FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Assessing the Effectiveness of Defect Prediction-based Test Suites at Localizing Faults

Orlando Macedo

Mestrado em Engenharia de Software

Supervisor: Prof. José Campos

Second Supervisor: Prof. Rui Abreu

September 20, 2023

Assessing the Effectiveness of Defect Prediction-based Test Suites at Localizing Faults

Orlando Macedo

Mestrado em Engenharia de Software

September 20, 2023

Abstract

Debugging a software program constitutes a significant and laborious task for programmers, often consuming a substantial amount of time. The need to identify faulty lines of code further compounds this challenge, leading to decreased overall productivity. Consequently, the development of automated tools for fault localization becomes imperative to streamline the debugging process and enhance programmer productivity.

In recent years, the field of automatic test generation has witnessed remarkable advancements, significantly improving the efficacy of automatic tests in detecting faults. The localization of faults can be further optimized through the utilization of such sophisticated tools.

This dissertation aims to conduct an experimental study that assembles specialized automatic test generation tools designed to detect faults by estimating the likelihood of code being faulty. These tools will be compared against each other to discern their relative performance and effectiveness. Additionally, the study will comprehensively compare developer-generated tests with automatically generated tests to evaluate their respective aptitude for fault localization. Through this investigation, we seek to identify the most effective automated test generation tool while providing valuable insights into the relative merits of developer-generated and automatically generated tests for fault localization.

Acknowledgements

I would like to thank my supervisors, Professor José Campos and Professor Rui Abreu, for their guidance and support throughout the development of this thesis. I would also like to thank all the teachers and mentors who were present during my academic journey, as well as my family and friends for their support.

Orlando Macedo

Contents

List of Figures

Abbreviations and Symbols

TG	Test Generation
FD	Fault Detection
FL	Fault Localization
SFL	Software Fault Localization
SBFL	Spectrum-based Fault Localization
IDE	Integrated Development Environment
DS	Dynamic Slicing
SS	Static Slicing

Chapter 1

Introduction

Despite considerable progress in the domain of software testing technologies, there persist specific areas where substantial improvement is warranted. One such domain is automatic software fault localization, where promising technologies exist but are yet to achieve a commendable level of maturity. As a result, they are not yet integrated into the routine tasks of the majority of software developers.

There is empirical evidence that indicates a direct correlation between the quality of the test stack and the efficiency with which it exercises the codebase, ultimately leading to improved results in automatic fault localization [8]. Consequently, the utilization of automatic test generation tools is justified. Often, the task of conceptualizing and implementing tests is viewed with disfavor by software developers. In some instances, tests are relegated to the end of a sprint and are foregone altogether when the implementation consumes the entirety of the sprint, leaving no time for test creation. This scenario unfolds more frequently than not, thereby underscoring the imperative need to have at least automatically generated tests. These tests are particularly advantageous in regression testing, as they generate tests that reflect the current scope of the system. Should the scope unexpectedly alter, these tests would identify the change. Therefore, automatic tests serve as a reliable source of truth and should be utilized when manual test generation is neglected.

Taking into account the aforementioned correlation between a high-quality test suite and effective fault localization, coupled with the knowledge that automatic test generation can produce superior test suites, an examination is warranted to ascertain whether a fault-focused test suite enhances the accuracy and efficiency of fault localization. This hypothesis forms the crux of the thesis presented in this paper, which aims to determine whether test suites automatically generated with an emphasis on fault detection also aid in fault localization.

In the forthcoming sections, a succinct examination of fault detection and fault localization will be proffered, primarily aiming to illuminate the temporal progression of these domains. This paper will delve deeper into a summarization of pertinent tools imperative for executing the empirical investigation. Additionally, we will catalog a series of similar studies that will act as a foundational base for our won.

Ultimately, we will expound on the procedural aspects of the empirical investigation, including

the employed metrics and an in-depth analysis of the procured results. These actions serve not only to maintain the academic rigor inherent in the research process, but also to ensure the accessibility and reproducibility of the results presented in this scholarly discourse.

1.1 Context

Empirical scholarly research has demonstrated significant advancements in the realm of fault detection, specifically for tools concentrated on predictive analyses. These tools are poised to optimize the identification of bug residences within the code. Essentially, these mechanisms proactively anticipate probable locations of bugs and rigorously scrutinize those segments of the codebase.

Prior investigations [20] [27] [8] have yielded substantial positive outcomes for fault detection. However, a question remains as to the applicability and effectiveness of these tools in the context of fault localization.

1.2 Problem

In light of the ongoing advancements in technologies and their consequential achievements in the domain of fault detection, it is a logical progression to question the applicability of these tools to the area of fault localization. A notable dearth of empirical evidence in this domain serves as a substantial barrier to the assimilation of these tools within the software developers' toolkit. This lack of data highlights the necessity for further rigorous academic investigation in this sector to ascertain the potential utility of these fault detection tools in fault localization processes.

1.3 Motivation

The continued enhancement of fault localization tools is a necessity, given their potential to augment software developers' productivity. Building upon the accepted premise that a robust test stack results in superior troubleshooting outcomes, it is logical that this dissertation is proposed.

It is of paramount importance within the academic community and beyond to ascertain whether the generation of optimized tests specifically designed to exercise faulty code could aid in the reduction of effort expended to locate faults within the codebase. The implications of such findings could have far-reaching effects on software development practices and therefore warrant in-depth investigation.

1.4 Research Questions

The research questions of this dissertation are related to the applicability of defect prediction-based test suites in fault localization. Following that line of thought, the following research questions

RQ1: To what extent does the employment of defect prediction-based test suites assist in fault localization?

RQ2: Does the utilization of defect prediction-based test suites enhance fault localization compared to developer-based test suites?

RQ3: Is there a correlation between coverage and mutation score with enhanced fault localization?

RQ4: Is there a correlation between fault detection and fault localization?

RQ5: Does the volume of tests generated correlate with the effectiveness in fault detection and fault localization?

RQ6: Do the approaches complement each other? Should we use them together?

1.5 Contributions

The development of this dissertation resulted in other contributions besides what is presented in this document. In particular, the approaches used in the study were updated to the latest version of *EvoSuite*. As all used in one way or another *EvoSuite* at its core, the updates were applied to make the comparison between them fairer. This work can be found in the following repository: https://github.com/Orlando-pt/evosuite_defect-prediction-techniques.

An approach used in the study, *PreMOSA*, needed to have a list of all the methods in a class to execute properly. In that regard, a tool was developed for that purpose. It can be found in the following repository: https://bitbucket.org/rjust/fault-localization-data/src/d4j-2.0/utils/buggymethods/.

Still, in the context of *PreMOSA*, the approach needs to know which methods have bugs, so that the tests generated are focused on those methods. The study used the bugs from the *Defects4J* repository [32], there was already information on where the bugs reside for some of the projects. However, the latest version is fairly recent at the time of writing, so some bugs were not yet included. To that end, the remaining bugs were systematically added to the repository. This work can be found in the following repository: https://bitbucket.org/rjust/fault-localization-data/src/d4j-2.0/analysis/pipeline-scripts/buggy-lines/.

Every methodology employed in the research has undergone prior evaluation, albeit within varied contexts. A unifying theme among them is the enhancement in the number of bugs considered for evaluation in the present study, compared to previous ones. The *Defects4J* repository now encompasses a greater volume of real-world bugs, enriching the assessment and fostering the emergence of novel perspectives. The ensuing table encapsulates the individual contributions of each method within the purview of this dissertation.

Approach	**Previous Contributions**	**Current Contributions**

PreMOSA	• Evaluated on 420 bugs from *Defects4J*, referent to 6 real-world projects • Evaluated with defect predictions of 75% accuracy • Evaluated with defect predictions of 100% accuracy • Results gathered for FD	+ Evaluated on 764 bugs from *Defects4J* used for FD, referent to 17 real-world projects + Evaluated on 421 bugs from *Defects4J* used for FL, referent to 16 real-world projects • Evaluated with defect predictions of 100% accuracy + Results gathered both for FD and FL
EntBug	• Evaluated on 7 simple bugs, referent to 6 real-world projects. • Results gathered for FL (diagnosability) and cost	+ Evaluated on 761 bugs from *Defects4J* used for FD, referent to 17 real-world projects + Evaluated on 420 bugs from *Defects4J* used for FL, referent to 17 real-world projects + Results gathered both for FD and FL
DDU	• Evaluated on 1050 bugs generated artificially • Evaluated on 186 bugs from *Defects4J*, referent to 5 real-world projects • Results gathered for FD and FL (diagnosability)	+ Evaluated on 779 bugs from *Defects4J* used for FD, referent to 17 real-world projects + Evaluated on 310 bugs from *Defects4J* used for FL, referent to 16 real-world projects + Results gathered both for FD and FL

Ulysis	• Evaluated on 111 bugs from *Defects4J*, referent to 5 real-world projects • Results gathered for FD and FL (coverage, diagnosability, cost)	+ Evaluated on 777 bugs from *Defects4J* used for FD, referent to 17 real-world projects + Evaluated on 158 bugs from *Defects4J* used for FL, referent to 16 real-world projects + Results gathered both for FD and FL

1.6 Reproducibility

The study presented in this dissertation was developed with the utmost care to ensure that the results are reproducible. To that end, all the approaches used in the study are open-source and publicly available at: https://github.com/Orlando-pt/evosuite_defect-prediction-techniques. Each methodology is present in a different branch, and vanilla *EvoSuite* is present in the *master* branch.

The scripts used to run the study are also publicly available: https://github.com/jose/fault-detection-and-localization-of-defect-prediction-based-tests

To run one of the approaches, *PreMOSA*, it is necessary to have a list of all the methods that are buggy. The files with the buggy methods are also publicly available at: https://bitbucket.org/rjust/fault-localization-data/src/d4j-2.0/analysis/pipeline-scripts/buggy-methods/.

Chapter 2

Background

2.1 Introduction

As mentioned earlier, creating tests can be a very tedious and error-prone task. Thus, in recent years there has been a constant evolution of technologies to help code developers automatically generate tests. These tests can be used as a single test base but also as a complement to the tests created by the developers themselves.

Another big headache for programmers is related to debugging applications. It is known that most of the time spent by a programmer is in debugging the code written by himself or someone else. Some errors in certain situations are difficult to anticipate, so fault localization technologies are very important because they can locate in the code the statement or line that causes a given failure. This type of technology has great potential to significantly increase developer productivity.

It is therefore important to understand whether the generation of tests focused on detecting defects/failures in the source code also helps when it is time to locate them.

2.2 Fault Detection

Fault detection refers to the process of identifying defects or errors in software systems. Typically involves a combination of manual and automated approaches. Manual techniques may include code reviews, static analysis, and manual testing, where software engineers carefully inspect the code and execute test cases to identify potential faults. Automated techniques, on the other hand, utilize tools and frameworks specifically designed for fault detection, such as unit testing frameworks, code analyzers, and debugging tools.

Automated testing plays a crucial role in fault detection by executing predefined test cases and comparing the actual software behavior against expected results. This helps identify deviations, failures, or unexpected behaviors that indicate the presence of faults.

Manual creation of unit tests is a time-consuming and error-prone process. That is why several tools can be used to automatically generate unit tests for existing software systems or new ones.

These tools analyze the code, identify different execution paths, boundary conditions, and inputs, and generate test cases to cover those scenarios.

By automatically generating test suites, developers can save time and effort in writing individual test cases. It helps increase test coverage by exploring a broader range of code paths, thereby increasing the likelihood of detecting faults or unexpected behaviors.

Ultimately, the combination of unit testing and automatic generation of test suites contributes to more comprehensive testing, improved code quality, and faster development cycles by automating the process of test case creation and ensuring thorough coverage of code paths.

One of the most effective test generation tools is called *EvoSuite*. It is a tool that automatically generates unit tests for Java programs. Uses a genetic algorithm to evolve test cases that cover different execution paths and inputs. It is possible to specify the desired coverage criteria, such as statement coverage or branch coverage, and the tool will generate test cases that satisfy those criteria.

2.3 Software FL Techniques

Software Fault Localization (SFL) is a technique that allows one to find the location of faults in a program. From Fault Localization (FL) techniques, a programmer can easily check why a program is not working. Traditionally, if a program isn't working then the programmer has to spend time on *debugging*. Manual *debugging* is a task that can take a lot of time and not be effective in finding the bug. It is for this reason that FL tools are increasingly gaining popularity. Such tools have great potential to increase the productivity of software developers, as well as substantially reduce the likelihood of bugs being delivered.

In the next sections it is identified what are the main FL techniques and which are the main tools that implement these techniques.

2.3.1 Traditional FL Techniques

Since the creation of the first programming languages, there has been always a need to help programmers to debug their code. Traditional techniques to help with fault location go from logging, assertions, breakpoints or even profiling. In the next sections, we briefly explore what these techniques consist of.

Program Logging consists of simple syntactic methods, such as the use of prints in a program, it is possible to track the location of faults or at least have a better idea of where they might be located. In 1999 *Jermaine Edwards* wrote an article in which he proposed to patent and standardize how logs should be written [15]. After several decades these standards continue to be used daily by programmers around the world to help debug applications.

Certain conditions only manifest themselves at runtime. With **assertions**, it is possible to check whether in a given point of code some condition is met or not. If the condition is not met, the program signals that the condition at point x was not met and so the programmer has a better idea of where the fault might be located.

This FL technique goes back to the beginnings of programming languages, in 1992 *David Rosenblum* detailed how to use this kind of technique to make a program more resilient and reliable [33].

Breakpoints are another very useful technique that allows programmers to pause the execution of a program and check its current characteristics. By using *BreakPoints* it is possible to specify areas of the code where we want to pause the execution of the program, after pausing we can check which are the current values of the variables that make up the program, and we can also modify them to check program-specific behavior.

Even nowadays this is a very used technique to debug programs. *Intellij*, one of the most popular *IDE* [29] for languages related to *Java* has its own debugger with functionalities that allow to specify lines to stop the execution, analyze the current state of the program, run the program line by line and more [21]. *Visual Studio*, another very popular *IDE* belonging to *Microsoft* also has its own debugger with identical functionalities [25].

2.3.2 Advanced FL Techniques

The systems have evolved in complexity and size over the years, which has led to the techniques listed above decreasing their effectiveness. The area of SFL has evolved precisely because of this, there is a need to locate faults in larger and more complex systems.

In this section it is explored 6 FL categories, namely *Slice-based*, *Spectrum-based*, *Program State-based*, *Machine Learning-based*, *Data Mining-based*, *Model-based* and a brief comparison between them.

2.3.2.1 Slice-Based Techniques

This set of techniques seeks to abstract a program by reducing it by removing irrelevant slices in such a way that the resulting program continues to function as before concerning certain specifications. Since 1979, when Weiser first enunciated **Static slicing**, there has been a constant evolution of these techniques. Evolution brought techniques such as dynamic slicing and its subdivisions, namely *Relevant Slicing*, *Conditioned Slicing* and more [37].

2.3.2.2 Dynamic Slicing

Dynamic Slicing first proposed in 1990 by Korel and Laski [24] suggests that a *dynamic slice* it is a part of the program that affects a given variable in a particular execution of the program. As only one execution is taken into account, *dynamic slicing* can significantly reduce the slice size compared to *static slicing*.

Next, some of the *dynamic slicing* evolutions that have been proposed over the years are explored:

- **Relevant Slicing** proposed by Agrawal et al. in 1993, argues that a relevant slice contains not only the statement that influences the variable but also the executed statements

that did not affect the output, but which could have an effect if they had been evaluated differently [3].

- **Conditioned Slicing** proposed by Canfora et al. in 1998 stipulates that a *conditioned slice* is a subset of program statements that preserves the behavior of the original taking into account a *slicing criterion* for a given set of execution paths [9].

- **Union Slicing** was proposed by Hall in 1995 and introduces the notion of simultaneous dynamic program slicing to extract executable program subsets. Works by applying any kind of dynamic slicing algorithm that meets certain criteria (one of them stipulates the obligation to create executable slices) and incrementally it creates the *simultaneous slice* using an iterative algorithm for all test cases [17].

- **Hybrid Slicing** uses the advantages of *static slicing* and *dynamic slicing* to mitigate its disadvantages. *Static slicing* suffers from the problem of imprecision and *dynamic slicing* is specific to only one execution. Gupta et al. [16] proposed to improve *Dynamic Slicing*(DS) by incorporating dynamic information into *Static Slicing*(SS). This hybrid technique exploits the information that is available during the debugging phase to calculate the static slices.

In addition to these so-called traditional techniques, variations have emerged with premises that are also daring. Among them stand out *Amorphous Slicing* first explored by Harman et al. [18], *Denotation Slicing* introduced by P. A. Hauser [19], and finally, *Parametric program slicing* published by J. Field et al. in 1995 [14].

2.3.2.3 Program Spectrum-Based Techniques

It is a type of technique whose main ramifications are inspired by studies in probabilistic and statistical-based causality models.

The *program spectra* details the execution information of a program from certain points of view, such as execution information for conditional branches or loop-free intra-procedural paths. In the year 1987 Collofello et al. [12] suggested that such a kind of spectra could be used to locate faults.

When execution fails, this information can be used to identify the code that caused the failure. To that end, these techniques use *code coverage* metrics or *executable statement hit spectrum*(ESHS) that indicate which blocks were exercised during program execution. This information can later be used to narrow the search surface, decreasing the effort to find the fault.

As stated by Wong et al. [35], the oldest studies referring to this area only used failed tests to create the *program spectra*. But this way of seeing things proved to be very ineffective, so posthumous studies use information from both failed and successful tests.

Renieris and Reiss [303] proposed an ESHS technique called **nearest neighbor**. This technique seeks to obtain information of a failed test and a successful test that is as similar as possible

to the one that failed, so that it has the shorter distance. The bug is found in the differences between the two tests. If the bug is not located in the *difference set* then the program continues to run, building the *Program Dependence Graph* and checking adjacent nodes unchecked on the graph until all nodes in the graph have been examined.

Tarantula is an ESHS-based similarity coefficient-based technique that uses coverage and execution results, both failed and successful tests, in order to compute the suspiciousness of each statement [23]. This technique shows really good results, and despite being created in the yearly 2000's it still reaches good results in fault localization. The main advantage is related to the significant reduction of necessary computing power inherited from the algorithm itself.

Ochiai is another popular technique with some similarity to the *nearest neighbor* but with mainly 2 differences. The first one is that *Ochiai* uses multiple test cases, contrasting with the single test case from *nearest neighbor*. Also, *Ochiai* includes all successful test cases, on the other hand, *nearest neighbor* only considered the closest to the failed ones [2].

Spectrum-based techniques have been particularly compared in the hope of finding the one technique that solves all the needs. Despite some papers stating that one technique is better than the other, the reality until now is that there is no technique that wins in every scenario. What is clear is that some techniques are better than others in certain conditions, but lost when facing a completely different situation. The creation of software is chaotic and doesn't follow a specific characteristic so that's why it is impossible to state a clear winner [26].

2.3.2.4 Program State-Based Techniques

This technique uses the runtime state of the program to locate where the fault is. Through the monitorization of variables and their values at a certain point in runtime, it is possible to get an accurate prediction of where a bug might be.

Relative Debugging compares the internal state of a development version where was identified a bug with a version of the program that didn't contain it [1]. On the opposite side, other approaches modify the values of some variables to identify which one is causing the faulty execution.

2.3.2.5 Machine Learning-Based Techniques

Machine learning-based techniques refer to the use of various algorithms and statistical models to analyze and learn from data in order to make predictions or decisions. In the context of FL, these techniques can be used to automatically identify the source of errors or bugs in a software system. This is typically done by analyzing program execution traces and comparing them to expected behavior.

Some examples of machine learning-based techniques that can be used for FL include decision trees, neural networks, and support vector machines. These techniques can improve the efficiency and accuracy of FL compared to traditional ones.

One example of a neural network implementation can be found in the technique presented by Wong and Qi propose a technique based on back-propagation neural networks [36]. The BP neural network is trained by collecting the coverage data of each test and the corresponding execution result. With this, the network can learn what is the relationship between those tests and is able to make predictions based on that.

2.3.2.6 Data Mining-Based Techniques

Data mining and Machine learning techniques are closely related and in some cases, used interchangeably, but they have some differences. Machine learning techniques typically involve the use of a labeled dataset to train a model to make predictions or decisions, while data mining techniques are used to uncover patterns and relationships in the data without the need for labeled data.

One of the main advantages of data mining-based techniques is that they can handle large and complex datasets, which is often the case in software systems. For instance, in [10] is discussed a combination of association rules and formal concept analysis that help in fault localization.

2.3.2.7 Model-Based Techniques

In this technique a model is generated directly from a program, which means that the model itself can model bugs. In a perfect scenario, the model doesn't contain any fault, so it can be applied in other versions of the program, the difference in behaviors is used to find bugs. This technique has an enormous disadvantage, it supposes that a correct model of each program is available [31].

2.3.3 Technique Comparison

The survey made by Wong et al. [35] shows that in more recent years the research community has been focusing on *slice* and *program spectrum-based* techniques. On the opposite side, *static* or *dynamic slice-based* techniques have been decreasing constantly their popularity.

The first reason might be that slice and program spectrum-based techniques can be applied to a wider range of software systems than static and dynamic slicing. While static slicing is only applicable to the source code, program slicing-based techniques are applied to the execution trace of the program, and program spectrum-based techniques are applied to the program's control flow and data dependencies. This allows for a more comprehensive analysis of the system's behavior and increases the chances of identifying the root cause of a fault.

Second, slice and program spectrum-based techniques are more efficient and effective than traditional slicing techniques. For example, program slicing-based techniques use dynamic slicing which is based on running the program and obtaining a trace of the program execution, which allows for a more accurate analysis of the program's behavior. Program spectrum-based techniques are able to identify faults by analyzing the program's control flow and data dependencies, which can be more effective than traditional slicing techniques that only focus on the code.

Overall, slice and program spectrum-based techniques are more widely used in recent years because they are more comprehensive, efficient, and effective than traditional slicing techniques and are better suited to the complexity and scale of modern software systems.

Chapter 3

State of The Art

With this dissertation, we propose to study whether the approaches that currently generate tests with a view to the detection of faults also helps when locating the fault itself. Previous studies of *Fault Detection* (FD) and FL explored very interesting techniques and tools, comparing them and taking their own conclusions.

In the next sections, we will explore some of the main studies related to fault detection.

3.1 Related Work on Fault Detection

In 2016 Shamshiri et al. [34] studied the effectiveness of 3 test generation tools checking the ratio of detected failures to those not detected. The tools were *Randoop* [30], *Agitar One* [1] and *EvoSuite* [13].

The procedure for generating the tests had 357 faults from the open database *Defects4j* [32] with origin in 5 projects. 26 flaws of the JFreeChart Project [2], 133 of the Google Closure Compiler [3], 65 of Apache Commons Lang [4], 106 from Apache Commons Math [6] and 27 from Joda Time [22].

In order to generate tests the study accounted for the randomness of both *Randoop* and *EvoSuite* by generating 10 test suites in 10 different executions. As for *Agitar One*, it was only done 1 run, since it is considered "fairly" deterministic.

Flaky tests and **false positives** are something that must be taken into account when generating tests. Flaky tests are tests that are unstable and may be successful on some occasions and unsuccessful on others. It is relatively frequent in tests involving comparisons of time, the system time is constantly changing and may cause test failure at a given time. False positives are related to tests that fail without having to do with the faults that actually exist. This type of situation happens mostly when the tools break the principles of object-oriented programming, for instance, calling private methods.

[1] http://www.agitar.com/solutions/products/automated_junit_generation.html
[2] https://github.com/jfree/jfreechart
[3] https://github.com/google/closure-compiler
[4] https://github.com/apache/commons-lang

After filtrating the tests that were flaky and the tests that were false positives, it was possible to calculate the faults detected with the tools by comparing existing faults with the reason why the generated tests failed. If a test fails and the reason why it failed was related to a fault that actually exists, then the fault was considered covered.

PreMOSA is an approach that was also tested in terms of its effectiveness in detecting faults. It was created by Perera et al. [27] and is a methodology that uses the information of defect predictors to guide the search for faults.

The defect predictors are used to predict the most likely locations of faults in the code. The information from that is then fed to PreMOSA, which uses a genetic algorithm to generate tests that will reveal those faults.

The whole point of this process is to focus the search on a predetermined set of components that are more likely to contain faults. Because *defect predictors* also have a margin of error, in a posterior phase the approach also exercises components that are labeled as not faulty.

The premise of *PreMOSA* is that there is no need to waste time and resources on components that do not contain bugs. *DynaMOSA* is an alternative that uses coverage information to evolve its tests, but every component has an equal probability of being faulty. *Perera* et al. [27] argue that there should be a way to focus the search where it matters, mainly because metrics such as coverage are proven to be misleading when it comes to finding faults.

3.2 Related Work on Fault Localization

Campos, Abreu et al. proposed a metric called **Entbug** in order to reduce the entropy of a diagnostic ranking and consequently, also reduce the number of candidates needed to inspect the reason for a failure [8]. The prototyping of the technique was achieved through the use of entropy to customize the genetic algorithm of *EvoSuite*.

EntBug is based on *spectrum-based reasoning* approach to multiple FL. This technique is an approximation to FL based on the probability theory. The underlying strategy is based on *Model-Based Diagnosis*, which uses logical reasoning to find faults. This approximation orders a set of program components by the probability that each of them explains a failure. The results are obtained in two distinct phases: candidate generation and candidate ranking.

For **Candidate Generation** it is necessary to understand the concept of Minimal Hitting Set (MHS). This is a problem that starts by finding a *hitting set* that consists of finding a set of values that intersects another family of sets with its own values. The *minimal hitting set* is a hitting set that cannot be made smaller without losing this last property [5]. As this is a problem that requires great computational effort, a technique called *STACCATO* was applied that significantly reduces the computational effort by computing only a relevant set of *multiple-fault candidates* [8].

As far as **Candidate Ranking** is concerned, *Candidate Generation* can generate a long list of candidates. In this phase, it is verified the probability of a given component being the cause of the failure. Afterward, those components are ordered having that probability as the ordering factor.

[5] https://github.com/VeraLiconaResearchGroup/Minimal-Hitting-Set-Algorithms/blob/master/README.md

EntBug is a technique that takes a test suite as input and produces additional tests for that test suite, which after adding the necessary tests will translate into a decrease in the entropy of the diagnosis.

As mentioned before, the objective of the technique is to reduce entropy, which can be translated into other words as information gain. The information gain that a new test case provides is determined by the reduction of the most likely suspect components. This capability reduces the diagnostic ranking entropy and consequently improves the diagnostic quality of spectrum-based reasoning. With the calculated generation of these tests, the number of tests in the test suite balances the density of the coverage matrix.

Input: Program Π, Test Suite T, Search Budget Δt, Stopping Condition C
Output: Extended Test Suite T'
1: $T' \leftarrow T$
2: $d \leftarrow |0.5 - \text{DENSITY}(T)|$
3: $\delta \leftarrow$ GETFITNESSFUNCTION(T')
4: **while** $\neg C$ **do**
5: $\quad t_c \leftarrow \text{EVOSUITE}(\Pi, \delta, \Delta t)$
6: $\quad$ **if** $d > |0.5 - \text{DENSITY}(T' \cup \{t\})| - \epsilon$ **then**
7: $\quad\quad T' \leftarrow T' \cup \{t_c\}$
8: $\quad\quad d \leftarrow |0.5 - \text{DENSITY}(T')|$
9: $\quad\quad \delta \leftarrow$ GETFITNESSFUNCTION(T')
10: $\quad$ **end if**
11: **end while**
12: **return** T'

Figure 3.1: Entbug Algorithm

The algorithm is depicted in the figure 3.1. At the beginning, it has as input a possible empty test suite T, the search budget Δt that you want to spend to generate each individual test, and condition C evaluates to true when the process has to end. The output is an extension of T.

The density of T is calculated using a *Density* function. *EvoSuite* is then called with the fitness function δ to generate a test case with the effort Δt. *EvoSuite* returns the test case that maximizes diagnostic information, which minimizes fitness function. If no tests are returned, the test case (t_c) is ignored. If t_c improves the test suite, then a new fitness function is added to it. δ is created and the value of d is updated. The steps described above are repeated until the condition C is fulfilled.

EntBug was tested using a vending machine program [7] and 6 more bugs were selected of 4 open-source projects [4] [6] [22] [5]. To solve the *Oracle Problem* it was used the version of the program where the fault existed and the later version with the correction. The experiment was repeated 10 times to account for the randomness of the results.

To check how the diagnosis improved over time, the genetic algorithm of *EvoSuite* was set to run for 10 seconds for each trial and measured the diagnostic accuracy of the evolving test suite at 5-minute intervals.

More recently, the metric **DDU** was proposed by Perez, Abreu et al. [28] that aims to verify which is the effectiveness of applying *spectrum-based fault localization* in tests generated from an automatic tool.

The main objective was to increase the value of the tests created for troubleshooting, creating specially optimized tests to not only detect failures but also serve as an effective help in the precise location of faults in the system.

Traditionally used metrics focus on the coverage of the program itself. For instance, most use branch/path coverage, modified/condition coverage and mutation coverage. These criteria do not take into account the needs inherent in locating program failures.

To create a spectra that has practical and efficient diagnosability the study took into account 3 aspects. Firstly, it is necessary to ensure that the components are frequently involved in the tests (density). Secondly, it is necessary to test the components in the most diverse possible combinations (test diversity). Lastly, spectra with less ambiguity should be favored by providing a notion of component distinguishability (uniqueness). *DDU* was designed to implement these 3 aspects.

To verify that the metric brought value to FL, an empirical study was carried out comparing the metric with other frequently used metrics. The metric implementation is incorporated in the test generation tool *EvoSuite* [13], already mentioned before. As the tool has an indeterministic component in the generation of tests, 10 executions of each experiment were carried out. Lastly, each run was limited to one run of 600 seconds.

The experiments were carried out in both a controlled environment and an uncontrolled one. The controlled environment aims to sow faults that create spectrums that are somewhat predictable. It used open-source code namely the Apache Commons-Codec [4], Apache Commons-Compress projects [5], Apache Commons-Math [6] and JodaTime [22]. From these projects, some faulty components were considered, when a test went through these components there was a 75% probability that the test failed. The uncontrolled scenarios used projects from database *Defects4J* [32] but without any change in source code.

The tool Crowbar[6] was used to apply spectrum-based techniques to create a ranked list of diagnostic candidates for the observed failures. With the sorted list, it is possible to calculate the diagnosability.

Chatterjee et al. [11] proposed an evolution of DDU, called **Ulysis**. It uses a concept of *Multiverse Analysis* that considers multiple hypothetical universes, each corresponding to a scenario where one of the components is assumed faulty.

[6]https://github.com/TQRG/crowbar-maven-plugin

Chapter 4

Empirical Study

This chapter is devoted to a comprehensive presentation of the empirical study conducted. The primary objective of this study was to compare the approaches and techniques delineated in chapter 3. Through this process, we succeeded in addressing the research questions outlined in chapter 1.4. Additionally, this methodology facilitated the provision of a structured guide designed for prospective researchers seeking to replicate, validate, or extend this work. This chapter thus serves as a bridge between our methodology and its practical implications within the broader field of study.

4.1 Experimental Setup

The empirical study was conducted utilizing the **Defects4J** database [32], a comprehensive collection of faults derived from various Java projects. This database, renowned within the research community, offers unique dual versions of each project - one version containing the bug and another rectified version. This distinctive characteristic is particularly advantageous in assessing the efficacy of the approaches subjected to testing.

However, it is pertinent to mention that not all bugs available in the database were included in this study due to certain technical constraints associated with *EvoSuite* and *GZoltar*.

As demonstrated in Figure 4.1, each methodology employed in this research utilized the same database and the same subset of bugs. Considering this fact, each methodology was executed **5 times** to mitigate the uncertainties inherent to the non-deterministic characteristic of *EvoSuite*. Each approach under evaluation can be considered a modification of *EvoSuite*, implying that the element of randomness permeates each approach under examination. To maintain the fairness of the study, an attempt was made to update all methodologies to the **latest version** of *EvoSuite*. Another critical factor is the scope within which the methodologies were assessed.

EvoSuite allows the generation of tests for a specific class, package, or even the entire project. In this study, the scope was narrowed to **specific classes** associated with the bug under investigation. This decision was made to minimize the time required to generate the test suites and to reduce the complexity of the generated test suites.

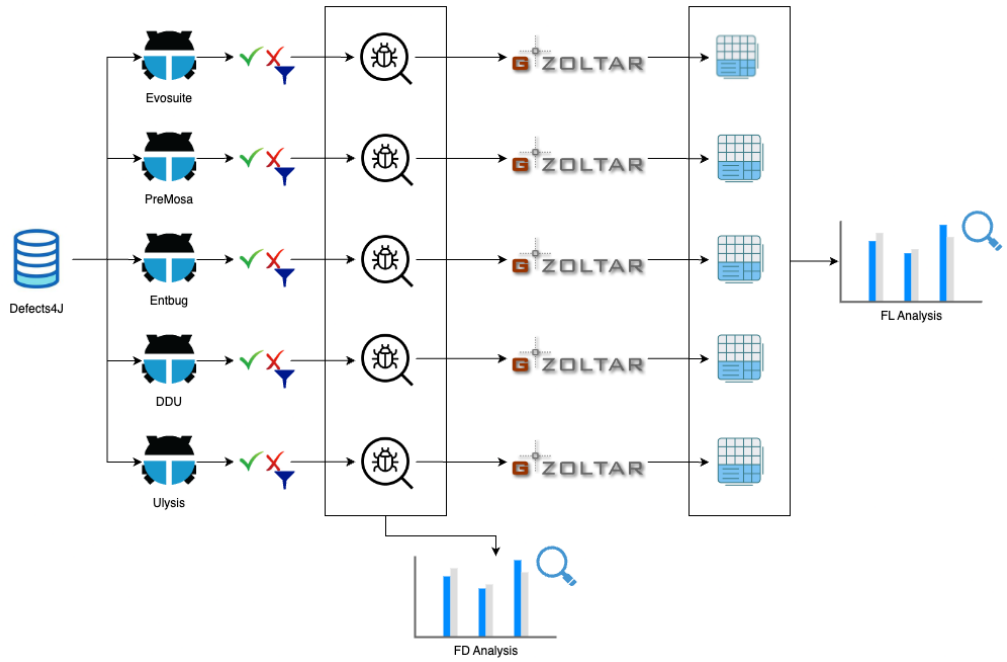


Figure 4.1: Study Architecture

The output of *EvoSuite* and its variants is a test suite. In our case, each approach was executed 5 times, yielding 5 test suites for each bug. The tests were generated using the **fixed** version of the bugs, implying that the test suites generated comprise **regression tests**. This is notably significant as it allows for the assessment of the quality of the generated tests in terms of their execution, the extent of code they cover, and the mutation score they achieve. Another critical step in this stage involves the exclusion of tests that fail to compile, are flaky, or simply fail. Section 4.2.1 delineates the metrics that can be extracted at this stage of test generation.

The final step involves filtering the tests capable of execution and those that contribute value to the study. These tests were subsequently utilized to gather data on **fault detection**. Section 4.2.2 elucidates the metrics that aid in understanding the effectiveness of the approach in terms of fault detection. Unlike TG, FD is conducted on the **buggy** version of the code. The regression tests created earlier are executed to verify their capability to detect the bug.

The generated test suites further contributed to the collection of data on **fault localization**, facilitated by **GZoltar**. The output from *GZoltar* comprises a ranked list of suspicious lines of code. Given this information, one can infer that each bug will correspond to 5 such lists. Section 4.2.3 explains how to extrapolate meaningful data for fault localization from these results.

It was deemed essential to establish a **baseline** to gauge whether the approaches were indeed efficient at detecting bugs, and more importantly, if they assisted in localizing them. To this end, the results generated from the vanilla *EvoSuite* were declared the reference point for comparisons. As all other approaches claim superior bug detection capabilities, presuming these claims to be valid, it becomes critical to understand the efficacy of each approach in localizing those bugs compared to a more generic approach such as *EvoSuite*.

4.2 Metrics

Fault detection and fault localization are two distinct concepts, each possessing unique methods for assessing their effectiveness. The primary objective of fault detection (FD) is to ascertain whether the bug has been detected, while for fault localization (FL), the goal is to determine if the bug has been accurately localized. The ensuing sections provide detailed explanations of how metrics were established for each domain of the study. Both of these stages are prefaced by the test generation phase, during which the test suites are generated and subsequently filtered.

4.2.1 Test Generation

During the generation of tests, it is imperative to ascertain the quality of the generated tests and their ability to execute successfully. Although this is not a primary objective of this study, it does enable the collection of data on parameters such as **coverage**, **mutation score**, **compilation errors**, and more.

The most straightforward metric to comprehend is the number of tests generated. Despite its simplicity, this metric is of significant value as it enables us to ascertain if the approach generates an optimal, excessive, or insufficient number of tests.

Compilation errors are relatively common during the generation of tests. This phenomenon arises because the generated tests are not always syntactically correct. Similarly, test 'flakiness' is a prevalent issue, characterized by intermittent test failures and successes. Such irregularities can be attributed to concurrent execution issues or variations in the testing environment.

Understanding the number of tests discarded due to the aforementioned reasons is also vital. Essentially, this metric evaluates the approach's efficacy in generating tests that are capable of successful execution.

Coverage denotes the proportion of code lines executed by the test suite. This metric is integral to comprehending whether the test suite effectively exercises the code under test. Hence, it is beneficial to discern whether the generated tests can cover all lines of code. Failing to cover lines where the bug resides implies that the bug will remain undetected. Consequently, a higher coverage percentage is typically associated with improved fault detection outcomes [34].

Similarly, **mutation score** plays a critical role. This metric represents the proportion of source code modifications, or 'mutants', eliminated by the test suite. The mutation score carries comparable significance to coverage as it aids in determining whether the test suite exercises the code under test. Moreover, it allows for the evaluation of the test suite's capability to exercise edge conditions, such as assessing the outcome of altering a **<=** comparison to **<**. A higher mutation score is generally linked to better fault detection performance [34].

4.2.2 Fault Detection

Evaluating the efficacy of methodologies for fault detection may appear straightforward, primarily focusing on the number of bugs each methodology identifies. However, some finer points warrant

careful consideration.

One such detail pertains to understanding why a bug was detected by one approach and not another. This distinction may be chalked up to sheer chance or may stem from a systemic issue evident in the data. The inherent randomness of the approaches can result in certain bugs being detected only in some runs, which may not reflect the approach's capabilities accurately. To prevent such scenarios, it is essential to scrutinize the results meticulously.

Tests **can fail** due to assertions, exceptions, or timeouts. Assertions and exceptions hold the most significance, as they directly relate to bug detection. If a test fails due to a timeout, it does not definitively indicate bug detection. The test's inability to conclude could stem from various causes, such as code sections that require a significant time to execute or limited machine resources leading to protracted execution times that exceed the timeout threshold. Given the array of factors causing a timeout failure, such failures are not considered reliable.

With respect to fault detection, we can once again examine the **coverage percentage**, this time using the **buggy** version of the project. This second evaluation aims to understand whether the tests covered the buggy lines and, if so, why they did not detect the bug. Were the inputs insufficiently robust? Or were the inputs appropriate but the assertions inadequate? These crucial questions can be answered with the collected data. It is intriguing to explore the correlation between **coverage** and **FD**. Does a test suite with high coverage detect more bugs than a test suite with low coverage?

4.2.3 Fault Localization

The primary aim of **fault localization (FL)** is to ascertain whether the predicted bug locations correspond to the actual region in the code where the bug was introduced. *GZoltar* produces a ranked list of suspicious lines of code, thus assessing the approach's effectiveness merely necessitates verifying the bug location's position within this list.

The methodology followed in this study to evaluate the efficacy of the approaches for fault localization is based on the work presented by Pearson et al. [26].

Pearson et al. evaluated fault localization effectiveness by employing a metric called **Exam Score**. This metric can apply various levels of granularity — lines, methods, classes, or even packages — for score calculation. In this study, **line granularity** was deemed most suitable given that only tests for one class were generated at a time. The *Exam Score* signifies the probability that a particular line of code is the origin of the observed failure. This score is typically computed by contrasting the observed failure behavior with the program's anticipated behavior. By gathering execution data, such as test case outcomes (pass/fail) and code coverage information, the fault localization algorithm calculates scores for distinct code locations. Higher scores suggest a higher probability of a line being faulty, whereas lower scores indicate a lower likelihood of the line being the source of the failure.

4.3 Results

This section presents the findings of the study, which are categorized into three distinct sections corresponding to each area of investigation. The initial section details the results obtained from the test generation stage. Subsequently, the outcomes of the fault detection stage are delineated in the second section, while the final section discloses the results from the fault localization stage.

4.3.1 Test Generation

In this phase, our primary emphasis is on the volume of test cases generated by each approach, as well as the quantity of test cases that were subsequently eliminated. It is crucial to first scrutinize the vanilla variant of *EvoSuite*, which serves as the baseline for comparison. This configuration yielded a total of 151,257 test cases, of which 2,260 were found to be defective. This translates to a defect rate of approximately 1.49%, which is a commendable outcome.

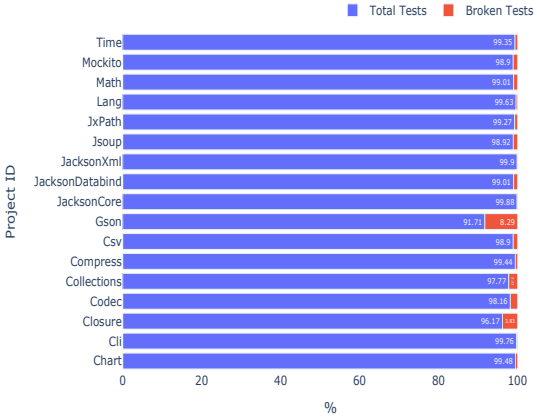
As demonstrated in Figure 4.2a, the vanilla variant of *EvoSuite* yielded a reasonable count of test cases, with a relatively low fraction of broken tests in most instances. The exception is the **Gson** project, where, on average, 8.29% of the tests generated were categorized as broken. This proportion is substantially high in comparison with other projects, in that sense it is important to understand why this occurs.

In a comprehensive analysis, it was determined that *EvoSuite* possesses a mechanism to identify inner classes and subsequently generate tests for them. The library employed by *EvoSuite* for Java bytecode manipulation designates inner classes with a dollar sign ($). Notably, certain defects within the **Gson** project are associated with a class labeled as *$Gson$Types*. These defects compromise the generated tests, given that the system misinterprets this class as an inner class and attempts to replace the ($) with a (.). This would be a valid operation for an actual inner class. However, the resultant issue is not merely the generation of tests with incorrect assertions but the production of tests that fail to compile. This shortcoming is inherent to *EvoSuite*, suggesting that subsequent approaches might also encounter this challenge.

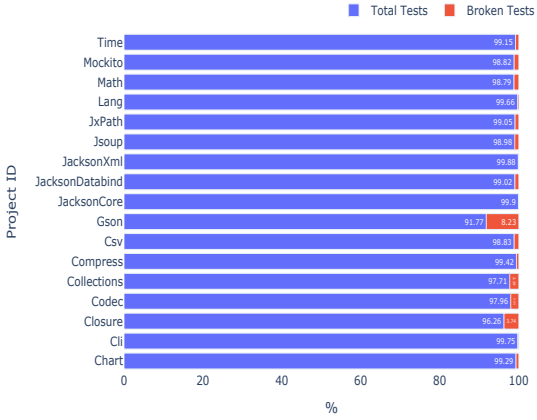
Nevertheless, vanilla *EvoSuite* had a positive result globally which is indicative of the approach's efficacy at generating runnable tests.

The results yielded by **EntBug**, as depicted in Figure 4.2b, display a substantial resemblance to those from vanilla *EvoSuite*. The percentage of broken tests shows minor variance, occasionally more, sometimes less. Regrettably, the findings related to the **Gson** project remain less than satisfactory. The proportion of broken tests remains exceedingly high, presenting a mean value marginally better than that from vanilla *EvoSuite*. But as it was said before, this is a problem of *EvoSuite* itself, it was already expected unsatisfactory results. Though **EntBug** generates slightly more tests, with an average total of 154,964 tests, it also records a higher number of broken ones, with an average total of 2,342 tests, which stands for 1.51% of broken tests. Very similar to vanilla *EvoSuite*.

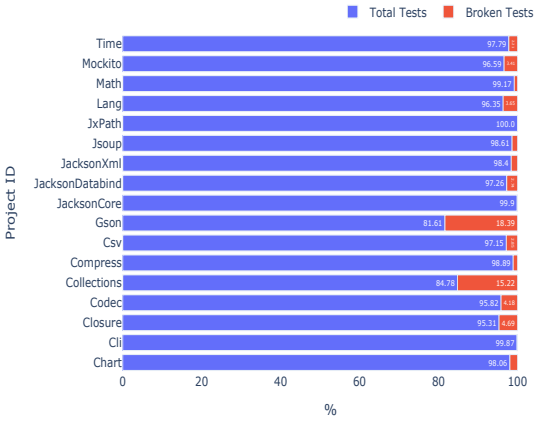
Figure 4.2c illustrates a considerable reduction in the number of generated tests when using **DDU**. Notably, **DDU** displays a greater level of conservatism compared to both vanilla *EvoSuite*



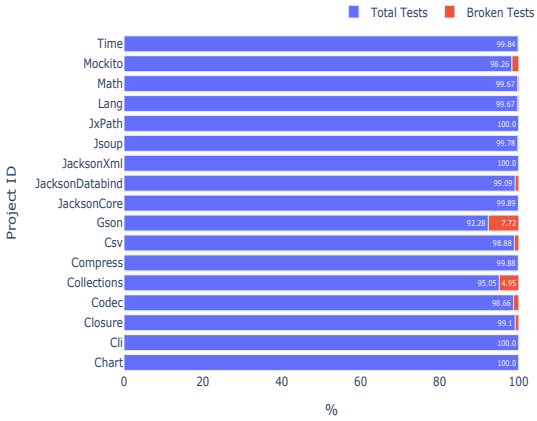
(a) Vanilla EvoSuite



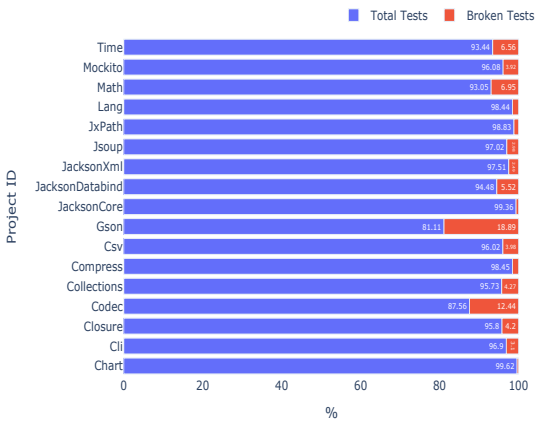
(b) EntBug



(c) DDU



(d) Ulysis



(e) PreMOSA

Figure 4.2: Broken Tests

and **EntBug** with only a total of 22,454 tests generated and 618 broken (2.75% of broken tests). This observation is of considerable interest as it underscores the more liberal nature of the other approaches in generating tests. The implications of this characteristic can be either advantageous or disadvantageous, contingent upon the results of FD and FL. Assuming that **DDU** is capable of detecting and localizing bugs with similar proficiency to the preceding two approaches, then **DDU** would demonstrate superior efficiency due to its generation of fewer tests, thereby consuming less time and resources.

The **Gson** project poses a substantial challenge to **DDU**. In comparison to earlier examined approaches, it exhibits a heightened proportion of broken tests, averaging at 18.39%. This increased percentage can be attributed statistically to **DDU** generating a smaller number of tests, thereby elevating the ratio of broken tests. However, the specific bugs implicated remain consistent when analyzed across different frameworks.

A subsequent observation pertains to the **Collections** project, which, when subjected to this approach, begins to exhibit suboptimal results. A significant 15.22% of the tests were found to be faulty, necessitating an exploration into the underlying causes. As previously noted, the central reason is the relatively lower number of tests generated by **DDU**. Specifically, **DDU** produced an average total of 58 tests, whereas the standard *EvoSuite* generated a substantially larger set of 646 tests. Nonetheless, both approaches exhibited broken tests, albeit in varying ratios.

It can already be observed that adopting a more conservative approach in test generation presents certain disadvantages. While **DDU** is optimized in this respect, it demonstrates challenges in test generation. At times, the proportion of successful tests generated by **DDU** can be statistically less favorable compared to other approaches, which can lead to less variability.

Figure 4.2d further illustrates a reduction in the number of tests generated, in this instance by **Ulysis**. Similar to *DDU*, the beneficial or detrimental nature of this reduction will be evaluated further. The average total number of tests produced stands at 20554, juxtaposed with an average total of 140 defective tests. This facet of the approach is noteworthy; **Ulysis** yields an exceptionally low count of malfunctioning tests, the lowest among all approaches under review, with only 0.68% being broken. This reduced count of tests, combined with the minuscule quantity of defective ones, suggests the approach's efficacy in test generation. It remains crucial to ascertain its proficiency in test detection and localization.

An additional noteworthy characteristic of *Ulysis* pertains to its performance on the **Gson** project. The approach exhibits a reduction in the percentage of non-functional tests, registering at 7.72%. Such results position *Ulysis* favorably, underscoring its proficiency in producing tests that compile and execute successfully for **Gson**.

The last approach under examination is illustrated in Figure 4.2e. **PreMOSA** produced an average of 57,372 tests, of which 2,278 were broken. These statistics are somewhat disconcerting, with 3.97% of the tests being broken. Notably, this constitutes the highest percentage of broken tests among all the approaches analyzed. Based on this preliminary assessment, it may be inferred that **PreMOSA** is less effective in generating tests that compile or execute successfully. It remains

imperative to further investigate its efficacy in bug detection and localization, and subsequently determine the approach's overall utility.

Consistent with the overall trend, **PreMOSA** encountered difficulties with the **Gson** project, yielding an average of 18.89% broken tests. Surprisingly, the **Codec** project posed a unique challenge, with 12.44% of its tests being broken. This marked the first instance of such suboptimal results for this project, thus necessitating a deeper exploration of the underlying causes. Closer scrutiny of the project results reveals that the bug *Codec-1* predominantly contributes to the high number of broken tests. This specific bug is related to a very long and complex method, replete with loops and conditional statements.

This complexity, combined with the fact that **PreMOSA** was provided with a list of faulty methods (with this method being the sole defective one), steered the evolution of tests predominantly towards this intricate method. Such a singular focus adversely affected the outcomes in terms of broken tests, as considerable time and resources were expended on this specific method. This overemphasis during test generation led to the creation of a higher number of flawed tests. While this approach is suboptimal for generating functional tests, it is conceivable that some of these tests might aid in the detection and localization of the bug.

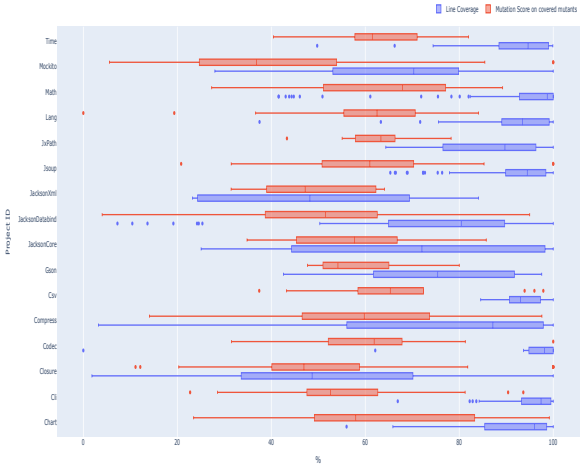
The results distinctly bifurcate the approaches into two categories. The first category encompasses vanilla **EvoSuite** and **EntBug**, while the second contains **DDU**, **Ulysis**, and **PreMOSA**. The former group displays a more aggressive approach to test generation, whereas the latter adopts a more conservative stance in terms of the quantity of tests. This differentiation is critical as it may significantly impact the results of both FD and FL.

It is noteworthy to mention that, in general, all the approaches produced a relatively small number of broken tests. However, the occurrence of such tests is not entirely absent. Consequently, it is crucial to ascertain whether integrating these approaches into a testing pipeline is advantageous or counterproductive. If these approaches are demonstrably effective in detecting and localizing bugs, their utilization might be justified. Nonetheless, one must remain cognizant of the potential frequency of broken tests they may generate.

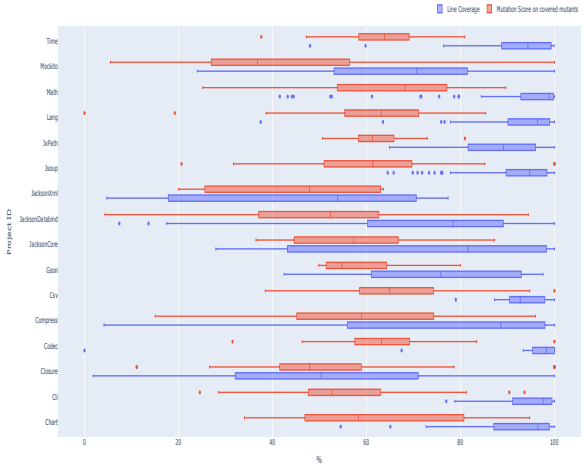
During the experiment, a discernible reduction in the number of available bugs was observed, attributed to varying complexities at different stages. In the current phase, there exists a probability that certain bugs might not yield results if they remain unexecuted within these approaches. In a synthesis of the present stage of the pipeline, both **DDU** and **Ulysis** generated data points for 833 bugs, **EntBug** accounted for 822, **PreMOSA** captured 831, and the standard *EvoSuite* tallied 823. It is imperative to highlight that a subset of these data points encompasses executions linked to broken tests, such data will be excluded in subsequent stages.

The process of test generation also yields valuable data pertaining to **coverage** and **mutation score**. It is important to remember that these metrics are derived from the **fixed version** of each bug. Figure 4.3a presents the results associated with vanilla *EvoSuite*.

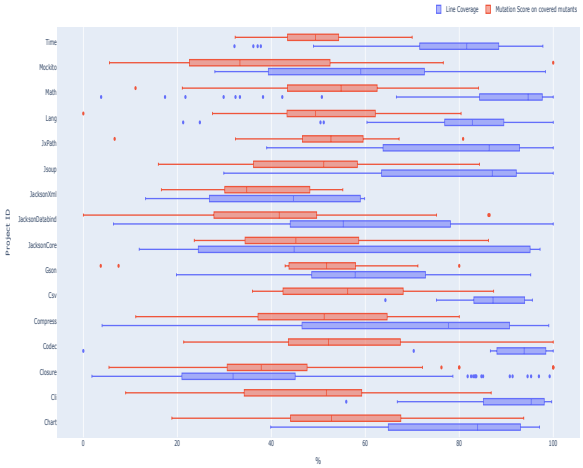
The chart illustrates a significantly high percentage of **line coverage** for the majority of the projects, with **JacksonXml** and **Closure** being the only two exceptions where improvements could be seen. The former exhibits a median coverage of merely 48.2% with a third quartile of 69.4%.



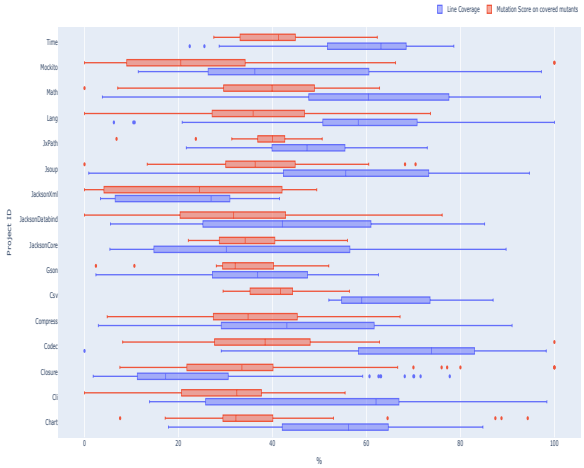
(a) Vanilla EvoSuite



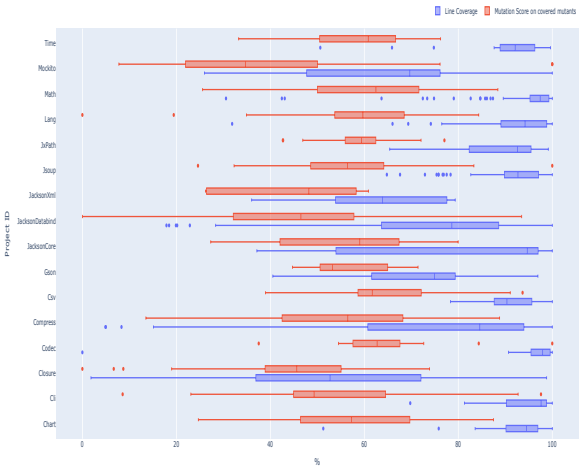
(b) EntBug



(c) DDU



(d) Ulysis



(e) PreMOSA

Figure 4.3: Coverage & Mutation Score

The latter demonstrates a median coverage of 48.7% and a third quartile of 70.1%. In academic literature, it is well-established that higher coverage typically contributes to improved FD outcomes. Consequently, it is hypothesized that enhanced FD and FL results should be observed for projects with higher coverage. However, corroborating this assumption within our research remains intriguing.

As anticipated, the percentage for **mutation score** is generally lower than that of *coverage*. Obtaining a high mutation score is acknowledged as a challenging endeavor, thus the observed outcomes align with expectations. The research literature also affirms that a higher *mutation score* is typically associated with superior FD outcomes. With the available FD and FL data, our study can examine the validity of this assertion. Despite the lower percentages of *mutation score* being expected, the projects **Mockito**, **JacksonXml**, **JacksonDatabind**, and **Closure** display worryingly low values. Among this group, *JacksonDatabind* exhibits the highest median value at 51.5%, which remains quite low. It presents an intriguing scenario to observe how these projects perform in terms of FD and FL.

When examining both *line coverage* and *mutation score*, the results suggest that these metrics are predominantly influenced by the complexity of the code under scrutiny. Consider, for instance, the **JacksonXml** project. Bugs labeled *JacksonXml-1* and *JacksonXml-3* pertain to the class *FromXmlParser*. The former registered an average line coverage of 24.3%, while the latter marked 23.3%. Conversely, *JacksonXml-6*, which is associated with the *ToXmlGenerator* class, boasted an average line coverage of 84.1%.

Upon a detailed examination of both classes, a stark disparity in their complexities becomes evident. The *ToXmlGenerator* class predominantly comprises straightforward getter and setter functions, with its methods generally being concise and encompassing rudimentary conditional statements and occasional loops. In stark contrast, the *FromXmlParser* class features two particularly intricate methods, namely *nextTextValue* and *nextToken*. Each of these methods spans approximately 100 lines of code, interwoven with multiple loops and conditional clauses.

This comparison serves as a salient illustration of the relationship between code complexity and metrics such as line coverage and mutation score. However, it is paramount to underscore that suboptimal results cannot be solely attributed to the approaches. Drawing from the aforementioned example, it becomes palpable that certain methods, especially the latter two, could benefit from refactoring to enhance clarity and maintainability.

Figure 4.3b presents the results for **EntBug**. The outcomes exhibit substantial similarity to those of vanilla *EvoSuite*, making it challenging to discern considerable disparities between the two charts. However, upon close inspection, minor differences do emerge. For instance, it can be observed that the distribution of both the *mutation score* and *line coverage* is more uniform in vanilla *EvoSuite* for the *JacksonXml* project. Conversely, the data for the *Csv* project appears to have a more balanced distribution in *EntBug*.

The quality of the tests generated by **DDU** appears to diminish, as evident in Figure 4.3c. Both the *line coverage* and *mutation score* percentages are observed to be inferior to the previous two approaches. As discussed earlier, the literature suggests a strong correlation between these

metrics and FD. Consequently, these findings do not portend well for *DDU*. Nevertheless, it will be intriguing to assess how the results for FD and FL unfold.

The underlying cause for this observed decline in quality is not immediately evident. However, one may postulate that, in contrast to preceding approaches, metrics like coverage and mutation score might not be paramount for **DDU**. This approach employs algorithms oriented towards assessing the efficacy of current tests in bug **localization**, potentially relegating the significance of more conventional metrics such as coverage and mutation score to a secondary role.

Ulysis demonstrates a further decline in test quality. As Figure 4.3d illustrates, the percentages for both metrics are even lower than those of *DDU*. Particularly alarming are the data for projects like *JacksonXml* and *Mockito*, where the percentages for both metrics primarily populate the lower end of the distribution.

This approach builds upon the foundational work of **DDU**, adopting a more advanced methodology wherein distinct universes are delineated and a singular component is deemed faulty within a given universe. This strategy diverges from an emphasis on conventional metrics, instead centering on the identification of optimal tests to aid in bug localization. This shift in focus partially explains the observed suboptimal outcomes.

It remains an open question whether relegating traditional metrics to a peripheral status enhances the efficacy of bug localization.

In Figure 4.3e, the outcomes for **PreMOSA** are delineated. The data bear a resemblance to those of vanilla *EvoSuite* and **EntBug**. Given that the foundational algorithm for **PreMOSA** is **DynaMOSA**, a parity in results with vanilla *EvoSuite* was anticipated. The integration of *DynaMOSA*, particularly targeting methods perceived to be more prone to faults, does not significantly alter the results. As evidenced by the data, certain projects depict metrics that lean in favor of **PreMOSA**, while others display a proclivity towards vanilla *EvoSuite*.

Given the preceding analyses, there exists a discernible divergence in priorities between vanilla **EvoSuite**, **EntBug**, and **PreMOSA** on one hand, and **Ulysis** and **DDU** on the other. The former group emphasizes code coverage and the generation of tests that probe boundary conditions, whereas the latter primarily aims to produce tests with a higher likelihood of localizing bugs.

This divergence in focus presents a compelling dimension for this study, as it potentially illuminates which approach is more effective for FD and FL.

In the subsequent phase of evaluation, certain bugs were excluded due to reasons ranging from compilation errors (observed in the prior stage) to issues pertaining to the approaches responsible for computing line coverage and mutation score. At this stage, *DDU* accounted for the highest number of bugs, totaling 773. This was succeeded by *Ulysis* with 770, vanilla *EvoSuite* with 745, *EntBug* registered 739, and *PreMOSA* concluded the list with 733.

4.3.2 Fault Detection

Shifting the focus toward the buggy version of the projects. At this juncture, the aim is to ascertain if the tests generated in the preceding stage can successfully identify the implanted bugs. By

deploying the **regression tests** formulated in the prior phase, executing them in the corresponding buggy version and verifying the detection of the bug.

There are instances where the implantation of a new bug disrupts the code compilation process. For instance, consider an extreme case where an entire method or even a class is eliminated. The *Java* compiler will halt the compilation due to the absent code during the compile time. Such an issue is prevalent and hence, comprehending the number of test suites that cannot be executed solely due to compilation impediments is fundamental.

As depicted in Figure 4.4a, it is observable that a significant percentage of test suites generated by *EvoSuite* in its vanilla form failed to compile. This scenario is particularly detrimental, considering the consequent loss of valuable data points that would otherwise enhance the comprehensiveness of the study. Each discarded test suite represents a missed opportunity to evaluate both fault detection and subsequent fault localization.

During this phase, vanilla *EvoSuite* encountered compilation errors for a total of 59 bugs. Consequently, subsequent FD and FL evaluations will be limited to an analysis of the remaining 764 bugs.

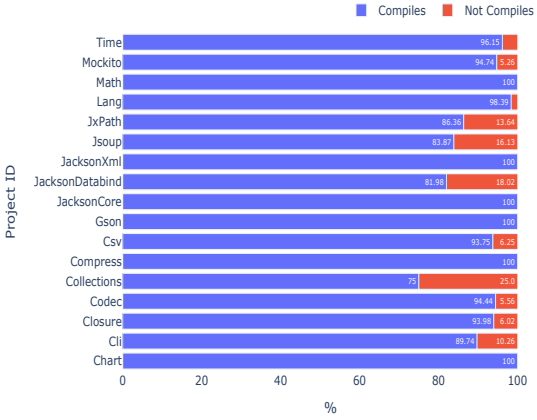
In the analysis of the results, the **Collections** project manifests the most pronounced percentage of non-compiling tests, with 25.0% of the test suites not compiling successfully. Upon a meticulous examination of the data, the primary cause of this phenomenon can be attributed to the project's limited number of bugs, specifically four. One notable bug involved a method that transitioned from a private to a protected access modifier between the buggy and the fixed versions. Test suites generated based on the fixed version, which treated the method as protected, faced compilation issues in the buggy version due to the method's private accessibility.

Similarly, the **JacksonDatabind** project exhibited compilation challenges. A case in point is the bug *JacksonDatabind-75*. This bug involves a modification in the method declaration by introducing an additional parameter. The resultant test suite, constructed based on the rectified method declaration, confronts compilation hindrances since it is incongruent with the original buggy version of the method.

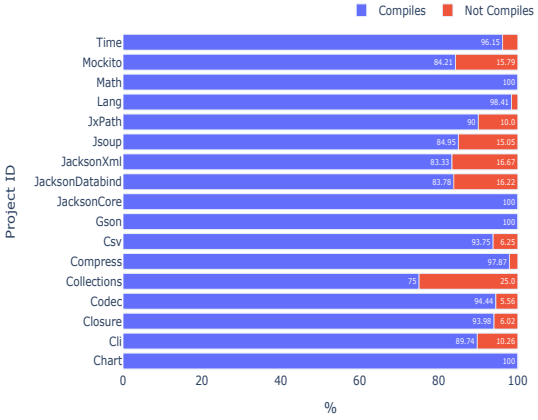
Upon examination of these compilation graphs, it becomes apparent that the performance across all approaches is comparably consistent, with identical projects demonstrating similar behaviors. In Figure 4.4b, it can be seen that **EntBug** presents *Collections* as the project suffering the highest proportion of compilation errors, as was the case with vanilla *EvoSuite*. Despite that, **EntBug** dropped more data points than vanilla *EvoSuite*, with 61 bugs that could not be executed.

For **DDU**, as depicted in Figure 4.4c, the proportion of test suites that fail to compile diminishes for certain projects. Notably, the *Mockito* project, which previously exhibited a moderate rate of non-compilation, now demonstrates a significant 39.47% of tests failing to compile.

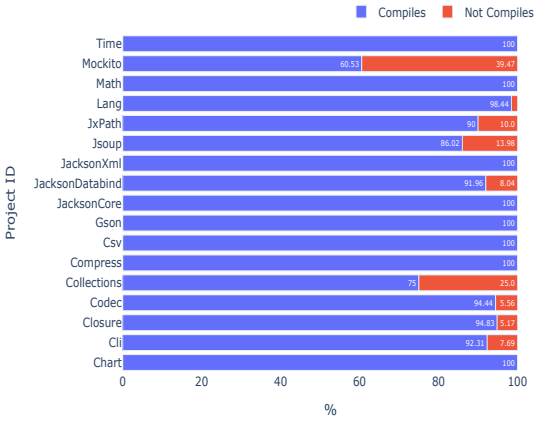
Upon a detailed examination of the results, the specific cause for this surge in non-compiling tests remained elusive. However, it is evident that **DDU** evolves its tests in a manner that appears more susceptible to compilation errors compared to the preceding approaches. For instance, vanilla *EvoSuite* consistently generates tests for *Mockito* 4 to 8 without encountering any compilation issues. Such outcomes were anticipated, given the absence of syntactical issues associated



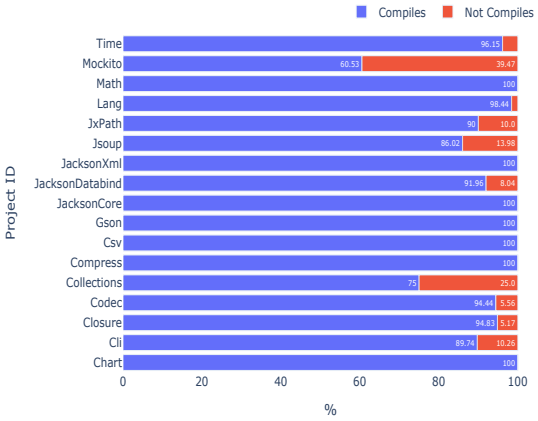
(a) Vanilla EvoSuite



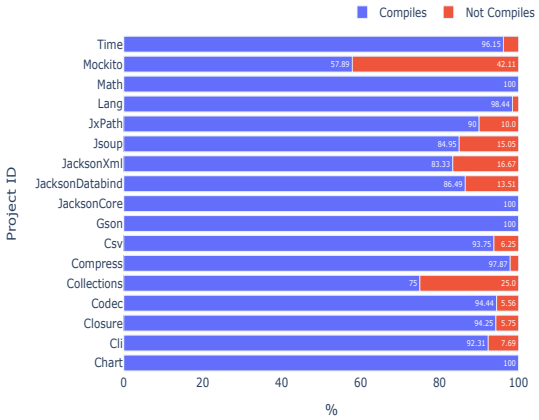
(b) EntBug



(c) DDU



(d) Ulysis



(e) PreMOSA

Figure 4.4: Compiled Bugs

with these bugs. Contrarily, **DDU** fails to produce tests for these bugs, a limitation that underscores potential issues within its test evolution algorithm. Although, the approach maintains the most bugs across all approaches, with only 54 non-compiling bugs.

A similar scenario unfolds for **Ulysis**, as illustrated in Figure 4.4d, with *Mockito* again constituting the highest percentage of compilation errors at an alluring equal of 39.47%. This observation is consistent with the understanding that **Ulysis** represents an advancement of the work initiated by **DDU**. Consequently, they share similar vulnerabilities, as underscored in this research. Similarly, **Ulysis** also does a good job at keeping data points, with only 56 bugs that could not be executed.

As depicted in Figure 4.4e, **PreMOSA** manifests the least favorable performance among all the approaches assessed. Specifically, the *Mockito* project remains particularly challenging for this approach, registering a compilation failure rate of 42.11%. Correspondingly, **PreMOSA** also exhibits the highest number of discarded bugs, totaling 67.

The worst results observed for **PreMOSA** can be attributed to its inherent knowledge of the faulty components, for which it specifically seeks to evolve tests. In contrast, other approaches lack this specific information and thus aim to evolve tests for all components indiscriminately. This broad focus might inadvertently confer an advantage, as these approaches could bypass some of the intricate challenges that **PreMOSA** confronts.

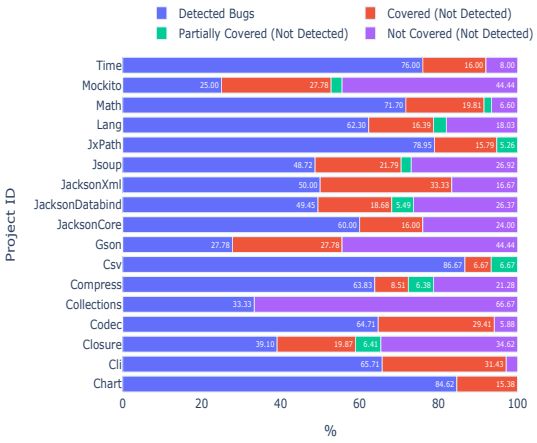
Turning the attention to the critical metric of **fault detection**, Figure 4.5a reveals a commendable performance of *EvoSuite* in its vanilla form, with a sizeable proportion of bugs detected. This is particularly evident in projects such as *Csv*, *Chart*, and *JxPath*, where the detection percentages surpass 78%. Conversely, projects such as *Mockito*, *Gson*, and *Closure* demonstrate evident difficulties in bug detection, none reaching a detection percentage exceeding 40%. Two bugs from *Collections* went undetected, while the one that remained was detected successfully. The approach exhibited the third-highest bug detection rate, identifying 424 out of 764 bugs. This translates to a detection rate of 55.5%.

A somewhat encouraging insight drawn from this data is the recognition that, despite the failure to detect some bugs, a significant proportion of these bugs are still covered by the tests. This understanding is vital, as it confirms that the tests are indeed probing the flawed code, thereby emphasizing the necessity to refine the assertions for more effective bug detection.

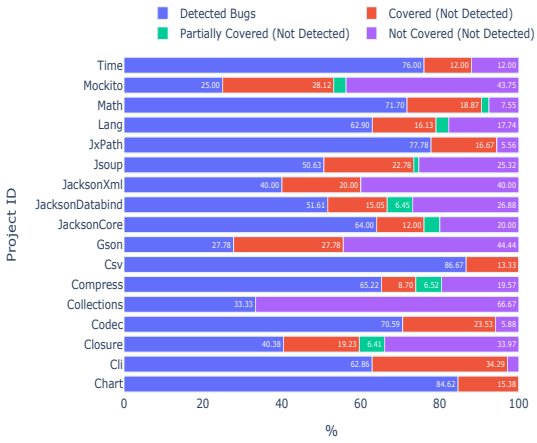
However, it is crucial to recognize that a significant proportion of bugs continue to not be detected. Having an overall detection rate of 55.5% is less than optimal, implying that the tests generated by vanilla *EvoSuite* do not effectively identify the bulk of the bugs.

EntBug's performance, as demonstrated in Figure 4.5b, once again closely mirrors that of vanilla *EvoSuite*. The analysis reveals that *EntBug* excels and faces difficulties in the same projects as its vanilla counterpart. **EntBug** shows a similar performance to vanilla *EvoSuite* in the total number of bugs detected, identifying 430 out of 761 bugs, resulting in a detection rate of 56.5%, the highest overall.

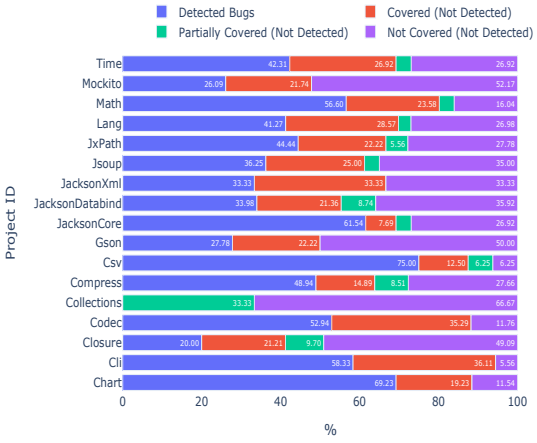
The approach has been predominantly evaluated in the context of FL. However, it stands to reason that to localize a bug, it must first be detected. For instance, *Codec-7* from the *Defects4J*



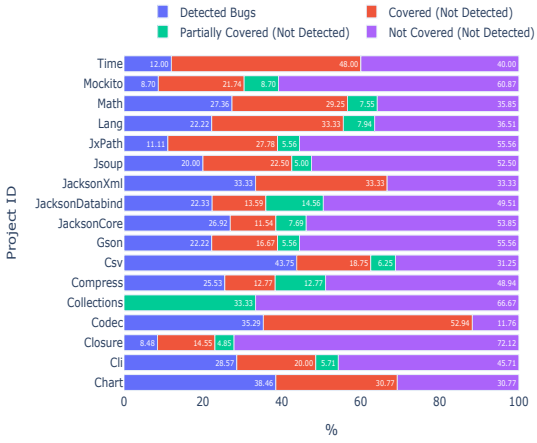
(a) Vanilla EvoSuite



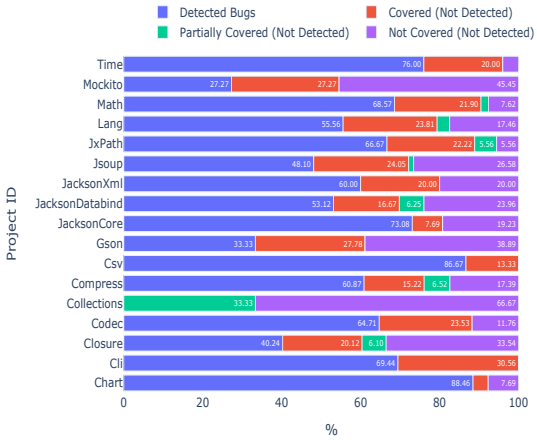
(b) EntBug



(c) DDU



(d) Ulysis



(e) PreMOSA

Figure 4.5: Detected Bugs

repository [32] corresponds to bug 99 in the *Codec* project [4]. Gratifyingly, this bug was identified in our study, aligning with expectations given its emphasis in the original paper [8]. Similarly, *Compress-7*, corresponding to bug 114 of the *Compress* project [5], and *Math-27*, relating to bug 835 of the *Math* project [6], were both detected. *Math-9*, which is linked to bug 938 in the *Math* project was also detected.

Of the four bugs referenced in the original paper, all four were successfully detected, marking a positive outcome. The status of the other bugs from the original study remains unresolved, as they were not subjected to this experiment.

A shift in performance can be discerned with **DDU**, as illustrated in Figure 4.5c. Notably, the detection rate of bugs is considerably low when compared with the preceding two approaches. While maintaining some level of proficiency in projects like *Chart* and *Csv*, **DDU**'s overall results leave much to be desired. All projects reflect a bug detection rate of less than 75%. A concerning observation is a substantial increase in the percentage of **non-covered** bugs. This suggests that the tests generated by **DDU** fail to address the lines of code associated with the bugs. This aspect warrants serious consideration as a test that fails to cover the buggy code is fundamentally incapable of detecting the bug.

The foundational paper introducing the approach [28] does not offer directly comparable data to our findings. However, it does present a list of bugs sourced from the *Defects4J* repository [32]. This list served as a benchmark to assess the efficacy of our results against the prior version of *Defects4J*. Employing the complete dataset, **DDU** managed to identify 314 out of the original 779 bugs, accounting for 40.3% of the total. The authors of [28] incorporated 186 bugs in their study, with our experiment excluding 4 from this number, leading to an intersection of 182 bugs. Based on the present data points, **DDU** identified 69 bugs, amounting to 37.9% of the total. While these findings are congruent with the initial paper, they register a marginal decline. It is crucial to consider that the original study utilized a distinct version of *EvoSuite*.

The results derived from **Ulysis**, as presented in Figure 4.5d, demonstrate a marked deterioration in performance compared to *DDU*. The detection rate of bugs is decidedly low, falling under 44% for all projects. This approach yields rather unfavorable results; most notably, it does not detect any bugs *Collections* and very few for *Mockito* and *Closure*. Adding to this, the percentage of **non-covered** bugs is significantly high. Several factors could contribute to these outcomes. Firstly, it is noteworthy that this approach/technique was incorporated into the latest version of *EvoSuite*. The evolution of *EvoSuite* from its initial version, where *Ulysis* was originally developed, might have led to critical divergences, resulting in subpar performance. Furthermore, human error associated with migrating the technique to the latest version of *EvoSuite* may have induced certain discrepancies, thereby influencing the results negatively. Despite these possible justifications, it is imperative to acknowledge that the observed results were indeed disappointing.

The paper introducing **Ulysis** [11] does not provide specific data regarding FD, nor does it offer any directly comparable metrics. Nevertheless, the performance of **Ulysis** in bug detection appears to be markedly suboptimal. Out of a total of 777 bugs, the approach only identified 161, constituting a mere 20.7% of the entirety. Such a low detection rate raises questions about the

approach's efficacy. While potential reasons for this performance were previously explored, it's unequivocal that **Ulysis** is not yet equipped for real-world applications.

In Figure 4.5e, **PreMOSA** displays commendable results, even somewhat superior in some projects when juxtaposed with vanilla *EvoSuite*.The outcomes remain appreciably respectable.

However, the paper introducing **PreMOSA** [27] posited distinct findings. The authors contended that **PreMOSA** could detect 8.3% more bugs than the standard test evolution algorithm, *DynaMOSA*. In contrast, our results illustrate that **PreMOSA** detected 427 out of 764 bugs, yielding a detection rate of 55.9%. This rate is 0.4% higher than that of vanilla *EvoSuite*, which employs the same algorithm as the baseline.

It's noteworthy to mention that since the publication of the original paper, both *DynaMOSA* and *EvoSuite* have undergone developments. This evolution might account for the disparity in performance observed for the approach. Furthermore, the contemporary *Defects4J* dataset encompasses a broader range of bugs, leading to potential new scenarios that **PreMOSA** might not have been originally designed to handle. Another crucial aspect to consider is the integration of **PreMOSA** with the latest iteration of *EvoSuite*, which includes the foundational algorithm. It's plausible that this algorithm may not be optimally suited to the current version of *EvoSuite*.

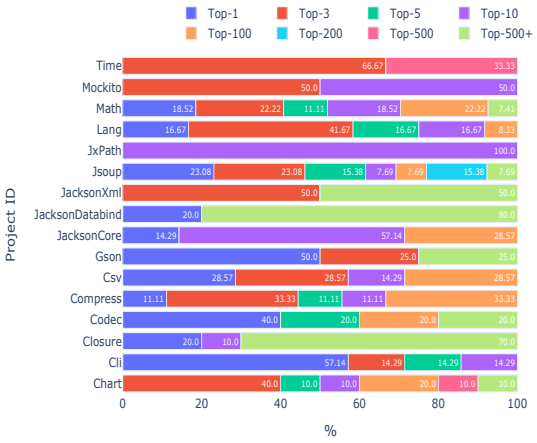
Contrary to expectations, *EntBug* emerged as the most proficient approach in terms of FD. While it was anticipated that approaches like **PreMOSA**, with its knowledge of faulty components, would surpass vanilla *EvoSuite* in the number of bugs detected by some commendable margin, the results did not corroborate this assumption.

4.3.3 Fault Localization

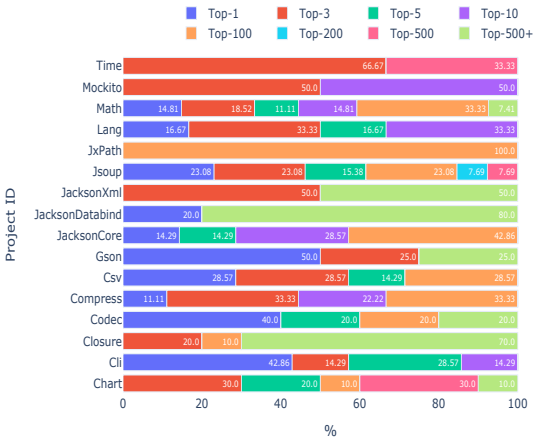
The final phase of the investigation involves Fault Localization. In this stage, we endeavor to ascertain whether the bug location predictions are capable of correctly identifying the code section where the bug was originally introduced.

It is essential to clarify that the subsequent analysis focuses on the intersection of the bugs identified by all approaches. Specifically, the evaluation will encompass only 124 bugs that every approach could pinpoint. However, vanilla **EvoSuite** produced location predictions for 411 bugs, **DDU** for 310, **Ulysis** for 158, **EntBug** for 420, and **PreMOSA** for 421. Given this context, certain approaches may exhibit inferior results due to the selection of bugs, although the converse could also be true. Comprehensive results for all bugs are available in the appendix, offering a comparative perspective on approach performance.

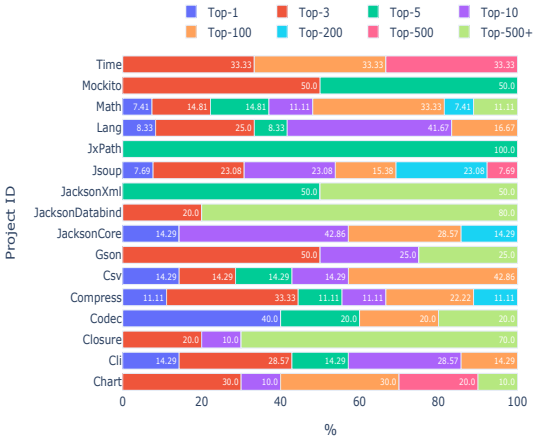
Figure 4.6a presents the findings for vanilla **EvoSuite**. The data reflect a relatively successful outcome for most projects. A good percentage of the bugs are located within the first 10 suspected lines of code. There are, however, a few exceptions. For instance, in the case of *JacksonDatabind*, 20% are located in the first position. However, the remaining bugs are located below 500. *Closure* and *JacksonXml* have also similar disappointing results. A significant portion of bugs is found towards the lower end of the ranked list of suspicious code lines, suggesting that most bugs were detected beyond the top 500. This kind of result is undesirable as, in a real-world scenario, it may not aid developers in bug detection and could potentially compound confusion.



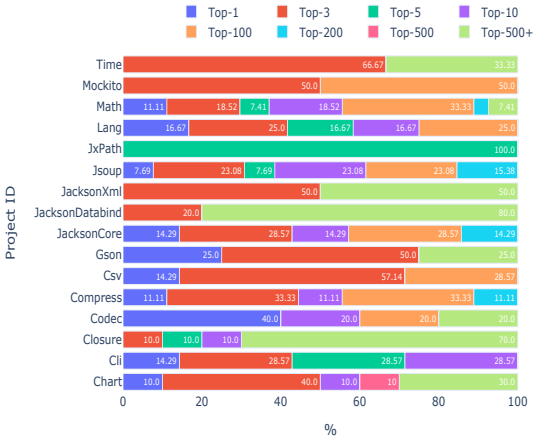
(a) Vanilla EvoSuite



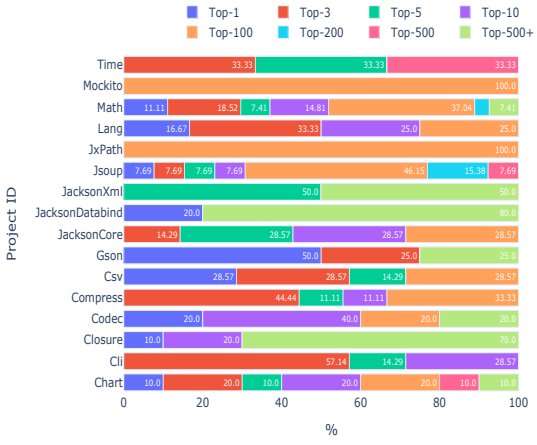
(b) EntBug



(c) DDU



(d) Ulysis



(e) PreMOSA

Figure 4.6: Located Bugs

The outcomes for **EntBug** 4.6b are, predictably, quite comparable to those of vanilla *EvoSuite*. However, *EntBug* exhibits a slight disadvantage in most projects. Notably, in the *Csv* project, **EntBug** surpasses vanilla *EvoSuite*. In this context, **EntBug** identifies 14.29% of the bugs within the top-5 rankings, whereas vanilla *EvoSuite* locates them within the top-10 rankings.

The study presented in [8] primarily emphasized the entropy of tests, thus leaving no direct metric for result comparison. Nevertheless, an examination of bug localization in our current experiment against the ones mentioned in the paper is feasible. Specifically, *Codec-7*, *Compress-7*, and *Math-9* are located in the first position. In contrast, *Math-27* is situated further down, occupying the 27th position.

In the original study, the bugs are accurately pinpointed with *Math-27* serving as the sole exception. This suggests that the approach retains its capacity for precise bug localization. However, the increased complexity and quantity of current bugs might be influencing its overall performance.

As anticipated, **DDU** presents a decline in the quality of FL outcomes. Previous data indicated a possible decline, and Figure 4.6c further confirms that this approach, on the whole, was less proficient at locating bugs compared to the preceding two. Several notable observations arise from this analysis.

DDU, while not particularly adept at pinpointing the bug within the initial suspected lines of code, demonstrated a reasonable proficiency at mitigating the presence at the lower end of the ranked list. As depicted in Figure 4.6c, the percentage of bugs discovered in the top 500+ is lesser compared to vanilla *EvoSuite* and *EntBug*. This indicates that the approach succeeds in shrinking the lower end of the ranked list of suspicious lines of code, although it falls short in locating the bug within the initial suspected lines of code.

Lastly, it is worth noting that, *DDU* excels in the *Mockito* project. The approach manifests superior performance in this project, locating one more bug within the top 5 while vanilla *EvoSuite* locates it in the top 10. But is also important to note that only 2 bugs were located for this project, which is a very small sample size.

Turning the attention to the analysis of **Ulysis**. Given the previous results, it would be prudent not to anticipate favorable outcomes. Regrettably, Figure 4.6d confirms this expectation. The majority of projects indicate a deterioration in quality when compared to the vanilla *EvoSuite* and **EntBug**.

The approach's performance is notably subpar, especially regarding the sample size it produces. Specifically, **Ulysis** located a mere 158 bugs, the smallest count among all evaluated approaches. This limited sample further restricts the observations of its comparative efficacy with other approaches. Such lackluster performance further reinforces reservations against its utilization, especially when considered alongside its other deficiencies.

The results for **PreMOSA**, as depicted in Figure 4.6e, are less than promising. Across most projects, the approach's performance lags behind that of vanilla *EvoSuite*, with the sole exception being the *Chart* project.

In the current dataset, the *Chart* project encompasses a total of 10 bugs. Notably, for one of these bugs, **PreMOSA** located the bug precisely at the first position, whereas vanilla *EvoSuite* identified it within the top three positions. This stands as the singular project where **PreMOSA** demonstrates a superior performance.

One might anticipate that **PreMOSA**, given its knowledge of where to search for bugs, would outperform vanilla *EvoSuite* in terms of bug localization. However, this expectation was not realized, which is unfortunate given the potential utility of this approach.

It is pertinent at this juncture to examine the distribution of results to ascertain which approaches normalize their bug localization more effectively. The kernel density estimation for vanilla **EvoSuite** and **DDU** is depicted on a logarithmic scale in Figure 4.7a. Observationally, vanilla **EvoSuite** demonstrates a superior distribution, characterized by a pronounced peak and reduced variance across the majority of projects.

Examining the peaks from vanilla *EvoSuite* reveals a trend where they often correspond to an exam score of one. This implies that the bug was identified in the first position. However, projects like *Closure* and *JacksonDatabind* raise concerns due to their peaks aligning with considerably elevated Exam Score values. The *Math* project presents a dichotomy: certain bugs manifest excellent results, while others are disappointing. Yet, reassuringly, the magnitude of the peaks representing commendable results surpasses those of less favorable outcomes. This suggests a higher number of successfully located bugs as opposed to unsuccessful attempts.

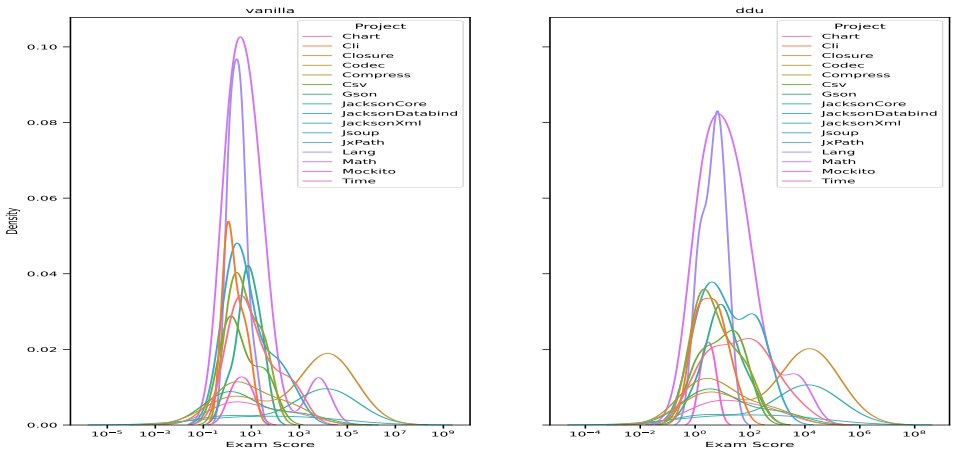
In juxtaposing vanilla *EvoSuite* with **DDU**, it becomes evident that **DDU** generally underperforms. Furthermore, it exhibits similar trends as observed for the *Closure* and *JacksonDatabind* projects.

EntBug's performance parallels that of vanilla *EvoSuite*, as delineated in Figure 4.7b. Primarily, the approach's results cluster around an Exam Score of 1. However, unlike vanilla *EvoSuite*, most projects lack pronounced peaks, suggesting that **EntBug** often fails to locate bugs in the first positions. Furthermore, a noticeable shift towards the spectrum's right end implies that **EntBug** frequently ranks bugs lower on its list.

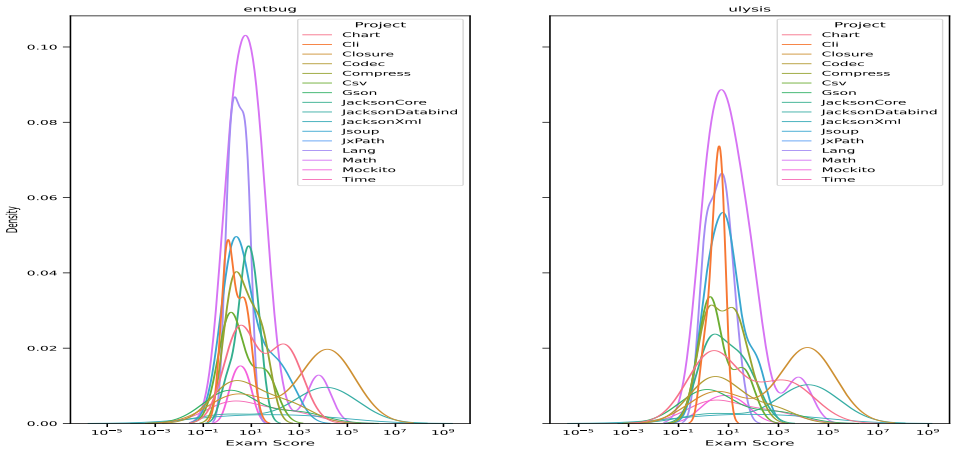
In Figure 4.7b, **Ulysis** exhibits enhanced peak prominence for several projects, showcasing admirable outcomes. However, a discernible observation is that these peaks correspond to worse Exam Scores, especially when juxtaposed against vanilla *EvoSuite* and *EntBug*. Notably, **Ulysis** crafted tests for the fewest bugs, signaling potential inadequacies in its test generation capabilities. Nonetheless, on the occasions it succeeds, its performance is reasonably satisfactory.

Figure 4.7c depicts the kernel density estimation for **PreMOSA** in comparison with test suites crafted by human developers. **PreMOSA**'s outcomes are noticeably underwhelming, exhibiting a sample distribution that doesn't gravitate toward favorable Exam Scores. This is particularly evident for the *Jsoup* project, where a pronounced distribution suggests a propensity toward higher Exam Scores, indicating a diminished proficiency in bug localization.

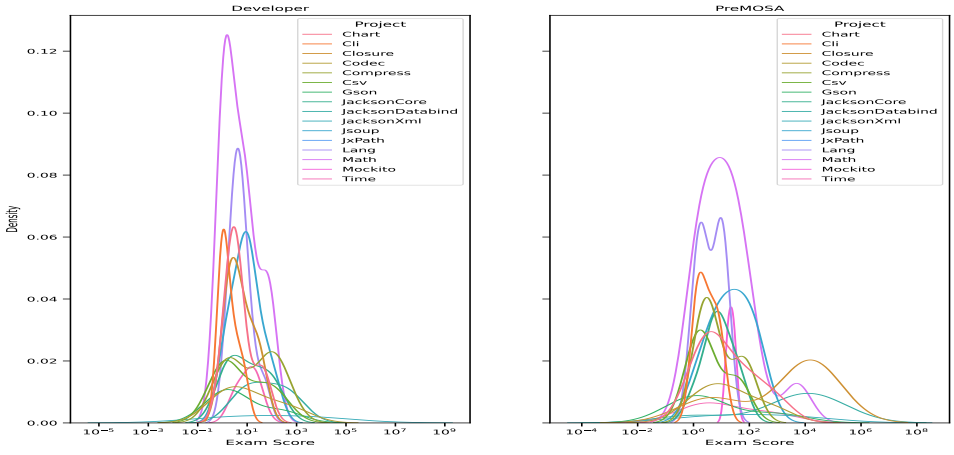
The distribution of results for test suites crafted by human developers is notably commendable, demonstrating the most consistent performance in comparison to all other approaches being evaluated. These insights will be instrumental in addressing the research question **RQ3**.



(a) Vanilla EvoSuite & DDU



(b) EntBug & Ulysis



(c) Developer & PreMOSA

Figure 4.7: Kernel Density

4.4 Result Evaluation

Based on the findings thus far, it is already possible to address the research questions delineated in section 1.4.

In response to research question **RQ1**, specifically, *"To what extent does the employment of defect prediction-based test suites assist in fault localization?"*, the findings indicate a nuanced outcome. When juxtaposing the performance of vanilla **EvoSuite** against **PreMOSA**-the latter being primarily designed to harness defect prediction-based test suites—vanilla **EvoSuite** exhibits superior efficacy in fault localization. Specifically, the baseline version identifies 20.2% of the bugs in the primary position, in contrast to **PreMOSA**'s 11.3%. Within the interval of one to three, vanilla **EvoSuite** accounted for 23.4% while **PreMOSA** logged 20.2%. Notably, both approaches converge at an 8.9% rate of bug localization within the top-5 rankings.

Upon examination, the performance metrics for *PreMOSA* appear to be less than optimal. One might argue that identifying bug locations beyond the top-3 rankings holds limited utility for developers. Considering this perspective, vanilla *EvoSuite* consistently outperforms across the board. This observation thus suggests that the adoption of defect prediction-based test suites does not inherently enhance fault localization.

The other approaches in consideration also integrate algorithms tailored for bug localization. However, as illustrated in section 4.3.3, they fall short of outperforming vanilla *EvoSuite*. Notably, **EntBug** demonstrated noteworthy results, yet it still did not exceed the established baseline.

The second research question, **RQ2**, inquires: *"Does the utilization of defect prediction-based test suites enhance fault localization compared to developer-based test suites?"*. Analyzing outcomes from section 4.3.3 and the localization data in Figure 4.8, it's discernible that the response is not straightforward. While certain developer-written tests outperform in FL metrics, other instances reveal superior results from defect prediction-based suites. For a more specific comparison, consider *EntBug*, which is tailored for pinpointing faulty components during test generation. This approach detects 49 out of 124 bugs within the top 3 positions of the suspicious code line ranking, in contrast to developer test suites that identify 53 bugs. This translates to *EntBug* having a 39.5% accuracy in the top 3 rankings, whereas developer suites achieve a 42.7% accuracy.

While developer test suites exhibit superior results in the given instance, it's imperative to highlight that the identified bugs differ between the two, with unique bugs detected exclusively by each method. Intriguingly, vanilla *EvoSuite* identifies an identical count of 64 bugs within the top 5 out of a total of 124. Hence, the efficacy may vary depending on the project, implying a potential synergistic approach using both methods.

A pertinent question arises regarding the relationship between coverage and mutation score in relation to fault localization. **RQ3** seeks to elucidate: *"Is there a correlation between coverage and mutation score with enhanced fault localization?"*. Comprehensive results, detailed in appendix A.1, offers a holistic perspective devoid of bug-specific influences. Notably, a discernible correlation exists between code coverage and fault localization efficacy. An analysis of the data in section 4.3.1 underscores that projects boasting superior code coverage concurrently exhibit

optimal fault localization outcomes. Projects such as *Mockito* and *JacksonXml* underscore this correlation, as their relative code coverage is inferior compared to other projects, mirroring their suboptimal performance in fault localization.

Regarding mutation score, a parallel observation emerges. Both aforementioned projects register as the least proficient in terms of mutation score, and consequently, they also exhibit subpar fault localization results. In summation, a definitive correlation exists between **code coverage** and **mutation score** in relation to enhanced fault localization. Such an inference aligns with anticipatory reasoning; comprehensive program execution across diverse situations logically facilitates more precise bug localization.

Do superior detection outcomes necessarily imply enhanced localization results? **RQ4** investigates this interplay by posing the question: *"Is there a correlation between fault detection and fault localization?"*. The empirical evidence does not suggest a direct correlation between these two metrics. For instance, while the *Gson* project excels in fault localization, its performance in fault detection ranks near the bottom among all approaches assessed. In contrast, the *Chart* project, despite demonstrating robust fault detection capabilities, fares suboptimally in fault localization. These observations underscore that the two metrics, although related, address distinct aspects of the problem space, and proficiency in one does not guarantee success in the other.

An essential facet for examination is the number of tests produced and their efficacy in both the detection and localization of faults. Research Question 5, denoted as **RQ5**, poses the question: *"Does the volume of tests generated correlate with the effectiveness in fault detection and fault localization?"*. As documented in section 4.3.1, a distinct variation was noted in the number of tests produced by each approach. Particularly, **DDU** and **Ulysis** generated the fewest tests, aligning with their inferior performance in FD. In the domain of FL, **Ulysis** yielded satisfactory results, yet both approaches still ranked among the least effective. An additional point concerning **Ulysis** is its unique behavior of generating tests targeting only 158 faulty components, results substantially unsatisfactory relative to its counterparts. Inferred from these findings, there is a tangible association between the number of tests produced and the efficiency in fault detection and localization. Predominantly, approaches that generated an extensive quantity of tests demonstrated enhanced performance across both evaluative metrics.

In addressing the sixth research question, denoted as **RQ6**, the inquiry arises: *"Do the approaches complement each other? Should we use them together?"*. The overlap of bugs located primarily by each approach is depicted in Figure 4.9. It is evident from the data that **EvoSuite** and **EntBug** identify almost identical bugs, suggesting minimal added value when combined. Notably, **Ulysis** is singular in its capability to locate three bugs that remain undetected by the other approaches, positioning it as a potential complement to the aforementioned approaches. The concurrent application of these approaches carries merit. However, the prioritization methodology for the results from each approach remains ambiguous. Consequently, while certain approaches present complementary attributes, the current absence of a definitive result prioritization framework makes their integrated utilization challenging.

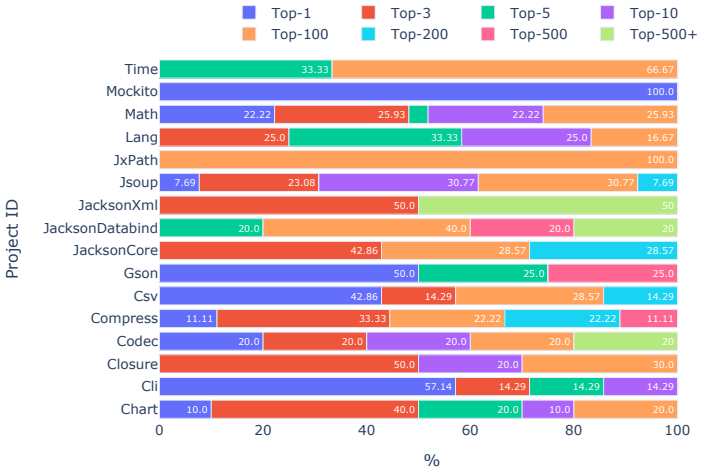


Figure 4.8: Fault Localization on Developer Test Suites

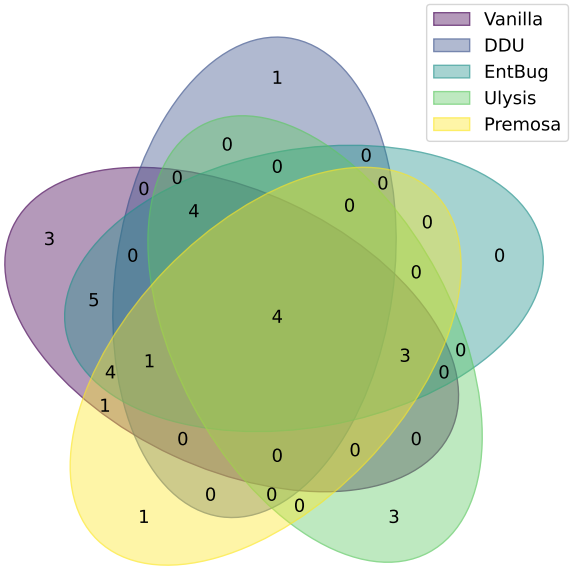


Figure 4.9: Venn diagram of Bugs Located on the First Position

4.5 Threats to Validity

The presented results come with certain limitations that warrant acknowledgment. Notably, all the approaches were updated to the latest version of *EvoSuite*. Such an update could introduce various implications. To illustrate, every approach in this analysis was originally developed for earlier versions of *EvoSuite*. The current iteration might exhibit behaviors that may not necessarily align with the design intents of these approaches. Furthermore, the manual process required for updating could introduce human errors into the approaches. Moving forward, it would be prudent to evaluate the native version of each approach.

Another dimension to consider pertains to the quantity of bugs included in the study. Earlier studies utilized fewer bugs from the *Defects4J* repository [32]. Employing all 835 bugs from the repository might yield unforeseen outcomes, given that the approaches weren't specifically tailored for this expanded set. Had they been benchmarked with this volume of bugs initially, the design approach for some approaches might have diverged.

Each approach possesses its unique configuration. For optimal outcomes, they need to operate with the most suitable set of parameters. This study couldn't experiment with varying configurations for each approach. Thus, when it came to parameter tuning, the study leaned on the prior work of *Shamshiri et al.* [34], running each approach with the advised parameters. However, certain specific parameters exclusive to some approaches were exempted from this approach.

Chapter 5

Conclusion

This thesis aims to critically assess the efficacy of test suites generated via the integration of defect prediction tools in accurately locating bugs. A pivotal component of this research was predicated on achieving robust fault localization (FL) outcomes from **PreMOSA**, given its alignment with the foundational premise of the thesis. Contrary to expectations, as delineated in the section 4.3.3, *PreMOSA* exhibited challenges in effective bug localization, with outcomes not surpassing the established baseline. It was postulated that the incorporation of a defect predictor would augment the results by offering specific insights into the bug's location. The underlying rationale was that if an approach possesses prior knowledge of which components are buggy, the resultant test suite should be inherently more adept at bug localization. Regrettably, this was not observed with *PreMOSA*. The potential underlying factors for this discrepancy are explored in detail in section 4.3.2, further emphasizing the unanticipated outcomes in the area of fault detection.

While the other approaches under consideration do not place an equivalent emphasis on defect prediction, they incorporate algorithms intended to assist in bug localization. Of these, **EntBug** is the sole approach that exhibits commendable performance in both FL and FD. However, its results do not surpass those of the established baseline for either metric. This underscores the potential avenues for enhancement in the predictive capacity of these approaches concerning test suite performance in FL.

A notable observation pertains to the performance of the baseline variant, vanilla **EvoSuite**. Astonishingly, it outperformed other specialized approaches despite lacking specific optimizations for bug localization. This reflects the variant's evolutionary progress over the years, and its ability to competently locate bugs without targeted specialization. However, the results from all evaluated approaches remain suboptimal for practical applications. For integration into real-world development environments, tools must precisely pinpoint bug locations, ideally ranking them at the top of the list of suspicious code lines. Given that none of the assessed approaches consistently achieved this, there remains a compelling argument against their immediate adoption in real-world scenarios.

5.1 Future Work

Going forward, there remain aspects related to this research that warrant further exploration. Crucially, assessing the original versions of each approach used in this study is essential. Given the modifications made to update them to the most recent version of *EvoSuite*, testing their initial forms would offer insights into their inherent efficacy.

As evident from the results section 4.3, nearly all approaches exhibited somewhat surprising outcomes, not aligning with the superior performance reported in their respective original papers. It might prove advantageous to re-evaluate these methodologies under their initial settings, such as using the original bug count and adhering to the original parameter configurations.

Regrettably, the study couldn't assess a wider range of approaches. Notably, there's commendable research in this field; for instance, Hershkovich et al.'s [20] approach, *Quadrant*, showed significant promise and would have been intriguing to evaluate. Unfortunately, due to time constraints, its assessment was omitted. This is just one of many examples, as numerous other approaches will provide further insights into the domain.

References

[1] David Abramson, Ian Foster, John Michalakes, and Rok Sosic. Relative debugging and its application to the development of large numerical models. *Proceedings of the ACM/IEEE Supercomputing Conference*, 2:1405–1418, 1995.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J.C. Van Gemund. An evaluation of similarity coefficients for software fault localization. *Proceedings - 12th Pacific Rim International Symposium on Dependable Computing, PRDC 2006*, pages 39–46, 2006.

[3] Hiralal Agrawal, Joseph R. Horgan, Edward W. Krauser, and Saul A. London. Incremental regression testing. *Conference on Software Maintenance*, pages 348–357, 1993.

[4] Apache. apache/commons-codec: Apache commons codec.

[5] Apache. apache/commons-compress: Apache commons compress.

[6] Apache. apache/commons-math: Apache commons math.

[7] Martin Burger and Andreas Zeller. Minimizing reproduction of software failures. *2011 International Symposium on Software Testing and Analysis, ISSTA 2011 - Proceedings*, pages 221–231, 2011.

[8] Jose Campos, Rui Abreu, Gordon Fraser, and Marcelo D'Amorim. Entropy-based test generation for improved fault localization. *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, pages 257–267, 2013.

[9] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11-12):595–607, 1998.

[10] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. Formal concept analysis enhances fault localization in software. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4933 LNAI:273–288, 2008.

[11] Prantik Chatterjee, Abhijit Chatterjee, José Campos, Rui Abreu, and Subhajit Roy. Diagnosing software faults using multiverse analysis. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, IJCAI'20, 2021.

[12] James S. Collofello and Larry Cousins. Towards automatic software fault location through decision-to-decision path analysis. *Managing Requirements Knowledge, International Workshop on*, pages 539–539, 12 1987.

[13] EvoSuite. Evosuite | automatic test suite generation for java.

[14] John Field, G. Ramalingam, and Frank Tip. Parametric program slicing. *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1995.

[15] Fig. Method, system, and program for logging statements to monitor execution of a program. *Method, system, and program for logging statements to monitor execution of a program*, pages 501–502, 12 1999.

[16] Rajiv Gupta and Mary Lou Soffa. Hybrid slicing: An approach for refining static slices using dynamic information. *SIGSOFT Softw. Eng. Notes*, 20(4):29–40, oct 1995.

[17] Robert J. Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering*, 2(1):33–53, 1995.

[18] Mark Harman and Sebastian Danicic. Amorphous program slicing. *Program Comprehension, Workshop Proceedings*, pages 70–79, 1997.

[19] Philip A. Hausler. Denotational program slicing. *Proceedings of the Hawaii International Conference on System Science*, 2:486–494, 1989.

[20] Eran Hershkovich, Roni Stern, Rui Abreu, and Amir Elmishali. Prioritized test generation guided by software fault prediction. *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 218–225, 2021.

[21] JetBrains. Tutorial: Debug your first java application | intellij idea documentation.

[22] JodaOrg. Jodaorg/joda-time: Joda-time is the widely used replacement for the java date and time classes prior to java se 8.

[23] James A. Jones, M. J. Harrold, and J. Stasko. Visualization for fault localization, 2003.

[24] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[25] Microsoft. First look at the debugger - visual studio (windows) | microsoft learn.

[26] Spencer Pearson, Jose Campos, Rene Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017*, pages 609–620, 7 2017.

[27] Anjana Perera, Aldeida Aleti, Burak Turhan, and Marcel Böhme. An experimental assessment of using theoretical defect predictors to guide search-based software testing. *IEEE Transactions on Software Engineering*, 49(1):131–146, 2023.

[28] Alexandre Perez, Rui Abreu, and Arie Van Deursen. A test-suite diagnosability metric for spectrum-based fault localization approaches. *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017*, pages 654–664, 7 2017.

[29] PYPL. Top ide index.

[30] Randoop. Randoop | automatic unit test generation for java.

[31] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[32] Rjust. Defects4j github repository.

[33] David S. Rosenblum. Towards a method of programming with assertions. *Proceedings - International Conference on Software Engineering*, pages 92–104, 1992.

[34] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. *Proceedings - 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pages 201–211, 1 2016.

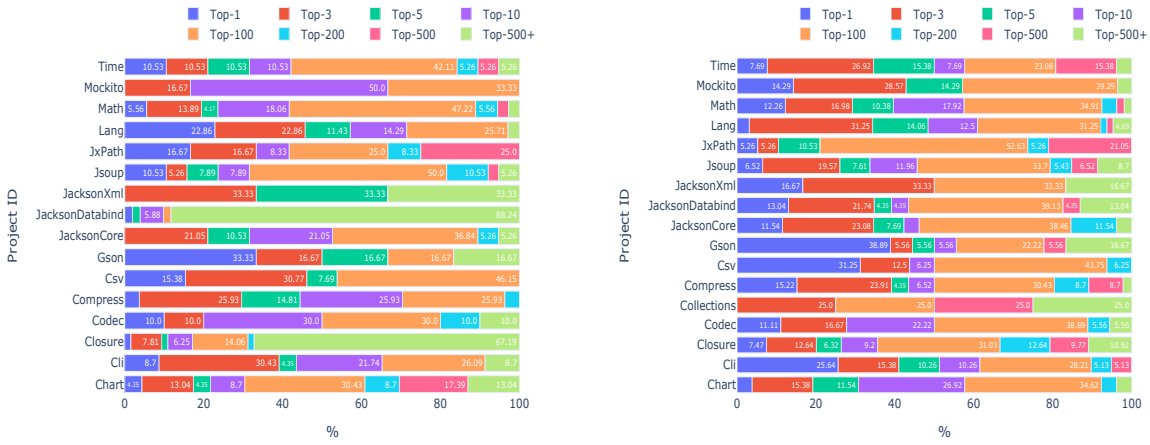
[35] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42:707–740, 8 2016.

[36] W. Eric Wong and Yu Qi. Bp neural network-based effective fault localization. *https://doi.org/10.1142/S021819400900426X*, 19:573–597, 11 2011.

[37] XuBaowen, QianJu, ZhangXiaofang, WuZhongqiang, and ChenLin. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30:1–36, 3 2005.

Appendix A

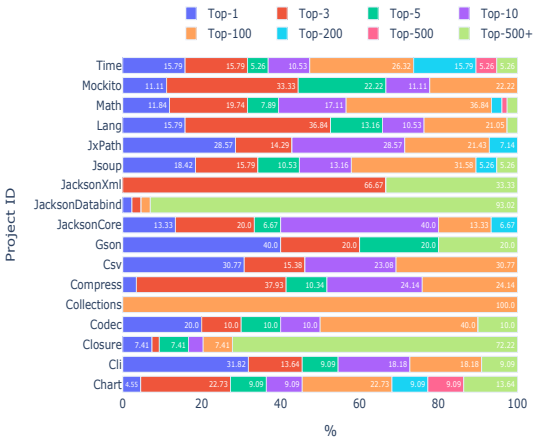
A.1 Plots for fault localization results without bug interception between tools.



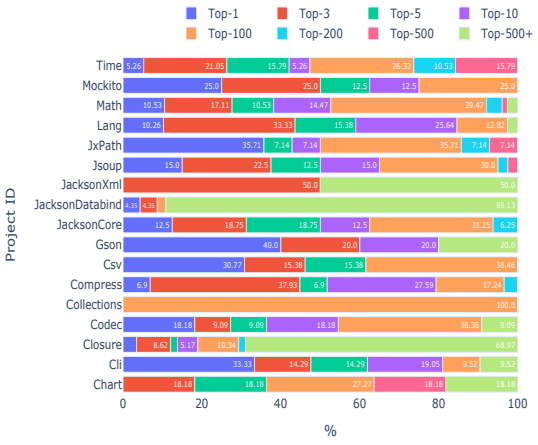
(a) PreMOSA

(b) Developer

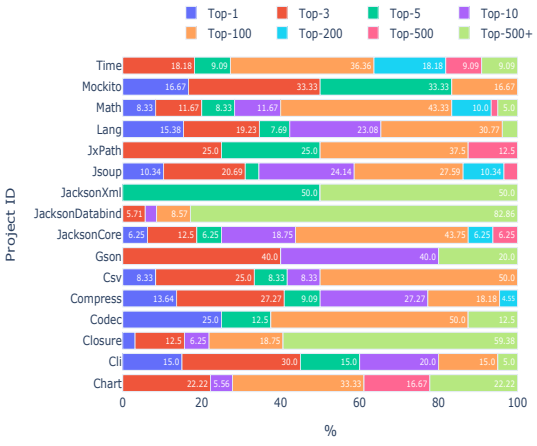
Figure A.1: All Located Bugs



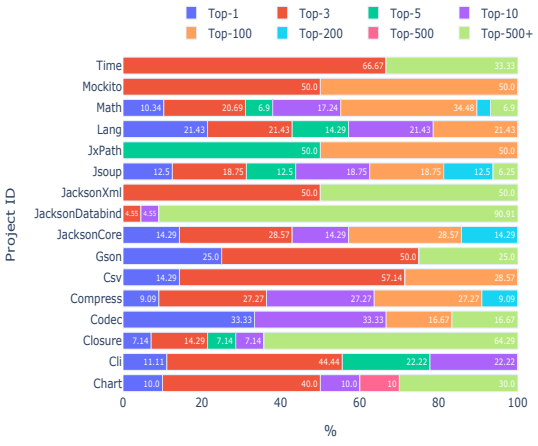
(a) Vanilla EvoSuite



(b) EntBug



(c) DDU



(d) Ulysis

Figure A.2

A.2 Considered bugs for fault detection

A.2.1 Vanilla EvoSuite

Project	Bug ID
Chart	1-26
Cli	1-5, 7-12, 14-15, 17-20, 22-38, 40
Closure	1-9, 11-26, 28-50, 52-59, 62, 64-74, 76-79, 81-85, 87-92, 95-136, 138-143, 145-148, 150-154, 156, 159-162, 164, 166-168, 170-176
Codec	1-12, 14-18
Collections	25, 27-28
Compress	1-47
Csv	1-15
Gson	1-18
JacksonCore	1-25
JacksonDatabind	1-9, 11-13, 16-17, 19-21, 23-29, 32-49, 51-52, 54, 57-58, 61-68, 70-74, 76, 80-95, 97-103, 105-107, 109-112
JacksonXml	1-6
Jsoup	1-2, 4-10, 12-13, 15-20, 24-27, 29-55, 57, 59-66, 68-70, 72-73, 75-86, 88-90, 93
JxPath	1-3, 5-6, 8-12, 14-22
Lang	1, 3-28, 30-34, 36, 38-65
Math	1-106
Mockito	1-18, 20-29, 31-38
Time	1-20, 22-25, 27

A.2.2 PreMOSA

Project	**Bug ID**
Chart	1-26
Cli	1-5, 7-12, 14-15, 17-20, 22-40
Closure	1-50, 52-62, 64-92, 94-136, 138-143, 145-148, 150-154, 156, 159-162, 164, 166-168, 170-176
Codec	1-12, 14-18
Collections	25, 27-28
Compress	1-32, 34-47
Csv	1-15
Gson	1-18
JacksonCore	1-26
JacksonDatabind	1-9, 11-13, 16-17, 19-29, 31-49, 51, 53-54, 56-58, 61-74, 76-78, 80-95, 97-103, 105-107, 110-112
JacksonXml	1, 3-6
Jsoup	1-2, 4-10, 12-20, 22, 24-27, 29-30, 32-55, 57, 59-66, 68-70, 72-73, 75-86, 88-90, 93
JxPath	1-3, 5-6, 8-12, 14-21
Lang	1, 3-28, 30-65
Math	1-10, 12-106
Mockito	12-17, 22-29, 31-38
Time	1-20, 22-25, 27

A.2.3 EntBug

Project	Bug ID
Chart	1-26
Cli	1-5, 7-12, 14-15, 17-20, 22-38, 40
Closure	1-9, 11-26, 28-50, 52-59, 62, 64-74, 76-79, 81-85, 87-92, 95-143, 145-148, 150-154, 156, 159-161, 164, 166-168, 170-176
Codec	1-12, 14-18
Collections	25, 27-28
Compress	1-32, 34-47
Csv	1-15
Gson	1-18
JacksonCore	1-25
JacksonDatabind	1-9, 11-13, 16-17, 19-21, 23-30, 32-49, 51-54, 57-58, 61-68, 70-74, 76, 80-95, 97-103, 105-107, 109-112
JacksonXml	1, 3-6
Jsoup	1-2, 4-10, 12-13, 15-20, 22, 24-27, 29-55, 57, 59-66, 68-70, 72-73, 75-86, 88-90, 93
JxPath	1-3, 5-6, 8-12, 14-18, 20-22
Lang	1, 3-28, 30-34, 36-65
Math	1-106
Mockito	1-2, 4-5, 7-17, 20, 22-29, 31-38
Time	1-20, 22-25, 27

A.2.4 DDU

Project	**Bug ID**
Chart	1-26
Cli	1-5, 7-12, 14-15, 17-20, 22-40
Closure	1-50, 52-62, 64-92, 94-143, 145-148, 150-154, 156, 159-161, 163-164, 166-168, 170-176
Codec	1-12, 14-18
Collections	25, 27-28
Compress	1-47
Csv	1-16
Gson	1-18
JacksonCore	1-26
JacksonDatabind	1-9, 11-14, 16-21, 23-51, 54-58, 60-74, 76-78, 80-108, 110-112
JacksonXml	1-6
Jsoup	1-2, 4-10, 12-13, 15-22, 24-27, 29-30, 32-55, 57, 59-66, 68-70, 72-86, 88-90, 93
JxPath	1-3, 5-6, 8-12, 14-21
Lang	1, 3-28, 30-65
Math	1-106
Mockito	12-17, 22-38
Time	1-20, 22-27

A.2.5 Ulysis

Project	Bug ID
Chart	1-26
Cli	1-5, 7-12, 14-15, 17-20, 22-38, 40
Closure	1-50, 52-62, 64-92, 94-136, 138-143, 145-148, 150-154, 156, 158-161, 163-164, 166-168, 170-176
Codec	1-12, 14-18
Collections	25, 27-28
Compress	1-47
Csv	1-16
Gson	1-18
JacksonCore	1-26
JacksonDatabind	1-9, 11-14, 16-52, 54-58, 60-74, 76-77, 80-103, 105-108, 110-112
JacksonXml	1-6
Jsoup	1-2, 4-10, 12-13, 15-22, 24-27, 29-55, 57, 59-66, 68-70, 72-73, 75-86, 88-90, 93
JxPath	1-3, 5-6, 8-12, 14-15, 17-22
Lang	1, 3-28, 30-65
Math	1-106
Mockito	12-17, 22-38
Time	1-20, 22-25, 27

A.3 Detected bugs

A.3.1 Vanilla EvoSuite

Project	Bug ID
Chart	1-6, 8-11, 14-19, 21-26
Cli	1, 5, 8, 12, 14-15, 17-20, 22-25, 29-34, 36, 38, 40
Closure	2, 7, 12, 19, 21-23, 26, 28, 30, 39, 41, 46, 49, 52, 54, 56, 68, 71-73, 76-77, 79, 81-82, 85, 91, 95, 98, 100, 103-104, 116, 124, 128, 131, 134-136, 139-141, 143, 145-148, 150-151, 153-154, 159-161, 164, 167-168, 173-175
Codec	4-7, 9, 11-12, 15-18
Collections	25
Compress	1-4, 6-9, 11, 14, 17-19, 21, 23-24, 27, 29, 34-35, 38-47
Csv	2-14
Gson	4, 7, 9, 12, 17
JacksonCore	1-4, 7-8, 10-12, 14-17, 20, 22
JacksonDatabind	2-4, 6, 8, 11-13, 16-17, 23, 32, 35-36, 38, 40-42, 44, 46, 49, 51, 57, 61-63, 70, 72, 84, 86, 88-92, 95, 97, 99-100, 102-103, 105-107, 111
JacksonXml	3-4, 6
Jsoup	5-8, 16-20, 25-27, 30, 36, 40-42, 46-50, 52-54, 59-60, 64, 70, 72, 75, 79-80, 83, 85-86, 88-89
JxPath	1-3, 5-6, 8-9, 11-12, 14, 16-18, 21-22
Lang	1, 4-5, 7-9, 11-12, 14-20, 22-24, 27, 32-34, 36, 39, 41, 45-49, 52, 54, 57-61, 65
Math	1-6, 9, 11, 13-14, 21-27, 29, 32, 35-38, 40, 42, 45-49, 51-56, 59-68, 70-73, 75-81, 83-87, 89-90, 92, 94-99, 101-102, 104-106
Mockito	2, 4, 17, 23, 29, 34-35, 37-38
Time	1-2, 4-9, 11-17, 20, 24-25, 27

A.3.2 PreMOSA

Project	Bug ID
Chart	1, 4-19, 21-26
Cli	1-2, 5, 8, 12, 14-15, 17-20, 22-25, 29-32, 34-35, 37-40
Closure	6-7, 19, 22, 26, 28, 30, 33, 39-41, 46, 49, 52, 54, 56, 58, 62, 65, 68, 72-73, 75-76, 79-82, 85-86, 91, 94-96, 98-100, 103-104, 106, 109, 124, 128, 131, 134-136, 139-141, 145-148, 150, 152-154, 159, 161, 164, 166-167, 173-175
Codec	2, 4, 6-7, 9, 11-12, 15-18
Compress	1-4, 6, 8-9, 11, 14, 16-19, 23-24, 27-29, 34-35, 38-44, 46
Csv	2-14
Gson	4, 6-7, 9, 12, 17
JacksonCore	1-5, 7-8, 10-18, 20, 22, 26
JacksonDatabind	2-6, 9, 11-13, 16-17, 22-23, 25, 31-33, 36, 38-46, 48-49, 51, 56-57, 61, 63, 70, 72, 84-86, 88, 90-91, 95, 97, 99, 102-103, 105-107, 111
JacksonXml	3-4, 6
Jsoup	5, 8, 14, 16-20, 22, 25-26, 30, 36, 40-44, 46-47, 50, 52-53, 59-60, 63-64, 72-73, 77, 79-80, 83-84, 86, 88-89, 93
JxPath	1, 3, 5-6, 8-9, 12, 14, 16, 18, 20-21
Lang	1, 4-5, 7, 9-12, 14, 16, 18-19, 22-23, 27, 32-33, 35-37, 39, 41, 44-49, 52-54, 57-59, 65
Math	1-6, 8-9, 13-14, 21-25, 27-30, 32, 35-38, 40, 42, 45-49, 51-53, 55-56, 59-67, 70-73, 75, 77-78, 80-81, 83, 85-90, 92, 94-99, 101-102, 105-106
Mockito	17, 23, 29, 35, 37-38
Time	1-6, 8-9, 11-15, 17, 20, 23-25, 27

A.3.3 EntBug

Project	Bug ID
Chart	1-8, 10-11, 14-19, 21-26
Cli	1-2, 5, 8, 14-15, 17-20, 22-25, 29, 31-34, 36, 38, 40
Closure	1, 6-7, 12, 17, 19, 21-23, 26, 28, 30, 39, 41-43, 49, 52, 54, 56, 62, 65, 68, 72-73, 76-77, 79, 81-82, 85, 91, 95, 98-100, 104, 109, 115, 128, 131, 134-135, 137, 139-141, 143, 145-148, 150-154, 159, 161, 164, 167, 173-174
Codec	2, 4-7, 9, 11-12, 15-18
Collections	25
Compress	1-9, 11, 14, 16-19, 23-24, 27, 29, 34-35, 38-46
Csv	2-14
Gson	4, 7, 9, 12, 17
JacksonCore	1-5, 7-8, 10-11, 14-17, 20-22
JacksonDatabind	2-4, 6-8, 11-13, 16-17, 23, 32, 35-36, 38, 40-46, 49, 51, 57, 61, 64, 70, 72, 80, 84, 86, 88-92, 95, 97, 99-100, 102-103, 105-107, 111
JacksonXml	4, 6
Jsoup	4-5, 7-8, 12, 16, 18-20, 22, 25-27, 30, 36, 40-41, 44, 46-47, 49-50, 52-55, 59-60, 63-64, 72, 75, 77, 79-80, 83, 86, 88-89, 93
JxPath	1, 3, 5-6, 8-10, 12, 14, 16-18, 21-22
Lang	1, 4-5, 7-9, 11-12, 14-20, 22-24, 27-28, 32-34, 36-37, 39, 41, 45-49, 52, 57-61, 65
Math	1-6, 9, 11, 13-14, 21-27, 29, 32, 35-38, 40, 42, 45-49, 51-56, 59-68, 70-73, 75-77, 79-81, 83-87, 89-92, 94-99, 101-102, 104-106
Mockito	2, 4, 17, 23, 29, 35, 37-38
Time	1-2, 4-9, 11-17, 20, 23-25

A.3.4 DDU

Project	Bug ID
Chart	4-6, 8, 10-12, 14-19, 21-25
Cli	1-2, 5, 8, 15, 17-20, 22-25, 27, 29, 31-32, 34, 38-40
Closure	19, 22, 28, 33, 39, 41, 49, 54, 56, 68, 73, 77, 80, 85, 100, 104, 106, 131, 134, 140-141, 146-148, 150-151, 153, 164, 166-167, 173-175
Codec	2, 4, 6-7, 11-12, 16-18
Compress	1-4, 6, 8-9, 11, 14, 17-19, 26-27, 29, 34, 38, 40-42, 44-46
Csv	2-13
Gson	4, 7, 9, 12, 17
JacksonCore	1-8, 10-11, 14-17, 20, 22
JacksonDatabind	2-4, 6, 8, 11-13, 16, 35-36, 38, 41, 44, 46, 49, 55, 57, 61, 63, 72, 84, 86, 88-90, 92, 97, 99-100, 104-107, 111
JacksonXml	4, 6
Jsoup	5, 8, 16, 18-22, 26, 30, 34, 36, 40-41, 46-47, 50, 52-53, 60, 64, 72, 75, 77, 79, 83, 86, 88-89
JxPath	3, 5-6, 9, 11-12, 16, 18
Lang	1, 7, 9, 11-12, 16, 18-19, 22-23, 32-33, 36, 39, 41, 45-49, 52, 57, 59, 61, 63, 65
Math	1, 3-6, 8, 11, 13-14, 21-24, 26-27, 29, 32-33, 35, 37-38, 40, 42-43, 45-47, 49, 51, 53, 55, 59-60, 62-67, 70-71, 73, 77-78, 80-81, 83, 85-89, 92, 95-98, 101-102, 105
Mockito	17, 23, 29, 35, 37-38
Time	1-2, 4-6, 8-9, 11, 14, 17, 24

A.3.5 Ulysis

Project	**Bug ID**
Chart	4, 10, 14-19, 22, 24
Cli	5, 8, 23-25, 27, 31-32, 34, 40
Closure	21-22, 27, 54, 56, 73, 104, 106, 128, 140-141, 146, 148, 174
Codec	4, 7, 11-12, 17-18
Compress	1, 4, 8-9, 11, 14, 18, 29, 34-35, 42, 45
Csv	3-5, 8-9, 12-13
Gson	4, 9, 12, 17
JacksonCore	1-2, 10, 16-17, 20, 22
JacksonDatabind	2, 4, 12-14, 16, 32, 35-36, 43, 46, 55, 57, 61, 63, 72, 88-90, 92, 97, 99, 105
JacksonXml	4, 6
Jsoup	8, 16, 18, 21-22, 26, 30, 40, 49-50, 52, 79, 83, 86, 88-89
JxPath	8-9
Lang	10-12, 18, 32-34, 41, 46-48, 52, 57, 65
Math	1, 3-6, 11, 13-14, 22, 27, 29, 37, 45-46, 56, 59, 66-67, 70-71, 77, 85-86, 89, 92, 96-98, 101
Mockito	35, 38
Time	2, 8, 11

A.4 Considered bugs in Fault Localization

A.4.1 Vanilla EvoSuite

Project	Bug ID
Chart	1-6, 8-11, 14-19, 21-26
Cli	1, 5, 8, 12, 14-15, 17-20, 22-25, 29-33, 36, 38, 40
Closure	2, 7, 12, 19, 22-23, 26, 28, 30, 39, 41, 46, 49, 52, 54, 56, 68, 72-73, 76-77, 81-82, 85, 91, 95, 100, 103-104, 124, 128, 131, 134-136, 139-141, 143, 145-148, 150-151, 153-154, 159-161, 164, 167, 173-174
Codec	4-7, 9, 12, 15-18
Collections	25
Compress	1-4, 6-8, 11, 14, 17-19, 21, 23-24, 27, 29, 34-35, 38-47
Csv	2-14
Gson	4, 7, 9, 12, 17
JacksonCore	1-4, 7-8, 10-12, 14-17, 20, 22
JacksonDatabind	2-4, 6, 8, 11-13, 16-17, 23, 32, 35, 38, 40-42, 44, 46, 49, 51, 57, 61-63, 70, 72, 84, 86, 88-92, 95, 97, 99-100, 102, 105-107, 111
JacksonXml	3-4, 6
Jsoup	5-8, 16-20, 25-27, 30, 36, 40-42, 46-50, 52-54, 59-60, 64, 70, 72, 75, 79-80, 83, 85-86, 88-89
JxPath	1, 3, 5-6, 8-9, 11-12, 14, 16-18, 21-22
Lang	1, 4-5, 7-9, 11-12, 14-20, 22-24, 27, 32-34, 36, 39, 41, 45-49, 52, 54, 57-61, 65
Math	1-6, 9, 11, 13-14, 21-27, 29, 32, 35-38, 40, 42, 45-49, 51-56, 59-68, 70-73, 75-81, 83-87, 89-90, 92, 94-99, 101-102, 104-106
Mockito	2, 4, 17, 23, 29, 34-35, 37-38
Time	1-2, 4-9, 11-17, 20, 24-25, 27

A.4.2 PreMOSA

Project	**Bug ID**
Chart	1, 4-19, 21-26
Cli	1-2, 5, 8, 12, 14-15, 17-20, 22-25, 29-31, 35, 37-40
Closure	7, 19, 22, 26, 28, 30, 33, 39-41, 46, 49, 52, 54, 56, 58, 62, 65, 68, 72-73, 75-76, 79-82, 85-86, 91, 94-96, 99-100, 103-104, 106, 109, 124, 128, 131, 134-136, 139-141, 145-148, 150, 152-154, 159, 161, 164, 166-167, 173-175
Codec	2, 4, 6-7, 9, 12, 15-18
Compress	1-4, 6, 8, 11, 14, 16-19, 23-24, 27-29, 34-35, 38-44, 46
Csv	2-14
Gson	4, 6-7, 9, 12, 17
JacksonCore	1-5, 7-8, 10-18, 20, 22, 26
JacksonDatabind	2-6, 9, 11-13, 16-17, 22-23, 25, 31-33, 36, 38-46, 48-49, 51, 56-57, 61, 63, 70, 72, 84-86, 88, 90-91, 95, 97, 99, 102-103, 105-107, 111
JacksonXml	3-4, 6
Jsoup	5, 8, 14, 16-20, 22, 25-26, 30, 36, 40-44, 46-47, 50, 52-53, 59-60, 63-64, 72-73, 77, 79-80, 83-84, 86, 88-89, 93
JxPath	1, 3, 5-6, 8-9, 12, 14, 16, 18, 20-21
Lang	1, 4-5, 7, 9-12, 14, 16, 18-19, 22-23, 27, 32-33, 35-37, 39, 41, 44-49, 52-54, 57-59, 65
Math	1-6, 8-9, 13-14, 21-25, 27-30, 32, 35-38, 40, 42, 45-49, 51-53, 55-56, 59-67, 70-73, 75, 77-78, 80-81, 83, 85-90, 92, 94-99, 101-102, 105-106
Mockito	17, 23, 29, 35, 37-38
Time	1-6, 8-9, 11-15, 17, 20, 23-25, 27

A.4.3 EntBug

Project	Bug ID
Chart	1-8, 10-11, 14-19, 21-26
Cli	1-2, 5, 8, 14-15, 17-20, 22-25, 29, 31-33, 36, 38, 40
Closure	1, 6-7, 12, 19, 22-23, 26, 28, 30, 39, 41-43, 49, 52, 54, 56, 62, 65, 68, 72-73, 76-77, 81-82, 85, 91, 95, 99-100, 104, 109, 115, 128, 131, 134-135, 139-141, 143, 145-148, 150-154, 159, 161, 164, 167, 173-174
Codec	2, 4-7, 9, 12, 15-18
Collections	25
Compress	1-8, 11, 14, 16-19, 23-24, 27, 29, 34-35, 38-46
Csv	2-14
Gson	4, 7, 9, 12, 17
JacksonCore	1-5, 7-8, 10-11, 14-17, 20-22
JacksonDatabind	2-4, 6-8, 11-13, 16-17, 23, 32, 35, 38, 40-46, 49, 51, 57, 61, 64, 70, 72, 80, 84, 86, 88-92, 95, 97, 99-100, 102, 105-107, 111
JacksonXml	4, 6
Jsoup	4-5, 7-8, 12, 16, 18-20, 22, 25-27, 30, 36, 40-41, 44, 46-47, 49-50, 52-55, 59-60, 63-64, 72, 75, 77, 79-80, 83, 86, 88-89, 93
JxPath	1, 3, 5-6, 8-10, 12, 14, 16-18, 21-22
Lang	1, 4-5, 7-9, 11-12, 14-20, 22-24, 27-28, 32-34, 36-37, 39, 41, 45-49, 52, 57-61, 65
Math	1-6, 9, 11, 13-14, 21-27, 29, 32, 35-38, 40, 42, 45-49, 51-56, 59-68, 70-73, 75-77, 79-81, 83-87, 89-92, 94-99, 101-102, 104-106
Mockito	2, 4, 17, 23, 29, 35, 37-38
Time	1-2, 4-9, 11-17, 20, 23-25

A.4.4 DDU

Project	**Bug ID**
Chart	4-6, 8, 10-12, 14-19, 21-25
Cli	1-2, 5, 8, 15, 17-20, 22-25, 27, 29, 31-32, 38-40
Closure	19, 22, 28, 33, 39, 41, 49, 54, 56, 68, 73, 77, 80, 85, 100, 104, 106, 131, 134, 140-141, 146-148, 150-151, 153, 164, 166-167, 173-174
Codec	2, 4, 6-7, 12, 16-18
Compress	1-4, 6, 8, 11, 14, 17-19, 26-27, 29, 34, 38, 40-42, 44-46
Csv	2-13
Gson	4, 7, 9, 12, 17
JacksonCore	1-8, 10-11, 14-17, 20, 22
JacksonDatabind	2-4, 6, 8, 11-13, 16, 35-36, 38, 41, 44, 46, 49, 55, 57, 61, 63, 72, 84, 86, 88-90, 92, 97, 99-100, 104-107, 111
JacksonXml	4, 6
Jsoup	5, 8, 16, 18-22, 26, 30, 34, 36, 40-41, 46-47, 50, 52-53, 60, 64, 72, 75, 77, 79, 83, 86, 88-89
JxPath	3, 5-6, 9, 11-12, 16, 18
Lang	1, 7, 9, 11-12, 16, 18-19, 22-23, 32-33, 36, 39, 41, 45-49, 52, 57, 59, 61, 63, 65
Math	1, 3-6, 8, 11, 13-14, 21-24, 26-27, 29, 32-33, 35, 37-38, 40, 42-43, 45-47, 49, 51, 53, 55, 59-60, 62-67, 70-71, 73, 77-78, 80-81, 83, 85-89, 92, 95-98, 101-102, 105
Mockito	17, 23, 29, 35, 37-38
Time	1-2, 4-6, 8-9, 11, 14, 17, 24

A.4.5 Ulysis

Project	Bug ID
Chart	4, 10, 14-19, 22, 24
Cli	5, 8, 23-25, 27, 31-32, 40
Closure	21-22, 27, 54, 56, 73, 104, 106, 128, 140-141, 146, 148, 174
Codec	4, 7, 11-12, 17-18
Compress	1, 4, 8, 11, 14, 18, 29, 34-35, 42, 45
Csv	3-5, 8-9, 12-13
Gson	4, 9, 12, 17
JacksonCore	1-2, 10, 16-17, 20, 22
JacksonDatabind	2, 4, 12-14, 16, 32, 35, 43, 46, 55, 57, 61, 63, 72, 88-90, 92, 97, 99, 105
JacksonXml	4, 6
Jsoup	8, 16, 18, 21-22, 26, 30, 40, 49-50, 52, 79, 83, 86, 88-89
JxPath	8-9
Lang	10-12, 18, 32-34, 41, 46-48, 52, 57, 65
Math	1, 3-6, 11, 13-14, 22, 27, 29, 37, 45-46, 56, 59, 66-67, 70-71, 77, 85-86, 89, 92, 96-98, 101
Mockito	35, 38
Time	2, 8, 11

A.5 Interception of bugs between tools for FL

Project	**Bug ID**
Chart	4, 10, 14-19, 22, 24
Cli	5, 8, 23-25, 31, 40
Closure	22, 54, 56, 73, 104, 140-141, 146, 148, 174
Codec	4, 7, 12, 17-18
Compress	1, 4, 8, 11, 14, 18, 29, 34, 42
Csv	3-5, 8-9, 12-13
Gson	4, 9, 12, 17
JacksonCore	1-2, 10, 16-17, 20, 22
JacksonDatabind	2, 4, 12-13, 16
JacksonXml	4, 6
Jsoup	8, 16, 18, 26, 30, 40, 50, 52, 79, 83, 86, 88-89
JxPath	9
Lang	11-12, 18, 32-33, 41, 46-48, 52, 57, 65
Math	1, 3-6, 13-14, 22, 27, 29, 37, 45-46, 59, 66-67, 70-71, 77, 85-86, 89, 92, 96-98, 101
Mockito	35, 38
Time	2, 8, 11