some of the class's fields. The original definition of this metric, which is the one used in this work, considers all pairs of a class's methods. In some of these pairs both methods access at least one common field of the class, while in other pairs the two methods to not share any common field accesses. The lack of cohesion in methods is then calculated by subtracting from the number of method pairs that don't share a field access to the number of method pairs that do [20].

16 **Afferent couplings** ▷ A class's afferent couplings (CA) is a measure of how many other classes use the specific class. CA is calculated using the same definition as that used for calculating CBO.

17 **Efferent couplings** ▷ A class's efferent couplings (CE) is a measure of how many other classes is used by the specific class. Coupling has the same definition in context of CE as that used for calculating CBO.

18 **Lack of cohesion in methods 3** ▷ LCOM3 varies between 0 and 2. m is the number of procedures (methods) in class, a is the number of variables (attributes) in class, $\mu(A)$ is the number of methods that access a variable (attribute). The constructors and static initializations are taking into accounts as separately methods.
$LCOM3 = \frac{\left(\frac{1}{2}\sum_{j=1}^{a}\mu\left(A_j\right)\right)-m}{1-m}$

19 **Lines of Code** ▷ The lines are counted from Java binary code and it is the sum of number of fields, number of methods and number of instructions in every method of given class.

20 **Data Access Metric** ▷ This metric is the ratio of the number of private and protected attributes to the total number of attributes declared in the class. A high value for DAM is desired. (Range 0 to 1).

21 **Aggregation** ▷ This metric measures the extent of the part-whole relationship, realized by using attributes. The metric is a count of the number of data declarations (class fields) whose types are user defined classes.

22 **Functional Abstraction** ▷ This metric is the ratio of the number of methods inherited by a class to the total number of methods accessible by member methods of the class. The constructors and the java.lang.Object (as parent) are ignored. (Range 0 to 1).

23	**Cohesion Among Methods of Class** ▷ This metric computes the relatedness among methods of a class based upon the parameter list of the methods. The metric is computed using the summation of number of different types of method parameters in every method divided by a multiplication of number of different method parameter types in whole class and number of methods. A metric value close to 1.0 is preferred. (Range 0 to 1).
24	**Inheritance Coupling** ▷ This metric (IC) provides the number of parent classes to which a given class is coupled. A class is coupled to its parent class if one of the following conditions is satisfied: (a) One of its inherited methods uses a variable (or data member) that is defined in a new /redefined method; (b) One of its inherited methods calls a redefined method; (c) One of its inherited methods is called by a redefined method and uses a parameter that is defined in the redefined method.
25	**Coupling Between Methods** ▷ The metric measure the total number of new/redefined methods to which all the inherited methods are coupled.

Table 10.2: Predicate code-based features.

	Feature ▷ Definition
1	**Comparators** ▷ Number of comparators inside predicates in a class: equalities, inequalities and ranges (higher and less than)
2	**Equalities** ▷ Number of equalities inside predicates in a class
3	**Inequalities** ▷ Number of inequalities inside predicates in a class
4	**Ranges** ▷ Number of ranges inside predicates in a class (e.g., $1<x)
5	**Variables** ▷ Number of variables inside predicates in a class
6	**Boolean Variables** ▷ Number of boolean variables inside predicates in a class
7	**Integer Variables** ▷ Number of integer variables inside predicates in a class
8	**Long Variables** ▷ Number of long variablesinside predicates in a class
9	**Double Variables** ▷ Number of double variables inside predicates in a class
10	**Equalities with Boolean** ▷ Number of equalities with boolean variables inside predicates in a class
11	**Inequalities with Boolean** ▷ Number of inequalities with boolean variables inside predicates in a class
12	**Equalities with Integer** ▷ Number of equalities with integer variables inside predicates in a class
13	**Inequalities with Integer** ▷ Number of inequalities with integer variables inside predicates in a class
14	**Ranges with Integer** ▷ Number of ranges with integer variables inside predicates in a class
15	**Equalities with Long** ▷ Number of equalities with long variables inside predicates in a class

16 **Inequalities with Long** ▷ Number of inequalities with long variables inside predicates in a class

17 **Ranges with Integer** ▷ Number of ranges with long variables inside predicates in a class

18 **Equalities with Double** ▷ Number of equalities with double variables inside predicates in a class

19 **Inequalities with Double** ▷ Number of inequalities with double variables inside predicates in a class

20 **Ranges** with Double ▷ Number of ranges with double variables inside predicates in a class

Table 10.3: CFG-based features.

	Feature ▷ Definition
1	**CFGs count** ▷ The number of control flow graphs
2	**Number of Vertices in a CFG** ▷ The number of vertices in the control flow graph reported as average, standard deviation, sum, maximum, and minimum values
3	**Number of Edges in a CFG** ▷ The number of edges in the control flow graph reported as average, standard deviation, sum, maximum, and minimum values.
4	**Radius** ▷ The minimum eccentricity of the CFG. Eccentricity is the maximum graph distance between any two vertices.
5	**Diameter** ▷ The maximum eccentricity.
6	**Center Size** ▷ The set of nodes with eccentricity equal to radius.
7	**Periphery Size** ▷ The set of nodes with eccentricity equal to the diameter.
8	**Average Shortest Path Length** ▷ This measure is equal to $a = \sum_{s,t \in V} \frac{d(s,t)}{n(n-1)}$ where V is the set of nodes, $d(s,t)$ is the shortest path from s to t, and n is the number of nodes in the CFG.
9	**Largest Eigenvalue of the Laplacian** ▷ The largest eigenvalue of the Laplacian of the CFGs [8]
10	**Second Largest Eigenvalue of the Laplacian** ▷ The second largest eigenvalue of the Laplacian of the CFGs [8].
11	**Algebraic Connectivity** ▷ The second smallest eigenvalue of the Laplacian [8]. This reflects how well connected a graph is.
12	**Smallest Non-Zero Eigenvalue of the Laplacian** ▷ The smallest non zero eigenvalue of the Laplacian of the graph [8].
13	**Eigenvalue Gap of the Laplacian** ▷ The difference between the largest and the second largest eigenvalue of the Laplacian of the graph.

14 **Largest Eignevalue Adjacency** ▷ The largest eigenvalue of the adjacency matrix of the graph. The elements of the adjacency matrix indicate whether pairs of vertices are adjacent or not in the graph.

15 **Second Largest Eignevalue Adjacency** ▷ The second largest eigenvalue of the adjacency matrix of the graph.

16 **Smallest Eigenvalue Adjacency** ▷ The smallest eigenvalue of the adjacency matrix of CFGs.

17 **Second Smallest Eigenvalue Adjacency** ▷ The second smallest eigenvalue of the adjacency matrix CFG.

18 **Eigenvalue Gap Adjacency** ▷ The difference between the largest and the second largest eigenvalue of the adjacency matrix of CFG.

19 **Standard Eigenvalue Deviation Adjacency** ▷ The standard deviation of the eigenvalues of the adjacency matrix of CFGs.

20 **Energy** ▷ The mean of the eigenvalues of the adjacency matrix of the CFGs.

21 **Node Degree** ▷ The number of edges adjacent to that node.

22 **Minimum Node Degree** ▷ The lowest node degree of the graph.

23 **Maximum Node Degree** ▷ The highest node degree of the graph.

24 **Density of the CFG** ▷ This measures how many edges are in the graph compared to the maximum possible number of edges, reported as average, standard deviation, sum, maximum, and minimum values.

25 **Edge Connectivity** ▷ It is equal to the minimum number of edges that must be removed to disconnect the graph, reported as average, standard deviation, sum, maximum, and minimum values.

26 **Node Connectivity** ▷ The minimum number of nodes that must be removed to disconnect the graph, reported as average, standard deviation, sum, maximum, and minimum values.

27	**Clustering Coefficient** ▷ The local clustering of each node in a graph is the fraction of triangles that actually exist over all possible triangles in its neighbourhood. The average clustering coefficient of the CFG is the mean of local clusterings.
28	**Transitivity** ▷ The fraction of possible triangles present in the CFG, reported as average, standard deviation, sum, maximum, and minimum values over all CFGs.
29	**McCabe's cyclomatic complexity** ▷ It is computed as $CC = E - V + P$, where P is the number of connected components, E are edges in the CFG and V are nodes.
30	**Class Complexity** ▷ The percentage of methods in the class with CC higher than 10. Class Complexity is a new proposed feature that aims to classify a CUT according to the number of complex CFGs.

Table 10.4: EvoSuite default settings.

Name	Type	Default
algorithm	Algorithm	MONOTONIC_GA
archive_type	ArchiveType	COVERAGE
array_limit	int	1000000
assertion_minimization_fallback	double	0.5
assertion_minimization_fallback_time	double	0.6666666667
assertion_strategy	AssertionStrategy	MUTATION
assertion_timeout	int	60
assertions	boolean	TRUE
bloat_factor	int	2
branch_comparison_types	boolean	FALSE
branch_eval	boolean	FALSE
branch_statement	boolean	FALSE
break_on_exception	boolean	TRUE
breeder_truncation	double	0.5
call_probability	double	0
carve_object_pool	boolean	FALSE
carving_timeout	int	120
catch_undeclared_exceptions	boolean	TRUE
check_best_length	boolean	TRUE
check_contracts	boolean	FALSE
check_contracts_end	boolean	FALSE
check_max_length	boolean	TRUE
check_parents_length	boolean	FALSE
chop_carved_exceptions	boolean	TRUE
chop_max_length	boolean	TRUE
chromosome_length	int	40
classpath	String[]	[]
client_on_thread	boolean	FALSE
cluster_recursion	int	10
concolic_mutation	double	0
concolic_timeout	int	15000
connection_data	String	connection.xml
consider_main_methods	boolean	TRUE
constraint_solution_attempts	int	3
coverage	boolean	TRUE
coverage_matrix	boolean	FALSE

coverage_matrix_filename	String	matrix
covered_goals_file	String	evosuite-report/covered.goals
cpu_timeout	boolean	FALSE
criterion	Criterion[]	BRANCH
crossover_function	CrossoverFunction	SINGLEPOINTRELATIVE
crossover_rate	double	0.75
ctg_bests_folder	String	best-tests
ctg_cores	int	1
ctg_delete_old_tmp_folders	boolean	TRUE
ctg_dir	String	.evosuite
ctg_memory	int	1000
ctg_min_time_per_job	int	1
ctg_project_info	String	project_info.xml
ctg_schedule	AvailableSchedule	BUDGET
ctg_seeds_dir_name	String	seeds
ctg_seeds_ext	String	seed
ctg_time	int	3
ctg_tmp_logs_dir_name	String	logs
ctg_tmp_pools_dir_name	String	pools
ctg_tmp_reports_dir_name	String	reports
ctg_tmp_tests_dir_name	String	tests
debug	boolean	FALSE
decomposition_threshold	int	500
defuse_aliases	boolean	TRUE
defuse_debug_mode	boolean	FALSE
double_precision	double	0.01
dse_constant_probability	double	0.5
dse_constraint_length	int	100000
dse_constraint_solver_timeout_millis	long	1000
dse_keep_all_tests	boolean	FALSE
dse_negate_all_conditions	boolean	TRUE
dse_probability	double	0.5
dse_rank_branch_conditions	boolean	TRUE
dse_solver	SolverType	EVOSUITE_SOLVER
dse_variable_resets	int	2
dynamic_limit	boolean	FALSE
dynamic_pool	double	0.5

dynamic_pool_size	int	50
dynamic_seeding	boolean	TRUE
eclipse_plugin	boolean	FALSE
elite	int	1
enable_alternative_fitness_calculation	boolean	FALSE
enable_alternative_suite_fitness	boolean	FALSE
enable_asserts_for_evosuite	boolean	FALSE
enable_asserts_for_sut	boolean	TRUE
enable_secondary_objective_after	int	0
enable_secondary_starvation	boolean	FALSE
epsilon	double	0.001
epson	double	0.01
error_branches	boolean	FALSE
evosuite_use_uispec	boolean	FALSE
exception_branches	boolean	FALSE
exclude_ibranches_cut	boolean	FALSE
exploitation_starts_at_percent	double	0.5
extra_timeout	int	60
filter_assertions	boolean	FALSE
filter_sandbox_tests	boolean	FALSE
float_precision	float	0.01
functional_mocking_input_limit	int	5
functional_mocking_percent	double	0.5
global_timeout	int	120
group_id	String	none
handle_servlets	boolean	FALSE
handle_static_fields	boolean	TRUE
headless_chicken_test	boolean	FALSE
headless_mode	boolean	TRUE
hierarchy_data	String	hierarchy.xml
honour_data_annotations	boolean	TRUE
ignore_missing_statistics	boolean	FALSE
ignore_threads	String[]	[]
initial_kinetic_energy	double	1000
initialization_timeout	int	120
inline	boolean	TRUE
insertion_score_object	int	1

insertion_score_parameter	int	1
insertion_score_uut	int	1
insertion_uut	double	0.5
insertion_uut	double	0.4
insertion_uut	double	0.1
instrument_context	boolean	FALSE
instrument_libraries	boolean	FALSE
instrument_method_calls	boolean	FALSE
instrument_parent	boolean	FALSE
instrumentation_skip_debug	boolean	FALSE
is_running_a_system_test	boolean	FALSE
jee	boolean	TRUE
jmc	boolean	FALSE
junit_check	boolean	TRUE
junit_check_on_separate_process	boolean	FALSE
junit_check_timeout	int	60
junit_failed_suffix	String	_Failed_ESTest
junit_strict	boolean	FALSE
junit_suffix	String	_ESTest
junit_tests	boolean	TRUE
keep_regression_archive	boolean	FALSE
kincompensation	double	1
kinetic_energy_loss_rate	double	0.2
lambda	int	1
lm_iterations	int	1000
lm_mutation_type	MutationType	EVOSUITE
lm_src	String	ukwac_char_lm
lm_strings	boolean	FALSE
local_search_adaptation_rate	double	2
local_search_arrays	boolean	TRUE
local_search_budget	long	5
local_search_budget_type	LocalSearchBudgetType	TIME
local_search_dse	DSEType	TEST
local_search_ensure_double_execution	boolean	TRUE
local_search_expand_tests	boolean	TRUE
local_search_primitives	boolean	TRUE
local_search_probability	double	1

local_search_probes	int	10
local_search_rate	int	-1
local_search_references	boolean	TRUE
local_search_restore_coverage	boolean	FALSE
local_search_selective	boolean	FALSE
local_search_selective_primitives	boolean	FALSE
local_search_strings	boolean	TRUE
log_goals	boolean	FALSE
log_timeout	boolean	FALSE
make_accessible	boolean	FALSE
max_array	int	10
max_attempts	int	1000
max_coverage_depth	int	-1
max_delta	int	20
max_generic_depth	int	3
max_initial_tests	int	10
max_int	int	2048
max_length	int	0
max_length_test_case	int	2500
max_loop_iterations	long	10000
max_mutants	int	100
max_mutants_per_class	int	1000
max_mutants_per_method	int	700
max_mutants_per_test	int	100
max_num_fitness_evaluations_before_giving_up	int	10
max_num_mutations_before_giving_up	int	10
max_recursion	int	10
max_replace_mutants	int	100
max_size	int	100
max_stalled_threads	int	10
max_started_threads	int	100
max_string	int	1000
min_free_mem	int	50000000
min_initial_tests	int	1
minimization_timeout	int	60
minimize	boolean	TRUE
minimize_old	boolean	FALSE

minimize_second_pass	boolean	TRUE
minimize_skip_coincidental	boolean	TRUE
minimize_sort	boolean	TRUE
minimize_strings	boolean	TRUE
minimize_values	boolean	FALSE
mock_if_no_generator	boolean	TRUE
molecular_collision_rate	double	0.2
mu	int	1
mutation_generations	int	10
mutation_probability_distribution	MutationProbabilityDistribution	UNIFORM
mutation_rate	double	0.75
mutation_timeouts	int	3
neighborhood_model	CGA_Models	LINEAR_FIVE
new_object_selection	boolean	TRUE
new_statistics	boolean	TRUE
no_runtime_dependency	boolean	FALSE
null_probability	double	0.1
num_random_tests	int	20
num_tests	int	2
number_of_mutations	int	1
number_of_tests_per_target	int	10
object_reuse_probability	double	0.9
output_granularity	OutputGranularity	MERGED
p_change_parameter	double	0.1
p_functional_mocking	double	0
p_object_pool	double	0.3
p_random_test_or_from_archive	double	0.5
p_reflection_on_private	double	0
p_special_type_call	double	0.05
p_statement_insertion	double	0.5
p_test_change	double	0.3333333333
p_test_delete	double	0.3333333333
p_test_insert	double	0.3333333333
p_test_insertion	double	0.1
parent_check	boolean	TRUE
plot	boolean	FALSE
population	int	50

population_limit	PopulationLimit	INDIVIDUALS
port	int	1044
primitive_pool	double	0.5
primitive_reuse_probability	double	0.5
print_covered_goals	boolean	FALSE
print_current_goals	boolean	FALSE
print_goals	boolean	FALSE
print_missed_goals	boolean	FALSE
print_to_system	boolean	FALSE
process_communication_port	int	-1
pure_equals	boolean	FALSE
pure_inspectors	boolean	TRUE
random_perturbation	double	0.2
random_tests	int	0
randomize_difficulty	boolean	TRUE
rank_bias	double	1.7
ranking_type	RankingType	PREFERENCE_SORTING
recycle_chromosomes	boolean	TRUE
reflection_start_percent	double	0.8
regression_analysis_branchdistance	int	0
regression_analysis_combinations	int	0
regression_analysis_objectdistance	int	0
regression_analyze	boolean	FALSE
regression_branch_distance	boolean	FALSE
regression_different_branches	boolean	FALSE
regression_disable_special_assertions	boolean	FALSE
regression_diversity	boolean	FALSE
regression_fitness	RegressionMeasure	RANDOM
regression_random_strategy	int	3
regression_skip_different_cfg	boolean	FALSE
regression_skip_similar	boolean	FALSE
regression_statistics	boolean	FALSE
remote_testing	boolean	FALSE
replace_calls	boolean	TRUE
replace_gui	boolean	FALSE
replace_system_in	boolean	TRUE
replacement_function	TheReplacementFunction	DEFAULT

report_dir	String	evosuite-report
reset_all_classes_during_assertion_generation	boolean	TRUE
reset_all_classes_during_test_generation	boolean	FALSE
reset_standard_streams	boolean	FALSE
reset_static_field_gets	boolean	FALSE
reset_static_fields	boolean	TRUE
reset_static_final_fields	boolean	TRUE
restrict_pool	boolean	FALSE
reuse_budget	boolean	TRUE
reuse_leftover_time	boolean	FALSE
sandbox	boolean	TRUE
sandbox_mode	SandboxMode	RECOMMENDED
save_all_data	boolean	TRUE
scaffolding_suffix	String	scaffolding
search_budget	long	60
secondary_objectives	SecondaryObjective[]	[TOTAL_LENGTH]
seed_clone	double	0.2
seed_dir	String	evosuite-seeds
seed_mutations	int	3
seed_probability	double	0.1
seed_types	boolean	TRUE
selection_function	SelectionFunction	RANK
serialize_ga	boolean	FALSE
serialize_regression_test_suite	boolean	FALSE
serialize_result	boolean	FALSE
show_progress	boolean	TRUE
shuffle_goals	boolean	TRUE
shutdown_hook	boolean	TRUE
shutdown_timeout	int	1000
skip_covered	boolean	TRUE
sort_calls	boolean	FALSE
sort_objects	boolean	FALSE
sourcepath	String[]	[]
starvation_after_generation	int	500
starve_by_fitness	boolean	TRUE
statistics_backend	StatisticsBackend	CSV
stop_zero	boolean	TRUE

stopping_condition	StoppingCondition	MAXTIME
stopping_port	int	-1
strategy	Strategy	EVOSUITE
string_length	int	20
string_replacement	boolean	TRUE
synthesis_threshold	int	10
test_archive	boolean	TRUE
test_carving	boolean	FALSE
test_comments	boolean	FALSE
test_dir	String	evosuite-tests
test_excludes	String	test.excludes
test_factory	TestFactory	ARCHIVE
test_format	OutputFormat	JUNIT4
test_includes	String	test.includes
test_naming_strategy	TestNamingStrategy	NUMBERED
test_scaffolding	boolean	TRUE
testability_transformation	boolean	FALSE
timeline_interpolation	boolean	TRUE
timeline_interval	long	60000
timeout	int	3000
timeout_reset	int	2000
tournament_size	int	10
track_boolean_branches	boolean	FALSE
track_covered_gradient_branches	boolean	FALSE
track_diversity	boolean	FALSE
tt_scope	TransformationScope	ALL
usage_rate	double	0.5
use_deprecated	boolean	FALSE
use_existing_coverage	boolean	FALSE
use_separate_classloader	boolean	TRUE
validate_runtime_variables	boolean	TRUE
variable_pool	boolean	FALSE
virtual_fs	boolean	TRUE
virtual_net	boolean	TRUE
write_all_goals_file	boolean	FALSE
write_cfg	boolean	FALSE
write_covered_goals_file	boolean	FALSE

write_individuals	boolean	FALSE
write_junit_timeout	int	60

Table 10.5: SF110 classes with Ciclomatic Complexity equal to or higher than five

Project	Class
squirrel-sql	net.sourceforge.squirrel_sql.client.session.EditableSqlCheck
squirrel-sql	net.sourceforge.squirrel_sql.client.session.SQLEntryPanelUtil
squirrel-sql	net.sourceforge.squirrel_sql.client.session.mainpanel.overview.datascale.IndexedColumnFactory
squirrel-sql	net.sourceforge.squirrel_sql.client.session.parser.kernel.ErrorStream
squirrel-sql	net.sourceforge.squirrel_sql.client.session.parser.kernel.Scanner
squirrel-sql	net.sourceforge.squirrel_sql.client.update.gui.installer.event.InstallStatusListenerImpl
squirrel-sql	net.sourceforge.squirrel_sql.client.util.codereformat.CodeReformator
squirrel-sql	net.sourceforge.squirrel_sql.client.util.codereformat.CodeReformatorKernel
squirrel-sql	net.sourceforge.squirrel_sql.fw.datasetviewer.cellcomponent.StringFieldKeyTextHandler
squirrel-sql	net.sourceforge.squirrel_sql.fw.gui.CascadeInternalFramePositioner
squirrel-sql	net.sourceforge.squirrel_sql.fw.gui.action.TableCopyCommand
squirrel-sql	net.sourceforge.squirrel_sql.fw.gui.action.TableCopyHtmlCommand
squirrel-sql	net.sourceforge.squirrel_sql.fw.gui.action.TableCopyInStatementCommand
squirrel-sql	net.sourceforge.squirrel_sql.fw.gui.action.TableCopyInsertStatementCommand
squirrel-sql	net.sourceforge.squirrel_sql.fw.gui.action.TableCopySqlPartCommandBase
squirrel-sql	net.sourceforge.squirrel_sql.fw.gui.action.TableCopyWhereStatementCommand
squirrel-sql	net.sourceforge.squirrel_sql.fw.gui.action.exportData.JTableExportData
sweethome3d	com.eteks.sweethome3d.j3d.Ground3D
sweethome3d	com.eteks.sweethome3d.j3d.PhotoRenderer
sweethome3d	com.eteks.sweethome3d.j3d.Room3D
sweethome3d	com.eteks.sweethome3d.j3d.Wall3D
sweethome3d	com.eteks.sweethome3d.swing.AutoCommitSpinner
vuze	com.aelitis.azureus.core.dht.netcoords.vivaldi.ver1.impl.tests.VivaldiTest
vuze	com.aelitis.azureus.core.metasearch.impl.DateParserClassic
vuze	com.aelitis.azureus.core.metasearch.impl.DateParserRegex
vuze	com.aelitis.azureus.core.networkmanager.impl.tcp.VirtualChannelSelectorImpl
vuze	com.aelitis.azureus.core.peermanager.piecepicker.impl.PiecePickerImpl
vuze	com.aelitis.azureus.core.peermanager.unchoker.DownloadingUnchoker
vuze	com.aelitis.azureus.core.peermanager.unchoker.UnchokerUtil
vuze	com.aelitis.azureus.core.peermanager.utils.BTPeerIDByteDecoderUtils
vuze	com.aelitis.azureus.core.util.GeneralUtils
vuze	com.aelitis.azureus.plugins.net.netstatus.swt.NetStatusPluginTester
vuze	com.aelitis.azureus.plugins.startstoprules.defaultplugin.DefaultRankCalculator
vuze	com.aelitis.azureus.plugins.startstoprules.defaultplugin.SeedingRankColumnListener
vuze	com.aelitis.azureus.plugins.startstoprules.defaultplugin.StartStopRulesDefaultPlugin
vuze	com.aelitis.azureus.ui.swt.UIConfigDefaultsSWTv3
vuze	com.aelitis.azureus.ui.swt.browser.listener.MetaSearchListener

Project	Class
	SF110 classes with Ciclomatic Complexity equal to or higher than five
vuze	com.aelitis.azureus.ui.swt.browser.listener.VuzeListener
vuze	com.aelitis.azureus.ui.swt.columns.torrent.ColumnProgressETA
vuze	com.aelitis.azureus.ui.swt.columns.utils.TableColumnCreatorV3
vuze	com.aelitis.azureus.ui.swt.columns.vuzeactivity.ColumnActivityActions
vuze	com.aelitis.azureus.ui.swt.skin.SWTBGImagePainter
vuze	com.aelitis.azureus.ui.swt.skin.SWTSkinImageChanger
vuze	com.aelitis.azureus.ui.swt.skin.SWTSkinUtils
vuze	com.aelitis.azureus.ui.swt.views.skin.sidebar.SideBarToolTips
vuze	com.aelitis.azureus.util.DataSourceUtils
vuze	org.bouncycastle.asn1.util.ASN1Dump
vuze	org.bouncycastle.asn1.x509.X509DefaultEntryConverter
vuze	org.bouncycastle.util.Strings
vuze	org.gudy.azureus2.cl.Main
vuze	org.gudy.azureus2.core3.config.impl.ConfigurationChecker
vuze	org.gudy.azureus2.core3.disk.impl.DiskManagerUtil
vuze	org.gudy.azureus2.core3.disk.impl.resume.RDResumeHandler
vuze	org.gudy.azureus2.core3.html.HTMLUtils
vuze	org.gudy.azureus2.core3.ipchecker.natchecker.NatChecker
vuze	org.gudy.azureus2.core3.ipfilter.impl.IpFilterAutoLoaderImpl
vuze	org.gudy.azureus2.core3.tracker.client.impl.bt.TrackerStatus
vuze	org.gudy.azureus2.core3.tracker.host.impl.TRHostConfigImpl
vuze	org.gudy.azureus2.core3.tracker.server.impl.TRTrackerServerTorrentImpl
vuze	org.gudy.azureus2.core3.tracker.server.impl.tcp.blocking.TRBlockingServerProcessor
vuze	org.gudy.azureus2.core3.util.AEMonSem
vuze	org.gudy.azureus2.core3.util.StringInterner
vuze	org.gudy.azureus2.platform.unix.ScriptBeforeStartup
vuze	org.gudy.azureus2.pluginsimpl.local.disk.DiskManagerRandomReadController
vuze	org.gudy.azureus2.pluginsimpl.remote.RPRequestHandler
vuze	org.gudy.azureus2.pluginsimpl.update.PluginUpdatePlugin
vuze	org.gudy.azureus2.ui.common.Main
vuze	org.gudy.azureus2.ui.console.commands.Plugin
vuze	org.gudy.azureus2.ui.console.commands.Priority
vuze	org.gudy.azureus2.ui.console.commands.Set
vuze	org.gudy.azureus2.ui.console.commands.Share
vuze	org.gudy.azureus2.ui.console.commands.Show
vuze	org.gudy.azureus2.ui.swt.ImageRepository
vuze	org.gudy.azureus2.ui.swt.KeyBindings

Project	Class
	SF110 classes with Ciclomatic Complexity equal to or higher than five
vuze	org.gudy.azureus2.ui.swt.Main
vuze	org.gudy.azureus2.ui.swt.MenuBuildUtils
vuze	org.gudy.azureus2.ui.swt.Messages
vuze	org.gudy.azureus2.ui.swt.PropertiesWindow
vuze	org.gudy.azureus2.ui.swt.TorrentUtil
vuze	org.gudy.azureus2.ui.swt.URLTransfer
vuze	org.gudy.azureus2.ui.swt.components.CompositeMinSize
vuze	org.gudy.azureus2.ui.swt.components.graphics.PingGraphic
vuze	org.gudy.azureus2.ui.swt.components.graphics.ScaledGraphic
vuze	org.gudy.azureus2.ui.swt.components.graphics.SpeedGraphic
vuze	org.gudy.azureus2.ui.swt.debug.UIDebugGenerator
vuze	org.gudy.azureus2.ui.swt.mainwindow.SelectableSpeedMenu
vuze	org.gudy.azureus2.ui.swt.pluginsimpl.BasicPluginConfigImpl
vuze	org.gudy.azureus2.ui.swt.shells.GCStringPrinter
vuze	org.gudy.azureus2.ui.swt.views.FilesViewMenuUtil
vuze	org.gudy.azureus2.ui.swt.views.GeneralView
vuze	org.gudy.azureus2.ui.swt.views.ViewUtils
vuze	org.gudy.azureus2.ui.swt.views.table.impl.TableTooltips
vuze	org.gudy.azureus2.ui.swt.views.table.impl.TableViewSWT_Common
vuze	org.gudy.azureus2.ui.swt.views.table.impl.TableViewSWT_EraseItem
vuze	org.gudy.azureus2.ui.swt.views.table.impl.TableViewSWT_PaintItem
vuze	org.gudy.azureus2.ui.swt.views.table.impl.TableViewSWT_TabsCommon
vuze	org.gudy.azureus2.ui.swt.views.tableitems.mytorrents.HealthItem
vuze	org.gudy.azureus2.ui.swt.views.tableitems.mytorrents.PiecesItem
vuze	org.gudy.azureus2.ui.swt.views.utils.CategoryUIUtils
vuze	org.gudy.azureus2.update.CoreUpdateChecker
freemind	accessories.plugins.ChangeNodeLevelAction
freemind	freemind.controller.NodeDragListener
freemind	freemind.controller.filter.condition.ConditionFactory
freemind	freemind.main.Base64Coding
freemind	freemind.modes.XMLElementAdapter
freemind	freemind.modes.common.CommonNodeKeyListener
freemind	freemind.modes.mindmapmode.actions.ApplyPatternAction
freemind	freemind.modes.mindmapmode.actions.ExportBranchAction
freemind	freemind.preferences.layout.KeyEventTranslator
freemind	freemind.preferences.layout.KeyEventWorkaround
checkstyle	com.atlassw.tools.eclipse.checkstyle.config.gui.widgets.ConfigPropertyWidgetFactory

Project	Class
	SF110 classes with Ciclomatic Complexity equal to or higher than five
checkstyle	com.atlassw.tools.eclipse.checkstyle.config.savefilter.FileContentsHolderSaveFilter
checkstyle	com.atlassw.tools.eclipse.checkstyle.config.savefilter.TreeWalkerModuleSaveFilter
checkstyle	com.atlassw.tools.eclipse.checkstyle.projectconfig.filters.UnOpenedFilesFilter
weka	weka.Run
weka	weka.attributeSelection.CfsSubsetEval
weka	weka.classifiers.CheckClassifier
weka	weka.classifiers.bayes.net.search.ci.ICSSearchAlgorithm
weka	weka.classifiers.functions.supportVector.RegSMO
weka	weka.classifiers.lazy.kstar.KStarNominalAttribute
weka	weka.classifiers.lazy.kstar.KStarNumericAttribute
weka	weka.classifiers.pmml.consumer.GeneralRegression
weka	weka.classifiers.trees.j48.C45ModelSelection
weka	weka.clusterers.CheckClusterer
weka	weka.core.ContingencyTables
weka	weka.core.Statistics
weka	weka.core.json.JSONInstances
weka	weka.core.matrix.EigenvalueDecomposition
weka	weka.core.matrix.QRDecomposition
weka	weka.core.matrix.SingularValueDecomposition
weka	weka.core.stemmers.LovinsStemmer
weka	weka.estimators.UnivariateKernelEstimator
weka	weka.gui.AttributeVisualizationPanel
weka	weka.gui.graphvisualizer.HierarchicalBCEngine
weka	weka.gui.sql.ResultSetHelper
weka	weka.gui.treevisualizer.TreeBuild
weka	weka.gui.treevisualizer.TreeVisualizer
weka	weka.gui.visualize.VisualizeUtils
liferay	com.liferay.portal.dao.db.DBFactoryImpl
liferay	com.liferay.portal.dao.orm.hibernate.LockModeTranslator
liferay	com.liferay.portal.dao.orm.hibernate.TypeTranslator
liferay	com.liferay.portal.jsonwebservice.JSONRPCResponse
liferay	com.liferay.portal.kernel.cal.RecurrenceSerializer
liferay	com.liferay.portal.kernel.dao.orm.QueryUtil
liferay	com.liferay.portal.kernel.search.facet.AssetEntriesFacet
liferay	com.liferay.portal.kernel.search.facet.MultiValueFacet
liferay	com.liferay.portal.kernel.search.facet.RangeFacet
liferay	com.liferay.portal.kernel.search.facet.ScopeFacet

Project	Class
	SF110 classes with Ciclomatic Complexity equal to or higher than five
liferay	com.liferay.portal.kernel.servlet.filters.invoker.FilterMapping
liferay	com.liferay.portal.kernel.util.URLCodec
liferay	com.liferay.portal.kernel.workflow.WorkflowConstants
liferay	com.liferay.portal.model.impl.UserCacheModel
liferay	com.liferay.portal.security.pacl.PACLClassUtil
liferay	com.liferay.portal.security.pacl.checker.BaseReflectChecker
liferay	com.liferay.portal.spring.hibernate.DialectDetector
liferay	com.liferay.portal.spring.jpa.DatabaseDetector
liferay	com.liferay.portal.spring.jpa.LocalContainerEntityManagerFactoryBean
liferay	com.liferay.portal.struts.StrutsURLEncoder
liferay	com.liferay.portal.tools.LangBuilder
liferay	com.liferay.portal.util.EntityResolver
liferay	com.liferay.portal.webdav.WebDAVServlet
liferay	com.liferay.portal.xml.NodeList
liferay	com.liferay.portlet.dynamicdatamapping.storage.FieldConstants
liferay	com.liferay.portlet.expando.model.ExpandoColumnConstants
liferay	com.liferay.portlet.expando.util.ExpandoConverterUtil
liferay	com.liferay.portlet.shopping.model.impl.ShoppingOrderCacheModel
liferay	com.liferay.taglib.portlet.DefineObjectsTag
liferay	com.liferay.util.CreditCard
pdfsam	it.pdfsam.gnu.gettext.GettextResource
pdfsam	it.pdfsam.plugin.split.listener.RadioListener
pdfsam	jcmdline.StringFormatHelper
pdfsam	org.pdfsam.guiclient.business.PagesWorker
pdfsam	org.pdfsam.guiclient.commons.renderers.JPdfSelectionTableRenderer
pdfsam	org.pdfsam.i18n.GettextResource
pdfsam	org.pdfsam.plugin.coverfooter.listeners.RunButtonActionListener
pdfsam	org.pdfsam.plugin.docinfo.listeners.RunButtonActionListener
pdfsam	org.pdfsam.plugin.encrypt.listeners.RunButtonActionListener
pdfsam	org.pdfsam.plugin.merge.listeners.RunButtonActionListener
pdfsam	org.pdfsam.plugin.mix.listeners.RunButtonActionListener
pdfsam	org.pdfsam.plugin.setviewer.listeners.RunButtonActionListener
pdfsam	org.pdfsam.plugin.split.listeners.RadioListener
pdfsam	org.pdfsam.plugin.split.listeners.RunButtonActionListener
pdfsam	org.pdfsam.plugin.vcomposer.listeners.RunButtonActionListener
pdfsam	org.pdfsam.plugin.vpagereorder.listeners.RunButtonActionListener
firebird	org.firebirdsql.encodings.EncodingFactory

Project	Class
SF110 classes with Ciclomatic Complexity equal to or higher than five	
dsachat	dsachat.client.gui.MainFrame
dsachat	dsachat.gm.gui.GmFrame
beanbin	net.sourceforge.beanbin.search.WildcardSearch
jsecurity	org.jsecurity.util.AntPathMatcher
jmca	com.soops.CEN4010.JMCA.JMCAAnalyzer
jmca	com.soops.CEN4010.JMCA.JParser.JavaParserTokenManager
jmca	com.soops.CEN4010.JMCA.JParser.ParseException
tullibee	com.ib.client.Contract
tullibee	com.ib.client.Order
byuic	com.yahoo.platform.yui.compressor.JavaScriptCompressor
byuic	com.yahoo.platform.yui.compressor.YUICompressor
saxpath	org.saxpath.Axis
gangup	gui.GroupPanel
gangup	map.VisibilityMap
gangup	module.BasicRules
gangup	module.GameModule
apbsmem	apbs_mem_gui.Run
a4j	net.kencochrane.a4j.DAO.Product
httpanalyzer	httpanalyzer.HttpPreference
javaviewcontrol	com.pmdesigns.jvc.tools.HtmlEncoder
javaviewcontrol	com.pmdesigns.jvc.tools.JVCParserTokenManager
javaviewcontrol	com.pmdesigns.jvc.tools.ParseException
corina	corina.editor.EditorTabSetFactory
corina	corina.editor.SamplePrintEditor
corina	corina.editor.SamplePrinter
corina	corina.manip.Truncate
corina	corina.map.SiteRenderer
corina	corina.prefs.components.FontPopup
corina	corina.util.NaturalSort
schemaspy	net.sourceforge.schemaspy.view.DotNode
javabullboard	framework.MainClass
lilith	de.huxhorn.lilith.Lilith
lilith	de.huxhorn.lilith.data.logging.MessageFormatter
lilith	de.huxhorn.lilith.data.logging.protobuf.LoggingEventProtobufDecoder
lilith	de.huxhorn.lilith.data.logging.protobuf.LoggingEventProtobufEncoder
lilith	de.huxhorn.lilith.swing.table.renderer.ApplicationRenderer
lilith	de.huxhorn.lilith.swing.table.renderer.IdRenderer

Project	Class
	SF110 classes with Ciclomatic Complexity equal to or higher than five
lilith	de.huxhorn.lilith.swing.table.renderer.LevelRenderer
lilith	de.huxhorn.lilith.swing.table.renderer.LoggerNameRenderer
lilith	de.huxhorn.lilith.swing.table.renderer.MarkerRenderer
lilith	de.huxhorn.lilith.swing.table.renderer.MessageRenderer
lilith	de.huxhorn.lilith.swing.table.renderer.MethodRenderer
lilith	de.huxhorn.lilith.swing.table.renderer.NdcRenderer
lilith	de.huxhorn.lilith.swing.table.renderer.ProtocolRenderer
lilith	de.huxhorn.lilith.swing.table.renderer.RemoteAddrRenderer
lilith	de.huxhorn.lilith.swing.table.renderer.RequestUriRenderer
lilith	de.huxhorn.lilith.swing.table.renderer.SourceRenderer
lilith	de.huxhorn.lilith.swing.table.renderer.StatusCodeRenderer
lilith	de.huxhorn.lilith.swing.table.renderer.ThreadRenderer
lilith	de.huxhorn.lilith.swing.table.renderer.ThrowableRenderer
lilith	de.huxhorn.lilith.swing.table.renderer.TimestampRenderer
lilith	de.huxhorn.lilith.swing.table.tooltips.MessageTooltipGenerator
lilith	de.huxhorn.lilith.swing.table.tooltips.ThreadTooltipGenerator
summa	dk.statsbiblioteket.summa.common.util.PayloadMatcher
summa	dk.statsbiblioteket.summa.common.util.RAMEater
summa	dk.statsbiblioteket.summa.ingest.split.XMLSplitterParserTarget
summa	dk.statsbiblioteket.summa.search.tools.QuerySanitizer
summa	org.apache.lucene.search.exposed.ExposedTimSort
nutzenportfolio	ch.bfh.egov.nutzenportfolio.service.projekt.ProjektDaoService
dvd-homevideo	Convert
dvd-homevideo	Menu
diebierse	bierse.controller.DefaultSettingsController
biff	Scanner
jiprof	com.mentorgen.tools.profile.Controller
jiprof	com.mentorgen.tools.profile.instrument.PerfMethodAdapter
jiprof	com.mentorgen.tools.util.profile.Client
jiprof	org.objectweb.asm.jip.ClassReader
jiprof	org.objectweb.asm.jip.FieldWriter
jiprof	org.objectweb.asm.jip.Frame
jiprof	org.objectweb.asm.jip.MethodWriter
lagoon	nu.staldal.lagoon.LagoonCLI
db-everywhere	com.gbshape.dbe.mysql.MysqlTableStructure
db-everywhere	com.gbshape.dbe.sapdb.SapdbTableStructure
jhandballmoves	visu.handball.moves.actions.NewPassEventAction

Project	Class
SF110 classes with Ciclomatic Complexity equal to or higher than five	
jhandballmoves	visu.handball.moves.controller.MouseController
jhandballmoves	visu.handball.moves.model.PassEvent
hft-bomberman	server.ServerGameModel
templateit	org.templateit.WorkbookParser
noen	fi.vtt.noen.mfw.bundle.server.plugins.persistence.PersistencePluginImpl
openjms	Browser
openjms	DurableSubscriber
openjms	Listener
openjms	Receiver
openjms	Sender
openjms	org.exolab.jms.tools.admin.AdminInfo
echodep	edu.uiuc.ndiipp.hubandspoke.profile.HaSMETSValidator
echodep	edu.uiuc.ndiipp.hubandspoke.profile.HaSMETSWebValidator
echodep	edu.uiuc.ndiipp.hubandspoke.workflow.WorkflowManager
battlecry	bcry.battlecry
battlecry	bcry.bcGenerator
fixsuite	org.fixsuite.message.parsers.fpl.MsgContentsParser
openhre	com.browsersoft.openhre.hl7.impl.parser.HL7ParserImpl
wheelwebtool	wheel.ErrorPage
wheelwebtool	wheel.asm.ClassReader
wheelwebtool	wheel.asm.FieldWriter
wheelwebtool	wheel.asm.Frame
wheelwebtool	wheel.asm.MethodWriter
wheelwebtool	wheel.xhtmlConversion.Node
javathena	org.javathena.core.utiles.Constants
javathena	org.javathena.login.parse.FromAdmin
javathena	org.javathena.login.parse.FromChar
javathena	org.javathena.login.parse.FromClient
ipcalculator	ipac.BinaryCalculate
ipcalculator	ipac.URLOpener
xbus	net.sf.xbus.admin.html.AdminDispatcherServlet
xbus	net.sf.xbus.base.xml.IteratedWhitespaceInElementAndCommentDeletion
ifx-framework	org.sourceforge.ifx.basetypes.IFXObject
shop	umd.cs.shop.JSListAxioms
shop	umd.cs.shop.JSTerm
jaw-br	jaw.entrada.Salvar
jopenchart	de.progra.charting.model.StackedChartDataModelConstraints

Project	Class
	SF110 classes with Ciclomatic Complexity equal to or higher than five
jopenchart	de.progra.charting.render.InterpolationChartRenderer
jopenchart	de.progra.charting.render.LineChartRenderer
jiggler	jigl.image.ColorHistogram
jiggler	jigl.image.levelSetTool.LevelSetNudge
jiggler	jigl.image.levelSetTool.LevelSetSharpen
jiggler	jigl.image.levelSetTool.LevelSetSmooth
jiggler	jigl.image.levelSetTool.LocalMedianSmooth
jiggler	jigl.image.ops.ConnectedComponents
jiggler	jigl.image.types.ComplexMIPMap
jiggler	jigl.image.types.MIPMap
jiggler	jigl.image.types.TiledComplexMIPMap
jiggler	jigl.image.utils.ImageGenerator
jiggler	jigl.image.utils.LocalDifferentialGeometry
gfarcegestionfa	fr.unice.gfarce.dao.OracleFormationDao
gfarcegestionfa	fr.unice.gfarce.dao.OracleIdentiteDao
gfarcegestionfa	fr.unice.gfarce.interGraph.CreerUnEtudiantAction
gfarcegestionfa	fr.unice.gfarce.interGraph.CreerUneFormationAction
gfarcegestionfa	fr.unice.gfarce.interGraph.ModifTableStockage
dcparseargs	de.devcity.parseargs.ArgsParser
celwars2009	Entity
heal	org.heal.module.search.AdvSearchDAO
heal	org.heal.module.search.SimpleSearchDAO
heal	org.heal.servlet.WSSimpleSearchAction
heal	org.heal.servlet.cataloger.MetadataRecordModifier
feudalismgame	src.Battle
feudalismgame	src.Purchase
newzgrabber	Newzgrabber.GroupsDialog
newzgrabber	Newzgrabber.LineData
newzgrabber	Newzgrabber.Newzbatch
newzgrabber	Newzgrabber.OptionsPanel