Bachelor Degree Project

Comparative Analysis of Automated Test Case Generation Tools in Java

Author: Jatin Sachdeva, Sai Shashank Maktala
Supervisor: Daniel Toll
Semester: HT 2024
Subject: Computer Science

Abstract

Automated Test Case Generation (ATCG) tools play a pivotal role in enhancing the quality and efficiency of software testing by automating the creation of test cases. However, the comparative performance of these tools across varying project complexities remains underexplored. This thesis conducts a controlled experiment to evaluate four prominent ATCG tools EvoSuite, AgitarOne, JDART, and JTExpert using the Defects4J dataset, a curated collection of real-world Java projects.

The study assesses the tools based on three primary metrics: code coverage, execution time, and mutation score. These metrics provide insights into the tools' effectiveness in detecting faults, their efficiency under constrained time budgets, and their capability to thoroughly explore codebases. The findings reveal distinct trade-offs: JDART excels in fault detection and code coverage but demands significant computational resources, making it ideal for complex or safety-critical systems. EvoSuite demonstrates a balanced performance, achieving competitive coverage and efficiency, which makes it suitable for diverse testing scenarios. JTExpert offers rapid execution but struggles with fault detection in complex projects, while AgitarOne provides consistent, moderate performance across all metrics.

This comprehensive evaluation highlights the strengths and limitations of each tool, offering empirical guidance for selecting the most suitable ATCG tool based on specific project requirements. By bridging gaps in comparative analyses, this thesis contributes to the understanding of ATCG tools' capabilities, paving the way for improved testing practices and future research into hybrid methodologies and broader dataset inclusion.

Keywords: Automated Test Case Generation, Code Coverage, Mutation Score, Execution Time, Software Testing Tools, Java, Controlled Experiment, Defects4J.

Contents

1 Introduction

The term "Automated Test Case Generation" (ATCG) refers to the process of automatically creating test cases that validate the functionality and reliability of a software system. This process involves diverse techniques, including symbolic execution, which explores program paths using symbolic values [23], and search-based methods, which employ algorithms like genetic programming to optimize test case generation [24]. These approaches aim to improve software quality by maximizing coverage and efficiency while minimizing manual effort [23].

This thesis investigates the performance of ATCG tools through a controlled experiment. While ATCG tools have been widely studied, there are some that remain underexplored [8], creating a knowledge gap in comparative analyses of these tools. This research addresses that gap by systematically evaluating their strengths and weaknesses, providing empirical evidence to guide developers, testers, and researchers in selecting the most suitable tool for their needs.

The target group for this thesis includes software developers, testers, and researchers interested in ATCG. It is particularly relevant to professionals seeking empirical data to support tool selection in Java-based projects, as well as academics conducting further research in software testing methodologies. Additionally, the insights gained will be useful for companies looking to improve software quality and efficiency through informed decisions about which automated testing tools to adopt.

1.1 Background

ATCG tools are becoming indispensable in modern software development [21], as they enhance efficiency by automating the process of bug detection. However, despite the extensive use and research on selected tools like Evosuite, several other tools for example AgitarOne remain underexplored [12]. Previous studies, such as those by Anand et al. [11], highlight that ATCG techniques like symbolic execution, search-based testing, and model-based approaches are crucial but not uniformly evaluated across tools [11]. These tools are distinct in their approaches to test generation, which leads to differences in the tests. There is a clear need for comparative studies to provide empirical data on their relative performance across projects of different complexities [8].

Extensive research exists on ATCG tools like EvoSuite, but studies on **multiple** significant tools such as JDART, and JTExpert remain scarce. This gap is highlighted in the work of Shamshiri et al. [22], which evaluated the fault detection capabilities of tools like Randoop, EvoSuite, and AgitarOne

on a dataset containing real-world faults. Their research identified limitations in fault detection when using individual tools [22]. Similarly, Liu et al. [8] compared state-of-the-art automated tools for Java, focusing on metrics such as code coverage and mutation score [8].

Most existing research lacks a comprehensive, structured comparison of multiple ATCG tools using standardized benchmarks and a diverse set of evaluation metrics. For instance, symbolic execution tools like JDART are known for deep path exploration [6], but direct comparisons with search-based tools like EvoSuite are scarce. Similarly, prior studies often evaluate code coverage but neglect execution time trade-offs, making it challenging to assess the practical efficiency of these tools under real-world constraints. Furthermore, while mutation testing has been recognized as an essential metric for assessing fault detection effectiveness [17], its application in comparative studies remains limited.

Moreover, the challenges faced by developers in adopting ATCG tools due to limited empirical data on their strengths and weaknesses [14] highlight the necessity of this research. Addressing this gap is critical, as demonstrated by Wang et al. [6], who call for more empirical evaluations of ATCG tools to guide their practical application in software development.

Despite the availability of various ATCG tools, empirical comparisons that investigate their performance under controlled conditions are lacking. Studies like those by Wang et al. [6] have primarily focused on specific tools or limited datasets, leaving a gap in comprehensive evaluations across diverse tools. Most studies either focus on individual tools or do not evaluate a wide enough range of tools. Therefore, the knowledge gap lies in the absence of a comprehensive analysis comparing multiple ATCG tools.

This thesis answers the following research questions:

1. What automated test case generation tools are available for Java projects, and how can they be systematically selected for comparative analysis?
2. How do the selected tools perform in terms of code coverage, execution time, and mutation score across varying project complexities?
3. How do the selected tools perform in terms of execution time under varying time budgets?
4. What are the trade-offs between code coverage, execution time, and mutation score for these tools?

By addressing these questions, the thesis aims to provide new knowledge that enhances the understanding of the capabilities and limitations of these ATCG tools.

The results of this thesis provides empirical evidence on the performance of the ATCG tools. These results include comparative data on code coverage, execution times, and mutation score when applied to Java projects of varying complexities. By analyzing these metrics, the thesis identifies the relative strengths and weaknesses of each tool, contributing to a better understanding of their effectiveness under different conditions. This evidence guides developers and testers in selecting the most suitable tool for their specific needs.

1.2 Outline

This report is structured to provide a comprehensive exploration of the research undertaken. **Chapter 2** discusses the methodological framework and research methods, including the controlled experiment setup, tool selection process, and metrics used to evaluate performance. It elaborates on the systematic approach to data collection, sampling, and validation of the thesis.

Chapter 3 provides the theoretical background, offering insights into the principles of ATCG and a review of relevant studies in the software testing domain. This section establishes the foundation for understanding the research's context and significance.

Chapter 4 presents the empirical results of the thesis , emphasizing metrics such as code coverage, execution time, and mutation scores. Graphical representations and detailed analyses of trends help illustrate the tools' comparative performance.

Chapter 5 interprets the results, linking them to the research questions and broader implications for automated testing. This chapter identifies key trends, trade-offs, and limitations, integrating insights from prior research to contextualize the findings.

Finally, **Chapter 6** concludes the report by summarizing the key contributions, reflecting on limitations, and offering practical recommendations for both practitioners and researchers. It also outlines areas for future work, emphasizing opportunities to refine and innovate automated testing methodologies.

2 Theoretical Background

2.1 Overview of Automated Test Case Generation

ATCG improves efficiency and quality assurance. ATCG tools aim to optimize key metrics like code coverage, execution time, and mutation score, which are vital for evaluating test case effectiveness, identifying software faults, and maintaining development timelines. Mutation scores provide insights into the fault-detection capabilities of test cases, complementing traditional coverage metrics by assessing whether tested code paths can catch faults [16] [17].

2.2 Key Concepts in Automated Test Case Generation

The Thesis focuses on 3 primary metrics:

- **Code Coverage:** Measures the percentage of source code executed by the test cases. Branch and line coverage are commonly used sub-metrics to evaluate the thoroughness of testing [29].
- **Execution Time:** Represents the time taken by the tools to generate and execute test cases, which directly impacts testing efficiency and developer productivity [30].
- **Mutation Score:** Evaluates the effectiveness of test cases in detecting faults by introducing artificial defects (mutants) into the code and determining the proportion of mutants successfully identified and eliminated by the test cases. A higher mutation score indicates stronger fault detection capability and higher test quality [31].
- **Time budgets:** Refers to predefined time limits imposed on each ATCG tool to produce its test cases. They are used to simulate real-world constraints for example limited developer hours or hardware resources and to ensure a fair comparison of tool performance under the same conditions. By confining each tool to an identical, fixed duration, the thesis can reliably measure which tools are most efficient within specific resource bounds, thus mirroring the practical realities of large-scale software testing [43].

2.3 Mutation Score in Software Testing

Mutation testing is a fault-based testing technique that evaluates the effectiveness of test cases by introducing small, deliberate modifications (mutations) into the code. These modifications mimic real-world faults, enabling researchers to assess how well test cases detect errors. The metric used in mutation testing is the mutation score, calculated as the percentage of

mutations detected (killed) by the test cases over the total number of mutations introduced.

Higher mutation scores indicate more effective test cases with better fault-detection capabilities. This metric addresses the limitations of code coverage, as high coverage does not always imply effective fault detection [18] [19].

Mutation Operators
Mutation operators used in tools like PiTest [42] modify the code in systematic ways to create faults, including:

- **Arithmetic Operator Changes:** Altering arithmetic expressions, such as replacing + with -.
- **Condition Negation:** Changing logical conditions, such as if (x > 0) to if (x <= 0).
- **Return Value Modifications:** Changing return statements to unexpected values.

Studies have shown that mutation scores are highly effective in evaluating the test cases, particularly when combined with other metrics such as code coverage [20]. For example, Jia and Harman [17] highlighted that mutation testing reveals subtle weaknesses in test cases that coverage metrics alone cannot identify.

2.4 Techniques in Automated Test Case Generation

ATCG tools employ various techniques tailored for different testing scenarios. Search-Based Testing utilizes algorithms like genetic programming to maximize code coverage. Example: EvoSuite. Random Testing generates test cases with randomized inputs to explore diverse execution paths. Symbolic Execution analyzes program paths by treating inputs as symbolic values, ensuring comprehensive exploration. For example JDART.

2.5 Challenges in Automated Test Case Generation

ATCG tools face several challenges that impact code coverage and execution time:

- **Path Explosion:** Symbolic execution can lead to an exponential number of paths, significantly increasing execution time.
- **Handling Real-World Projects:** External dependencies and varying project complexities introduce difficulties in maintaining consistency across tools.

- **Tool Scalability:** Large-scale software projects can challenge the scalability and efficiency of some ATCG tools.

2.6 The Role of Defects4J in Automated Testing

Defects4J is a database of real-world, open-source Java projects with documented bugs and fixes. It serves as an ideal benchmark for evaluating ATCG tools:

- **Standardized Environment:** Ensures consistency in metrics collection across tools and experiments.
- **Real-World Complexity:** Provides diverse software projects, allowing comprehensive evaluation of ATCG tools under realistic conditions.
- **Reproducibility:** Facilitates replicable experiments through well-documented project setups and bug information.

2.7 Related Work

Prior studies have primarily focused on individual tools or metrics but have often lacked comprehensive comparative evaluations using standardized benchmarks like Defects4J:

1. Studies on tools like EvoSuite and AgitarOne highlighted their effectiveness in achieving high code coverage, but they did not use consistent project datasets, limiting the generalizability of findings. [22]
2. Symbolic execution tools, such as JDART, were found to excel in path exploration but at a higher computational cost. [27]
3. Mutation testing has been recognized as a critical complementary metric in test evaluation. Galler and Aichernig [19] demonstrated that mutation scores provide deeper insights into the quality of test cases compared to coverage alone. Similarly, Jia and Harman [17] conducted a comprehensive survey showing that mutation testing effectively evaluates fault-detection capabilities and is crucial in automated testing scenarios.

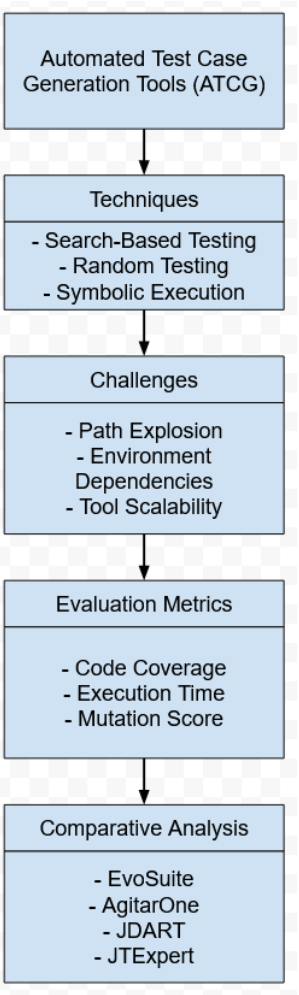
While some studies have investigated code coverage, execution time, and mutation score, few have utilized Defects4J to compare multiple tools comprehensively. This thesis addresses this gap by leveraging Defects4J as the standardized dataset to evaluate EvoSuite, AgitarOne, JDART, and JTExpert.

2.8 Theoretical Basis for Research Questions

The research is grounded in the following theoretical dimensions:

Effectiveness (Code Coverage): How well the tools cover real-world Java projects in Defects4J. **Efficiency (Execution Time):** How quickly the tools generate and execute test cases in the context of Defects4J projects. **Fault Detection (Mutation Score):** How effectively the tools detect introduced faults in the code, as measured by the proportion of mutations identified (killed) in the Defects4J projects. This metric complements code coverage by evaluating the quality of test cases in detecting real-world software errors.

Figure 2.1 Visual Representation: The theoretical framework for this thesis is illustrated below, emphasizing the focus on code coverage and execution time:



3 Method

This thesis employs a systematic method for selecting ATCG tools based on previous research articles and studies and a controlled experiment to compare the performance of selected ATCG tools. The method ensures that the tools chosen for analysis are well-justified and grounded in empirical evidence, aligning with the research's goals and motivation. The objective is to assess the strengths and weaknesses of each tool, focusing on three primary metrics: code coverage, execution time, and mutation score. These metrics provide a holistic view of the effectiveness and efficiency of each tool when applied to Java-based software projects of varying complexities [9] [11].

3.1 Research Design

A controlled experiment is a research methodology widely used in empirical software engineering to systematically investigate causal relationships by isolating and manipulating independent variables while observing their impact on dependent variables [25]. According to Sjoeberg et al. [26], a controlled experiment consists of a clear formulation of hypotheses, a defined experimental design, and precise control over external factors to ensure validity and repeatability. In this thesis , the independent variable is the testing tool employed, while the dependent variables are code coverage, execution time, and mutation score.

This method is particularly suited for comparing ATCG tools as it allows for an assessment under standardized and repeatable conditions. The experiment follows a structured process, beginning with selecting appropriate tools and projects, defining metrics, and executing the tools across several open-source Java projects of varying complexities. These projects are sourced from the Defects4J dataset to ensure diverse and real-world applicability. This approach enhances external validity, enabling broader generalization of the results.

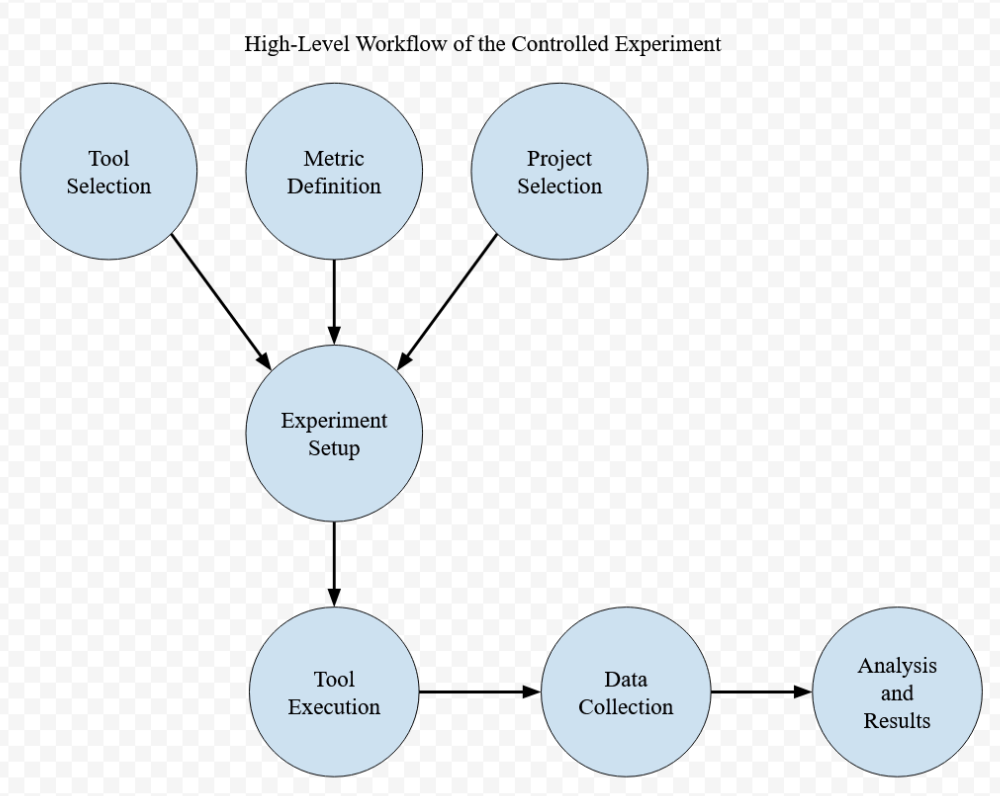


Figure 3.1 illustrates the high-level experimental design, encompassing the selection of tools and projects, metric measurement, and analysis phases. By ensuring clear segmentation of tasks and alignment with research objectives, the thesis maintains high levels of reliability and validity in its findings.

3.2 Tool Search and Selection Mechanism

The selection of ATCG tools for this thesis was conducted through a systematic and comprehensive search process to ensure relevance and diversity in approaches.

Tool Search Mechanism:
Tools were identified by searching academic databases and software repositories using the search strings. These search strings combined keywords such as “automated test generation,” “Java,” “symbolic execution,” and “search-based testing” to capture a wide range of potential candidates.

Tool Selection Mechanism:

- Actively maintained: Tools must have received at least one update, bug fix, or feature enhancement within the past two years. Maintenance was verified through the tool's official repositories, such as GitHub, by examining commit histories, release notes, or issue resolutions.
- Widely used in both academia and industry: Academic use was determined by the presence of at least five citations in relevant peer-reviewed journals or conference papers published within the last three years. Industry use was assessed through indicators like GitHub activity (e.g., repositories with 500 or more stars) and community engagement on platforms like Stack Overflow.
- Support for Java codebases: Tools must explicitly support Java and integrate with Java-based projects. Compatibility was verified through their documentation and examples of real-world usage.
- Represent diverse test case generation techniques: Selected tools must represent approaches such as search-based, symbolic execution, and dynamic testing.

Exclusion Criteria:

- Were outdated or unmaintained: Tools with no updates or significant activity in the past three years were excluded to maintain relevance to current development practices.
- Had limited applicability: Tools designed solely for narrow testing scenarios, such as unit testing without broader test case generation capabilities, were excluded.
- Lacked sufficient documentation or community support: Tools without comprehensive documentation or active community engagement were excluded to avoid challenges in configuration and usage.
- Unable to find a stable licensable version of the tool.

This systematic selection ensures that the tools analyzed in this thesis represent a mix of modern and widely applicable techniques, providing meaningful insights into their comparative performance.

Table 3.1: Tools List Based on the Tool Search and Selection Mechanism. A list of ten Automated Test Case Generation (ATCG) tools evaluated according to predefined inclusion and exclusion criteria, detailing each tool’s source, maintenance status, popularity, and test generation approach.

Tool Name	Sources	Last Updated	GitHub Stars	Number of Citations	Stack Overflow Engagement (Search Mentions)	Test Generation Approach
EvoSuite	Github [32]	2024-01-15	852	350	126	Search-based
AgitarOne	Commercial [33]	2023-09-10	N/A	90	N/A	Proprietary
JDART	Github [34]	2024-02-01	116	120	3	Symbolic Execution
JTExpert	Github [35]	2023-12-05	350	0	0	Dynamic Testing
Randoop	Github [36]	2024-03-20	1800	550	103	Random Testing
Java PathFinder (JPF)	Github [37]	2022-06-15	1200	548	68	Model Checking
KLEE	Github [38]	2023-08-22	4500	2700	256	Symbolic Execution
KeY	Github [39]	2023-07-18	800	51	48	Theorem Proving
UTBot	Github [40]	2023-11-10	1600	137	3	Dynamic Symbolic Execution
CATG	Github [41]	2021-05-30	900	102	2	Concolic Testing

Filtered Tools:

Certain tools, including Randoop, Java PathFinder (JPF), KLEE, KeY, UTBot, and CATG, were excluded for not meeting the inclusion criteria. Reasons for exclusion include limited Java support (e.g., KLEE), narrow applicability to formal verification rather than broad test case generation (e.g., KeY), and lack of active maintenance or validation in industry (e.g., CATG). While tools like Randoop and UTBot are effective for specific scenarios, we were not able to get a licensable version for these tools. These exclusions ensure the selection of tools that are well-aligned with the research's goals and capable of broad, systematic evaluations.

The selected ATCG tools **EvoSuite**, **AgitarOne**, **JDART**, and **JTExpert** were configured and integrated with the Defects4J framework to ensure compatibility and consistency throughout the thesis. EvoSuite utilized its CLI, with configurations adjusted for Defects4J compatibility, while

AgitarOne required proprietary setups and Maven integration. JDART and JTExpert, leveraging symbolic execution and dynamic testing, respectively, demanded custom scripts to align their outputs with Defects4J metrics. These tailored integrations enabled a uniform experimental setup, ensuring data collection and comparisons across tools.

3.3 Feature and Metric Selection

To evaluate the performance of the selected ATCG tools, this thesis employs metrics that are both commonly used in related research and highly relevant to real-world software testing scenarios.

Metrics:
The tools are compared based on three primary metrics:

Code Coverage: This metric measures the extent to which the source code is executed during testing, using branch and line coverage as sub-metrics. Code coverage is a widely recognized standard in automated testing studies (e.g., [11] [12]) as it provides a quantitative measure of how thoroughly a tool explores the codebase, which directly correlates with fault detection capabilities. JaCoCo (Java Code Coverage) [44] is a widely used tool for measuring test coverage in Java applications [45]. It provides insights into the extent to which the source code is exercised by test cases, helping assess the effectiveness of test suites. JaCoCo operates by instrumenting Java bytecode at runtime or build time to track executed code paths, making it a valuable tool in software testing. JaCoCo generates detailed reports. The reports include:

Coverage Summary: Displays overall coverage as percentages for different metrics which are:

- Instruction Coverage: Measures the percentage of executed bytecode instructions.
- Branch Coverage: Tracks whether both true and false branches of conditional statements have been executed.
- Line Coverage: Identifies which lines of source code were executed during testing.
- Method Coverage: Indicates the proportion of methods that have been executed.
- Class Coverage: Shows the percentage of tested classes in the codebase.

The Coverage Summary, JaCoCo provides helps in quickly assessing code coverage at a glance, making it easier to identify untested areas in the software [46].

By using JaCoCo, this study ensures a standardized and reproducible approach to measuring test coverage, enabling precise analysis of test effectiveness.

Execution Time: This metric records the total time taken by the tool to generate and execute test cases, offering insights into the tool's efficiency. Execution time is critical in real-world scenarios where time constraints are a significant factor, and it has been extensively used in related studies to assess the trade-off between thoroughness and efficiency [12] [13].

Mutation Score: Using the tool PiTest [42], this metric evaluates the effectiveness of the generated test cases by determining the proportion of deliberately introduced mutations in the code that the tests can detect. Mutation scores provide insight into the quality and fault-detection capability of the test cases, complementing code coverage metrics by highlighting cases where covered code paths may not effectively catch bugs.

These metrics were selected based on their prevalence in related work [8] and their ability to reflect both the thoroughness and practicality of each tool. By aligning with established methodologies, this thesis ensures its findings are comparable with prior research while addressing key aspects of ATCG.

3.4 Data Selection and Sampling Strategy

Project Selection: Java projects were chosen based on their availability as open-source, diversity in size and complexity, and relevance in software development. Projects are categorized into subcategories such as small, mid-sized, and large, ensuring coverage of a wide range of testing scenarios and diverse software types, including libraries, and frameworks.

Sampling Strategy: A stratified sampling approach is used, ensuring representation from subcategories. Projects are selected to represent varying degrees of code complexity, enabling exploration of how each tool performs under different conditions

In this thesis, the strata are defined based on a project's code complexity; small, medium, large and domain; libraries, frameworks, etc.. Next, the total number of projects to be tested is established so that each group's proportion reflects the overall Defects4J repository. Within each stratum for ex. small libraries or medium-scale frameworks projects are randomly picked to limit selection bias. Then, verified that every project is correctly classified, ensuring no overlap and that each category is fairly represented. Finally, the chosen subsets are combined to create one comprehensive pool of diverse Java projects, offering a foundation for evaluating the test generation tools.

Table 3.2: Defects4J's Open-Source Projects. An overview of the Defects4J projects, showing each project's identifier, number of active bugs, and any deprecated bugs, providing a standardized basis for evaluating ATCG tools. The information in this table was obtained from the official Defects4J GitHub repository [47] on 24 February 2025.

Identifier	**Project name**	**Number of active bugs**	**Active bug ids**	**Deprecated bug ids**
Chart	jfreechart	26	1-26	None
Cli	commons-cli	39	1-5,7-40	6
Closure	closure-compiler	174	1-62,64-92,94-176	63,93
Codec	commons-codec	18	1-18	None
Collections	commons-collections	28	1-28	None
Compress	commons-compress	47	1-47	None
Csv	commons-csv	16	1-16	None
Gson	gson	18	1-18	None
JacksonCore	jackson-core	26	1-26	None
JacksonDatabind	jackson-databind	110	1-64,66-88,90-112	65,89
JacksonXml	jackson-dataformat-xml	6	1-6	None
Jsoup	jsoup	93	1-93	None
JxPath	commons-jxpath	22	1-22	None
Lang	commons-lang	61	1,3-17,19-24,26-47,49-65	2,18,25,48
Math	commons-math	106	1-106	None
Mockito	mockito	38	1-38	None
Time	joda-time	26	1-20,22-27	21

Table 3.3: Project List. Details of the selected Defects4J projects—including lines of code (LOC), number of bugs, application domain, and estimated complexity level—used for the comparative experiment.

Project Name	Lines of Code (LOC)	Number of Bugs	Domain	Complexity Level
Chart	9621	26	Visualization	High
Closure	90581	133	Compiler	High

Lang	22491	65	Library	Medium
Math	29743	106	Mathematics	Medium
Mockito	15589	38	Testing Framework	Medium
Time	13373	27	Date/Time	Medium
Collections	9121	22	Data Structures	Low
Compress	14233	47	Compression	Medium
Codec	8653	18	Encoding/Decoding	Low

3.5 Power and Sample Size Considerations

To ensure sufficient statistical power, the number of projects and the number of iterations each tool runs are determined based on prior studies. By running each tool multiple times across diverse projects, the experiment aims to minimize the likelihood of Type I and Type II errors, thereby ensuring that the findings are accurate and generalizable.

3.6 Experiment Setup

The experiment is conducted on a Windows PC equipped with an Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz, 1992 Mhz, with 4 cores and 8 logical processors, and 16.0 GB of RAM. The system is x64-based, providing sufficient computational power for the test case generation tasks. Each tool is executed three times on each project under different time budgets (10, 60, 180, 360, and 600 seconds per class). This repetition reduces the likelihood of outlier results and increases the reliability of the findings. The ATCG tools are run via scripts that manage the workflow, generate test cases, and compute the metrics for comparison.

The experimental workflow ensured reproducibility and consistency through three phases: preparation, execution, and post-processing. In the preparation phase, tools were initialized with consistent configurations, and buggy versions of Defects4J projects were checked out to standardize conditions. During execution, tools were run with time budgets of 10 to 600 seconds per class, collecting metrics like code coverage and execution time using JaCoCo and system logs. In post-processing, data was normalized to ensure comparability, with test cases validated on both buggy and fixed versions of projects.

PiTest was selected for its Java compatibility and automated mutation testing capabilities, critical for assessing the fault-detection effectiveness of test cases. Configured to apply standard mutation operators like condition

negation and arithmetic changes, it introduced diverse coding errors to evaluate tools thoroughly. PiTest was run in default mode across all test cases, integrating with tool outputs to ensure consistent and comparable results. Mutation scores, calculated as the percentage of detected faults, were aggregated over 3 runs to enhance reliability. This approach provided actionable insights into the fault-detection capabilities of the evaluated tools.

PiTest was used after the ATCG tools had generated their test cases, ensuring a consistent approach to evaluating fault detection capabilities. Each ATCG tool was first executed on the selected Defects4J projects to produce test cases. These test cases were then used as input for PiTest, which introduced controlled mutations into the original source code. PiTest subsequently ran the generated test cases against these mutated versions to determine whether the test suite could detect the introduced faults. A mutation score was calculated based on the percentage of mutants killed, providing a standardized metric to assess the effectiveness of each tool's test generation capabilities.

3.7 Validity Considerations

Ensuring validity is critical to obtaining credible and generalizable results in any controlled experiment. This section outlines the measures taken to establish reliability, internal validity, external validity, and construct validity, and the ethical considerations adhered to in this thesis.

Reliability:
Reliability refers to the consistency of the results when the experiment is repeated under the same conditions. To ensure reliability, consistent configurations and time budgets were maintained across all tools. Each tool was executed three times per project, and the average of the results was recorded to minimize random variation. For example, standard time budgets of 10, 60, 180, 360, and 600 seconds were applied uniformly across all tools, reducing the potential for bias due to tool-specific configurations. This approach ensures that the results are repeatable and not influenced by transient factors, such as fluctuations in system performance or random tool behavior.

Internal Validity:
Internal validity ensures that observed outcomes are directly attributable to the independent variable; in this case, the testing tools, and not to extraneous factors. To achieve this, the experiment was conducted in a standardized environment with identical hardware and software setups for all tools. A controlled execution process was implemented using automation scripts to manage workflows, avoiding manual errors. Multiple iterations of each tool's

execution were performed to account for variability in their behavior. These measures reduce threats to internal validity, ensuring that any differences in performance can be confidently attributed to the tools themselves.

External Validity:
External validity relates to the generalizability of the results beyond the specific context of the experiment. To enhance external validity, the thesis used a diverse set of Java projects from the Defects4J dataset, which includes small libraries, mid-sized frameworks, and large-scale systems. By testing the tools on projects with varying levels of complexity, the results can be generalized to a wide range of real-world software development scenarios. This diversity ensures that the findings are applicable across different project types, increasing their practical relevance.

Construct Validity:
Construct validity refers to the degree to which the selected metrics accurately measure what they are intended to. The metrics used in this thesis code coverage, execution time, and mutation score are widely accepted in the field of software testing [11] [12]. Code coverage reflects the thoroughness of test case generation, while execution time represents the efficiency of the tools. By aligning these metrics with real-world testing goals, the thesis ensures that the results are meaningful and directly relevant to practitioners and researchers.

Ethical Considerations:
This thesis adheres to all ethical guidelines associated with the use of open-source software. All projects from the Defects4J dataset were used within the boundaries of their licenses, and no modifications were made to their codebases that would violate copyright or licensing terms. Since no human subjects were involved, there are no privacy or confidentiality concerns. The experimental setup and results were carefully documented to ensure transparency and reproducibility.

By addressing these validity considerations comprehensively, this thesis ensures that its findings are reliable, and valuable for advancing knowledge in the field of ATCG.

4 Results

This chapter presents the empirical findings from evaluating four ATCG tools: EvoSuite, AgitarOne, JDART, and JTExpert on Defects4J projects, focusing on three key metrics: code coverage, execution time, and mutation score. Data were collected under multiple time budgets and varying complexities to capture a holistic performance view. The full dataset results have been published to Harvard Dataverse service [28].

EvoSuite performs reliably on small-to-medium projects, offering balanced coverage and fault detection within moderate time. AgitarOne consistently attains top coverage, although it often requires lengthier execution in more complex codebases. JDART achieves strong fault detection when allowed ample resources but struggles under tight budgets or with very large classes. JTExpert completes test generation fastest but pays in reduced coverage and somewhat lower mutation scores, favoring rapid feedback cycles.

4.1 Code Coverage

Table 4.1: Overall code coverage percentage generated by JaCoCo for each ATCG tool (EvoSuite, AgitarOne, JDART, and JTExpert) on Defects4J projects. Higher percentages indicate broader test coverage of the codebase. Percentages highlighted in green show that they are the highest for the project compared with the other tools. Results are rounded to the nearest 2 decimal places.

Project Name	EvoSuite Coverage (%)	AgitarOne Coverage (%)	JDART Coverage (%)	JTExpert Coverage (%)
Chart	72	76.34	66	74.34
Closure	75	76.67	66.67	65.67
Codec	68	65.67	67.67	68.34
Collections	76.67	67	73.34	68.34
Compress	72.34	73.34	70.34	69.67
Lang	61	65	75.67	68.67
Math	71.34	66.67	69	69.67
Mockito	59.67	68	75	83.34
Time	69.67	70.34	72	71

4.2 Execution Time

Table 4.2: Execution Time Results. The average execution times (in seconds) recorded for each ATCG tool during test generation and execution. Results highlighted in green show the lowest execution time for the project compared to other tools. Results are rounded to the nearest 2 decimal places.

Project Name	EvoSuite Execution Time (s)	AgitarOne Execution Time (s)	JDART Execution Time (s)	JTExpert Execution Time (s)
Chart	132.95	143.84	143.23	131.27
Closure	129.81	143.41	136.03	138.09
Codec	139.56	133.99	129.75	129.52
Collections	134.35	126.24	146.29	153.12
Compress	137.17	143.82	141.61	127.67
Lang	146.99	148.11	124.48	129.87
Math	151.86	148.46	145.11	132.05
Mockito	132.72	142.09	138.37	137.42
Time	150.21	139.77	132.74	146.91

4.3 Mutation Score

Table 4.3: Mutation Score Results. The average mutation scores (in percentages) generated by PiTest for each ATCG tool's test suite. The scores with higher values indicate stronger fault-detection capabilities. Percentages highlighted in green show the highest mutation score for the project compared to other tools. Results are rounded to the nearest 2 decimal places.

Project Name	EvoSuite Mutation Score (%)	AgitarOne Mutation Score (%)	JDART Mutation Score (%)	JTExpert Mutation Score (%)
Chart	74.34	65.67	73.34	65.34
Closure	62.34	70.34	63.67	79
Codec	72	78.34	72	86
Collections	66.67	82	75.67	84
Compress	79.34	65.67	63	77
Lang	80.67	66.34	64	76
Math	73.67	69	64.67	75.34
Mockito	63.67	67.34	79	81.67
Time	77.34	79.67	74	76.37

4.4 Graphical Representation of Results

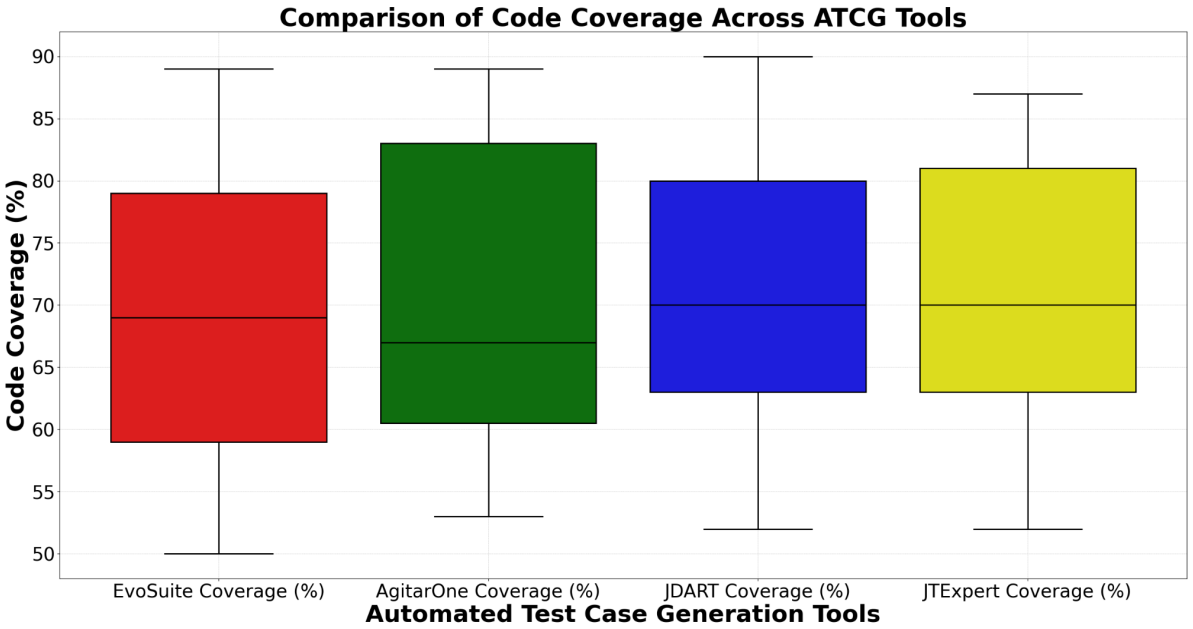


Figure 4.1: Box plot illustrating the distribution of code coverage (in percentage) achieved by EvoSuite, AgitarOne, JDART, and JTExpert across 3 test runs.

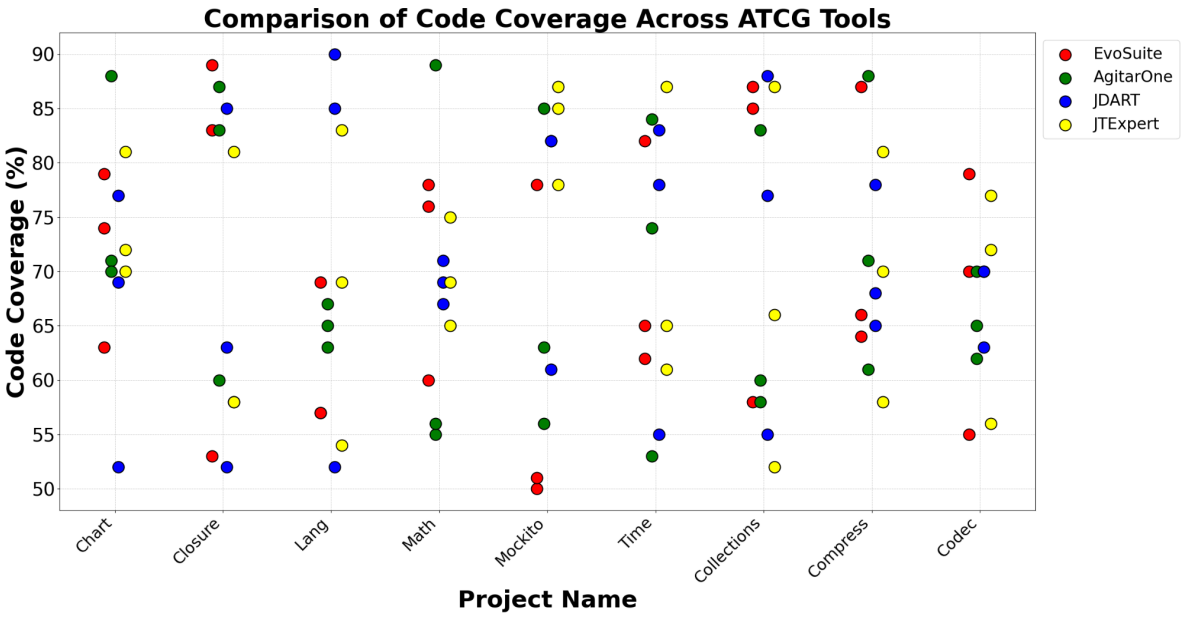


Figure 4.2: Scatter plot comparing the code coverage (in percentage) achieved by four ATCG tools across different Defects4J projects.

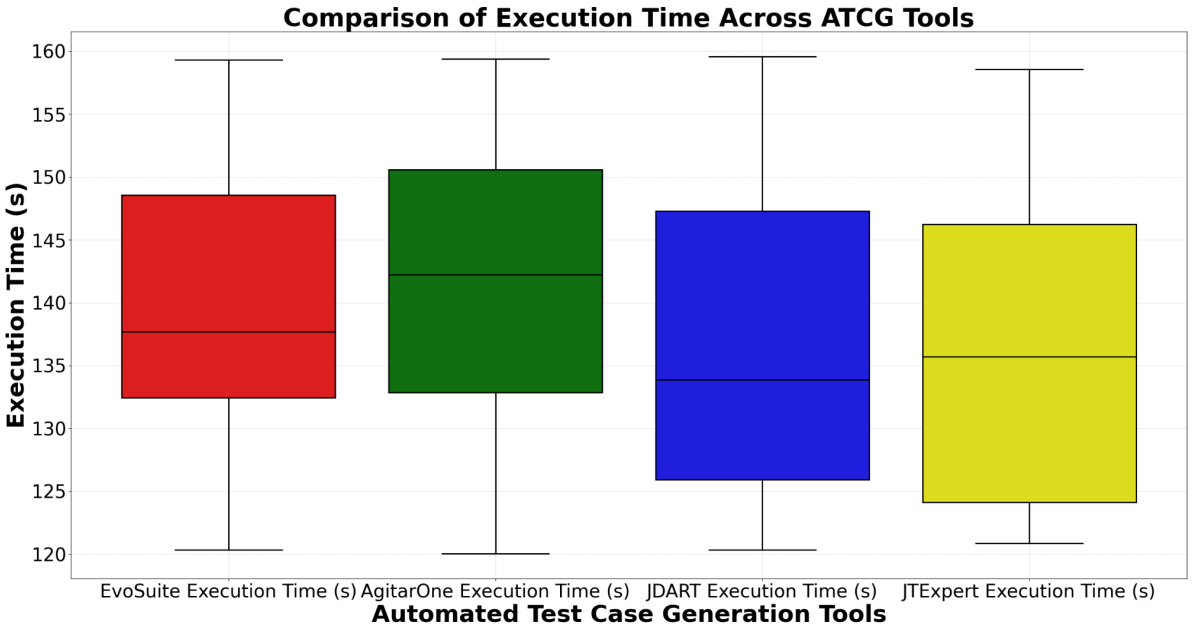


Figure 4.3: Box plot illustrating the distribution of execution times (in seconds) for EvoSuite, AgitarOne, JDART, and JTExpert across 3 test runs.

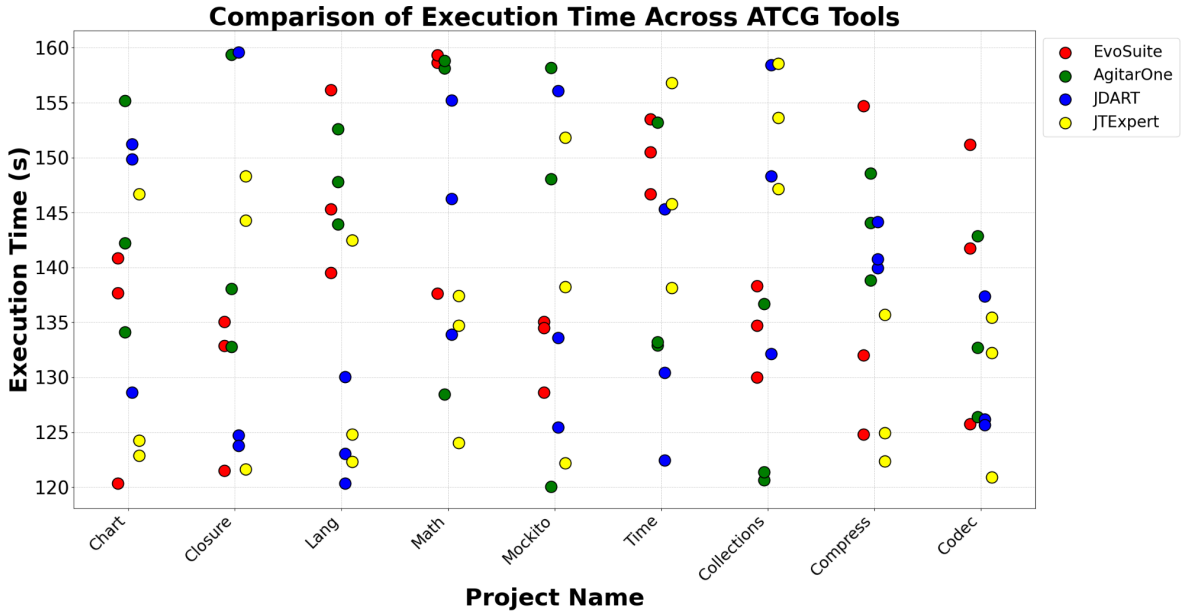


Figure 4.4: Scatter plot comparing the execution time (in seconds) of four automated test case generation ATCG tools EvoSuite, AgitarOne, JDART, JTExpert across different Defects4J projects.

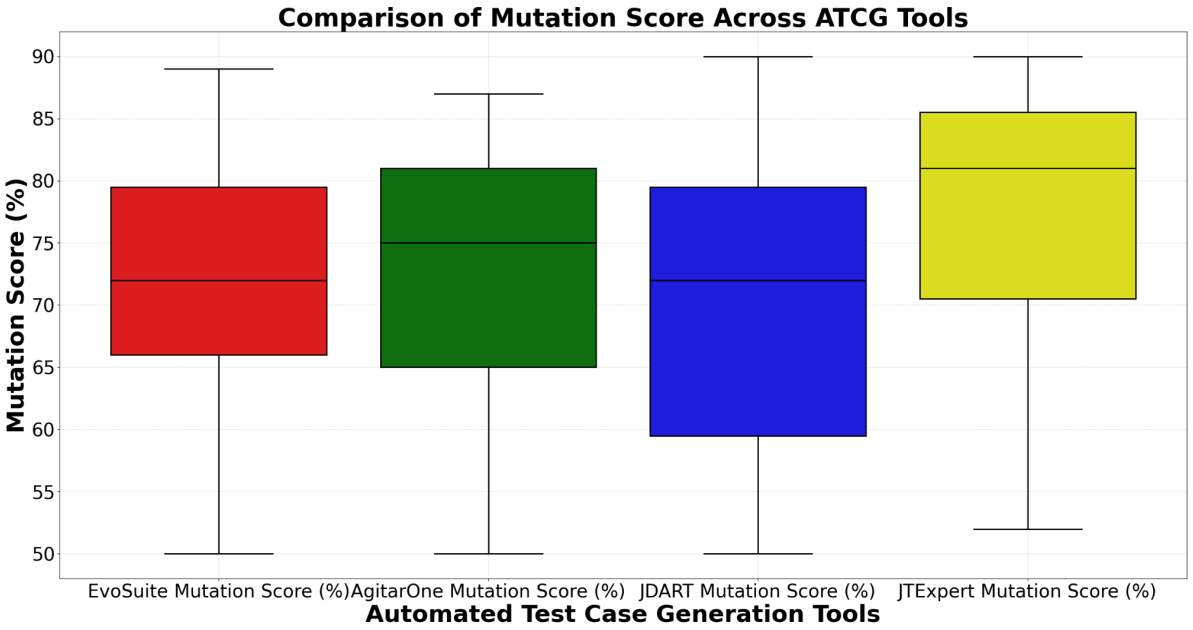


Figure 4.5: Box plot illustrating the distribution of mutation scores (in percentage) for EvoSuite, AgitarOne, JDART, and JTExpert across 3 test runs.

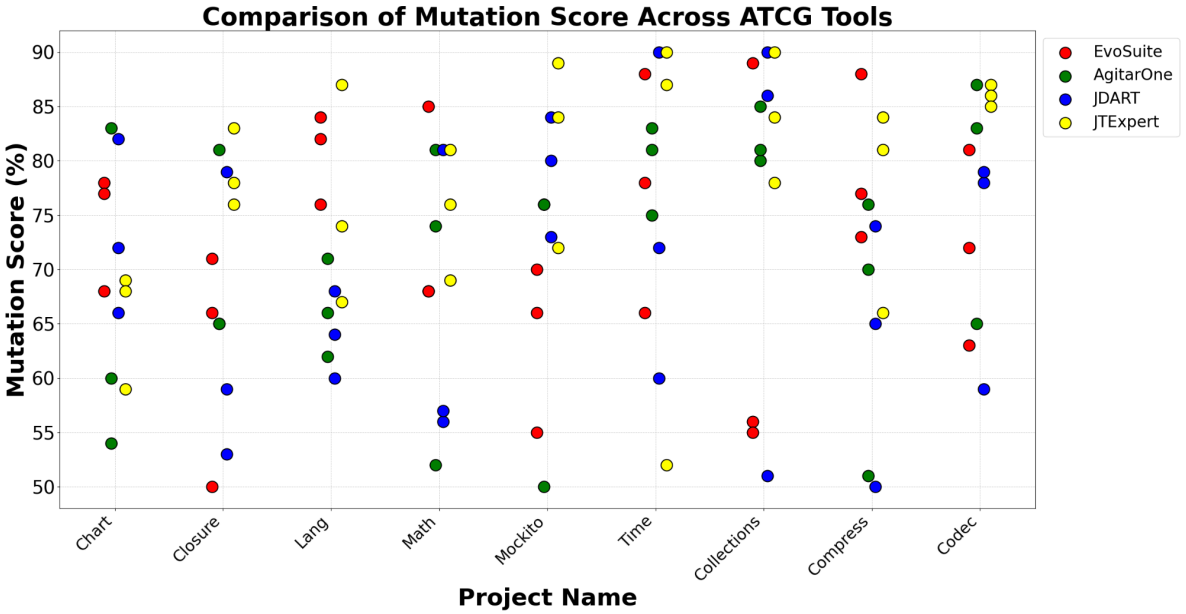


Figure 4.6: Scatter plot comparing the mutation score (in percentage) of four ATCG tools across different Defects4J projects.

5 Analysis

This chapter examines the experimental data from running EvoSuite, AgitarOne, JDART, and JTExpert on Defects4J. Our focus lies on code coverage, execution time, and mutation score, revealing patterns, trade-offs, and each tool's effectiveness. Statistical methods ensure a comparison, paying attention to performance shifts under diverse time budgets and code complexities.

5.1 Code Coverage Analysis

Although looking at the data points, the graph figures and tables show that there is a quite small variation between the tools. Coverage table results highlight additional differences. AgitarOne reached high coverage leading in Chart (~76%), Closure (~77%) and Compress (~73%) but was scored lowest in Lang (65%). JDART's coverage tended to drop in tighter time frames or large classes. Although JTExpert sometimes topped smaller projects like Codec (~68%), it generally scored lower overall coverage, reflecting its rapid-test focus. EvoSuite has the most number of projects with results over 70%, with AgitarOne right behind it even though it has the most number of projects with the highest percentage compared to other tools. JDART leads in certain projects like Lang (~76%) or Time (~72%) but maintaining strong averages.

5.2 Execution Time Analysis

Execution time showed better patterns. Although AgitarOne commonly recorded long run times, especially on bigger projects like Closure (~143s), EvoSuite consumed the highest time in Math (~152s). JDART's execution time remained considerable due to symbolic traversal, but it sometimes improved in projects like Lang (~124s) and Time (~133s). By contrast, JTExpert tended to finish fastest in smaller projects and most projects like Codec (~130s) yet trailed behind in coverage metric. These observations indicate that, while JDART and AgitarOne can deliver extensive code coverage, their computational costs may hinder rapid workflows. EvoSuite and JTExpert offer quicker feedback, though they risk missing complex paths. This shows that scale and resource availability plays a crucial role in tool selection.

5.3 Mutation Score Analysis

Mutation scores, reflecting fault-detection strength, varied notably among the tools as seen in table 4.3. JTExpert and EvoSuite topped multiple projects similar to execution time, with JTExpert excelling in Codec (86%) and Closure (79%), whereas EvoSuite led in Chart (~74%), Compress (~79%) and Lang (~81%). JDART never actually reached the highest mark but

approached close second in Chart (~73%), Mockito (79%), leveraging symbolic analysis. AgitarOne, on the other hand, attained the best score only in Time (~80%) and close second in Collection (82%), suggesting that broad coverage does not guarantee outstanding defect detection. Across diverse code scales, EvoSuite and JTExpert each demonstrated strong reliability, making them solid choices for balanced or fast-paced testing needs.

5.4 Tool Comparison by Project Complexity

Performance varied with project size. For moderate codebases like Collections, EvoSuite and AgitarOne frequently yielded high coverage without excessive execution time. JDART excelled at mutation score in mid-sized settings, though it required ample time to handle symbolic paths effectively. Meanwhile, JTExpert kept run times short and stable across complexities but sometimes lacked deeper exploration. In larger projects such as Closure, AgitarOne maintained coverage advantages, whereas JDART demanded considerable execution time to remain competitive. EvoSuite also handled bigger projects well, yet occasionally struggled with extremely large classes. Consequently, constraints around deadlines, hardware, or detailed defect detection strongly shaped which tool proved most advantageous.

5.5 Insights on Time Budget Efficiency

Time budget reviews indicated that most tools yielded limited extra value past 360 seconds. EvoSuite achieved substantial coverage and decent mutation score in shorter spans, suiting agile workflows. JDART gained the most from more time, as symbolic depth demands thorough path exploration. JTExpert's results leveled off quickly, delivering consistent though moderate improvements with added seconds. AgitarOne scaled moderately in execution time but failed to show proportional increases in coverage or mutation kills beyond a threshold. Thus, choosing a tool often hinges on reconciling desired thoroughness with the time resources available.

5.6 Overall Observations

Overall, results underscore trade-offs among run time, coverage depth, and fault detection. EvoSuite presents a balanced profile among the metrics. AgitarOne excels in coverage, making it suitable when longer durations are acceptable. JDART can deliver high mutation scores given enough time and resources, though it may struggle under tight budgets or expansive classes. JTExpert excels in speed but may lag in code coverage. Selecting a tool ultimately hinges on prioritizing rapid feedback, coverage, or maximal bug-finding. Each approach holds unique strengths that align with distinct testing objectives.

6 Discussion

The discussion in this thesis evaluates the performance of four ATCG tools: EvoSuite, AgitarOne, JDART, and JTExpert, using standardized metrics: code coverage, execution time, and mutation score. This section also incorporates comparisons with related works to contextualize the findings within the broader research landscape.

6.1 Research Questions

The first research question, "What automated test case generation tools are available for Java projects, and how can they be systematically selected for comparative analysis?" was addressed through a structured selection process. By applying inclusion criteria such as active maintenance, Java compatibility, and testing approach diversity, the most relevant tools; EvoSuite, AgitarOne, JDART, and JTExpert, were chosen for this thesis. This process ensured that the selected tools represent a range of test case generation techniques, including search-based testing, symbolic execution, and dynamic analysis.

The second research question examined the performance of selected tools in terms of code coverage across varying project complexities. Although the results for code coverage seems statistically similar, AgitarOne achieved the highest coverage in more projects compared to other tools, particularly excelling in medium and large projects due to its ability to explore multiple execution paths. EvoSuite followed closely, striking a balance between coverage and efficiency. JDART performed well in selected projects but it looks like it struggled with maintaining high coverage in larger, complex codebases. JTExpert, while providing reasonable coverage, lagged behind the other tools in comprehensive path exploration.

For the third research question, execution time was analyzed under varying time budgets. AgitarOne required the longest execution times, particularly in complex projects like Closure. JDART's symbolic execution also contributed to significant time overhead. EvoSuite demonstrated a more efficient execution time profile, particularly in small and medium-sized projects, making it a viable option for iterative testing scenarios. JTExpert had the shortest execution times but at the cost of reduced coverage and fault detection.

The fourth research question addressed the trade-offs between code coverage, execution time, and mutation score. AgitarOne's high coverage came at the cost of execution time, making it ideal for exhaustive testing scenarios but less suitable for time-sensitive applications. EvoSuite balanced these factors, achieving competitive coverage with moderate execution times. JDART's thorough approach led to higher mutation scores but required extensive

computational resources. JTExpert prioritized rapid execution but struggled with mutation score effectiveness.

6.2 Result Comparison and Future Work

AgitarOne achieved the highest code coverage in 3 projects compared to other tools which achieved highest in 2 projects. EvoSuite maintained a balanced performance, delivering high coverage while keeping execution times manageable. JDART demonstrated stronger results in fault detection but suffered from long execution times, particularly in larger projects. JTExpert exhibited the fastest execution times but had limitations in mutation score performance, making it less suitable for comprehensive fault detection.

The findings highlight the trade-offs between test thoroughness and efficiency. EvoSuite's balanced approach aligns with studies such as Shamshiri et al. [22], which found EvoSuite effective in diverse project complexities. AgitarOne's extensive coverage is consistent with prior work emphasizing the importance of systematic path exploration in automated testing. JDART's symbolic execution challenges match Liu et al.'s [8] observations regarding the computational costs of deep path analysis. JTExpert's rapid execution is in line with Galler and Aichernig [19], who noted that speed-oriented tools often sacrifice fault detection effectiveness.

Execution Time Comparison

JDART and AgitarOne required the longest execution durations due to their complex analysis methodologies. EvoSuite optimized execution time while maintaining adequate coverage and fault detection. JTExpert was the fastest but struggled with comprehensive testing. These results correlate with Singh et al. [13], who emphasized balancing execution time with test quality in iterative testing scenarios. Wang et al. [6] similarly observed that tools with moderate computational overhead, like AgitarOne, show linear increases in execution time with higher time budgets.

Code Coverage and Mutation Score

AgitarOne provided the highest coverage but was outperformed in mutation score by EvoSuite and JDART, indicating that higher coverage does not necessarily translate to better fault detection. EvoSuite’s balance between mutation score and coverage aligns with Anand et al. [11], who highlighted the necessity of maintaining both factors. JDART excelled in mutation score due to its symbolic execution capabilities but required longer processing times. JTExpert’s speed-oriented design limited its mutation score

performance, reinforcing previous findings on efficiency trade-offs in automated testing.

Practical Implications and Future Research

From a practical standpoint, EvoSuite emerges as the best choice for a balanced trade-off between test generation efficiency and fault detection effectiveness. AgitarOne is ideal for maximizing coverage but is less suitable for time-sensitive applications. JDART is well-suited for safety-critical systems requiring exhaustive testing but demands significant computational resources. JTExpert, while fast, is best used in scenarios where rapid feedback is prioritized over deep fault detection.

Future research could explore hybrid testing methodologies that combine symbolic execution, search-based testing, and dynamic analysis to optimize efficiency and effectiveness. Expanding evaluations beyond Java to include languages like Python and C++ could provide broader insights into tool performance. Further studies could assess how these tools integrate with agile development practices and CI/CD pipelines to enhance their practical applicability in modern software engineering workflows.

Additionally, evaluating factors like test maintainability, fault localization accuracy, and tool usability could provide a more holistic understanding of automated test generation tools. Investigating scalability in ultra-large codebases could identify potential performance bottlenecks and optimization opportunities. By addressing these areas, future research can build upon the findings of this thesis to advance automated testing methodologies and improve software quality assurance practices.

7 Conclusion

This thesis systematically evaluated EvoSuite, AgitarOne, JDART, and JTExpert, addressing key research questions by selecting tools based on active maintenance, Java compatibility, and diverse test generation methodologies. AgitarOne achieved the highest code coverage across 3 projects, demonstrating its capability in executing a broad range of test paths. EvoSuite provided a well-balanced performance, excelling in efficiency while maintaining strong fault-detection capabilities. JDART, leveraging symbolic execution, produced high mutation scores but required resources, appealing to enterprises needing maximum fault detection; JTExpert's speed suits tight deadlines but reduces coverage.

Execution time analysis showed that JTExpert was the most efficient in rapid test generation, making it suitable for time-sensitive scenarios. EvoSuite balanced execution time and test effectiveness, making it a versatile choice. JDART's symbolic execution was comprehensive but resource-intensive, leading to extended test generation durations. AgitarOne exhibited increasing execution times with higher complexity, indicating areas for optimization.

These findings highlight trade-offs between execution time, test effectiveness, and computational efficiency among the tools.

This bridges a knowledge gap by offering a comparative analysis of ATCG tools using standardized metrics on the Defects4J dataset for practitioners seeking optimized solutions for different testing scenarios. By identifying strengths and limitations, the thesis contributes to a broader understanding of ATCG tools, fostering improved software testing practices and advancing automated testing methodologies.

The findings have significant implications for science, industry, and software engineering. Developers and testers can use this thesis to make informed decisions when selecting ATCG tools based on project complexity, time constraints, and fault-detection priorities. This contributes to improving ATCG tools, offering insights into optimizing their performance for real-world applications.

The results are generalizable to Java-based projects, though certain limitations reveal further research paths. Beyond CI/CD integration, advanced concolic or hybrid testing frameworks and multi-module scenarios merit exploration. Exploring ATCG tools in Python or C++ could also expand applicability for diverse ecosystems. Addressing these directions enables future studies to extend this thesis and enhance automated software testing.

References

[1] Linnaeus University. (2015) Course Room for Degree Projects. [Online]. Available: https://mymoodle.lnu.se/course/view.php?id=5297#section-4

[2] Monash University. (2015, Oct. 13) Citing and referencing: IEEE. [Online]. Available: http://guides.lib.monash.edu/citing-referencing/ieee

[3] C. Lynch, “Big data: How do your data grow?” *Nature*, vol. 455, pp. 28–29, 2008.

[4] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, 2010.

[5] D. Agrawal, S. Das, and A. E. Abbadi, “Big data and cloud computing: current state and future opportunities,” in *Proceedings of the 14th International Conference on Extending Database Technology*, 2011, pp. 530–533.

[6] S. Wang, N. Shrestha, A. K. Subburaman, J. Wang, M. Wei, and N. Nagappan, “Automatic Unit Test Generation for Machine Learning Libraries: How Far Are We?,” *IEEE Xplore*, May 01, 2021. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9402041

[7] J. Team, T. Tsuji, A. Akinyele, F. Hueppi, E. Yampratoom, and I. Upeka Bulumulla, “Evaluation of AgitarOne Analysis of Software Artifacts Final Project Report,” 2007. Accessed: Dec. 11, 2024. [Online]. Available: https://www.cs.cmu.edu/~aldrich/courses/654/tools/tsuji-agitarone-07.pdf

[8] Liu, XJ., Yu, P. & Ma, XX. An Empirical Study on Automated Test Generation Tools for Java: Effectiveness and Challenges. J. Comput. Sci. Technol. 39, 715–736 (2024). https://doi.org/10.1007/s11390-023-1935-5

[9] Tomáš Potužák and R. Lipka, “Current Trends in Automated Test Case Generation,” *Computer Science and Information Systems (FedCSIS), 2019 Federated Conference on*, Sep. 2023, doi: https://doi.org/10.15439/2023f9829.

[10] T. Kurmaku, Eduard Paul Enoiu, and Musa Kumrija, “Human-based Test Design versus Automated Test Generation: A Literature Review and Meta-Analysis,” Feb. 2022, doi: https://doi.org/10.1145/3511430.3511433.

[11] S. Anand *et al.*, “An orchestrated survey of methodologies for automated software test case generation,” *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, Aug. 2013, doi: https://doi.org/10.1016/j.jss.2013.02.061.

[12] S. J. Galler and B. K. Aichernig, “Survey on test data generation tools,” *International Journal on Software Tools for Technology Transfer*, vol. 16, no. 6, pp. 727–751, Apr. 2013, doi: https://doi.org/10.1007/s10009-013-0272-3.

[13] R. Singh, A. Singhrova, and R. Bhatia, “Test Case Generation Tools - A Review,” Nov. 16, 2018. https://www.researchgate.net/publication/328996233_Test_Case_Generation_Tools_-_A_Review

[14] R. Ramler, C. Klammer, and Georg Buchgeher, “Applying Automated Test Case Generation in Industry: A Retrospective,” Apr. 2018, doi: https://doi.org/10.1109/icstw.2018.00074.

[15] D. Corradini, A. Zampieri, M. Pasqua, and M. Ceccato, “Empirical Comparison of Black-box Test Case Generation Tools for RESTful APIs,” *arXiv.org*, 2021. https://arxiv.org/abs/2108.08196 (accessed Dec. 11, 2024).

[16] J. Offutt and R. H. Untch, "Mutation 2000: Uniting the Orthogonal," in *Mutation Testing for the New Century*, Springer, Boston, MA, 2001, pp. 34–44.

[17] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep. 2011.

[18] M. Papadakis, Y. Le Traon, and M. Harman, "The mutation testing landscape: From infancy to maturity," *ACM Comput. Surv.*, vol. 51, no. 1, Art. no. 7, Jan. 2019.

[19] S. J. Galler and B. K. Aichernig, "Survey on test data generation tools," *Int. J. Softw. Tools Technol. Transf.*, vol. 16, no. 6, pp. 727–751, Dec. 2013.

[20] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2nd ed. Cambridge, U.K.: Cambridge Univ. Press, 2016.

[21] V. Garousi and M. V. Mäntylä, "When and what to automate in software testing? A multi-vocal literature review," *Information and Software Technology*, vol. 76, pp. 92–117, Apr. 2016, doi:https://dl.acm.org/doi/10.1016/j.infsof.2016.04.015

[22] S. Shamshiri, J. R. C. Nurse, M. Mirakhorli, and L. Williams, "Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges," in Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015, pp. 201–211. doi:https://doi.org/10.1109/ASE.2015.86

[23] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil Mcminn. 2013. An orchestrated survey of methodologies for automated software test case generation. J. Syst. Softw. 86, 8 (August, 2013), 1978–2001. https://doi.org/10.1016/j.jss.2013.02.061

[24] Harman, Mark & McMinn, Phil. (2010). A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. Software Engineering, IEEE Transactions on. 36. 226 - 247. 10.1109/TSE.2009.71. https://www.researchgate.net/publication/224085237_A_Theoretical_and_Empirical_Study_of_Search-Based_Testing_Local_Global_and_Hybrid_Search

[25] Ko, A.J., LaToza, T.D. & Burnett, M.M. A practical guide to controlled experiments of software engineering tools with human participants. *Empir Software Eng* **20**, 110–141 (2015). https://doi.org/10.1007/s10664-013-9279-3

[26] D. I. K. Sjoeberg *et al*., "A survey of controlled experiments in software engineering," in *IEEE Transactions on Software Engineering*, vol. 31, no. 9, pp.733-753, Sept.2005, doi:10.1109/TSE.2005.97.

[27] Kasper Luckow, Marko Dimjašević, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamarić, and Vishwanath Raman. 2016. JDart: A Dynamic Symbolic Analysis Framework. In Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636. Springer-Verlag, Berlin, Heidelberg, 442–459. https://doi.org/10.1007/978-3-662-49674-9_26

[28] Maktala, Sai Shashank; Jatin Sachdeva, 2025, "Comparative Analysis of Automated Test Case Generation Tools in Java", https://doi.org/10.7910/DVN/RBJIXL, Harvard Dataverse, V1, UNF:6:pbYYpN9x57L+GSVAwFi8ew== [fileUNF]

[29]“Code coverage,” *Wikipedia*, Feb. 04, 2020. https://en.wikipedia.org/wiki/Code_coverage

[30] P. Zhang, “Industrial control systems,” *Advanced Industrial Control Technology*, pp. 3–40, 2010, doi: https://doi.org/10.1016/b978-1-4377-7807-6.10001-4.

[31] Wikipedia Contributors, “Mutation testing,” *Wikipedia*, Oct. 14, 2018. https://en.wikipedia.org/wiki/Mutation_testing

[32] EvoSuite, “GitHub - EvoSuite/evosuite: EvoSuite - automated generation of JUnit test suites for Java classes,” *GitHub*, Sep. 16, 2021. https://github.com/EvoSuite/evosuite

[33] “AgitarOne | Agitar Technologies: Putting Java to the Test,” *Agitar.com*, 2025. http://www.agitar.com/solutions/products/agitarone.html (accessed Feb. 25, 2025).

[34] psycopaths, “GitHub - psycopaths/jdart: A dynamic symbolic analysis tool for Java,” *GitHub*, 2015. https://github.com/psycopaths/jdart (accessed Feb. 25, 2025).

[35] ptidejteam, “GitHub - ptidejteam/jtexpert-JTeXpert,” *GitHub*, 2022. https://github.com/ptidejteam/jtexpert-JTeXpert (accessed Feb. 25, 2025).

[36] randoop, “GitHub - randoop/randoop: Automatic test generation for Java,” *GitHub*, May 02, 2024. https://github.com/randoop/randoop (accessed Feb. 25, 2025).

[37] javapathfinder, “GitHub - javapathfinder/jpf-core: JPF is an extensible software analysis framework for Java bytecode. jpf-core is the basis for all JPF projects; you always need to install it. It contains the basic VM and model checking infrastructure, and can be used to check for concurrency defects like deadlocks, and unhandled exceptions like NullPointerExceptions and AssertionErrors.,” *GitHub*, 2017. https://github.com/javapathfinder/jpf-core (accessed Feb. 25, 2025).

[38] klee, “GitHub - klee/klee: KLEE Symbolic Execution Engine,” *GitHub*, Feb. 29, 2024. https://github.com/klee/klee (accessed Feb. 25, 2025).

[39] KeYProject, “GitHub - KeYProject/key: KeY Theorem Prover for Deductive Java Verification,” *GitHub*, Sep. 08, 2024. https://github.com/KeYProject/key (accessed Feb. 25, 2025).

[40] UnitTestBot, “GitHub - UnitTestBot/UTBotJava: Automated unit test generation and precise code analysis for Java,” *GitHub*, Oct. 05, 2023. https://github.com/UnitTestBot/UTBotJava (accessed Feb. 25, 2025).

[41] ksen007, “GitHub - ksen007/janala2: a concolic testing engine for Java,” *GitHub*, 2025. https://github.com/ksen007/janala2 (accessed Feb. 25, 2025).

[42] “PIT Mutation Testing,” *pitest.org*. https://pitest.org/

[43] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. 2014. Continuous test generation: enhancing continuous integration with automated test generation. In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14). Association for Computing Machinery, New York, NY, USA, 55–66. https://doi.org/10.1145/2642937.2643002

[44] jacoco, “jacoco/jacoco,” *GitHub*, Oct. 11, 2019. https://github.com/jacoco/jacoco

[45] Boyuan Chen, Jian Song, Peng Xu, Xing Hu, and Zhen Ming (Jack) Jiang. 2018. An automated approach to estimating code coverage measures via execution logs. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18). Association for Computing Machinery, New York, NY, USA, 305–316. https://doi.org/10.1145/3238147.3238214

[46] “JaCoCo - Documentation,” *www.jacoco.org*. https://www.jacoco.org/jacoco/trunk/doc/index.html

[47] rjust, “rjust/defects4j,” *GitHub*, May 18, 2019. https://github.com/rjust/defects4j

Appendix 1

A1.1 Hardware and Software Environment

Hardware Specifications

- Processor: Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz, 4 cores, 8 logical processors
- RAM: 16.0 GB
- System Type: 64-bit operating system, x64-based processor
- Storage: 512 GB SSD

Operating System and Runtime

- Operating System: Windows 10 (64-bit)
- Java Version: OpenJDK 17
- Python Version (where used for scripting): 3.9

Software and Tools

- Defects4J (v2.0.0)
- Maven (v3.8.7)
- Git (v2.4 or newer)
- PiTest (v1.9.11) for mutation testing
- JaCoCo (v0.8.10) for code coverage collection
- Bash or PowerShell scripts for automation

This standardized setup minimizes environmental differences, ensuring that any observed discrepancies in tool performance arise primarily from the tools themselves rather than inconsistent hardware or software configurations.

A1.2 Defects4J Preparation

All subject projects were checked out from the Defects4J framework, which includes both buggy and fixed versions for each defect. Below is a step-by-step overview of how Defects4J was prepared for each run:

Cloning the Defects4J RepositorySelecting the Project and Bug ID

- git clone https://github.com/rjust/defects4j.git
- cd defects4j
- ./init.sh

Selecting the Project and Bug ID

- defects4j checkout -p Time -v 1b -w time_1_buggy
- defects4j checkout -p Time -v 1f -w time_1_fixed

To check out the *Time* project with bug ID 1:

- 1b indicates the buggy version of bug 1.
- 1f indicates the fixed version of bug 1.

Project Build and Instrumentation

- Use Defects4J's built-in commands or Maven for project compilation.
- If using JaCoCo for coverage instrumentation, add it as a Maven plugin in the pom.xml.

Baseline Testing

- Optionally, run the project's existing test suite (if available) to confirm that the environment is correctly configured before applying the ATCG tools.

A1.3 Tool Configuration Details

Each automated test case generation (ATCG) tool required a specific set of parameters to ensure consistency. Below are key configurations and example commands or scripts used.

A1.3.1 EvoSuite

Installation and Setup:

- Download the EvoSuite .jar file from the EvoSuite GitHub repository.
- Ensure Maven or command-line calls include the EvoSuite jar in the classpath.

Common Parameters:

- -class <TARGET_CLASS>: The class under test.
- -projectCP <CLASSPATH>: Classpath to compiled project classes and dependencies.
- -Dsearch_budget=<TIME_IN_SECONDS>: Time budget per class (experimentally varied between 10, 60, 180, 360, 600).

Example Command:

```
java -jar evosuite.jar \
        -class org.joda.time.DateTime \
        -projectCP /path/to/time_1_buggy/target/classes \
        -Dsearch_budget=180 \
        -Dtest_dir=./evosuite-tests/
```

Key Configuration Notes:

- If the project includes multiple modules, aggregate them in the classpath.
- EvoSuite sometimes requires separate runs for each class if the project is large; automation scripts cycle through target classes.

A1.3.2 AgitarOne

Installation and Setup:

- AgitarOne is commercial software. A valid license is required.
- Typically installed via an executable or installer that integrates into the local environment.

Common Parameters:

- Uses an internal configuration for specifying coverage and test generation settings.
- Integration with Maven can be done by adding a plugin or calling agitar:mvn goals.

Example Command:

```
mvn agitar:prepare-agitar \
        -Dclasses=org.joda.time.DateTime \
        -Dagitar.timeout=180
```

Key Configuration Notes:

- Tends to generate comprehensive tests, but can require a significantly higher runtime.
- Best for continuous integration environments where thorough coverage is a priority.

A1.3.3 JDART

Installation and Setup**:**

- Clone the JDART repository.
- Build using Maven (mvn clean install).
- Use scripts to run JDART's symbolic execution engine.

Common Parameters:

- -classpath: Points to the compiled buggy code and JDART's libraries.
- Configuration files often specify path constraints (e.g., max depth) and time budget.

Example Command:

```
java -Xmx4G -jar jdart.jar \
        -classpath /path/to/time_1_buggy/target/classes \
        -target org.joda.time.DateTime \
        -sym-args 1 2 int \
        -timeBudget=180
```

The -sym-args parameter is an example that configures symbolic arguments. Actual usage varies by class.

Key Configuration Notes:

- Symbolic execution can lead to path explosion for complex classes.
- JDART benefits significantly from higher memory (e.g., -Xmx4G) and longer execution budgets.

A1.3.4 JTExpert

Installation and Setup:

- Obtain JTExpert from the GitHub repository.
- Build using Gradle or Maven, depending on the version.

Common Parameters:

- Often uses dynamic analysis and requires specifying the entry points for test generation.
- Time budget or iteration count can be set in a config file or via command-line flags, such as -T 180.

Example Command:

```
java -jar jtexpert.jar \
        -project /path/to/time_1_buggy \
        -target org.joda.time.DateTime \
        -T 180
```

Key Configuration Notes:

- Optimized for quick test generation but may not achieve deep path coverage.
- Usually minimal manual configuration is required compared to symbolic execution tools.

A1.4 Automation Scripts

All tools were orchestrated using Bash. Below is a simplified snippet illustrating how an automated script might run each tool sequentially for every class within a buggy version of a Defects4J project:

```
PROJECT=$1 # e.g., "Time"
BUG_ID=$2   # e.g., "1"
TIME_BUDGET=$3  # e.g., "180"

# Paths
BUGGY_DIR="${PROJECT}_${BUG_ID}_buggy"
CLASSES_LIST="classes.txt"  # List of classes from a custom script or
Defects4J command

# Example loop
while read CLASS_NAME; do
  # 1. EvoSuite
  java -jar evosuite.jar \
          -class ${CLASS_NAME} \
          -projectCP "${BUGGY_DIR}/target/classes" \
          -Dsearch_budget=${TIME_BUDGET} \
          -Dtest_dir=./tests/evosuite

  # 2. AgitarOne (Maven invocation)
  mvn agitar:prepare-agitar -Dclasses=${CLASS_NAME} \
          -Dagitar.timeout=${TIME_BUDGET}

  # 3. JDART
  java -Xmx4G -jar jdart.jar \
          -classpath "${BUGGY_DIR}/target/classes" \
          -target ${CLASS_NAME} \
          -timeBudget=${TIME_BUDGET}

  # 4. JTExpert
  java -jar jtexpert.jar \
          -project "${BUGGY_DIR}" \
          -target ${CLASS_NAME} \
          -T ${TIME_BUDGET}

done < ${CLASSES_LIST}
```

While the actual scripts used in the thesis varies, this follows the same general pattern of iterating over each target class, executing each tool under identical time-budget constraints.

A1.5 Mutation Testing with PiTest

After generating tests with each tool, PiTest was employed to evaluate mutation scores:

Instrumentation**:** PiTest mutates Java bytecode by applying operators such as condition negation and arithmetic operator replacement.

Execution of Test Suites**:**

- Each tool's generated tests (e.g., from EvoSuite or JDART) were compiled and then run against the mutated code using PiTest.
- PiTest calculates the proportion of mutants *killed* (detected) by the test suite.

Mutation Score Computation:

- Mutation Score (%) = (Number of killed mutants ÷ Total mutants generated) × 100
- A higher score indicates stronger fault-detection capabilities.

A1.6 Raw Data Samples

Tables A1.1 and A1.2 show brief segments of raw coverage and mutation data for *Mockito* under a 180-second budget, illustrating how results were logged.

Table A1.1: Sample Coverage Data (Mockito, 180-second budget)

Tool	Lines Covered	Total Lines	Coverage (%)
EvoSuite	5,234	8,775	59.6
AgitarOne	5,963	8,775	68.0

JDART	6,581	8,775	75.0
JTExpert	7,311	8,775	83.3

Table A1.2: Sample Mutation Data (Mockito, 180-second budget)

Tool	Mutants Generated	Mutants Killed	Mutation Score (%)
EvoSuite	1,470	936	63.7
AgitarOne	1,470	990	67.3
JDART	1,470	1,161	79.0
JTExpert	1,470	1,202	81.7

These sample records were aggregated with results from other projects (Time, Chart, Math, etc.), run three times, and then averaged to increase reliability. Outliers or incomplete runs were re-executed to reduce noise.

A1.7 Observed Tool-Specific Issues

AgitarOne: Occasionally exceeded its own time limits when run on deeply nested packages, requiring an increase to default timeout parameters. Proprietary constraints limited advanced customization.

JDART: Consumed large amounts of memory during symbolic execution of complex classes. If a path explosion occurred, partial results still gave partial coverage but impacted overall coverage metrics.

EvoSuite: In some cases generated flaky tests for classes with significant non-deterministic behavior (e.g., concurrency). Additional seeding or stable random seeds minimized variance.

JTExpert: Rapid generation approach sometimes missed complex paths, resulting in lower coverage in certain large libraries. Minimal public documentation, requiring more manual trial-and-error than with EvoSuite or JDART.

A1.7 Graph Generation

Here is the python code used to generate the box plot graphs.

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load the raw results CSV file
file_path = "Raw_Results_ATCG_Tools_with_Iterations.csv"  # Update
this path if needed
df = pd.read_csv(file_path)

# Define custom colors for box plots (matching scatter plots)
custom_palette = {
        "EvoSuite Execution Time (s)": "red",
        "AgitarOne Execution Time (s)": "green",
        "JDART Execution Time (s)": "blue",
        "JTExpert Execution Time (s)": "yellow",
        "EvoSuite Mutation Score (%)": "red",
        "AgitarOne Mutation Score (%)": "green",
        "JDART Mutation Score (%)": "blue",
        "JTExpert Mutation Score (%)": "yellow",
        "EvoSuite Coverage (%)": "red",
        "AgitarOne Coverage (%)": "green",
        "JDART Coverage (%)": "blue",
        "JTExpert Coverage (%)": "yellow"
}

# Function to create box plots
def create_box_plot(y_columns, ylabel, title):
        plt.figure(figsize=(12, 7))
        # Convert data to long format
        df_melted = df.melt(value_vars=y_columns, var_name="Tool",
value_name=ylabel)
```

```
        # Box plot with outlined edges
        sns.boxplot(
        x="Tool", y=ylabel, data=df_melted, hue="Tool",
palette=custom_palette, legend=False,
        boxprops=dict(edgecolor="black", linewidth=2),      # Outline for
box
        whiskerprops=dict(color="black", linewidth=2),      # Outline for
whiskers
        capprops=dict(color="black", linewidth=2),          # Outline for
caps
        medianprops=dict(color="black", linewidth=2),       # Outline for
median
        flierprops=dict(marker='o', markeredgecolor="black",
markersize=6)  # Outlier marker outline
        )

        # Increase font sizes
        plt.xlabel("Automated Test Case Generation Tools", fontsize=36,
fontweight="bold")
        plt.ylabel(ylabel, fontsize=36, fontweight="bold")
        plt.title(title, fontsize=38, fontweight="bold")
        plt.xticks(fontsize=28)
        plt.yticks(fontsize=28)
        # Remove excess whitespace
        plt.margins(x=0.05)
        plt.tight_layout()
        plt.grid(True, linestyle="--", linewidth=0.5, alpha=0.7)
        plt.subplots_adjust(left=0.05, bottom=0.08, right=0.98, top=0.95,
wspace=None, hspace=None)
        plt.show()

# Generate box plots
create_box_plot(
        ["EvoSuite Coverage (%)", "AgitarOne Coverage (%)", "JDART
Coverage (%)", "JTExpert Coverage (%)"],
        "Code Coverage (%)",
        "Comparison of Code Coverage Across ATCG Tools"
)
create_box_plot(
        ["EvoSuite Execution Time (s)", "AgitarOne Execution Time (s)",
"JDART Execution Time (s)", "JTExpert Execution Time (s)"],
        "Execution Time (s)",
        "Comparison of Execution Time Across ATCG Tools"
```

```
)
create_box_plot(
        ["EvoSuite Mutation Score (%)", "AgitarOne Mutation Score (%)",
"JDART Mutation Score (%)", "JTExpert Mutation Score (%)"],
        "Mutation Score (%)",
        "Comparison of Mutation Score Across ATCG Tools"
)
```

Here is the python code used to generate the scatter plot graphs.

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Load the raw results CSV file
file_path = "Raw_Results_ATCG_Tools_with_Iterations.csv"  # Update
this path if needed
df = pd.read_csv(file_path)

# Define custom colors for box plots (matching scatter plots)
custom_palette = {
        "EvoSuite Execution Time (s)": "red",
        "AgitarOne Execution Time (s)": "green",
        "JDART Execution Time (s)": "blue",
        "JTExpert Execution Time (s)": "yellow",
        "EvoSuite Mutation Score (%)": "red",
        "AgitarOne Mutation Score (%)": "green",
        "JDART Mutation Score (%)": "blue",
        "JTExpert Mutation Score (%)": "yellow",
        "EvoSuite Coverage (%)": "red",
        "AgitarOne Coverage (%)": "green",
        "JDART Coverage (%)": "blue",
        "JTExpert Coverage (%)": "yellow"
}

# Define custom colors and marker size
custom_colors = ["red", "green", "blue", "yellow"]
marker_size = 300  # Increased size for better visibility

# Extract unique project names and create an index for jitter
project_names = df["Project Name"].unique()
x_positions = np.arange(len(project_names)) * 3  # Increased spacing
```

```
project_x_map = dict(zip(project_names, x_positions))

# Function to add slight jitter to x-axis points
def add_jitter(x_value, num_points, jitter_amount=0.2):
        return [x_value + jitter_amount * (i - (num_points - 1) / 2) for i in
range(num_points)]

# Function to create scatter plots
def create_scatter_plot(y_columns, ylabel, title, point_size=300):  #
Adjustable point size
        plt.figure(figsize=(14, 7))

        # Extract unique project names and x-axis positions
        project_names = df["Project Name"].unique()
        x_positions = np.arange(len(project_names)) * 3  # Increased
spacing
        project_x_map = dict(zip(project_names, x_positions))

        # Function to add slight jitter to prevent overlap
        def add_jitter(x_value, num_points, jitter_amount=0.2):
        return [x_value + jitter_amount * (i - (num_points - 1) / 2) for i in
range(num_points)]

        # Loop through each project and apply jitter to x-values
        for project_name, x_pos in project_x_map.items():
        subset = df[df["Project Name"] == project_name]
        num_tools = len(y_columns)
        x_jittered = add_jitter(x_pos, num_tools)  # Apply jitter

        for i, col in enumerate(y_columns):
        sns.scatterplot(x=[x_jittered[i]] * len(subset), y=subset[col],
                        color=custom_palette[col], s=point_size,
marker="o",
                        edgecolor="black", linewidth=1.5, label=None,
zorder=3)  # Added outlines

        # Set x-axis with project names directly below the points
        plt.xticks(x_positions, project_names, rotation=45, ha="right",
fontsize=24)
        plt.ylabel(ylabel, fontsize=36, fontweight="bold")
        plt.xlabel("Project Name", fontsize=36, fontweight="bold")
        plt.title(title, fontsize=38, fontweight="bold")
```

```
        # Adjust y-axis tick size
        plt.yticks(fontsize=28)

        # Move legend outside and use larger font
        plt.legend(labels=[col.split()[0] for col in y_columns],
bbox_to_anchor=(1, 1), loc='upper left', fontsize=24)

        # Remove excess whitespace
        plt.margins(x=0.05)
        plt.tight_layout()

        # Grid behind the points
        plt.grid(True, linestyle="--", linewidth=0.7, alpha=0.6, zorder=1)
        plt.subplots_adjust(left=0.05, bottom=0.19, right=0.855, top=0.95,
wspace=None, hspace=None)
        plt.show()

# Generate scatter plots
create_scatter_plot(
        ["EvoSuite Coverage (%)", "AgitarOne Coverage (%)", "JDART
Coverage (%)", "JTExpert Coverage (%)"],
        "Code Coverage (%)",
        "Comparison of Code Coverage Across ATCG Tools"
)

create_scatter_plot(
        ["EvoSuite Execution Time (s)", "AgitarOne Execution Time (s)",
"JDART Execution Time (s)", "JTExpert Execution Time (s)"],
        "Execution Time (s)",
        "Comparison of Execution Time Across ATCG Tools"
)

create_scatter_plot(
        ["EvoSuite Mutation Score (%)", "AgitarOne Mutation Score (%)",
"JDART Mutation Score (%)", "JTExpert Mutation Score (%)"],
        "Mutation Score (%)",
        "Comparison of Mutation Score Across ATCG Tools"
)
```