Bachelor Degree Project

Empirical Comparison Between Conventional and AI-based Automated Unit Test Generation Tools in Java

Author: Marios Gkikopouli
Author: Batjigdrel Bataa
Supervisor: Jonas Lundberg
Examiner: Faiz Ul Muram
Semester: VT 2023
Subject: Computer Science

Abstract

Unit testing plays a crucial role in ensuring the quality and reliability of software systems. However, manual testing can often be a slow and time-consuming process. With current advancements in artificial intelligence (AI), new tools have emerged for automated unit testing to address this issue. But how do these new AI tools compare to conventional automated unit test generation tools? To answer this question, we compared two state-of-the-art conventional unit test tools (EVOSUITE and RANDOOP) with the sole commercially available AI-based unit test tool (DIFFBLUE COVER) for Java. We tested them on 10 sample classes from 3 real-life projects provided by the Defects4J dataset to evaluate their performance regarding code coverage, mutation score, and fault detection. The results showed that EVOSUITE achieved the highest code coverage, averaging 89%, while RANDOOP and DIFFBLUE COVER achieved similar results, averaging 63%. In terms of mutation score, DIFFBLUE COVER had the lowest average score of 40%, while EVOSUITE and RANDOOP scored 67% and 50%, respectively. For fault detection, EVOSUITE and RANDOOP detected a higher number of bugs (7 out of 10 and 5 out of 10, respectively) compared to DIFFBLUE COVER, which found only 4 out of 10. Although the AI-based tool was outperformed in all three criteria, it still shows promise by being able to achieve adequate results, in some cases even surpassing the conventional tools while generating a significantly smaller number of total assertions and more comprehensive tests. Nonetheless, the study acknowledges its limitations in terms of the restricted number of AI-based tools used and the small number of projects utilized from Defects4J.

Keywords: **Software Testing; Automatic Test Case Generation; AI; Defects4J; Experiment;**

Contents

1 Introduction

This is a 15 HEC Bachelor thesis in Computer Science with a focus on software testing and specifically the use of AI tools for automated unit test generation in Java.

Software testing is a process or cycle of assessments aimed at evaluating a software system or its components to ensure that the said system behaves as intended and meets the specified requirements of its stakeholders [1]. The testing process involves the identification and detection of bugs, errors, or missing requirements within a developed system.

Unit testing is one of the fundamental steps in the software testing process that focuses on verifying the smallest units of software, typically modules or methods. It involves testing the individual units in isolation to uncover errors within the boundaries of the module [2]. Unit testing is typically conducted after the coding phase after the program has been reviewed and syntax errors have been corrected. Since modules are not standalone programs, mocks/stubs need to be developed to address dependencies and facilitate testing. Designing modules with high cohesion simplifies unit testing as it reduces the number of test cases and makes errors more predictable and detectable [2].

This chapter serves as an introduction to the thesis, covering background information, related work, problem formulation, motivation, results, scope/limitations, target audience, and overall outline. The background emphasizes the importance of software testing and the challenges it faces, such as budget and time constraints. The related work discusses previous studies on automated test generation tools as a means to address the budget and time challenges and improve the testing process. The problem formulation addresses the research gap and the need to evaluate AI-based tools for automated unit test generation. The motivation section explains the goals of the research and the insights to be gained. The results present the findings of this empirical study and draw conclusions. The scope/limitations acknowledge the study's limitations. The target audience section identifies who this study concerns, and the outline provides a structure for the thesis.

1.1 Background

In today's technology-driven world, computer science and specifically software testing plays a vital role in ensuring the quality and reliability of software systems. This is of utmost importance to businesses because the cost of resolving bugs escalates rapidly as the software life cycle progresses. A study conducted by Cambridge University in January 2013 estimated that debugging software globally costs \$312 billion annually and that developers spend 50% of their time on resolving errors [3]. Unit testing is an essential process in software development that aims to identify and eliminate errors, but it also is one of the most time-consuming and costly activities for developers and organizations [4].

1.2 Related work

To address this time-consuming nature of the testing process, many automated unit test generation tools for Java have been developed over the last few decades [5]. Some empirical studies have compared the effectiveness of manual and automated unit test generation tools in Java. In 2008, Bacchelli et al. conducted the first analysis and found that automated test generation tools can speed up test creation and help find defects, but they do not replace the need for manual testing to ensure thorough analysis of the tested system [6]. A follow-up study conducted in 2019 concluded that current automated test case generation tools are better at optimizing code coverage and mutation score than manually written tests, but still have little improvement when it comes to defect finding [7]. Shamshiri et al., in a study performed in 2015, also found that while the test suites generated automatically managed to identify 55.7% of the faults in total, only 19.9% of the individual test suites were able to detect a fault [8]. in 2020, Souza et al. conducted a study on a larger scale that aimed to compare the effectiveness of automatically generated test suites and manually written test suites when used as regression test suites and found that, in general, manually written test suites were more effective than automatically generated test suites in terms of both line coverage and mutation coverage [9]. Additionally, Almasi et al. conducted a study involving RANDOOP and EVOSUITE, up-to-date unit test generation tools at the time, and found that developers in the industry are troubled by the size and readability of the test suites [10].

1.3 Problem formulation

In recent years, there have been significant advancements in AI, with various fields taking advantage of it. As a result, new AI-based unit test generation tools have emerged, which could potentially further improve the software testing process. To our knowledge, there has not been any research evaluating the efficiency of these new AI tools, especially when compared to conventional automated unit test generation tools and the improvements they bring, if any. In this paper, we conducted an experiment to empirically evaluate the effectiveness of AI-based tools in regards to unit test generation and compared them to conventional tools using Defects4J, a data set that contains bugs from real word open-source repositories [11]. We generated test suites using two state-of-the-art unit test generation tools, EVOSUITE [12] and RANDOOP [13] [14], and one AI-based unit test generation tool, DIFFBLUE COVER [15], for Java for projects in the Defects4J dataset and analyzed how each of the tools performed against the developer-written tests. In more detail, we strived to answer the following questions: 1) "What is the code coverage achieved by the AI-generated test suite compared to the one generated by conventional tools?", 2) "What is the mutation score achieved by the AI-generated test suite compared to the one generated by conventional tools?", 3) "How many faults was the AI-generated test suite able to detect compared to the one generated by conventional tools?"

1.4 Motivation

By conducting this study, we aim to explore the potential of AI in improving the effectiveness of test suites and contribute to the understanding of strengths and weaknesses in AI-based software testing tools. Moreover, by understanding the strengths and weaknesses of AI-based compared to conventional tools, developers can make informed decisions about adopting and integrating these tools into their projects. Most importantly, identifying potential weaknesses of AI tools in the context of unit testing can lead to potential future improvement of the tools.

This study uses EVOSUITE and RANDOOP as the conventional tools for this experiment providing a baseline for comparison with AI-based automated generation tools. EVOSUITE and RANDOOP were part of the most recent Search-Based Software Testing competition in 2022 (SBST), with EVOSUITE being the winner [14]. Additionally, EVOSUITE and RANDOOP were selected due to their widespread utilization and established presence across academia and research [7], [8], [10], and [9]. Moreover, it should be noted that the availability of the other tools in the competition seemed to be limited, as most of them were not publicly accessible, further supporting our decision to choose EVOSUITE and RANDOOP.

1.5 Results

Our research findings show that DIFFBLUE COVER, the AI tool, was able to achieve code coverage averaging 63%, similar to RANDOOP. However, EVOSUITE was able to outperform both, averaging 89%. Compared to the study performed by Serra et al. (Table I in [7]), EVOSUITE still seems to surpass RANDOOP when it comes to code coverage. In terms of mutation score, DIFFBLUE COVER performed the worst, averaging at 40%, while EVOSUITE and RANDOOP achieved a mutation score of 67% and 50% respectively. Compared to the results achieved by Serra et al. (Table I in [7]), EVOSUITE once more performed better than RANDOOP concerning mutation testing. In regards to fault detection, DIFFBLUE COVER was able to detect 4 out of 10 bugs, while EVOSUITE and RANDOOP found 7 out of 10 and 5 out of 10 respectively. In comparison to the findings in the study by Shamshiri et al.[4], both EVOSUITE and RANDOOP identified faults for the same subject classes used in our experiment.

1.6 Scope/Limitation

Despite the insights gained from our results, it is important to acknowledge the limitations of our study. Unfortunately, we were unable to gain access to more than one AI-based unit test generation tool due to them being relatively new and with limited access. MACHINET was one promising AI tool that was considered, but due to restrictions on the number of generated tests to 50 per month, we were unable to include it [16]. Another limitation was that subject classes were only chosen from 3 different projects, due to limited familiarity with Apache Ant builds and time constraints.

1.7 Target group

Our research targets both researchers and practitioners in the field. This paper aims to provide valuable insights that could inform the direction of future efforts in this area, as well as inform developers about the effectiveness of using AI-based automated testing tools in production and the industry.

1.8 Outline

This paper is organized as follows. Section 2 provides the experiment methodology, including the experiment process, design, validity, and reliability. Section 3, presents the theoretical background including obstacles in the investigation area, related work, and research gap. Section 4 includes a description of the experiment setup and the steps followed to collect and validate the data. In addition, it gives a comprehensive report on the selected sample projects and the respective classes under investigation. Section 5 displays the results and analysis of the experiment and comparison with related work and Section 6 discusses our findings. Section 7 addresses the validity threats of the study, while in Section 8, we conclude and discuss future work.

2 Methodology

2.1 Research Project

The goal of this study is to compare AI-based unit test generation tools with conventional automated unit test generation tools to determine whether AI-based tools can increase the overall effectiveness of test suites in real-life open-source projects, as well as identify their strengths and weaknesses. This paper targets both researchers and practitioners in the field. Researchers are interested in gaining insights into the weaknesses of AI in the context of unit test generation, which could inform the direction of future development efforts in this area. The practitioners are concerned with the effectiveness of using AI-based automated testing tools in practical scenarios. To achieve our objective, we will perform an experiment to measure code coverage, mutation score, and fault detection capabilities of the generated test suites using the same methodology as Serra et al. [7]. which was built upon the work done by Bacheli et al. [6], and follows a similar experimental setup as the study conducted by Shamshiri et al. [8]. Additionally, we will select subject classes from a set of real-world open-source projects and defects using the Defects4J framework [11] same as the ones used by Shamshiri et al.

1. ***RQ1*** *- What is the code coverage of AI-generated tests compared to conventional tools?*
2. ***RQ2****- What is the mutation score of AI-generated tests compared to conventional tools?*
3. ***RQ3****- What are the fault detection capabilities of AI-generated tests compared to conventional tools?*

A common alternative approach to our experiment would be to only measure code coverage. However, research shows that code coverage alone is insufficient for effectively evaluating the quality of a test suite [17]. Similarly, mutation testing alone shows a weak correlation with real fault detection, as stated in the literature [18]. Therefore, to make a proper assessment, we have included three criteria: code coverage, mutation score, and fault detection capabilities. By answering the above research questions we hope to provide valuable insights for researchers and developers in the field regarding the potential of AI-based automated unit test generation tools when compared to unit tests generated by conventional tools.

2.2 Research Method

An experiment is a research method often conducted in a laboratory setting that grants the researcher control over the study and the tasks to be performed. It can include a comparison of various techniques, methods, or procedures to thoroughly investigate their effects on specific variables of interest. The primary objective of such research is to manipulate one or more independent variables, which effectively represent the diverse treatments or conditions being studied to observe and measure the effects it has on the dependent variables allowing the researcher to draw conclusions [19]. The data collected from the experiment is often analyzed using statistical methods to determine the effects of the independent variables on the dependent variables.

This study aims to compare AI-based unit test generation tools with conventional automated unit test generation tools. An experiment allows the researchers to manipulate the independent variable (type of test generation tool) by assigning subjects (projects) to different treatments (AI-based tools vs. conventional tools). Furthermore, experimenting provides researchers with control over various factors, such as project size, programming language, and defect types which allows for a direct comparison between the two types of tools. Lastly, through an experiment, the study aims to measure the dependent variables, which include code coverage, mutation score, and fault detection capabilities of the generated test suites, facilitating the collection of quantitative data and enabling statistical analysis of the results making it the ideal to method to answer this studies research questions.

Another method that could have been considered would be performing a case study. However, while a case study could provide insights into the effectiveness of AI-based test generation tools, it may not offer a controlled environment for a direct comparison with conventional tools, this is because case studies focus on in-depth investigations of a single entity or phenomenon and may not provide the necessary control, comparability, and generalizability required for evaluating and comparing different methods [19].

For this study, we followed the experimental procedure outlined by C. Wohlin et al. in their paper on empirical research methods in web and software engineering (Chapter 13.4) [19]. The procedure in the specific context of our experiment is as follows:

Planning and Design:

- *Sampling*: Select a sample of real-world open-source projects and defects using the Defects4J framework to consider the representativeness of the sample and its potential impact on the generalizability of the results.
- *Randomization*: Randomly assign participants (e.g., subjects, projects, or defects) to either the AI-based tool group or the conventional tool group to ensure unbiased results and minimize the influence of confounding factors.
- *Independent and Dependent Variables*: Clearly define and measure the independent variable (type of unit test generation tool) and the dependent variables (code coverage, mutation score, and fault detection capabilities).
- *Confounding Factors*: Identify and account for any potential confounding factors that may influence the dependent variables independently of the independent variable such as flaky tests, false positives, and project complexity
- *Standard Experimental Designs*: Choose an appropriate experimental design that suits the research objectives and variables being studied. In this study we are following the same experimental design performed by Serra et al., [7].

Operation:

- Ensure participants' commitment to the experiment and provide them with clear instructions and materials that define all the tasks to be performed and make sure participants record relevant data for later analysis.

Analysis and Interpretation:

- *Analysis*: Validate the collected data, and apply descriptive statistics to gain an overview. Choose appropriate statistical tests based on the data characteristics and the research design.

- *Validity Concerns*: Consider the validity of the results in terms of internal, external, conclusion, and construct validity. Address potential threats to validity, such as confounding factors and the representativeness of the sample.

- *Conclusions and Actions*: Based on the analysis results, draw appropriate conclusions regarding the effectiveness of AI-based unit test generation tools compared to conventional tools. Consider the strengths and weaknesses identified in the study.

2.2.1 Experimental Design

This experiment follows the same experimental design performed by Serra et al., [7], however, there is no need to manually identify the defective versions of classes or write manual tests since Defects4J can provide them for this experiment. Our study involves the following steps.

Subject classes: Defects4J[11] is a framework that includes a database and a flexible structure, that offers genuine software bugs written in Java to facilitate repeatable experiments in the field of software testing research [11]. At the time of writing, Defects4J contains up to 835 bugs from open-source repositories such as Collections, Lang, Time, Math, and more. Defects4J offers both (1) faulty versions of production classes, (2) corresponding developer-written manual tests, (3) and bug-free versions of the classes. We randomly selected 10 sample classes with a buggy and fixed version from 3 projects among others showcased in the study by Shamshiri et al. [8] specifically Time, Math, and Lang. More information about the subject classes can be found in Section 4.

Automated Unit test generation tools: For their study on automated unit test generation tools, Serra et al [7]. selected three tools: EVOSUITE, RANDOOP, and JTExpert. EVOSUITE and RANDOOP are still considered state-of-the-art tools in this field and will be included in our controlled experiment. However, we were unable to find information on how to obtain access to JTExpert and set it up. Automated unit test generation tools based on AI are still relatively new, so the number of options available is limited. Currently, the only publicly available AI-based tools for generating unit tests are DIFFBLUE COVER and MACHINET. Both of these tools offer a free version for users to try out but due to MACHINET's limitation of only 50 free generations per month, it was not included in the experiment.

Generating regression tests: Once the subject classes and their defects from Defects4J were identified we used the three automated unit test generation tools - EVOSUITE, RANDOOP, DIFFBLUE COVER - on both the faulty and fixed versions of the classes. We ran the tools for up to 180 seconds, using their default settings, and repeated the generation 10 times for each class to account for the randomness factor of random and search-based algorithms typically used in conventional automated unit test generation tools as mentioned by Serra et al. [7]

Flaky tests: To conduct regression tests and identify faults, it is important to have test suites that pass successfully on the fixed version of the subject classes. However, it is important to acknowledge that these tools have the potential to generate nondeterministic or flaky tests, which may also fail on the fixed version [8]. For example, flaky tests often occur in the Time project when a test case depends on the system's time. This dependency can cause the test to pass during generation but fail when executed at a later time. To mitigate this, we manually eliminated any failing tests in the fixed version before recompiling the test suite.

False positives: In certain cases, a test may fail on the buggy version, even if it is not classified as a flaky test. Nevertheless, the failing test might not be related to the actual bug under investigation. These false positives often occur due to the test breaking encapsulation or capturing outdated mocking behavior [8]. To overcome this, we followed the same procedure performed by Shamshiri et al. [8] and compared the failure message of the generated test with the failing messages of the developer-written test to determine if it was a false positive. We spend 15-20 minutes for each generated test suite depending on the number of failing tests.

Analysis To answer our research questions, we followed the same steps as those performed by Serra et al.[7] for data analysis. We started by measuring the line coverage of each tool - AI, conventional, and manually written tests. To ensure consistency, a widely used open-source code coverage tool, JaCoCo[20], was utilized across all tests. Next, we computed the mutation score (RQ2) for all test groups using PIT, a well-established tool for mutation testing [21]. To answer RQ3, we performed the same actions as Serra et al. [7]. We first generated tools for the fixed versions of classes and ran those tests against their faulty versions. Then we generated tests directly for the faulty versions of the classes because, as mentioned by Serra et al., "RANDOOP generates two kinds of suites: a regression suite that records the current behavior and an error-revealing suite that checks for specific specifications or contract violations" [7]. We considered both cases in the fault detection analysis and displayed them separately. Finally, we calculate the average for each tool regarding code coverage, mutation score, and fault detection to compare the performance of each tool. Additionally, we measure the recall, precision, and F-measure to evaluate the results of the fault detection (see Section 3).

2.3 Reliability and Validity

External validity: External validity is addressed by utilizing tools that are widely used and well-established in the field. Moreover, we followed the same methodology performed in the study by Serra et al. [7] and picked a randomized sample of classes from different real-world open-source projects provided by Defects4J, from the ones mentioned in the study by Shamshiri et al. [8], therefore allowing us to compare our findings with their results when applicable. The use of the Defects4J framework, which provides a database of real-world software bugs written in Java, ensures the repeatability of the experiment. In Section 4, the tables reference Bug IDs from Defects4J, which consist of an abbreviation of the project name (e.g., Time) followed by the bug number (e.g., 8). These Bug IDs can be used to retrieve further information about the bug and project from Defects4J. The appendix 1.7 provides the residing source file path and a link to the bug-specific report for convenience.

Internal validity: To ensure the validity of our results, we manually investigated any failing tests and dismissed any false positives or flaky tests. This involved comparing the failing error messages of the automated tests generated by the tools with the failing error messages of manually written developer tests in the buggy version of each class in order to accurately reflect the performance of the tools.

Construct validity: The use of established tools like JaCoCo and PIT for measuring the coverage and mutation score variables also contributes to ensuring construct validity. In addition, The constructs are clearly defined, aligning with established practices in the field ([6], [7], [10], [8]). Furthermore, the study replicates previous research, ensuring consistency and comparability ([6], [7], [8]). Moreover, confounding factors such as flaky tests and false positives, are addressed through manual investigation and eliminated.

Reliability: The study maintains reliability by making use of the publicly available Defects4J dataset and ensuring consistency through the utilization of publicly accessible tools for data gathering and analysis across all subject classes. Additionally, the generation of tests for each subject class is repeated 10 times to address the inherent randomness in conventional unit test generation tools, thereby improving the reliability and consistency of the obtained results. This repetition helps to provide more consistent and reliable results.

2.4 Ethical Considerations

Considering that we are conducting a controlled experiment, there are no significant ethical considerations in the project's research design to be mentioned. The experiment solely relies on publicly available open-source projects from Defects4J, which is licensed under the MIT License. Furthermore, all the tools used in the experiment are also publicly available open-source, with the exception of DIFFBLUE COVER which is a commercial tool. Moreover, the study does not involve human participants or sensitive data. The data collected and analyzed focuses on evaluating the effectiveness of different test generation tools regarding code coverage, mutation score, and fault detection capabilities. As long as the experiment adheres to the terms and conditions of Defects4J and the tools used the experiment's design may not introduce any direct ethical concerns.

3 Theoretical Background

3.1 Introduction to Software Testing and Automated Unit Test Generation

Software testing is a critical process in software development, that aims to identify defects and errors in software systems. It involves executing the software to find any inconsistencies between the expected and actual behavior of the codebase and ensure the quality and functionality of the software. There are various types of software testing, such as unit testing, which focuses on testing individual components in isolation. Integration testing verifies the interaction and compatibility between different components of the system. Regression testing involves retesting previously tested functionality to ensure that recent changes to the software have not introduced new defects. Other types of testing include performance testing, end-to-end testing, security testing, and mutation testing.

The creation of tests is a crucial but time-consuming process that has led to the development of many automated test generation tools to improve developer productivity and increase the quality of test suites. Automated unit test generation tools are tools that can automatically generate unit tests for software components. These tools primarily aim to assist developers in increasing code coverage and finding regressions

3.2 Conventional Automated Unit Test Generation Techniques in Java

3.2.1 Random-based Techniques for Unit Test Generation

Random-based techniques for unit test generation refer to approaches that make use of randomization to generate unit tests such as functions, methods, or classes. These techniques aim to explore different execution paths and input combinations by randomly generating test inputs.

RANDOOP, one of the conventional tools used in the experiment utilizes a technique called feedback-directed random testing which combines random-based algorithms and feedback mechanisms to generate tests [22]. In short, RANDOOP creates tests by putting together small pieces of code called method calls as shown in Figure 3.1. It chooses these method calls randomly and picks the values to use as inputs from the tests it has already created. Once it creates a new test, it runs it and checks if any rules or agreements are broken. If a test breaks a rule, it informs the user while if a test does something wrong or illegal, it gets rid of that test [22].

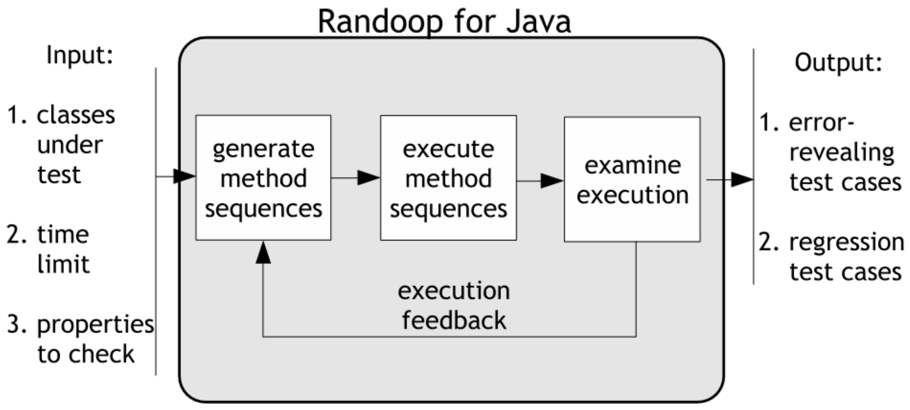


Figure 3.1: A simplified overview of RANDOOP's test generation process.
Source: [22]

3.2.2 Search-Based Technique for Unit Test Generation

A search-based technique for unit test generation refers to approaches that make use of searching algorithms to automatically generate unit tests. This technique explores the space of all possible test inputs and generates tests that satisfy certain criteria such as code coverage or a specific fault.

EVOSUITE, one of the conventional tools used in this experiment utilizes a search-based algorithm called Genetic Algorithm (GA) [23] as illustrated in Figure 3.2. In brief, this algorithm tries to solve Problems by imitating how living things adapt in nature. It starts with a random group of solutions and continues evolving them until it finds a solution that meets certain criteria, (e.g. code coverage) or until it has used all resources. For every step of the evolution, a new group is created and filled with the best solutions from the previous group where the fittest solutions are chosen (selection). Next, some of the solutions might be combined to create new solutions (crossover) and others might undergo small changes (mutation). Lastly, the new solutions are added to the new group and this process is repeated until a solution is found or resources are depleted [23].

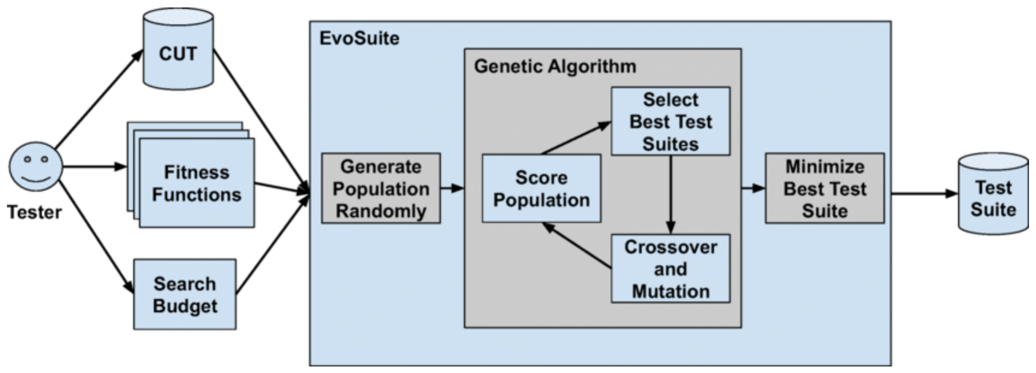


Figure 3.2: A simplified overview of EVOSUITES's test generation process.
Source: [24]

3.2.3 Dynamic Symbolic Execution (DSE)

Dynamic Symbolic Execution is another technique that can be used for automated unit test generation. It works by instrumenting and running a program while exploring all the different paths it can take and keeping track of all the input constraints, conditions, or other rules. After the program finishes executing and all the constraints and rules are collected, it uses an SMT (Satisfiability Modulo Theories) solver to figure out new inputs or data to guide the program towards new execution paths [25]. Utilizing DSE can result in high coverage percentages and is less prone to false positives. However, currently, there are no mature tools based on DSE available outside of academia [25].

3.3 AI-Based Unit Test Generation in Java

3.3.1 Reinforcement Learning Technique for Unit Test Generation

In recent years tools utilizing AI and machine learning models for unit test generation have emerged. DIFFBLUE COVER, the AI-based tool used in our study is an autonomous tool that uses AI to analyze the byte code of the methods in the code base to automatically generate tests as shown in Figure 3.3. It identifies multiple approaches to invoke each method and employs reinforcement learning to achieve comprehensive coverage of the entire codebase [26]. DIFFBLUE COVER is deterministic, meaning that each time it is used to generate tests, it will produce the same test suite unless changes are made in the codebase.

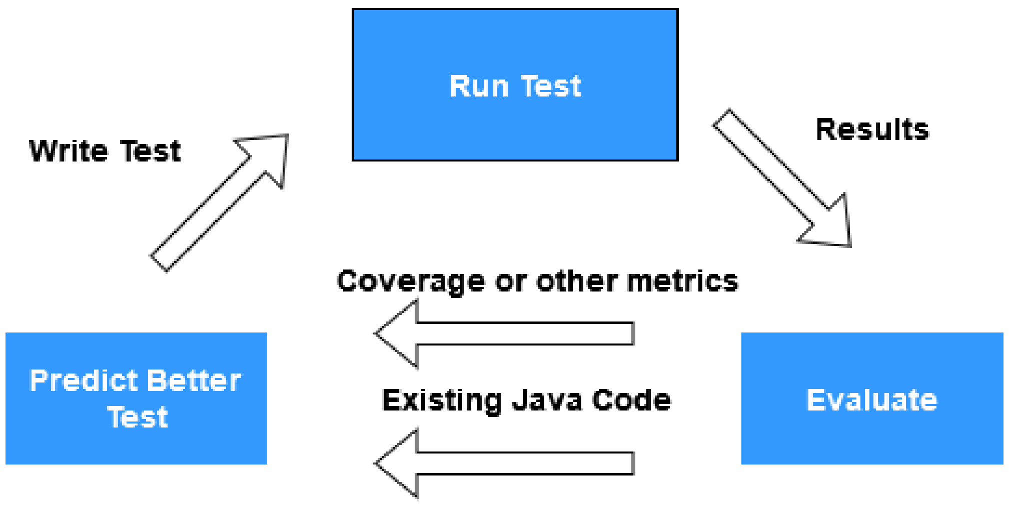


Figure 3.3: A simplified overview of DIFFBLUE's test generation process.
Source: Adapted from [27]

3.3.2 Large Language Models for Unit Test Generation

Another type of AI used for test generation is using transformer-based large language models (LLMs) such as MACHINET which we were unable to include in this study due to usage restriction. However, such tools have certain obvious limitations such as being unpredictable with minor prompt changes leading to different results. Furthermore, LLMs cannot reason Considering that Language Models solely offer the most probable text completion for a given prompt without providing substantial assurances on reasoning metrics [28].

3.4 Concepts in Test Suite Evaluation

3.4.1 Mutation Score

Mutation testing is a testing technique that creates deliberate modifications in the source code (mutants) with every mutant representing a single alteration from the original program. The mutants are then tested by the test suite to determine how many mutants can pass through the tests undetected (survived) and how many are caught (killed) [29]. The mutation score is calculated by dividing the number of killed mutants by the total number of generated mutants (as shown in Equation 1). Table 3.1 provides a list of the most common mutation operators in object-oriented programming (OOP) languages.

$$Mutation\ Score = (\frac{Number\ of\ Mutants\ Killed}{Total\ Number\ of\ Mutants\ Generated}) * 100 \tag{1}$$

Table 3.1: Mutation Operations in Mutation Testing for OOP Languages

Mutation Operation	**Description**
Arithmetic Operator Mutations	Mutations that alter arithmetic operators, such as replacing addition (+) with subtraction (-), multiplication (*) with division (/), or vice versa.
Relational Operator Mutations	Mutations that modify relational operators, such as changing less than (<) to greater than (>) or equal to (==) to not equal to (!=).
Conditional Operator Mutations	Mutations that affect conditional operators, such as replacing logical AND (&&) with logical OR (||) or vice versa.
Assignment Operator Mutations	Mutations that modify assignment operators, such as replacing "=" with "+=", "-=", "*=", or "/=".
Increment/Decrement Mutations	Mutations that change the increment (++) or decrement (–) operators to their counterparts, or remove them altogether.
Method Call Mutations	Mutations that modify method invocations, such as changing the method names, and adding, removing, or changing the order of arguments
Conditional Statement Mutations	Mutations that alter conditional statements, such as changing in if/else statements, reversing the condition, or removing it entirely.
Loop Statement Mutations	Mutations that affect loop statements, such as modifying loop conditions, variables, or boundaries.
Exception Handling Mutations	Mutations that modify exception handling code, such as replacing a thrown exceptions, or changing their handling strategy.
Object State Mutations	Mutations that modify object state, such as modifying the visibility of attributes and methods or removing, adding, and changing the values of attributes.
Inheritance Mutations	Mutations that affect inheritance relationships, such as changing the superclass or modifying method overrides in subclasses.
Encapsulation Mutations	Mutations that modify the visibility of attributes or methods, such as changing a method from public to private.
Interface Mutations	Mutations that alter interface implementations, such as changing method signatures or adding/removing implemented interfaces.
Polymorphism Mutations	Mutations that affect polymorphic behavior, such as changing method calls or adding/removing method overrides.
Object Creation Mutations	Mutations that modify object creation and instantiation, such as changing constructor arguments or removing object creation and instantiation.

Null Mutations	Mutations that add or remove null values, such as assigning null to variables or returning null from methods.
Boundary Mutations	Mutations that target boundary conditions, such as modifying loop boundaries, changing literals, or using extreme values in calculations or comparisons.
Control Flow Mutations	Mutations that modify control flow structures, such as adding/removing control flow statements or changing loop conditions.
Resource Management Mutations	Mutations that affect resource management, such as introducing resource leaks.
Thread Safety Mutations	Mutations that target thread safety issues, such as introducing race conditions, changing synchronization mechanisms, or modifying shared data access.

3.4.2 Code Coverage

Code coverage is a measure of how much of a software's source code is exercised during the execution of a test suite. Higher code coverage can be an indicator that a large portion of the code has been tested by the test suite. However, code coverage alone does not guarantee the absence of faults and the quality of a test suite [17]. Different types of code coverage focus on different aspects. For example, path coverage focuses on the percentage of unique paths through the code that are exercised during testing and is calculated as shown in Equation 2. Branch coverage is concerned with the percentage of branches or decision points within the code that is executed during testing. Branch coverage is measured as displayed in Equation 3. Lastly, line coverage measures the percentage of lines of code that are executed during testing by determining if a line of code has been executed at least once. Line coverage can be calculated as shown in Equation 4. In this experiment, we are measuring line coverage to align with the methodology used by Serra et al. [7].

$$Path\ Coverage = (\frac{Number\ of\ Unique\ Paths\ Covered}{Total\ Number\ of\ Unique\ Paths}) * 100 \tag{2}$$

$$Branch\ Coverage = (\frac{Number\ of\ Branches\ Covered}{Total\ Number\ of\ Branches}) * 100 \tag{3}$$

$$Line\ Coverage = (\frac{Number\ of\ Executed\ Lines}{Total\ Number\ of\ Lines}) * 100 \tag{4}$$

3.4.3 Fault Detection

The ability of a test suite to detect faults or regressions directly impacts the quality of a test suite. Fault detection in testing refers to the process of identifying defects or faults that cause the system to deviate from its intended functionality.

This process can involve various techniques. For example, dynamic testing may involve functional testing, integration testing, regression testing, stress testing, and exploratory testing in order to identify any existing faults in the system. Other ways of searching for faults in the system can be done by using static code analysis tools, monitoring, and logging to detect issues in real-time, or user feedback.

Automated unit test generation tools can often be included to further test any scenarios that might have been missed by the developers during manual testing, or to explore boundary conditions and edge cases by generating test inputs that test the limits of input ranges or system constraints through fuzz testing.

3.5 Related Research

In recent years, there have been several studies on automated unit test generation. In 2008, Baccheli et al. [6] performed a study in the FREENET project using EVOSUITE, RANDOOP, and JTEXPERT to compare manually and automatically generated in the tests considering code coverage, mutation score, and fault detection ability. The results showed that automated tools could generate tests for numerous classes in a short period. Furthermore, Baccheli et al. found that the regression tests generated by these tools achieved high code coverage and mutation scores as well as, helped uncover unexpected scenarios, revealing defects that other approaches might miss. However, the research also identified several drawbacks, such as that developers were not compelled to thoroughly study the tested code. Additionally, these tools required a significantly higher number of test cases compared to manual tests to detect defects. Furthermore, automatically generated tests often had confusing methods and variable naming lacking self-explanation making it hard for developers to read.

Ten years later, Serra et al [7]. reconstructed the same experiment by Baccheli et al. with the addition of direct fault detection to measure any improvements. The results conclude that there haven't been any dramatic improvements since the original study. However, both of these studies are limited to comparing conventional automated generation tools with manual tests and were conducted only in the context of the FREENET project.

In 2015, Shamshiri et al. [8] performed a comprehensive study on a larger scale using the Defects4J dataset using RANDOOP, EVOSUITE, and AGITA to evaluate how well the generated test suites perform at detecting faults. The study found that test suites were able to detect 55.7% of the faults on average, while only 19.9% of all the individual test suites detected a fault.

In 2020, Souza et al. [9] conducted a large-scale investigation using ten open-source projects, totaling 1,368 classes aiming to compare the effectiveness of manually written tests and automatically generated tests using EVOSUITE and RANDOOP. The study focused on regression tests and evaluated the test suites in terms of line coverage and mutation score. Contrary to the findings reported by Bacheli et al. [6] and Serra et al. [7], Souza et al. discovered that, overall, manual tests performed better than automatically generated tests in terms of both line coverage and mutation score.

Another study performed by Almasi et al [10] using EVOSUITE and RANDOOP in an industrial scale application found that EVOSUITE detected 56.40% and 38.00% of these faults respectively. Additionally, Almasi et al highlighted that the developers in the industry are concerned with the readability and number of tests generated by these tools making them hard to integrate.

However, none of the above-mentioned related work explored the performance of AI-based tools for unit test generation in any context which is what this study aims to investigate.

4 Research project - Implementation

In this section, we describe the implementation of our controlled experiment. The section highlights the setup environment, tools used, classes under investigation, and all the steps taken to implement our method of data collection.

For the experiment, we began by randomly selecting a set of open-source projects built with Maven from the Defects4J dataset. The selected projects are Time, Math, and Lang.

The Defects4Js *Time* identifier refers to the joda-time project. A quality replacement for the Java date and time classes for pre-Java 8 projects - its design allows for multiple calendar systems, including, but not limited to, Gregorian, Julian, Buddhist, and Islamic while still providing a simple API[30]. This project encases a total of 26 bugs 3 out of which are being used in the experiment - Time 8, Time 16, and Time 20. Depending on the bug, the overall project size ranges from 81385 Lines Of Code(LOC) to 80259. Averaging 315 different classes.

The commons-math project can be accessed from the Defects4J framework using *Math* identifier. The commons-math is a library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the Java programming language or commons-lang [31]. From this project, which has a total of 106 bugs, three bugs are being utilized in our experiment. Namely, Math 61, Math 35, and Math 3. Out of the 3 projects, commons-math is the largest in terms of both LOC and the number of classes. Adjusting to developments made on the project between bugs, the total LOC is 116735 with over nine hundred(935) different classes.

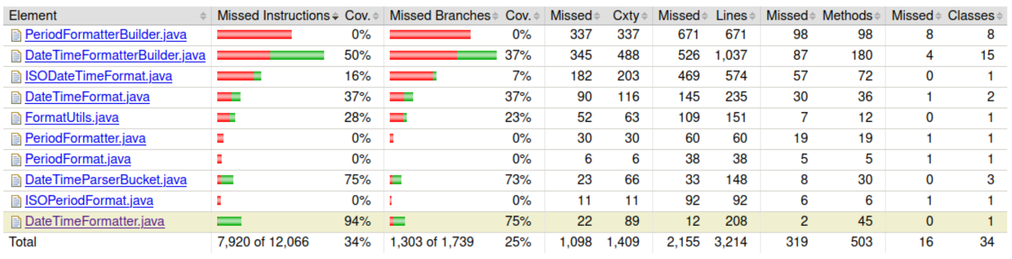
Lastly, the smallest project used in the experiment comes under 200 classes with an average LOC of 49799. The commons-lang project is abbreviated to *Lang* in the dataset. The main purpose of it is to provide extra methods to manipulate the core classes of Java, as standard Java libraries fail to do so. Notably, String manipulation methods, basic numerical methods, object reflection, concurrency, creation and serialization, and System properties [31]. Regardless of the size, this project houses a total of 64 bugs. Including bug Lang 4, Lang 34, and Lang 36 which will be used in the experiment.

All prior mentioned bugs are chosen based on a random selection. Each chosen bug has an accompanying manually written test to showcase it. Further details concerning the manual tests are exposed in Table 4.2 alongside the target class being tested.

Table 4.2: The details of the manual tests and target classes. The first column contains Bug ID. In the following column LOC of a class containing the given bug is provided. Next to it, is the number of methods the container class has. The last two columns have LOC and the number of assertions each manually-written test class has.

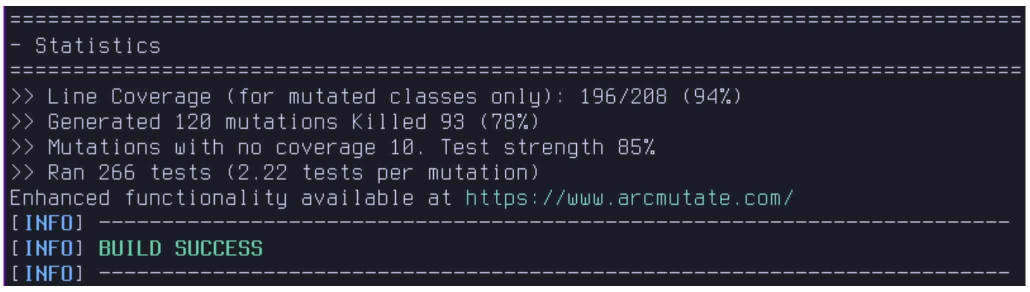
Bug ID	Container LOC	Methods	Test LOC	Assertions
Time 16	356	43	724	227
Math 66	151	2	106	13
Lang 4	44	2	23	4
Lang 36	635	46	995	502
Time 20	1786	181	279	78
Math 3	688	43	783	160
Lang 34	803	121	74	12
Math 35	38	4	69	2
Math 61	77	7	132	11
Time 8	586	58	790	233

With the selection of the bugs done, both fixed and buggy versions of the source project are generated. First, measurements were conducted to assess the quality of each manual test. The metrics used are Code Coverage and Mutation score. The Code Coverage is measured using JaCoCo while PIT is utilized to generate the Mutation score. Sample results from each tool are provided in Figure 4.4, and 4.5 respectively.



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
PeriodFormatterBuilder.java		0%		0%	337	337	671	671	98	98	8	8
DateTimeFormatterBuilder.java		50%		37%	345	488	526	1,037	87	180	4	15
ISODateTimeFormat.java		16%		7%	182	203	469	574	57	72	0	1
DateTimeFormat.java		37%		37%	90	116	145	235	30	36	1	2
FormatUtils.java		28%		23%	52	63	109	151	7	12	0	1
PeriodFormatter.java		0%		0%	30	30	60	60	19	19	1	1
PeriodFormat.java		0%		0%	6	6	38	38	5	5	1	1
DateTimeParserBucket.java		75%		73%	23	66	33	148	8	30	0	3
ISOPeriodFormat.java		0%		0%	11	11	92	92	6	6	1	1
DateTimeFormatter.java		94%		75%	22	89	12	208	2	45	0	1
Total	7,920 of 12,066	34%	1,303 of 1,739	25%	1,098	1,409	2,155	3,214	319	503	16	34

Figure 4.4: Sample result of JaCoCo: Manual test for bug Time 16.



```
================================================================================
- Statistics
================================================================================
>> Line Coverage (for mutated classes only): 196/208 (94%)
>> Generated 120 mutations Killed 93 (78%)
>> Mutations with no coverage 10. Test strength 85%
>> Ran 266 tests (2.22 tests per mutation)
Enhanced functionality available at https://www.arcmutate.com/
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

Figure 4.5: Sample result of PIT: Manual test for bug Time 16.

Afterward, we removed the manual tests completely and ran the selected automated unit test generation tools (EVOSUITE, RANDOOP) and the AI-based tool (DIFFBLUE

COVER) to generate tests for the fixed versions of the classes. We used the tools' default settings, allocated a time budget of 180 seconds, and repeated the process 10 times for each class while calculating line coverage and mutation score simultaneously. For illustration purposes, a small portion of the test suite generated by each mentioned tool is attached in Listing 1, 2, and 3. For the entire test class, please see [32].

```java
/**
 * Method under test: {@link MathArrays#scaleInPlace(double, double[])}
 */
@Test
public void testScaleInPlace() {
    // Arrange
    double[] arr = new double[]{10.0d, 1.0d, 10.0d, 1.0d};

    // Act
    MathArrays.scaleInPlace(10.0d, arr);

    // Assert
    assertEquals(100.0d, arr[0], 0.0);
    assertEquals(10.0d, arr[1], 0.0);
    assertEquals(100.0d, arr[2], 0.0);
    assertEquals(10.0d, arr[3], 0.0);
}

```

Listing 1: Portion of test suite generated for bug Math 3 by DIFFBLUE COVER.

```java
double[] doubleArray83 = org.apache.commons.math3.util.MathArrays.
    ebeAdd(doubleArray62, doubleArray77);
double[] doubleArray85 = org.apache.commons.math3.util.MathArrays.
    normalizeArray(doubleArray77, (-100.0d));
org.apache.commons.math3.util.MathArrays.scaleInPlace((-100.0d),
    doubleArray85);
double double87 = org.apache.commons.math3.util.MathArrays.distance(
    doubleArray14, doubleArray85);
double[] doubleArray88 = org.apache.commons.math3.util.MathArrays.scale
    (2.0d, doubleArray85);
...
org.junit.Assert.assertNotNull(doubleArray85);
org.junit.Assert.assertEquals(java.util.Arrays.toString(doubleArray85),
     "[10000.0]");
org.junit.Assert.assertTrue("'" + double87 + "' != '" + 9965.0d + "'",
    double87 == 9965.0d);

}

```

Listing 2: Portion of test suite generated for bug Math 3 by RANDOOP.

```java
@Test(timeout = 4000)
public void test174()  throws Throwable  {
    double[] doubleArray0 = new double[6];
    doubleArray0[0] = 202.0;
    doubleArray0[1] = 202.0;
    doubleArray0[2] = (-2005.874945627);
    doubleArray0[3] = 604.3128111436902;
    doubleArray0[4] = 202.0;
    doubleArray0[5] = 202.0;
    MathArrays.scaleInPlace(202.0, doubleArray0);
}

```

Listing 3: Portion of test suite generated for bug Math 3 by EVOSUITE.

The test suites were manually investigated for flaky tests, and any such tests were removed. Subsequently, we executed the generated test suites on the respective buggy versions. The results were manually analyzed for any false positives, and if any were found, they were removed. We spend 15-20 minutes investigating for false positives depending on the number of failing tests. Finally, we directly generated test suites for the buggy versions of the classes, following the same procedure. The Figure 4.6 illustrates the procedure.

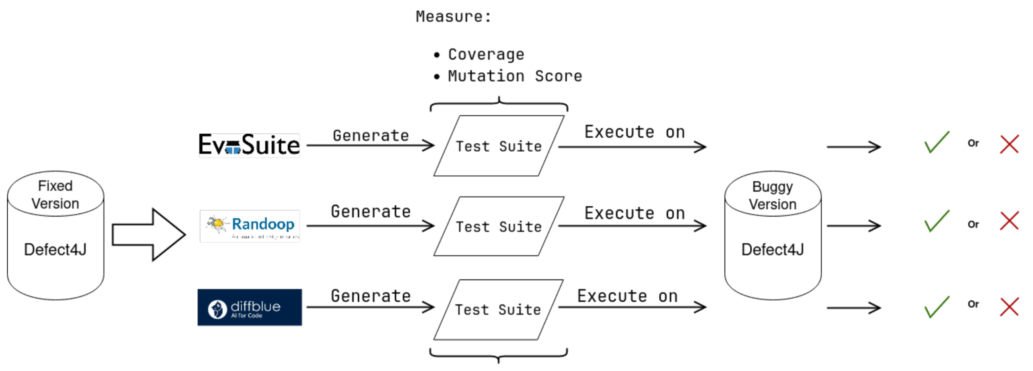


Figure 4.6: Summary of the experimental setup: For each subject class selected from the Defects4J dataset, we generate tests on the fixed version of the class, measure code coverage and mutation score, and execute the test on the buggy version.
Source: Adapted from [8]

The entire process is conducted on a system running Arch Linux with kernel version 6.3.5. Further specifications include 16 gigabytes of RAM, and 4 core CPU.

5 Analysis of the results

5.1 RQ1 - Coverage of AI Tools vs Conventional Automated Tests

Based on the data gathered for line coverage displayed in Table 5.3, it is evident that in multiple instances, all tools were able to achieve line coverage comparable to or higher than the manually written developer tests. However, there was an exception in the case of classes containing many private methods, such as Math 66, where only EVOSUITE outperformed the manually written tests. All three tools are unable to generate tests for private methods directly but they may indirectly cover them by testing the public methods that make calls to those private methods. EVOSUITE seems to be the only tool that performed well in this scenario. Overall, EVOSUITE achieved the highest coverage averaging 89%, followed by the manually written developer tests averaging 79%, while RANDOOP and DIFFBLUE COVER achieved the same coverage averaging 63%.

Our results indicate that the AI-based tool (DIFFBLUE COVER) did not surpass any of the conventional tools or the manual tests in terms of coverage. However, it is noteworthy that the AI-based tool generated significantly fewer tests on average, while still achieving high coverage percentages in many cases. For example, in the bug with ID Time 8, RANDOOP generated over 150,000 assertions with a coverage of 69%, EVOSUITE generated 3,147 assertions with a coverage of 86%, and DIFFBLUE COVER generated 128 assertions with a coverage of 70%. Additionally, DIFFBLUE COVER does not generate tests for trivial methods, which may contribute to an increase in overall coverage without necessarily improving test quality. This strongly suggests that the AI-based tool generates notably fewer redundant assertions.

Table 5.3: Results of line coverage achieved by the three testing tools along with the manually written tests on a set of bugs from the Defects4J dataset. The bug IDs are listed in the first column, followed by the coverage percentages obtained by each testing tool in the subsequent columns. The last row displays the average line coverage for each testing tool across all bugs.

Bug	Manual	Randoop	Evosuite	Diffblue Cover
Time 16	94%	27%	83%	83%
Math 66	95%	24%	100%	14%
Lang 4	92%	100%	100%	32%
Lang 36	97%	65%	86%	79%
Time 20	40%	30%	57%	43%
Math 3	82%	83%	96%	90%
Lang 34	23%	32%	91%	27%
Math 35	95%	100%	96%	94%
Math 61	88%	99%	98%	97%
Time 8	86%	69%	86%	70%
Overall	*79%*	*63%*	*89%*	*63%*

5.2 RQ2 - Mutation Score of AI Tools vs Conventional Automated Tests

In terms of mutation score, EVOSUITE achieved the highest score among the tools, averaging 67%. It was followed by RANDOOP with an average score of 50%. On the other hand, DIFFBLUE COVER had the lowest mutation score, averaging 40%. The inability of both DIFFBLUE COVER and RANDOOP to generate tests for private methods had an impact on the mutation score, as observed in the results for Math 66 in Table 5.4. However, DIFFBLUE COVER, the AI-based tool, was able to achieve a higher mutation score only for Lang 36, where the class under test consisted solely of static methods. However, among our selected test subjects, only EVOSUITE was able to outperform the manually written tests averaging 63% in terms of mutation score.

Overall, these findings suggest that, when it comes to effectively detecting faults through mutation testing, both manually written tests and conventional automated unit test generation tools outperformed the AI-based tool.

Table 5.4: Results of mutation score achieved by the three testing tools along with the manually written tests on a set of bugs from the Defects4J dataset. The bug IDs are listed in the first column, followed by the mutation score obtained by each testing tool in the subsequent columns. The last row displays the average mutation score for each testing tool across all bugs.

Bug	Manual	Randoop	Evosuite	Diffblue Cover
Time 16	79%	24%	55%	43%
Math 66	82%	8%	73%	6%
Lang 4	60%	80%	72%	7%
Lang 36	79%	38%	56%	58%
Time 20	25%	19%	27%	27%
Math 3	66%	83%	49%	54%
Lang 34	4%	6%	86%	13%
Math 35	79%	79%	83%	57%
Math 61	91%	98%	79%	91%
Time 8	62%	68%	92%	46%
Overall	*63%*	*50%*	*67%*	*40%*

5.3 RQ3 - Faults Detected by AI Tools vs Conventional Automated Tests

Regarding the fault detection capability, Table 5.5 provides information on whether the respective tools were able to identify the bugs. RANDOOP, which generated both regression and error-revealing tests, unfortunately, didn't generate any error-revealing tests (E.R) for the subject classes under investigation. In terms of overall performance, EVOSUITE demonstrated the highest fault detection capability for regression tests, successfully identifying 7 out of 10 faults. RANDOOP came in second place, detecting 5 out of 10 faults, while DIFFBLUE COVER had the lowest performance, identifying 4 out of 10 faults. It's worth noting that none of the tools were able to identify faults when generating bugs directly for the faulty version of the classes which does not come as a surprise since the primary usage of these tools is to identify regressions. Another possible reason could be not all bugs are necessarily caused by coding errors, but rather by deviations from the intended behavior or functionality.

Our results indicate that the AI-based tool, DIFFBLUE COVER, performed worse than the conventional tools in terms of fault detection using regression tests. It is worth mentioning that Papadakis et al. found that both the size of the test suite and the mutation score have an impact on fault detection, but weak correlations exist between mutation score and fault detection when controlling for test suite size [18]. DIFFBLUE COVER generated the fewest number of assertions and achieved the lowest mutation score among the tools, which could potentially be correlated with its lower number of detected faults.

Table 5.6 provides information on the number of flaky tests encountered when executing the test suite generated for the fixed version on the buggy version. Among the tested tools, DIFFBLUE COVER exhibited the highest number of flaky tests during regression, with a total of 5 cases observed across all classes, while RANDOOP had the smallest number of cases with only 1. EVOSUITE fell in the middle, with a total of 3 cases of flaky tests during regression testing. It is worth noting that DIFFBLUE COVER had flaky tests for only 2 bug instances, whereas EVOSUITE had flaky tests for 3 bug instances. Additionally, none of the tools produced any false positives. We did not investigate the test suites generated directly for the buggy version since none of the tools were able to identify any bugs in that case.

Table 5.5: Summary of fault detection achieved by three testing tools, along with manually written tests, for a set of bugs from the Defects4J dataset. The bug IDs are listed in the first column. The subsequent columns indicate whether the bug was detected (+), not detected (-), or if no tests were generated (*) by each testing tool. The last row displays the number of faults detected by each testing tool across all bugs, including regression and directly generated tests for the faulty classes.

Bug	**Randoop Reg.**		**Randoop E.R.**		**Evosuite**		**Diffblue Cover**	
	Reg	Direct	Reg	Direct	Reg	Direct	Reg	Direct
Time 16	-	-	*	*	-	-	-	-
Math 66	+	-	*	*	+	-	+	-
Lang 4	-	-	*	*	-	-	-	-
Lang 36	-	-	*	*	+	-	-	-
Time 20	-	-	*	*	-	-	-	-
Math 3	+	-	*	*	+	-	-	-
Lang 34	-	-	*	*	+	-	+	-
Math 35	+	-	*	*	+	-	+	-
Math 61	+	-	*	*	+	-	+	-
Time 8	+	-	*	*	+	-	-	-
Overall	*5/10*	*0*	*0*	*0*	*7/10*	*0*	*4/10*	*0*

Table 5.6: The number of false positives identified, i.e., the number of different types of failing tests when executing the test suite generated on the fixed version compared to the buggy version, is reported in the table. The term "different types" implies that if multiple test cases fail due to the same reason, they are considered as one instance of a false positive. It should be noted that flaky tests can technically be considered as false positives [8], but in this context, we distinguish them because false positives primarily occur due to the breaking of encapsulation or outdated mocking behavior as we mentioned in Section 2. Since none of the tools resulted in any false positives, the table only includes the number of flaky tests identified.

Bug	**Randoop Reg.**	**Evosuite**	**Diffblue Cover**
	Flaky Tests	Flaky Tests	Flaky Tests
Time 16	0	0	0
Math 66	1	1	0
Lang 4	0	1	0
Lang 36	0	0	3
Time 20	0	0	0
Math 3	0	0	0
Lang 34	0	1	0
Math 35	0	0	0
Math 61	0	0	0
Time 8	0	0	2
Overall	*1*	*3*	*5*

After analyzing the fault detection capabilities of the tested tools, we aimed to further evaluate their results using recall, precision, and F-measure. These metrics provide a more detailed assessment of the effectiveness of the tools in identifying faults and are defined as follows:

- **True Positive (TP):** The number of bugs correctly identified by the testing tool.
- **False Positive (FP):** The number of bugs incorrectly identified by the testing tool (i.e., flaky or false positive tests).
- **False Negative (FN):** The number of bugs not identified by the testing tool.

In our experiment, each class under investigation contained one bug, resulting in True Positives (TP) being either 1 or 0. The number of False Positives (FP) for each tool (RANDOOP, EVOSUITE, DIFFBLUE COVER) can be seen in Table 5.6. Similarly, the False Negatives (FN) would also be either 1 or 0, as each class only contained a single bug.

Recall (also known as Sensitivity) measures the proportion of bugs that were correctly identified by the testing tool out of all the bugs that exist. It is calculated using Equation 5.

$$Recall = \frac{TP}{TP + FN} \tag{5}$$

Precision measures the proportion of failing tests that were true bugs out of all the failures identified. The Precision value is calculated using Equation 6.

$$Precision = \frac{TP}{TP + FP} \tag{6}$$

To compute the F-measure, we first calculate the numerator using Equation 7, which is the harmonic mean of Precision and Recall. Then, the denominator is calculated using Equation 8, which is the sum of Precision and Recall. The F-measure provides a balanced measure between Precision and Recall and is computed using Equation 9.

$$Numerator = 2 \times (Precision \times Recall) \tag{7}$$

$$Denominator = Precision + Recall \tag{8}$$

$$F - measure = \frac{Numerator}{Denominator} \tag{9}$$

The results of recall, precision, and F-measure for each of the tools can be found in Figure 5.7. In brief, EVOSUITE demonstrated a relatively high recall rate of 0.7, indicating that it was able to identify a significant portion of the bugs and a precision rate of 0.7, suggesting that the identified bugs were mostly true positives. The F-measure for EVOSUITE was also 0.7, indicating a balanced performance between recall and precision.

RANDOOP Regression tests achieved a recall rate of 0.5, meaning it successfully identified half of the bugs and a relatively high precision rate of 0.833, suggesting a low number of false positives. The F-measure for RANDOOP Regression was 0.625, indicating a moderate balance between recall and precision.

DIFFBLUE COVER achieved a recall rate of 0.4, indicating it identified a lower proportion of the bugs compared to the other tools. The precision rate for DIFFBLUE COVER was at 0.444, suggesting a relatively higher number of false positives compared to EVOSUITE and RANDOOP. The F-measure for DIFFBLUE COVER was 0.42, indicating a moderate balance between recall and precision.

These results further support the conclusion that EVOSUITE performed the best among the tested tools in terms of fault detection, as it achieved the highest values for recall, precision, and F-measure, while on the other hand, DIFFBLUE COVER performed the lowest in terms of fault detection, with lower values for recall, precision, and F-measure.

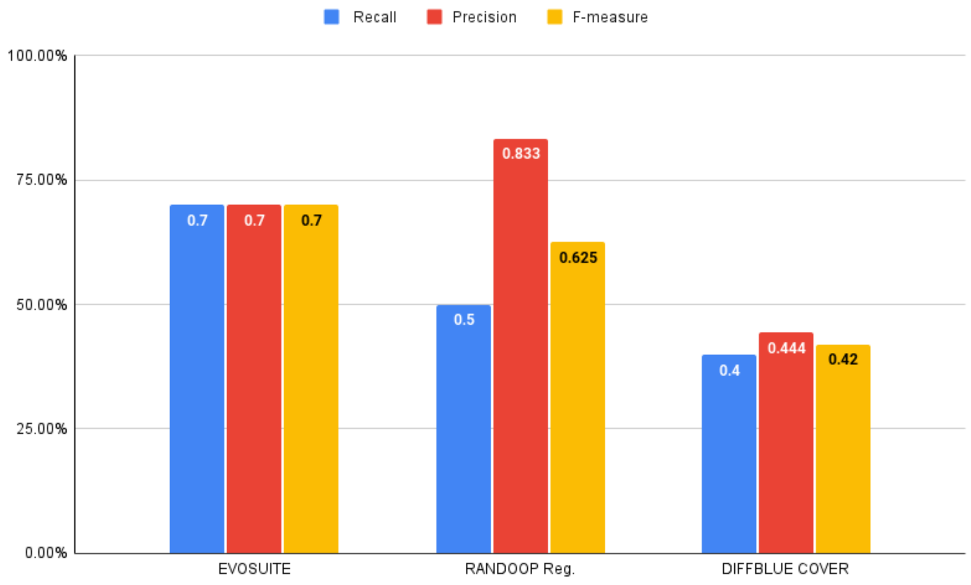


Figure 5.7: The results for recall, precision, and F-measure for each tool.

6 Discussion

Our empirical study compared manually written tests by developers, two conventional automated unit test generation tools, and one publicly available AI-based automated unit test generation tool. By conducting experiments on the Defects4J dataset, our results show the following: 1) In terms of code coverage, conventional tools outperformed both manual and AI-generated tests. EVOSUITE took the first place, averaging 89%, while the AI-based tool DIFFBLUE COVER shared the last place, averaging 63%. When comparing these results to the study conducted by Serra et al. (Table I in [7]), it is important to note that, although our experiment used a different set of subject classes, our findings indicate that EVOSUITE continues to outperform RANDOOP in terms of code coverage. 2) For mutation score, the AI-based tool performed the worst, averaging at 40%, while EVOSUITE performed the best, averaging at 67%. Similarly to the study conducted by Serra et al. (Table I in [7]), our results demonstrate that EVOSUITE achieved a higher mutation score and continued to outperform RANDOOP in this criterion. 3) Regarding fault detection using regression, the conventional tools surpassed the AI-based tool. EVOSUITE detected 7 out of 10 bugs, RANDOOP detected 5 out of 10, and DIFFBLUE COVER detected 4 out of 10. However, none of the tools were able to find any defects when generating tests directly for the buggy version of the subject classes. Comparing the obtained results with the findings of the study conducted by Serra et al. (Table I in [7]), it can be observed that EVOSUITE once again outperforms RANDOOP in terms of fault detection capabilities for regression tests. Furthermore, our findings for the subset of bugs we selected from the study conducted by Shamshiri et al. [8] align with their reported results. Both EVOSUITE and RANDOOP identified the same faults as reported by them, without any additional or fewer detections. Additionally, none of the tools produced any false positives, aligning with the findings of Shamshiri et al. [8].

Our findings suggest that conventional tools still outperform the AI-based tool regarding code coverage, mutation score, and fault detection. Nonetheless, it is important to note that the AI-based tool generated a significantly smaller number of assertions and flaky tests while still achieving high percentages of code coverage, mutation score, and detecting an adequate number of faults. Additionally, DIFFBLUE COVER generates far more readable tests but also includes comments. This can contribute to better integration, debugging, maintainability, and effectively addressing the concerns raised by developers, as mentioned by Almasi et al. [10].

7 Threats to validity

External validity: In our study, we selected a sample of 10 classes from 3 open-source Java projects thus, the findings and conclusions drawn from this study may not adequately represent the broader variety of projects, programming languages, and their specific characteristics and semantic differences. Another external validity threat is the limited usage of AI-based tools in the experiment. Due to the novelty of AI-based unit test generation tools, we were able to get access to only one tool, DIFFBLUE COVER. Other tools were either still under heavy development and not publicly available, or not accessible for free use (MACHINET).

Internal Validity: Even though we manually investigated all failure reasons exhibited by the generated test suites and compared them with the results from the manually written developer test to ensure their validity, there is a chance that some test failures may have been due to the same underlying issue as the bug, rather than a flaky test. As a result, a failure could have potentially indirectly discovered the bug but was overlooked by us during the investigation. Additionally, we spent approximately 15 to 20 minutes for each class iteration, depending on the number of failing tests, in order to manually verify whether a test failure was a false positive or not. The time constraint of this process could potentially have impacted the results.

Construct Validity: According to Shamshiri et al., "Each bug is represented by a minimized difference between the buggy version and a later fixed version, rather than the actual code change that introduced the bug. Consequently, while the bugs are indeed genuine, not all of them may accurately represent regression faults" [8]. Suggesting that the results might underestimate the capabilities of the tools. Furthermore, it is important to acknowledge that the primary usage of the selected automated unit testing tools is regression testing and not discovering existing bugs in the code base.

8 Conclusions and Future Work

We investigated the performance of an AI-based test generation tool regarding code coverage, mutation score, and fault detection by empirically comparing them to conventional unit test generation tools. Our results showed that even though the AI tool was surpassed in all three criteria by the conventional tools it was able to achieve satisfactory results while generating fewer redundant test cases and generating more readable code in certain cases, even surpassing conventional tools. This indicates that although conventional automated unit test generation tools currently hold an advantage in terms of performance, AI-based tools demonstrate promise and offer specific advantages that make them more attractive to developers and the industry. We hope that our findings will inspire further research to improve the capabilities of AI unit test generation tools. Additionally, we hope that our research will serve as a means to evaluate future advancements in this field.

In the future, we plan to repeat the experiment using a larger number of available AI tools for unit test generation, alongside a larger sample of projects from the Defects4J dataset. This will allow us to further explore and gather more comprehensive insights into the capabilities and improvements of AI in unit test generation.

References

[1] M. A. Jamil, M. Arif, N. S. A. Abubakar, and A. Ahmad, “Software testing techniques: A literature review,” in *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, 2016, pp. 177–182. [Online]. Available: https://doi.org/10.1109/ICT4M.2016.045

[2] S. S. R. Ahamed, “Studying the feasibility and importance of software testing: An analysis,” *CoRR*, vol. abs/1001.4193, 2010. [Online]. Available: http://arxiv.org/abs/1001.4193

[3] C. University. (2013) Financial content: Cambridge university study states software bugs cost economy $312 billion per year. [Online]. Available: https://www.prweb.com/releases/2013/1/prweb10298185.htm

[4] M. Ellims, J. Bridges, and D. C. Ince, “The economics of unit testing,” *Empirical Software Engineering*, vol. 11, no. 1, pp. 5–31, Mar 2006. [Online]. Available: https://doi.org/10.1007/s10664-006-5964-9

[5] P. McMinn, “Search-based software test data generation: a survey,” *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.294

[6] A. Bacchelli, P. Ciancarini, and D. Rossi, “On the effectiveness of manual and automatic unit test generation,” in *2008 The Third International Conference on Software Engineering Advances*, 2008, pp. 252–257. [Online]. Available: https://doi.org/10.1109/ICSEA.2008.66

[7] D. Serra, G. Grano, F. Palomba, F. Ferrucci, H. C. Gall, and A. Bacchelli, “On the effectiveness of manual and automatic unit test generation: Ten years later,” in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 121–125. [Online]. Available: https://doi.org/10.1109/MSR.2019.00028

[8] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, “Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t),” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 201–211. [Online]. Available: https://doi.org/10.1109/ASE.2015.86

[9] B. Souza and P. Machado, “A large scale study on the effectiveness of manual and automatic unit test generation,” in *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, ser. SBES ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 253–262. [Online]. Available: https://doi.org/10.1145/3422392.3422407

[10] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, “An industrial evaluation of unit test generation: Finding real faults in a financial application,” in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2017, pp. 263–272. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2017.27

[11] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 437–440. [Online]. Available: https://doi-org.proxy.lnu.se/10.1145/2610384.2628055

[12] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 416–419. [Online]. Available: https://doi.org/10.1145/2025113.2025179

[13] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, ser. OOPSLA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 815–816. [Online]. Available: https://doi.org/10.1145/1297846.1297902

[14] A. Gambi, G. Jahangirova, V. Riccio, and F. Zampetti, "Sbst tool competition 2022," in *Proceedings of the 15th Workshop on Search-Based Software Testing*, ser. SBST '22. New York, NY, USA: Association for Computing Machinery, 2023, p. 25–32. [Online]. Available: https://doi-org.proxy.lnu.se/10.1145/3526072.3527538

[15] Diffblue. (2020) Diffblue is a pioneering generative ai for code company with a mission to change the way software is written. [Online]. Available: https://www.diffblue.com/about-us

[16] (2022) Ai in java: How ai is revolutionizing software development. [Online]. Available: https://blog.machinet.net/post/ai-in-java-how-ai-is-revolutionizing-software-development

[17] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 72–82. [Online]. Available: https://doi-org.proxy.lnu.se/10.1145/2568225.2568278

[18] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, "Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 537–548. [Online]. Available: https://doi-org.proxy.lnu.se/10.1145/3180155.3180183

[19] C. Wohlin, M. Höst, and K. Henningsson, *Empirical Research Methods in Web and Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 409–430. [Online]. Available: https://doi.org/10.1007/3-540-28218-1_13

[20] A. Parsai and S. Demeyer, "Comparing mutation coverage against branch coverage in an industrial setting," *International Journal on Software Tools for Technology Transfer*, vol. 22, pp. 1–24, 08 2020. [Online]. Available: https://doi.org/10.1007/s10009-020-00567-y

[21] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: A practical mutation testing tool for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 449–452. [Online]. Available: https://doi-org.proxy.lnu.se/10.1145/2931037.2948707

[22] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, ser. OOPSLA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 815–816. [Online]. Available: https://doi.org/10.1145/1297846.1297902

[23] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013. [Online]. Available: https://doi.org/10.1109/TSE.2012.14

[24] H. Almulla and G. Gay, "Learning how to search: generating effective test cases through adaptive fitness function selection," *Empirical Software Engineering*, vol. 27, no. 2, p. 38, Jan 2022. [Online]. Available: https://doi.org/10.1007/s10664-021-10048-8

[25] T. Chen, X. song Zhang, S. ze Guo, H. yuan Li, and Y. Wu, "State of the art: Dynamic symbolic execution for automated test generation," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1758–1773, 2013, including Special sections: Cyber-enabled Distributed Computing for Ubiquitous Cloud and Network Services Cloud Computing and Scientific Applications — Big Data, Scalable Analytics, and Beyond. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X12000398

[26] Diffblue. Write java unit tests automatically with ai for code. [Online]. Available: https://info.diffblue.com/hubfs/Product/Cover%20Core%20feature%20sheet%20100223-05.pdf

[27] A. Piper. (2022, Oct.) Copilot to cover: Why ai can't replace developers with robots, but can make life better. [Online]. Available: https://www.slideshare.net/AndyPiper1/copilot-to-cover-why-ai-cant-replace-developers-with-robots-but-can-make-like-better

[28] K. Valmeekam, A. Olmo, S. Sreedharan, and S. Kambhampati, "Large language models still can't plan (a benchmark for llms on planning and reasoning about change)," 2023. [Online]. Available: https://arxiv.org/abs/2206.10498

[29] J. Možucha, e. P. Rossi, Bruno", A. Jedlitschka, A. Nguyen Duc, M. Felderer, S. Amasaki, and T. Mikkonen, "Is mutation testing ready to be adopted industry-wide?" in *Product-Focused Software Process Improvement*. Cham: Springer International Publishing, 2016, pp. 217–232. [Online]. Available: https://doi.org/10.1007/978-3-319-49094-6_14

[30] Joda-time. [Online]. Available: https://www.joda.org/joda-time/

[31] Apache commons. [Online]. Available: https://commons.apache.org/

[32] (2023) Sample of test suites generated by the tools used in the experiment. [Online]. Available: https://doi.org/10.6084/m9.figshare.23302085

A Appendix 1

Table 1.7: Modified source file and Bug report of each bug report in Defects4J.

Time 16	
Source	org.joda.time.format.DateTimeFormatter
Report	https://sourceforge.net/p/joda-time/bugs/148
Math 66	
Source	org.apache.commons.math.optimization.univariate.BrentOptimizer
Report	https://issues.apache.org/jira/browse/MATH-395
Lang 4	
Source	org.apache.commons.lang3.text.translate.LookupTranslator
Report	https://issues.apache.org/jira/browse/LANG-882
Lang 36	
Source	org.apache.commons.lang3.math.NumberUtils
Report	https://issues.apache.org/jira/browse/LANG-521
Time 20	
Source	org.joda.time.format.DateTimeFormatterBuilder
Report	https://sourceforge.net/p/joda-time/bugs/126
Math 3	
Source	org.apache.commons.math3.util.MathArrays
Report	https://issues.apache.org/jira/browse/MATH-1005
Lang 34	
Source	org.apache.commons.lang3.builder.ToStringStyle
Report	https://issues.apache.org/jira/browse/LANG-586
Math 35	
Source	org.apache.commons.math3.genetics.ElitisticListPopulation
Report	https://issues.apache.org/jira/browse/MATH-776
Math 61	
Source	org.apache.commons.math.distribution.PoissonDistributionImpl
Report	https://issues.apache.org/jira/browse/MATH-349
Time 8	
Source	org.joda.time.DateTimeZone
Report	https://github.com/JodaOrg/joda-time/issues/42