VALID 2024

The Sixteenth International Conference on Advances in System Testing and Validation Lifecycle

ISBN: 978-1-68558-199-2

September 29 - October 03, 2024

Venice, Italy

VALID 2024 Editors

Jos van Rooyen, Huis voor softwarekwaliteit, Nederland

Luigi Lavazza, Universita` degli Studi dell'Insubria, Italy

VALID 2024

Forward

The Sixteenth International Conference on Advances in System Testing and Validation Lifecycle (VALID 2024), held on September 29 – October 3, 2024 in Venice, Italy, continued a series of events focusing on designing robust components and systems with testability for various features of behavior and interconnection.

Complex distributed systems with heterogeneous interconnections operating at different speeds and based on various nano- and micro-technologies raise serious problems of testing, diagnosing, and debugging. Despite current solutions, virtualization and abstraction for large scale systems provide less visibility for vulnerability discovery and resolution, and make testing tedious, sometimes unsuccessful, if not properly thought from the design phase.

The conference on advances in system testing and validation considered the concepts, methodologies, and solutions dealing with designing robust and available systems. Its target covered aspects related to debugging and defects, vulnerability discovery, diagnosis, and testing.

The conference provided a forum where researchers were able to present recent research results and new research problems and directions related to them. The conference sought contributions presenting novel result and future research in all aspects of robust design methodologies, vulnerability discovery and resolution, diagnosis, debugging, and testing.

We welcomed technical papers presenting research and practical results, position papers addressing the pros and cons of specific proposals, such as those being discussed in the standard forums or in industry consortiums, survey papers addressing the key problems and solutions on any of the above topics, short papers on work in progress, and panel proposals.

We take here the opportunity to warmly thank all the members of the VALID 2024 technical program committee as well as the numerous reviewers. The creation of such a broad and high quality conference program would not have been possible without their involvement. We also kindly thank all the authors that dedicated much of their time and efforts to contribute to VALID 2024. We truly believe that thanks to all these efforts, the final conference program consists of top quality contributions.

This event could also not have been a reality without the support of many individuals, organizations and sponsors. We also gratefully thank the members of the VALID 2024 organizing committee for their help in handling the logistics and for their work that is making this professional meeting a success. We gratefully appreciate to the technical program committee co-chairs that contributed to identify the appropriate groups to submit contributions.

We hope the VALID 2024 was a successful international forum for the exchange of ideas and results between academia and industry and to promote further progress in system testing and validation. We also hope that Venice provided a pleasant environment during the conference and everyone saved some time for exploring this beautiful city

VALID 2024 Steering Committee

Lorena Parra Boronat, Instituto Madrileño de Investigación y Desarrollo Rural, Agrario y Alimentario and Universitat Politecnica de Valencia, Spain
Yan-Fu Li, Tsinghua University, China

Table of Contents

Analysis of Test Smell Impact on Test Code Quality

Ismail Cebeci and Tugkan Tuglular
Department of Computer Engineering
Izmir Institute of Technology
Izmir, Turkiye
e-mail: {ismailcebeci | tugkantuglular}@iyte.edu.tr

***Abstract*—Software testing is a crucial component of the software development life-cycle, playing a key role in ensuring the quality and robustness of software products. However, test code, like production code, is susceptible to poor design choices or "test smells", which can compromise its effectiveness and maintainability. This article investigates the prevalence and impact of various test smells across open-source software projects, using advanced detection tools such as JNose and TestSmellDetector. The research highlights that certain test smells, such as "Assertion Roulette," "Magic Number Test," and "Lazy Test," are notably prevalent. The study also examines the co-occurrence of different test smells, providing understanding of how these issues interrelate. Additionally, the article compares the effectiveness of JNose and TestSmellDetector in detecting test smells, providing insights into their strengths and limitations.**

Keywords-Test Smells; Software Testing; Empirical Software Engineering.

I. INTRODUCTION

Software testing is a fundamental part of the software development process and has significant importance in ensuring the quality of software [1]. Test cases exhibit a crucial role in the early detection of software bugs during the software development process.

The quality of the test suite is measured with test coverage analysis where the number of different structural components (functions, instructions, branches, and lines of code) included in the test suite is considered [2]. Nevertheless, despite having a large amount of code coverage, the test code may still contain design choices that are not well-executed, known as test smells. The inclusion of smells in test code has the potential to affect the overall quality of test suites, hence impacting the quality of the production code.

The motivation behind this research stems from the observation that despite the critical role of testing in software development, test smells are often overlooked. Developers and testers may inadvertently introduce these smells into the test code, not through a lack of skill, but due to pressures of deadlines, lack of awareness, or inadequate tool support.

This study contributes to the field by providing empirical data on the detection and impact of test smells across a broad spectrum of open-source software projects. It leverages modern test smell detection tools-JNose [3] and TestSmellDetector [4] tools-to gather insights into the prevalence and co-occurrence of different smells, thereby offering a granular understanding of how these smells interrelate and the potential for cascading effects within the test code. Moreover, for these two tools, a comparison was made on issues such as the differences between them, which test smells are detected better, which device detects more test smells.

The structure of this thesis is organized as follows: Following this introduction, Section II reviews STATE OF THE ART in the field, laying a theoretical foundation for understanding test smells. Section III describes the TOOL INFRASTRUCTURE used in the study, including a detailed examination of the JNose and TestSmellDetector tools. Section IV presents a CASE STUDY analysis, where these tools are applied to a dataset of software projects to identify and analyze test smells. Section V shows the observed RESULTS and Section VI concludes findings and directions for future research.

II. STATE OF THE ART

Modern studies are going in the direction of discovering, defining, and eliminating various categories of code smells, and explaining their origins and influence on the overall program quality. Such studies utilize several approaches, including empirical analysis of open-source software projects and constructing and testing elaborate security tools.

A study by Silva Junior et al. [5], the researchers examined the awareness of test practitioners and the unknowingly incorporation of smells to test code development. A survey is conducted with 60 chosen professionals from different organizations to investigate the frequency and situations in which they encounter smells, particularly 14 types of test smells, which are frequently used in cutting-edge test smell detection tools.

In another study [6] related to the severity of test smells by Campos et al., a set of tests that cause problematic consequences are targeted and the developers' point of view on the issue of test smells is mentioned. By working with its developer participants from six open-source software projects on GitHub, the study aims at characterizing to which extent developers perceive test smells to affect the test code they implement.

In a similar study by Davide Spadini et al. [7], severity thresholds for test smells are investigated. Using 1489 java projects from Apache and Eclipse ecosystems and TestSmellDetector tool, they considered 4 test smells-Assertion Roulette, Eager Test, Verbose Test, and Conditional Test Logic-are higher thresholds than others.

In our study, with extending the total number of test smell types, 21 types of test smells are used, and with using 500 open-source GitHub projects (more than 5000 Java test files), "Magic Number Test" and "Assertion Roulette" are detected as most frequent test smells. “Empty Test”, “Sleepy Test”, and “Mystery Guest” are 3 of the 5 lowest test smells detected using JNose tool [3] and TestSmellDetector tool [4].

Another study [8] by Michele Tufano et al. presented (i) a survey among 19 developers is carried out to find out how they rated test smells as design issues, and (ii) a huge empirical study based on commit history of 152 open source projects and focused on identifying aspects of both software systems such as when test smells are introduced, how long they last and their relationship with code smells affecting the classes tested.

In our study, to detect test smells, we used two different automated test smell detection tool "JNose Tool” and TestSmellDetector Tool" and the results show that all test files have at least one type of test smell, and to have better test code quality, all test smells should be resolved by developers.

In another study [9] by Soares et al., an innovative way to raise the quality of test code using the JUnit 5 features is described. As part of this research, a mixed-method survey is executed, covering 485 of the most widely used Java open-source projects, finding out that JUnit 5 is used by only a tiny share (15,9%).

In the paper [10] by Annibale Panichella et al., authors scrutinize test smells in the context of automatic test generation. They critically examine whether such smell detection tools work well on sets of tests generated by tool EVOSUITE that test 100 classes of Java programs, in which there are 2340 test cases. Two tools are used in the study. Static detection rules are the first one among the tools suggested by Bavota et al. [11], Grano et al. [12] also use this same tool to detect test smells in test codes. The next tool is TestSmellDetector tool, which is available on GitHub and can be used publicly. The frequency of detection of test smells in Static Detection rules is significantly lower if we compare the findings between Static Detection rules and TestSmellDetector tool. The TestSmellDetector tool demonstrates slightly superior outcomes. Martins et al. [13] also use TestSmellDetector tool to detect test smells and investigate co-occurrence values between different test smells.

Benefiting from previous articles, in addition to similarities, in this article, a research was conducted for the first time using the two mentioned tools and 21 types of test smells with using huge number of projects "around 500", and the results obtained for both tools were compared. Additionally, the co-occurance of the test smells for both tools were compared.

III. TOOL INFRASTRUCTURE

This section mainly explains the tool infrastructure used to detect test smells, in which a detailed analysis about JNose and TestSmellDetector tools are presented. It introduces the working principles of these tools by detailing how they analyze and recognize test smells in test code.

A. JNose Tool

The JNose Test tool enables testers to review the past versions of the software projects and find the test coverage

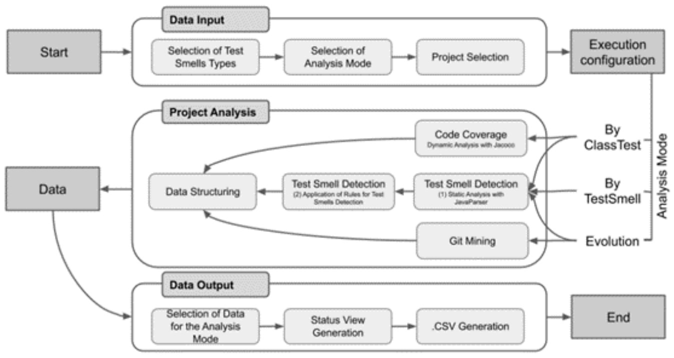


Figure 1. High-level architecture of Jnose tool

and the test smells that often bother the code quality. This fact enables us to compare various quality metrics of the project over the course of its development process. There are three crucial procedures in the JNose Test operation as shown in Figure 1.

- Data Input: This part receives the input set of command parameters for the tool execution, such as test smell types of lists, analysis mode (code coverage, test smells detection and evolution), and the project for analysis.
- Project Analysis: This component presents the analysis of the program by choosing the analysis mode.
- Data Output: By this component, the status of the execution is being rendered and the comma-separated value (CSV) file containing the results of the analysis is generated.

The JNose Tool offers the capability to detect and analyze smells in various ways. Firstly, it can detect smells in a specific test class using the TestClass method, which provides information about the quantity of each type of smell detected in the test class. Secondly, it can detect smells across multiple project versions using the Evolution method, which provides information about the authors and timestamps of the test smell’s insertion in the test code. Lastly, the detection can be used to identify the precise location of a test smell using the TestSmell method, which returns the method location of the smell for the purpose of analyzing the quality of the test code.

In accordance with the GNU General Public License, the JNose Test tool is licensed. The software tool is developed as a Java project and consists of four packages: (i) core, which is responsible for detecting test smells and coverage metrics; (ii) page, which is responsible for displaying web pages and their content; (iii) dto, which includes the classes used in data transfer (Data Transfer Object); (iV) util, which is responsible for identifying tests and production classes and saving results into CSV files.

B. TestSmellDetector Tool

The objective of including TestSmellDetector tool is to offer developers an automated methodology for enhancing the quality of their test suites. The TestSmellDetector tool can

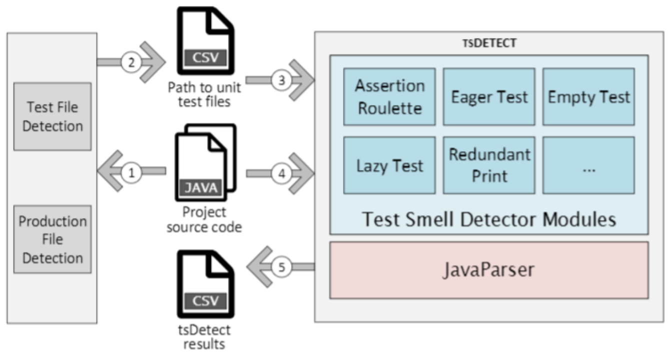


Figure 2. High-level architecture of TestSmellDetector tool

identify 19 smells present in Junit-based unit test files. The TestSmellDetector tool software provides a comprehensive list of detected smells, accompanied by their respective definitions and detection algorithms. The algorithm receives software project source code as input and initially distinguishes between unit test files and production source files.

TestSmellDetector tool is a Java jar file that is open-source and may be used as a command line program. The implementation of TestSmellDetector tool as a self-contained executable file, as opposed to a plugin, eliminates the need for users to own a dedicated Integrated Development Environment (IDE) on their system for the purpose of identifying smells in their test code.

Figure 2 illustrates a comprehensive overview of the architectural design of the TestSmellDetector tool. The project structure is used in 1 and 2 to identify the test and production files. TestSmellDetector tool determines whether test smells are present in the test files in 3 and 4. The test smell detection process findings are saved in 5.

IV. CASE STUDY

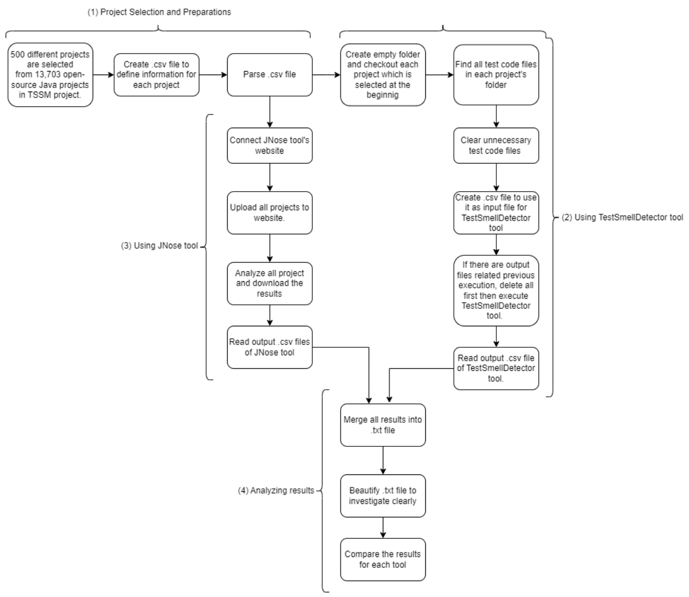
To understand test smell impaction of test code quality, we used two different tools which are JNose Tool and TestSmellDetector Tool then we analyzed the result of output files of both tools using projects that they used from Test Smells and Structural Metrics (TSSM) dataset [13].

Figure 3 shows an overview of our study. Mainly in this study, there are four parts to get results to compare and to answer our research questions.

- Project Selection and Preparations: to select projects and preparations to use JNose and TestSmellDetector tools.
- Using TestSmellDetector tool: to follow a way to get results after using TestSmellDetector tool.
- Using JNose tool: to follow a way to get results after using JNose tool.
- Analyzing results: to obtain results to answer research questions.

A. Project Selection

These procedures led to the collection of data from 13,703 open-source Java projects that make up the TSSM dataset.



(1) Project Selection and Preparations
500 different projects are selected from 13,703 open-source Java projects in TSSM project.
Create .csv file to define information for each project
Parse .csv file
Create empty folder and checkout each project which is selected at the beginnig
Find all test code files in each project's folder
Clear unnecessary test code files
Create .csv file to use it as input file for TestSmellDetector tool
If there are output files related previous execution, delete all first then execute TestSmellDetector tool.
Read output .csv file of TestSmellDetector tool.
(2) Using TestSmellDetector tool
(3) Using JNose tool
Connect JNose tool's website
Upload all projects to website.
Analyze all project and download the results
Read output .csv files of JNose tool
(4) Analyzing results
Merge all results into .txt file
Beautify .txt file to investigate clearly
Compare the results for each tool

Figure 3. High-level architecture of our study

500 distinct projects are randomly chosen from this collection of open-source Java projects. These projects work with the TestSmellDetector Tool as well as the JNose Tool. Java is among the most common languages today [14] and contains a large number of test codes. This gives us a lot of test code to examine. Additionally, since the two tools used work on Java codes, we decided to work with Java projects. Every project is tested separately at first, and if it works successfully with both tools, it is included in the list.

B. Implementation of Automated Scripts

In this study, four fundamental Python files were implemented. We will do the explanation of these files' roles and functions in detail. Each file has the sole aim of automating and facilitating a different aspect of testing smell analysis process, which in turn makes the identification, comparison, and understanding of test smells in many projects more efficient and accurate.

1) Python File for Preparation of Using Tools: In this file, six functions are created for preparation of using tools. These functions simply do these steps:

- Picking out necessary column names from input CSV file.
- Creating empty folder with using GitHub projects' names.
- Cloning GitHub projects into created empty folders one by one.
- Testing files and their associated source files within GitHub project folders.
- Removing the files, where the lines' sole content are comments.
- Creating a structured CSV file, which is originally named with output.csv and it is specifically designed to meet the given inputs of the TestSmellDetector application.

2) Python File for Using Tools: In this file, six functions are created for using tools. These functions simply do these steps:

- Executing TestSmellDetector Tool with 'output.csv' as a file input.
- Deleting files left over from past executions.
- Reading results clearly going through the created CSV file after executing TestSmellDetector tool. Then, creating txt file after reading CSV file.
- Reading results clearly going through the created CSV files after executing JNose tool. Then, creating txt file after reading all files.
- merging results by two different tools, into one conclusive file titled. After merging, findings might not be next to each other. Therefore, reorganizing findings after merging.

3) Python File for Comparing Results of Each Tools: Comparing the results of different testing methods, which are used in the detection of smells. Co-occurrence Analysis, Ratio Calculation and Comparison and Visualization are done in this file.

4) Python File for Connecting JNose Tool's Website: To accesses the webpage, which is related to Jnose Tool. It automatically inputs GitHub project links into the local server address "http://127.0.0.1:8080" and analyze each project. Then, it downloads results in the CSV format.

V. RESULTS

In this analysis, we compare the effectiveness of two software testing tools, JNose Tool and TestSmellDetector Tool, in identifying several types of test smells within software projects. Test smells play a critical role in ensuring the reliability and efficacy of software testing procedures by identifying any flaws in the test code that could undermine their quality or effectiveness.

The JNose Tool detected 81773 test smells in total using all files. The TestSmellDetector tool detected 89497 test smells in total using all files.

Figure 4 shows a comparative analysis of file affectation by test smells, the total number of files examined alongside those unaffected by test smells as identified by two separate tools: JNose and TestSmellDetector. It is evident that a comprehensive set of 5478 files were subjected to the analysis. JNose Tool identified 1550 files that exhibited no test smells, representing a significant portion of the total, yet still suggesting that many files could contain at least one form of test smell. In contrast, the TestSmellDetector Tool demonstrated a higher identification rate, with 1075 files reported as unaffected. Intriguingly, the bar labeled 'No Affected (Both)' is shown at a value of zero, indicating that there were no files, which both tools concurrently identified as free of test smells.

The data serves as a more encompassing and detailed view of the detection capabilities of both tools as they work across a range of test smells. The fact that different detection rates for various test smells are shown by the two tools indicates a noticeable difference as shown in Figure 5. The

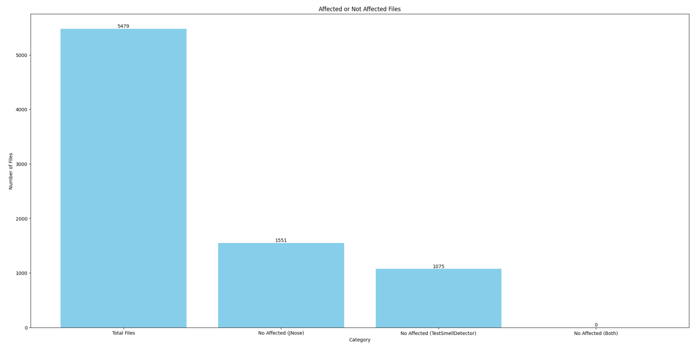


Figure 4. Number of Affected and not Affected Files

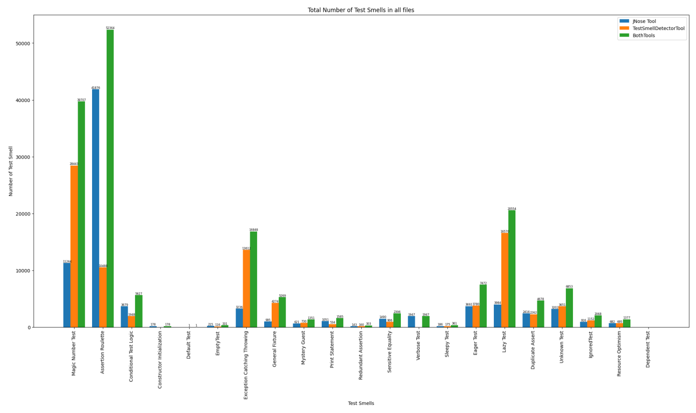


Figure 5. Total Number of Test Smells with using JNose and TestSmellDetector Tools in all files

TestSmellDetector Tool, for instance, is very effective in identifying 'Magic Number Test' smell with 28,443 instances detected entirely outperforming the 11,264 instances detected by the JNose Tool. The pattern of higher detection rates by the TestSmellDetector Tool is also observed in the other types of tests smells like 'Exception Catching Throwing' and 'Lazy Test', which the tool detected 13,612 and 16,570 occurrences, respectively and thus demonstrating its sensitivity towards these particular smells.For 'Assertion Roulette, TestSmellDetector Tool detected 10,488 occurence.

On the other hand, JNose Tool proved to be more effective than TestSmellDetector Tool in discovering the 'Assertion Roulette' instances, which were 41,876 compared to TestSmellDetector Tool, which discovered 10,488 instances as shown in Figure 5. This revelation of the JNose Tool's effectiveness in this case indicates that it can be particularly useful for scenarios where the tests contain multiple non-documented assertions, resulting in unclear test outcomes. In addition, the JNose Tool exhibits greater detection rates for various sorts of test smells, such as the 'Magic Number Test' and 'Lazy Test', with detection rates of 11,264 and 3984 occurrences, respectively. This demonstrates the tool's sensitivity towards these specific smells. JNose tool also performed high detection rates for 'Eager Test' with detection rate of 3692.

This analysis provides the absolute number of files affected

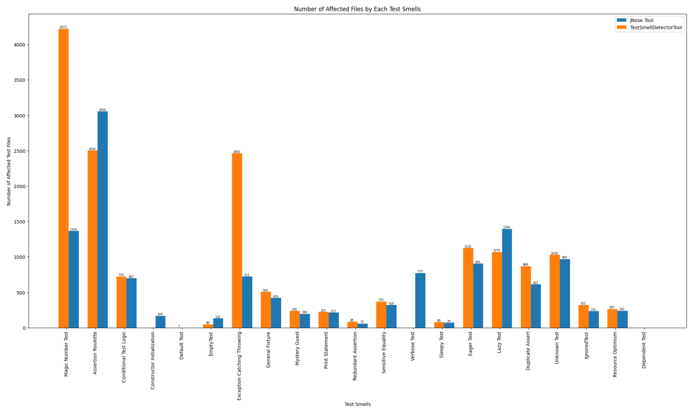


Figure 6. Number of Affected Files by Each Test Smells

by each test smell and allows an assessment of the extent of testing and detection of smell testing for both tools across various categories of test smell as shown in Figure 6.

By using the TestSmellDetector tool, highest numbers of affected files by ’Magic Number Test’, ’Assertion Roulette’, ’Exception Catching Throwing’, ’Eager Test’, ’Lazy Test’, and ’Unknown Test’ are detected as 4222, 2503, 2463, 1126, 1070, and 1030. On the other hand, by using the JNose tool, highest numbers of affected files by ’Assertion Roulette’, ’Lazy Test’, ’Magic Number Test’, ’Exception Catching Throwing’, ’Unknown Test’, and ’Eager Test’ are detected as 3056, 1396, 1364, 969, and 905.

The analysis also highlights test smells that are most and least prevalent in the datasets. ’Magic Number Test’, ’Assertion Roulette’, ’Exception Catching Throwing’, ’Eager Test’, ’Lazy Test’, and ’Unknown Test’ are among the most affecting test smells, with both tools identifying a considerable number of affected files. In contrast, ’Constructor Initialization’, ’Default Test’, and ’Dependent Test’ show minimal to no detection across both tools.

The utilization of co-occurrence matrices serves as an analytical cornerstone for uncovering the underlying patterns of test smell interactions within software testing environments. The matrices of The JNose Tool and TestSmellDetector Tool explain these patterns, illustrating both pronounced and negligible relationships among various test smells. In the interest of refining testing strategies, it becomes necessary to research into the specifics of these relationships.

Results for the JNose Tool as shown Figure 7, the one, which stands out the most is a correlation established between ’Conditional Test Logic’ and ’Eager Test’ with a co-occurrence value of [1.00], indicating a strong likelihood of these issues to arise simultaneously.

Similarly, the pairing of ’Exception Catching Throwing’ with ’Unknown Test’ and a high co-occurrence rate of [0.99] of using JNose Tool shows a strong correlation.

Next strong correlations are the one observed between ’Sleepy Test’ and ’Constructor Initialization’, with a co-occurrence value of [0.96] for the JNose Tool.

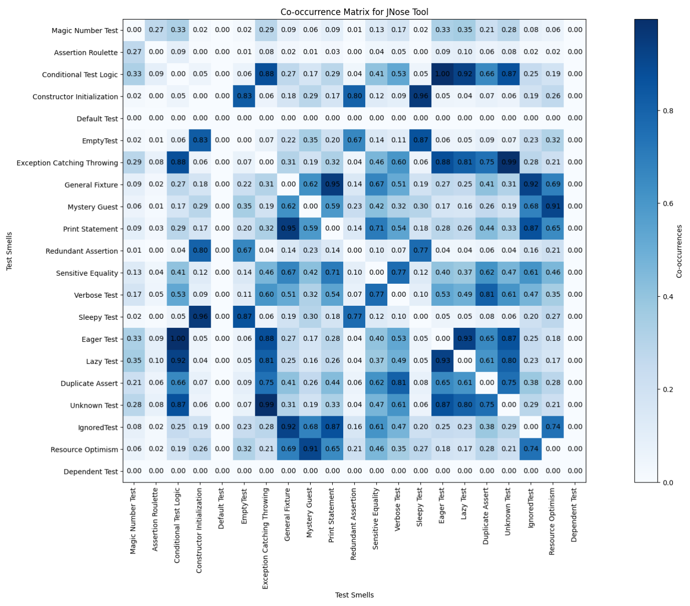


Figure 7. Co-occurrence Matrix for JNose Tool

Conversely for the JNose tool, a pair exposes relationships that are markedly tenuous, as is the case between ’Magic Number Test’ and ’Redundant Assertion’, with a negligible co-occurrence rate of [0.01]. Another pair exhibiting minimal interdependence comprises ’Mystery Guest’ and ’Assertion Roulette’ and, ’Empty Test’ and ’Assertion Roulette’ where the co-occurrence rate stands at [0.01] for both pairs.

Results for the TestSmellDetector Tool as shown in Figure 8, the notable correlation observed in this case is between ’Unknown Test’ and ’Eager Test’ and their co-occurrence value of [0.97].

The pairing of ’Source Optimism’ with ’Mystery Guest’ also has a strong co-occurrence rate of [0.95] with using TestSmellDetector Tool.

Conversely, the matrix unveils relationships that are markedly tenuous, as is the case between ’Magic Number Test’ and ’Redundant Assertion’, ’Magic Number Test’ and ’Sleepy Test’, ’Assertion Roulette’ and ’Empty Test’, ’Empty Test’ and ’Exception Catching Throwing’, ’Empty Test’ and ’Lazy Test’, so on with a negligible co-occurrence rate of [0.01] with using TestSmellDetector Tool.

VI. CONCLUSION AND FUTURE WORK

Testing is currently considered to be an essential process for improving the quality of software. Unfortunately, past literature has shown that test code can often be of low quality and may contain design flaws, also known as test smells. This paper presented a comparison of the results of the most well-known test smell detector tools (JNose and TestSmellDetector) using 500 distinct open-source GitHub projects. These results give us (i) the number of detection of test smells by each tool, (ii) the number of affected test code files by test smells, and (iii) the co-occurrence rate of detected test smells with the mentioned tools.

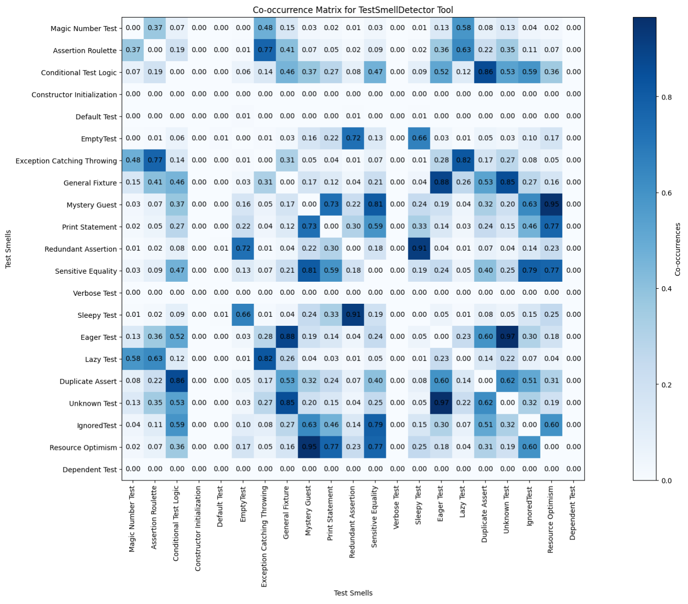


Figure 8. Co-occurrence Matrix for TestSmellDetector Tool

- (i) The 'Assertion Roulette' is the most prevalent smell in the JNose Tool with 41,876 detections. Like 'Assertion Roulette', other common the test smells 'Magic Number Test' with 11264 detections, 'Lazy Test' with 3984 detections, 'Eager Test' with 3692 detections, 'Conditional Test Logic' with 3679 detections, 'Exception Catching Throwing' with 3236 detections, and 'Unknown Test' with 3202 detections. On the other hand, the TestSmellDetector tool has found that the test smells 'Magic Number Test' with 28443 detections and 'Lazy Test' with 16570 detections are the most frequently observed. Furthermore, the test smells 'Exception Catching Throwing' with 13612 detections, 'Assertion Roulette' with 10488 detections, 'General Fixture' with 4274 detections, and 'Eager Test' with 3780 detections are observed in all files.
- (ii) The TestSmellDetector tool detected several files affected by the test smells ('Magic Number Test', 'Assertion Roulette', 'Exception Catching Throwing', 'Eager Test', 'Lazy Test', and 'Unknown Test'), with respective counts of 4222, 2503, 2463, 1126, 1070, and 1030. On the other hand, the JNose tool detected several affected files by 'Assertion Roulette', 'Lazy Test', 'Magic Number Test', 'Exception Catching Throwing', 'Unknown Test', and 'Eager Test' are detected as 3056, 1396, 1364, 969, and 905.
- (iii) The JNose tool showed that there is a strong correlation between the test smells 'Conditional Test Logic' and 'Eager Test', as indicated by a co-occurrence value of [1.00]. Furthermore, the JNose tool reveals a strong relationship between the pairs 'Exception Catching Throwing' and 'Unknown Test', as evidenced by a high co-occurrence rate of [0.99]. In contrast, a high-rated correlation was noticed in this significant relationship between the test smells 'Unknown Test' and 'Eager Test', with a co-occurrence value of [0.97] when using the TestSmellDetector tool. Furthermore, the TestSmellDetector Tool exhibited a combination of 'Source Optimism' and 'Mystery Guest', with a significant co-occurrence rate of [0.95].

As future work, we plan to replicate this study with larger projects, including a more extensive set of test smells. We also plan to implement a new tool to detect test smells and refactor them further. Then, we plan to compare these three tools with larger projects and to show decreased number of detected test smells after refactoring.

REFERENCES

[1] M. Aberdour, "Achieving quality in open-source software," *IEEE Software*, vol. 24, no. 1, pp. 58–64, 2007. DOI: 10.1109/MS.2007.2.

[2] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," ICSE 2014, pp. 72–82, 2014. DOI: 10.1145/2568225.2568278.

[3] T. Virgínio *et al.*, "Jnose: Java test smell detector," SBES '20, pp. 564–569, 2020. DOI: 10.1145/3422392.3422499.

[4] A. Peruma *et al.*, "Tsdetect: An open source test smells detection tool," ESEC/FSE 2020, pp. 1650–1654, 2020. DOI: 10.1145/3368089.3417921.

[5] N. S. Junior, L. Rocha, L. A. Martins, and I. Machado, "A survey on test practitioners' awareness of test smells," 2020. arXiv: 2003.05613.

[6] D. Campos, L. Rocha, and I. Machado, "Developers perception on the severity of test smells: An empirical study," 2021. arXiv: 2107.13902.

[7] D. Spadini, M. Schvarcbacher, A.-M. Oprescu, M. Bruntink, and A. Bacchelli, "Investigating severity thresholds for test smells," MSR '20, pp. 311–321, 2020. DOI: 10.1145/3379597.3387453.

[8] M. Tufano *et al.*, "An empirical investigation into the nature of test smells," pp. 4–15, 2016.

[9] E. Soares *et al.*, "Refactoring test smells: A perspective from open-source developers," SAST '20, pp. 50–59, 2020. DOI: 10.1145/3425174.3425212.

[10] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn, "Revisiting test smells in automatically generated tests: Limitations, pitfalls, and opportunities," pp. 523–533, 2020. DOI: 10.1109/ICSME46990.2020.00056.

[11] G. Bavota, A. Qusef, R. Oliveto, A. D. Lucia, and D. Binkley, "Are test smells really harmful?" *Empirical Software Engineering*, vol. 20, pp. 1052–1094, 2015. DOI: 10.1007/s10664-014-9313-0.

[12] G. Grano, F. Palomba, D. Di Nucci, A. De Lucia, and H. C. Gall, "Scented since the beginning: On the diffuseness of test smells in automatically generated test code," *Journal of Systems and Software*, vol. 156, pp. 312–327, 2019, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2019.07.016.

[13] L. Martins, H. Costa, and I. Machado, "On the diffusion of test smells and their relationship with test code quality of java projects," *Journal of Software: Evolution and Process*, vol. 36, no. 4, e2532, 2024. DOI: https://doi.org/10.1002/smr.2532. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2532.

[14] IEEE Spectrum, "Top programming languages 2024," 2024, [Online]. Available: https://spectrum.ieee.org/top-programming-languages-2024 (visited on 08/21/2024).

Precise Code Fragment Clone Detection

Mariam Arutunian
Center of Advanced Software Technologies
Russian-Armenian University
Yerevan, Armenia
mariam.arutunian@rau.am

Matevos Mehrabyan
Center of Advanced Software Technologies
Russian-Armenian University
Yerevan, Armenia
matevos.mehrabyan@rau.am

Sevak Sargsyan
Center of Advanced Software Technologies
Russian-Armenian University
Yerevan, Armenia
sevak.sargsyan@rau.am

Hayk Aslanyan
Center of Advanced Software Technologies
Russian-Armenian University
Yerevan, Armenia
hayk.aslanyan@rau.am

***Abstract*— Detecting duplicate code fragments referred as "clones", is essential for various aspects of software management, maintenance, and security. This article presents a novel method for detecting code fragment clones, applicable to source and binary code. The method addresses the limitations of existing tools, which often focus on detecting clones of entire functions and are typically specialized for either source or binary code, but not both simultaneously. The developed algorithm analyzes input code fragments against the target project, and outputs all detected fragment clones. For fragment clone detection, it uses program dependence graphs - a data structure unifying data and control flow for the function. In the first step source and binary code are converted to program dependence graph representation. Then unified algorithm is applied for maximal similar subgraphs detection. Code fragments corresponding to detected similar subgraphs are considered as clones. The experimental evaluation of the proposed method demonstrates its effectiveness providing an average 96.9% precision, 92.9% recall for binary code, and 96.5% precision, 93.8% recall for source code.**

Keywords- code clones; program static analysis; binary code; source code.

I. INTRODUCTION

Identifying copied code fragments, referred as fragment clones, are vital for software management, maintenance, and security. It can be applied for several purposes:

1. Software plagiarism detection: identifying copied code helps ensure originality and protect intellectual property,
2. Malware detection and classification: researchers can identify new malware variants by finding similar code patterns of known malicious software fragments,
3. Finding known vulnerabilities and avoiding bug propagation: Sometimes, code fragments containing bugs and vulnerabilities are also copied, making the detection of these fragments crucial for preventing the spread of bugs.

Beyond these specific applications, identifying and managing code clones improves overall software quality and reduces maintenance costs. Code clones can arise for a variety of reasons. For instance, they can occur when software developers copy-paste existing code fragments into their projects with or without modifications. Studies [1] show that about 20% of code is duplicated in software packages. In binary code, compiler optimizations like inlining, and transformations can also create clones.

Modern software projects highly use third-party packages and libraries. A 2024 report by Synopsys [2] revealed that over 96% of commercial software packages incorporate open-source code. Another study of 7,800 open-source projects has shown that 44% of them have at least one pair of identical code fragments [3]. These studies reveal the extensive use of code duplication in software development.

Despite the variety of code clone detection methods and tools, only a few can detect clones of fragments rather than whole functions. Besides, existing tools are focused either on source or binary code clone detection. There is no unified approach to detect both of them.

We propose a novel approach for accurate source and binary code fragments' clones' detection. For accuracy Program Dependence Graphs (PDGs) are utilized, which capture most of the software semantics and robust to code changes. Code clones are identified as maximum similar subgraphs for corresponding source and binary code. The core of the developed tools is the same for the source and binary code clones' detection, where the PDG creation parts are code specific. We consider code fragments as a sequence of instructions for binary or source code. A fragment can correspond to a function, basic blocks, or sequences of instructions in a function. Two code fragments are considered clones if they are similar or identical. Section II gives more strict definitions of both binary and source code fragment clones. The proposed method is implemented as a tool named Fragment Clone Detector (FCD) that takes as input a code fragment, a project, and a percentage of similarity. The tool then outputs all fragments from the target project that are clones of the given fragment with the given percentage of similarity.

In addition to evaluating the quality of the implemented method, we have designed and implemented a testing system, which generates tests, based on real-world projects. Then it executes FCD and calculates precision, recall, and Root Mean Square Error (RMSE) for it. The rest of the paper is organized as follows: Section II defines code clone types for binary and source code and describes PDG. Section III explores existing research in the field. Sections IV and V detail the proposed approach for detecting code fragment clones. The testing system structure is presented in Section VI. Section VII of the paper presents the results of the experimental evaluation. The final section concludes the paper.

II. BACKGROUND

In this section main ideas used in the work are introduced: code clone types and PDG. Both source and binary code clone types are defined in the Subsection A. And the Subsection B will cover the description of the PDG, its components, and its uses.

A. *Code clone types*

It is accepted [4] that source code clones have four types. While the definition of source code clones is well-established, the definition of binary code clones has minor differences due to its specifics. The definition of source code clone types:

- **Type 1**: Two source code fragments that are identical except for variations in whitespaces and comments,
- **Type 2**: Two source code fragments that can differ by identifiers, literals, and types. This type also includes Type 1 clones,
- **Type 3**: Two source code fragments with additions, deletions, or modifications of instructions. Includes Type 2 clones too. Type 3 clones are also referred to as non-exact clones,
- **Type 4**: Two source fragments that perform the same calculations but use different instructions. Type 4 clones are also referred as semantic clones.

TABLE I. EXAMPLE OF SOURCE CODE CLONE TYPES.

Original code	*Type-1*

float sum = 0.0;
for (int i = 0; i<n; i++){
sum = sum + F[i];
}
	float sum = 0.0; // Comment
	for (int i = 0; i<n; i++){
	___ sum = sum + F[i];
	}

Type-2	*Type-3*

int sum1 = 0; // Comment
for (int i = 0; i<n; i++){
___ sum1 = sum1 + F[i];
}
	int prod = 1; // Comment
	for (int i = 0; i<n; i++) {
	___ prod = prod * F[i];
	}

Type-4	

int factorial_rec (int n) {
if (n <= 1) {
return 1;
} else {
return n * factorial_rec (n - 1);
}
}
	int factorial_iterative(int n) {
	int result = 1;
	for (int i = 1; i <= n; ++i) {
	result *= i;
	}
	return result;
	}

As there are no comments and whitespaces in binary code, a slightly different definition for binary code clone types is used. Binary code clone types [5] are:

- **Type 1**: Two identical binary code fragments.
- **Type 2**: Two binary code fragments that can differ by registers, literals, and operand sizes. This type also includes Type 1 clones.
- **Type 3**: Two binary code fragments with additions, deletions, or modifications of instructions. Includes Type 2 clones too. Type 3 clones are also called non-exact clones.
- **Type 4**: Two binary fragments that have the same calculations but use different instructions.

TABLE I and TABLE II present examples of source and binary clone types, respectively. In both tables, original code and all clone types are presented.

TABLE II. EXAMPLE OF BINARY CODE CLONE TYPES.

Original code	*BinType-1*
mov [ebp+var_1], 5 mov eax, [ebp+var_1] iadd eax, [ebp+var_4]	mov [ebp+var_1], 5 mov eax, [ebp+var_1] iadd eax, [ebp+var_4]
BinType-2	*BinType-3*
mov [ebp+var_1], 10 mov ecx, [ebp+var_1] iadd ecx, [ebp+var_4]	~~mov [ebp+var_1], 10~~ mov ecx, [ebp+var_1] iadd ecx, [ebp+var_4]
BinType-4	
factorial_rec: pushq %rbp movq %rsp, %rbp subq $16, %rsp movl %edi, -4(%rbp) cmpl $1, -4(%rbp) jg .L2 movl $1, %eax jmp .L3 .L2: movl -4(%rbp), %eax subl $1, %eax movl %eax, %edi call factorial_rec imull -4(%rbp), %eax .L3: ret	factorial_O3: movl $1, %eax cmpl $1, %edi jle .L1 .p2align 4,,10 .p2align 3 .L2: movl %edi, %edx subl $1, %edi imull %edx, %eax cmpl $1, %edi jne .L2 .L1: ret

B. *Program dependence graph*

PDG is a directed graph that combines data and control dependencies. The vertices of PDGs are program statements and the edges are data and control dependencies between them. PDGs are used in various applications, such as compiler optimizations, program analysis, and software engineering tasks (like refactoring, debugging). As PDG makes explicit both the data and control dependencies between operations of the program, that makes it useful for understanding complex program behaviors and improving software quality and efficiency.

III. RELATED WORK

There are many works related to code clone detection. However, most of them can find only clones of a whole function. Our method deals with every fragment of code inside a function. Obviously, it also finds function clones.

Copyright (c) IARIA, 2024. ISBN: 978-1-68558-199-2

Code clone detection techniques are divided into the following groups: text-based, token-based, tree-based, metrics-based, graph-based, and machine-learning based. Also, there are numerous hybrid methods combining several techniques for clone detection.

In the case of a text-based approach [6] [7] [8] [9], two code fragments are compared in the form of text/strings. It only finds Type 1 clones. In the case of a token-based approach [10] [11] [12] [13], the entire code is transformed into a sequence of tokens. It is more robust against code changes than text-based techniques, which allows it to find Type 1 and Type 2 clones.

Tree-based approaches [14] [15] [16] [17] use parse trees or Abstract Syntax Trees (AST) of the analyzable code. Then, similar subtrees are detected using tree-matching algorithms. It can find all three types of clones. But as a rule, this approach suffers in precision for Type 3 clone detection, because instructions difference strongly changes the underlying tree structure.

In the case of a metrics-based approach [18] [19] [20] [21], different types of metrics are calculated for code fragments. Then these metrics are compared to find similar code fragments. Usually, for calculating different types of metrics the code is converted into some graph representation, such as AST or PDG. This approach suffers in precision and produces many false positives.

In the case of a graph-based approach [22] [23] [24] [25], a PDG or just a Control Flow Graph (CFG) is generated from the code. Then maximal isomorphic or similar (it may be defined differently for each method) subgraphs are searched. PDG-based approaches are robust to the insertion and deletion of code, reordered instructions, intertwined and non-contiguous code. However, they have higher asymptotic complexity and may not be scalable.

In the case of machine learning-based techniques [26] [27] [28] [29], the focus is on training models to classify or cluster similar code fragments. Patterns are learned from a dataset containing examples of both similar and dissimilar codes. Learning algorithms are well-suited for code clone detection tasks because they can learn and identify complex patterns. However, learning-based techniques need large and clean datasets of code clones to work properly, but these are not available for all programming languages. Many methods rely on existing code clone detection tools to gather data for machine learning, but these tools are often unreliable and prone to errors.

In addition, there are hybrid methods, which combine several techniques for clone detection. Some examples are text-based and tree-based [30], token-based and tree-based [31], metric-based and graph-based [32], tree-based and learning-based [33] [34], etc. They addresses the challenge of individual methods.

Thus, each of the discussed techniques has its advantages and disadvantages. An appropriate method can be selected based on the problem that needs to be solved.

IV. Code Fragment Clone Detection

The developed algorithm takes a code fragment, a project, and a percentage of similarity as its input. It analyzes all the functions within the project and identifies clones of the specified fragment. The identified clones must have at least the specified percentage of similarity. It is important to note that we assume the provided code fragment is within a single function. Figure 1 provides architecture of the proposed method. It has two primary components: the construction of PDGs and the matching of these graphs.

A. Construction of PDGs

PDGs are constructed for the specified fragment and all functions of the target program. Vertices of the PDG represent instructions of Intermediate Representation (IR), and edges are constructed based on data and control dependencies between them. The construction process of PDGs varies for binary and source code as the code representation differs, and the specific details are outlined in the implementation section. For the vertices of the PDG, instead of "original form" instructions of IR are used, as it simplifies and standardizes the code, allowing tools to be reused across different languages and architectures.

To construct the PDG for the specified fragment, the PDG for the entire function containing the fragment is first created. Then, a subgraph corresponding to the specified fragment is extracted to serve as the final PDG of the fragment. Basically, it is the smallest induced subgraph of the entire function's PDG that includes all instructions of the specified fragment. For simplicity, we will call it a fragment's PDG. The constructed graphs are then utilized in the next step, where instructions from the specified fragments are matched against all instructions within the functions throughout the entire project.

B. Graphs' matching

Once the PDGs are constructed, the algorithm starts matching the vertices of the fragment's PDG with the vertices

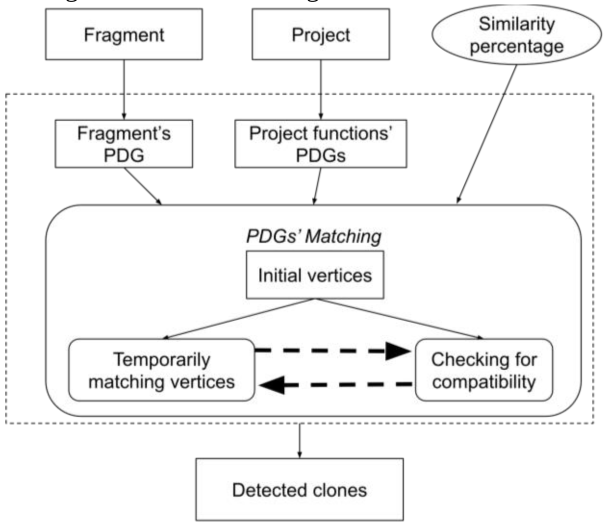


Figure 1. Architecture of the method.

of each function's PDG. It is important to note that within a single function's PDG, there can be detected multiple matches indicating the existence of several clones of the specified fragment within that function.

Similarity percentage for the detected fragment clone is calculated by the following formula:

$$similarity = \frac{matched\ common\ vertices\ count}{fragment\ PDG's\ vertices\ count} * 100\%$$

The matching algorithm between the fragment's and a function's PDGs involves the following phases:

1. Construction of the set of initial matched vertex pairs,
2. Iterative expansion of matched vertex pairs.

The first vertex of each pair is from the fragment's PDG and the second is from the function's PDG. Corresponding instructions for the vertices of each pair have the same operation code. The algorithm then selects one of the unconsidered pairs from the set to start expanding process. From the selected pair, the algorithm temporarily matches previously unmatched pairs of vertices using specific subroutines. These subroutines match vertices based on their features and adjacent edges, ensuring that vertices with identical operation codes are paired. If the temporarily matched vertices meet all specified conditions, they are finally matched. This process is repeated for all vertices that are not matched yet. The expanding phase stops when no new temporarily matched pairs can be identified. The output of this process is the list of sets, where each set contains matched vertex pairs. Further details will be provided later in the text. For simplicity, we will be using some notations that are described below:

- $function\_PDG$ - PDG of the given function,
- $fragment\_PDG$ - PDG of the given fragment,
- $initial\_pairs$ - the set of initial pairs of vertices $(v, v*)$, where $v \in fragment\_PDG$, $v* \in function\_PDG$,
- $temporarily\_matched\_pairs$ - the set of pairs $(v, v*)$, where $v \in fragment\_PDG$, $v* \in function\_PDG$, which are temporarily matched, but need to pass several checks before final matching,
- $matched\_pairs$ - the set of pairs $(v, v*)$, where $v \in fragment\_PDG$, $v* \in function\_PDG$, which are finally matched,
- $matched(G)$ – the set of finally matched vertices of graph G,
- $incompatible\_pairs$ - the set of $(v, v*)$ incompatible pairs of vertices, where $v \in fragment\_PDG$, $v* \in function\_PDG$,
- $opcode(v)$ - is an operation code corresponding to a vertex v,
- $pred\_ctrl(v)$/ $succ\_ctrl(v)$ - the set of predecessor / successor vertices of v by control dependence,
- $pred\_data(v)$ / $succ\_data(v)$ - the set of predecessor / successor vertices of v by data dependence,
- $bb(v)$ - the list of vertices in the same basic block as vertex v,
- $pred\_bb(v)$/ $succ\_bb(v)$ - the list of vertices in the predecessor / successor basic blocks of vertex v.

1) Construction of the set of initial matched vertex pairs.

The phase of selecting initial pairs of vertices aims to find such pairs of vertices from $fragment\_PDG$ and $function\_PDG$, which are likely to be matched together. Afterward, they are used as a starting point for the graphs' matching process. The amount of such vertices should be as small as possible for efficiency. To achieve this, the initial vertices in PDGs are selected using various subroutines, chosen based on their effectiveness during the experimental evaluation.

The first subroutine selects all vertices $(v, v*)$with no incoming edges in both PDGs, where $v \in fragment\_PDG$ and $v* \in function\_PDG$. These vertices typically correspond to the first instructions of the specified fragment and the function. Then, from the obtained sets of vertices, the subroutine constructs all possible combinations of pairs, where the corresponding instructions have the same operation code, and adds them to $initial\_pairs$.

The second subroutine collects vertices with the maximum incoming data dependencies in $fragment\_PDG$. Then it collects vertices from $function\_PDG$ that have an equal or greater number of incoming data dependencies. Like the first subroutine, this one also creates all possible combinations of pairs from the obtained sets (ensuring that the corresponding instructions have the same operation code) and adds them to $initial\_pairs$ set.

The third subroutine identifies all the instructions from the code fragment that have the maximum number of corresponding IR instructions. It then selects instructions from the function with the same number of corresponding IR instructions. Subsequently, the subroutine collects vertices corresponding to the first IR instructions of the mentioned instructions. Finally, similar to other subroutines, it generates all possible combinations of pairs from the obtained sets, ensuring that the corresponding instructions have the same operation code, and adds them to $initial\_pairs$ set.

2) Iterative expansion of matched vertex pairs.

The expanding phase temporarily matches unconsidered vertices from $fragment\_PDG$ and the $function\_PDG$. Next, it checks temporarily matched vertices for conditions. If a pair passes conditions checking, it is placed to $matched\_pairs$ list, otherwise it is placed to $incompatible\_pairs$ list. Expanding starts from $initial\_pairs$ and iteratively matches vertices until no temporarily matched vertices can be detected.

a) Temporarily matching.

The matching algorithm involves five temporary matching subroutines. The results obtained from these subroutines are then checked against several conditions (described in the next section), and some of the temporarily matched pairs may be filtered out. The matching process is complete when no new pairs of vertices are temporarily matched, meaning that the algorithm has exhausted all possible matches between the fragment's PDG and the function's PDG.

For each pair of vertices $(u, u*)$ temporary matching is allowed if $opcode(u) == opcode(u*)$, the size of $pred\_ctrl(u)$ equals to the size of $pred\_ctrl(u*)$, and the size of $succ\_ctrl(u)$ equals to the size of $succ\_ctrl(u*)$, where $(u, u*) \notin matched\_pairs$ and $(u, u*) \notin incompatible\_pairs$.

In all subroutines, two vertices $(v, v*)$ can be temporarily matched if $(v, v*) \notin matched\_pairs$, $(v, v*) \notin incompatible\_pairs$ and corresponding instructions have the same opcode. The subroutines are applied in the specific order, and if one of them temporarily matches a pair, the others will not be applied. At the beginning $temporarily\_matched\_pairs \leftarrow \emptyset$. Below are descriptions of five temporarily matching subroutines:

1. For each pair $(v, v*) \in matched\_pairs$ temporarily match vertices $(u, u*)$, where $u \in pred\_ctrl(v)$ and $u* \in pred\_ctrl(v*)$ sets, and add them to $temporarily\_matched\_pairs$ if temporary matching is allowed. Do the same for vertices $(u, u*)$ from $succ\_ctrl(v)$ and $succ\_ctrl(v*)$ sets. If $temporarily\_matched\_pairs$ is not empty, go to conditions checking phase.
2. For each pair $(v, v*) \in matched\_pairs$ temporarily match vertices $(u, u*)$, where $u \in bb(v)$ and $u* \in bb(v*)$ lists, and add them to $temporarily\_matched\_pairs$ if temporary matching is allowed. If $temporarily\_matched\_pairs$ is not empty, go to conditions checking phase.
3. For each pair $(v, v*) \in matched\_pairs$ temporarily match vertices$(u, u*)$, where $u \in pred\_bb(v)$ and $u* \in pred\_bb(v*)$ lists, and add them to $temporarily\_matched\_pairs$ if temporary matching is allowed. Do the same for vertices from $succ\_bb(v)$ and $succ\_bb(v*)$. If $temporarily\_matched\_pairs$ is not empty, go to conditions checking phase.
4. For each pair $(v, v*) \in matched\_pairs$ temporarily match vertices $(u, u*)$, where $u \in pred\_data(v)$ and $u* \in pred\_data(v*)$ sets, and add to $temporarily\_matched\_pairs$ if temporary matching is allowed. Do the same for vertices from $succ\_data(v)$ and $succ\_data(v*)$ sets. If $temporarily\_matched\_pairs$ is not empty, go to conditions checking phase.
5. Temporarily match pairs$(u, u*) \in initial_pairs$, and add to $temporarily\_matched\_pairs$, if $(u, u*) \notin matched\_pairs$ and $(u, u*) \notin incompatible\_pairs$.

b) Conditions checking.

The next stage is the checking of temporarily matched pairs. After each iteration of temporarily matching, each pair $(v, v*) \in temporarily\_matched\_pairs$ is checked for conditions. If the pair satisfies all conditions, it is moved to $matched\_pairs$, otherwise to $incompatible\_pairs$. The conditions are described below:

1. $pred\_condition(v, v*)$ returns $false$ if $\exists p \in pred\_ctrl(v)$ where $p \in matched(fragment\_PDG)$ and $\nexists p* \in function\_PDG$ such that $p* \in pred\_ctrl(v*)$ and $(p, p*) \in matched\_pairs$, otherwise returns $true$.
2. $succ\_condition(v, v*)$ returns $false$ if $\exists s \in succ\_ctrl(v)$ where $s \in matched(fragment\_PDG)$ and $\nexists s* \in function\_PDG$ such that $s* \in succ\_ctrl(v*)$ and $(s, s*) \in matched\_pairs$, otherwise returns $true$.

V. Implementation

We implemented the proposed method in a tool called FCD. It is a command-line tool, that receives the following inputs:

1. The project path and the function name containing the code fragment to be analyzed,
2. The boundaries of the code fragment: the start and end line numbers for source code, the start and end memory relative addresses for binary code,
3. The project in which to search for clones of the specified fragment,
4. An optional minimum similarity percentage parameter, which is used to filter out clones that are less similar than the specified value. This parameter belongs to (0, 100], and has a default value of 90. The 90% similarity is chosen to detect highly similar code fragments, which is more of the interest to developers.

The process of PDG's generation differs for source and binary code, however, the matching parts are the same. For source code PDG's generation FCD uses LLVM intermediate representation [35]. To get PDGs for source code a new pass is added in LLVM, which uses control flow information, use-def chains and alias analysis. For binary code PDGs generation FCD uses REIL [36] intermediate representation. At first, it uses IDA Pro [37] disassembler to restore assembler and control flow graphs. Then the obtained assembler is translated to the REIL intermediate language using Binnavi [38]. Lastly, it uses Binside [39] to generate PDGs, which was developed by our team previously.

Code fragment clone detection algorithm is implemented in C++ language. The output of the tool consists of a set of JSON files containing information about the detected clones. This information includes functions' names corresponding to matched fragments, similarity percentage, all pairs of matched instructions, and other relevant details.

VI. Testing System

To evaluate FCD algorithm, we have designed and implemented a testing system, which generates tests, executes FCD and calculates precision, recall, and Root Mean Square Error (RMSE) to assess their effectiveness. Test generation is done using PDGs of real-world projects. For each PDG, it creates a duplicate, removes some vertices, and considers it as fragment's PDG. It randomly selects a basic block and removes corresponding vertices until the desired similarity percentage is reached. After removing a vertex, its predecessor vertices are connected with the successor ones. If all vertices in the chosen basic block are removed and the provided similarity is still not met, the system randomly selects a new basic block and starts removing consecutive vertices from that block. This process continues until the required similarity percentage is not met.

It then runs the FCD algorithm on generated PDGs' pairs and compares the resulting similarity percentage with the one specified to testing system. Ideally, the similarity percentages of the created PDGs' pairs by the testing system should match with the results from the FCD algorithm. The testing system saves information about the correspondence of the original and the generated PDG vertices, which is used to calculate precision, recall, and RMSE.

VII. Results

FCD is tested with the discussed testing system on projects OpenSSL, JasPer, c-ares, Rsync. Tables Table *III* and Table *IV* present the results of source and binary code clone detection, respectively. The results are averaged across similarity thresholds 100%, 90%, 80%, and 70%.

The tool achieves perfect results when generated clones are 100% similar. Furthermore, FCD consistently demonstrated high accuracy across lower thresholds, as reflected in the averaged results in the tables. However, binary code clone detection's speed is slow compared to source code clone's detection time, as for binary bigger PDG's are generated.

TABLE III. Source Code Clone Results

Project	***C/C++ code lines***	***Precision***	***Recall***	***RMSE***	***FCD speed***
c-ares 1.15.0	61087	97.5	95.2	6.1	0m 0.29s
jasper 1.900.1	28279	95.4	93	6	0m 15s
openssl 1.0.2t	310922	97	95.1	7.7	0m 2s
rsync 3.1.3	44832	96	91.9	10.7	0m 26s

On average, FCD has 96.5% precision, 93.8% recall and 7.6% RMSE for source code. And on average, FCD has 96.9% precision, 92.9% recall and 5.4% RMSE for binary code. Despite high rates of the tool's precision and recall, there are still certain cases that the tool may not detect correctly. This occurs when the copied code is modified by adding a new instruction between each original instruction, i.e., one instruction from the original code, followed by one new instruction, then another from the original, and so on. However, if the copied code is modified in such a way that a whole basic block is added the tool identifies it correctly.

TABLE IV. Binary Code Clone Results

Project	***Size of the binary***	***Architecture***	***Precision***	***Recall***	***RMSE***	***FCD speed***
libcares 2.3.0 (c-ares 1.15.0)	86 KiB	x86-64	98.9	95.6	4.6	0m 41s
libcares 2.3.0 (c-ares 1.15.0)	96 KiB	x86	97.9	93.4	5.5	0m 43s
libcares 2.3.0 (c-ares 1.15.0)	146 KiB	ARM	98.9	95.6	4.6	0m 49s
jasper 1.900.1	1.5 MiB	x86-64	96	92.1	5.4	3m 5s
jasper 1.900.1	368 KiB	x86	95	90	6.5	2m 1s
jasper 1.900.1	478 KiB	ARM	94.1	89.8	6.1	2m 8s
openssl 1.0.2t	536 KiB	x86-64	99.9	98.1	3.8	1m 10s
openssl 1.0.2t	507 KiB	x86	98.8	95.8	3.9	0m 57s
openssl 1.0.2t	634 KiB	ARM	97.9	95.6	4.4	1m 25s
rsync 1.3.2	1.7 MiB	x86-64	96	91	6.6	3m 34s
rsync 1.3.2	1.6 MiB	x86	94.9	88.9	6.7	3m 21s
rsync 1.3.2	1.8 MiB	ARM	94.1	88.8	7.4	3m 58

The tool is not compared with the related tools as there is no common benchmark for evaluation. While there are some benchmarks available for C/C++ languages, they include only Type-4 clones, which our tool does not detect. Additionally, each tool uses its own method to calculate similarity levels, which results in inconsistent evaluations of the same code fragments.

VIII. Conclusion

The study proposes a novel technique to identify duplicated code fragments. It overcomes limitations of existing clone detection tools, which typically target only full functions and specialize in either source or binary code analysis. Experimental evaluation on real-world software projects demonstrates the high precision and effectiveness of the proposed clone detection approach for source and binary code. As conclusion we can clearly see that PDG captures enough information for source and binary code to enable accurate clone detection for both cases. Moreover, a unified algorithm can be used for maximal similar subgraphs detection in both cases.

ACKNOWLEDGMENT

The work was supported by the Science Committee of RA, in the frames of the research project 21SCG-1B003.

REFERENCES

[1] C. K. Roy and J. R. Cordy, "An empirical study of function clones in open source software systems," in *Proceedings of the 15th Working Conference on Reverse Engineering*, 2008, pp. 81-90.

[2] "Synopsis," 2024 Open Source Security and Risk Analysis Report, [Online]. Available: https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2024.pdf. [retrieved: 08.2024].

[3] R. Koschke and S. Bazrafshan, "Software-clone rates in open-source programs written in c or c++," *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 3, pp. 1-7, 2016.

[4] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470-495, 2009.

[5] H. K. Aslanyan, "Effective and Accurate Binary Clone Detection," *Mathematical Problems of Computer Science*, vol. 48, pp. 64-73, 2017.

[6] D. Tukaram and U. Maheswari B, "Design and development of software tool for code clone search, detection, and analysis," in *2019 3rd International conference on Electronics, Communication and Aerospace Technology (ICECA), pp. 1002-1006*, 2019.

[7] C. Ragkhitwetsagul and J. Krinke, "Using compilation/decompilation to enhance clone detection," in *2017 IEEE 11th International Workshop on Software Clones (IWSC), IEEE, pp. 1–7*, 2017.

[8] T. Kamiya, "An execution-semantic and content-and-context-based code-clone detection and analysis," in *2015 IEEE 9th International Workshop on Software Clones, IWSC 2015 - Proceedings, pp. 1–7*, 2015.

[9] J. Chen, M. H. Alalfi, T. R. Dean, and Y. Zou, "Detecting Android malware using clone detection," *Journal of Computer Science and Technology*, vol. 30, pp. 942-956, 2015.

[10] L. Yang, Y. Ren, J. Guan, B. Li, and J. Ma, "FastDCF: a partial index based distributed and scalable near-miss code clone detection," in *Parallel and Distributed Computing, Applications and Technologies: 22nd International Conference, PDCAT 2021, pp. 210-222*, Guangzhou, China, 2021.

[11] Y.-L. Hung and S. Takada, "CPPCD: a token-based approach to detecting potential clones," in *IEEE 14th International Workshop on Software Clones (IWSC), IEEE, pp. 26–32*, 2020.

[12] Y. Wu et al., "SCDetector: software functional clone detection based on semantic tokens analysis," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, , pp. 821–833*, New York, NY, USA: ACM, 2020.

[13] K. E. Rajakumari, "Comparison of token-based code clone method with pattern mining technique and traditional," in *Proceedings of 2019 3rd IEEE International Conference on Electrical, Computer and Communication Technologies, ICECCT 2019, pp. 1–6*, 2019.

[14] Y. Yu, Z. Huang, and G. Shen, "ASTENS-BWA: searching partial syntactic similar regions between source code fragments via," *Science of Computer Programming*, vol. 222, p. 102839, 2022.

[15] W. Wen et. al., "Cross-project software defect prediction based on class code similarity," *IEEE Access*, vol. 10, p. 105485–105495, 2022.

[16] Y. Gao et al., "TECCD: A Tree Embedding Approach for Code Clone Detection," in *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, pp. 145–156*, 2019.

[17] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD : Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering*, 2007.

[18] S. Parsa, M. Zakeri-Nasrabadi, and M. Ekht, "Method name recommendation based on source code metrics," *Journal of Computer Languages*, vol. 74, no. 101177, pp. 1-13, 2023.

[19] H. Jin, Z. Cui, S. Liu, and L. Zheng, "Improving code clone detection accuracy and efficiency based on code complexity analysis," in *n 2022 9th International Conference on Dependable Systems and Their Applications (DSA), IEEE, pp. 64–72*, 2022.

[20] K. W. Nafi, T. S. Kar, B. Roy, C. K. Roy, and K. A. Schneider, "CLCDSA: cross language code clone detection using syntactical features and API documentation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, pp. 1026–1037*, 2019.

[21] M. Sudhamani and L. Rangarajan, "Code similarity detection through control statement and program features," *Expert Systems with Applications*, vol. 132, no. 15, pp. 63-75, 2019.

[22] W. Wen et al., "Cross-project software defect prediction based on class code similarity," *IEEE Access*, vol. 10, pp. 105485-105495, 2022.

[23] H. K. Aslanyan, S. F. Kurmangaleev, V. G. Vardanya, M. S. Arutunian, and S. S. Sargsyan, "Platform-independent and scalable tool for binary code clone detection," in *Proceedings of the Institute for System Programming of the RAS, pp. 215-226*, 2016.

[24] Z. Xue et al., "SEED: semantic graph based deep detection for type-4 clones," in *International Conference on Software and Software Reuse, pp. 120–137*, 2022.

[25] N. Mehrotra et al., "Modeling functional similarity in source code with graph-based Siamese networks," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3771-3789, 2022.

[26] A. Zhang et al., "Learn to align: a code alignment network for code clone detection," in *2021 28th Asia-Pacific Software Engineering Conference (APSEC), pp. 1-11*, 2021.

[27] N. D. Q. Bui, Y. Yu, and L. Jiang, "InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 1186-1197*, 2021.

[28] Y. Li, C. Yu, and Y. Cui, "TPCaps: a framework for code clone detection and localization based on improved CapsNet," *Applied Intelligence*, vol. 53, p. 16594–16605, 2022.

[29] S. Patel and R. Sinha, "Combining holistic source code representation with siamese neural networks for detecting code clones," in *IFIP International Conference on Testing Software and Systems, pp. 148–159*, 2022.

[30] A. Schafer, W. Amme, and T. S. Heinze, "Stubber: compiling source code into bytecode without dependencies for Java code clone detection," in *2021 IEEE 15th International Workshop on Software Clones (IWSC), IEEE, pp. 29-35*, Oct. 2021.

[31] W. Wang, Z. Deng, Y. Xue, and Y. Xu, "CCStokener: Fast yet accurate code clone detection with semantic token," *Journal of Systems and Software*, vol. 199, p. 111618, May 2023.

[32] H. Aslanyan et al., "Scalable Framework for Accurate Binary Code Comparison," in *2017 Ivannikov ISPRAS Open Conference (ISPRAS), pp. 34-38*, 2017.

[33] A. Zhang, L. Fang, C. Ge, P. Li, and Z. Liu, "Efficient transformer with code token learner for code clone detection," *Journal of Systems and Software*, vol. 197, p. 111557, Mar. 2023.

[34] Y. Wu, S. Feng, D. Zou, and H. Jin, "Detecting semantic code clones by building AST-based Markov chains model," in *37th IEEE/ACM*

International Conference on Automated Software Engineering, pp. 1-13, New York, NY, USA, Oct. 2022.

[35] "The LLVM Compiler Infrastructure," [Online]. Available: www.llvm.org. [retrieved: 08.2024].

[36] "REIL - The Reverse Engineering Intermediate Language. Zynamics," [Online]. Available: https://www.zynamics.com/binnavi/manual/html/reil_language.htm. [retrieved: 08.2024].

[37] "IDA Pro," [Online]. Available: https://hex-rays.com/ida-pro/. [retrieved: 08.2024].

[38] "BinNavi," [Online]. Available: https://www.zynamics.com/binnavi.html. [retrieved: 08.2024].

[39] H. Aslanyan, M. Arutunian, G. Keropyan, S. Kurmangaleev, and V. Vardanyan, "BinSide : Static Analysis Framework for Defects Detection in Binary Code," in *2020 Ivannikov Memorial Workshop (IVMEM), pp. 9-14*, Orel, Russia, 2020.

Addressing EvoSuite’s Limitations: Method-Specific Test Case Generation and Execution in Controlled Environments

Carlos Galindo, Manuel Gregorio, Josep Silva
Valencian Research Institute for Artificial Intelligence
Universitat Politècnica de València
Valencia, Spain
e-mail: {carlosgalindo,magre1,josilga}@upv.es

***Abstract*— Unit testing is crucial for ensuring software quality and reliability. Although recent advancements in artificial intelligence, particularly Large Language Models (LLMs), offer promise for automating unit test generation, they often struggle with compilation due to an insufficient understanding of specific code rules and execution errors, primarily caused by incorrect assertions. This paper focuses on EvoSuite, a leading state-of-the-art Search-Based Software Testing (SBST) tool that originated in academic research and has proven to be a more reliable alternative for generating unit tests, particularly in Java. EvoSuite excels by directly targeting code coverage and optimizing test generation based on actual program behavior, overcoming many challenges LLMs face. We share our experiences and challenges with EvoSuite across various projects, which have provided valuable insights for its subsequent application in ASys, a system for automatically evaluating Java code. The study explores challenges such as generating tests for overloaded methods and running tests across different environments. We also discuss solutions for these challenges, including method-specific test generation strategies and ensuring test execution compatibility. Our findings highlight the limitations and potential improvements for EvoSuite, offering valuable insights for developers and researchers aiming to enhance automated unit test generation in their projects.**

Keywords- EvoSuite; automated test unit generation.

I. INTRODUCTION

Unit tests are a type of software testing that focuses on verifying the functionality of the smallest unit of a program, typically a single function or method. These tests are fundamental in the software development process to ensure the quality and reliability of systems. However, writing unit tests can be complex and time-consuming, especially as program complexity increases. With the advancement of Artificial Intelligence (AI), particularly Large Language Models (LLMs), new opportunities have emerged for automating the generation of unit tests. Recent studies have explored using ChatGPT [1] for this purpose, but the results have shown that the generated tests often have numerous compilation errors, mainly because the tool lacks a deep understanding of specific code rules, such as access restrictions and the proper use of abstract classes, and execution errors, primarily caused by incorrect assertions due to an inadequate grasp of the focal method's intention [2]. Tools like ChatTester [2] and ChatUnitTest [3] have been developed to address these limitations, improving the generated tests' accuracy. ChatUnitTest achieves this by integrating with the ChatGPT API, albeit at an additional cost.

Despite these advancements in AI, Search-Based Software Testing (SBST) techniques [3] remain the most effective solution for generating unit tests in Java. These techniques, used by various tools, have demonstrated superior results compared to LLMs, due to their specialized focus on testing [4]. One of the most powerful and extended techniques is EvoSuite [5], initially developed as an academic research tool to advance automated unit test generation techniques. EvoSuite has excelled in competitions such as the SBST Tool Competition 2022 [6] and the SBFT Tool Competition 2023 [7], demonstrating its effectiveness and obtaining the highest overall mark despite challenges related to usability and inherent limitations of the Java language [8]. Due to its open-source licensing, EvoSuite has not only become a cornerstone in academic research, where its testing architecture has been widely adopted and extended in various projects, but it has also been tested and applied in industrial contexts. This includes experiments on large-scale open-source projects and even some industrial systems, confirming its potential in practical applications [9]. While these industrial applications demonstrate the tool's versatility, they also highlight challenges in scaling up to the complexity of real-world systems, an area where continued research and development are essential.

Nevertheless, EvoSuite has its own issues. Despite being the leading tool in its field and having proven that individual developers may not be able to find more faults than EvoSuite [10], it faces challenges that reflect broader issues within automated test generation tools. For instance, while achieving a completely bug-free software might be unrealistic, the focus remains on identifying and mitigating specific challenges that can hinder fault detection. Studies, such as [11], have pointed out that automatically generated tests often struggle with issues like incorrect oracles and unexpected exceptions, which can significantly impact their effectiveness. Moreover, as highlighted in [12], although high code coverage is correlated with an increased likelihood of fault detection, it is not a definitive guarantee. In practice, this means that while EvoSuite can achieve high coverage, certain types of faults, particularly those related to more complex software behaviors, might still go undetected. The study shown in [13] further elaborates on this, indicating that code coverage serves as a moderate indicator of fault detection effectiveness, with its strength varying depending on the testing profile. Similarly, [14] discusses the link between coverage and software reliability, supporting the notion that focusing on coverage is still a practical approach, though not without its limitations.

Given these findings, while recognizing the limitations, our work continues to prioritize coverage in the use of EvoSuite, as it remains a practical and widely accepted measure of test suite effectiveness in detecting faults. However, we acknowledge that the ultimate goal is not solely to achieve high coverage but also to ensure that the generated tests effectively uncover real and critical bugs in the software. This dual focus on coverage and fault detection is crucial for improving the reliability of automated testing tools like EvoSuite. By refining these tools to better handle complex scenarios and enhance the accuracy of test oracles, we strive to contribute to the ongoing efforts in advancing automated testing practices, ultimately aiming for more dependable and effective software testing outcomes.

The contributions of this paper include a detailed exploration of the practical application of EvoSuite in ASys [15], a system designed to grade Java programs automatically. ASys relies heavily on reflection to inspect the source code of the target program and discover its internal structure and dependencies. With the information gathered, ASys can modify the target program’s source code at runtime to facilitate the generation of white-box unit tests. In this context, unit tests are crucial in validating students' code submissions by providing precise and targeted feedback on individual functions or methods. This targeted validation aligns with ASys's educational objectives, ensuring that each aspect of the student's solution is thoroughly evaluated. To achieve this, ASys leverages EvoSuite, which is executed by ASys at runtime on the user’s machine. To facilitate this integration, we conducted numerous tests to explore the feasibility of most of the options and facilities offered by EvoSuite. ASys began as a desktop application but has evolved into a client-server architecture with a third component installed on the end user’s machine. This third component is responsible for grading and evaluating programming exercises and has been extended to also handle the generation and execution of unit tests using EvoSuite. As a result, ASys now poses challenges on EvoSuite, such as the need to distinguish test cases generated for overloaded methods and the need for running the test cases on different environments (the teacher and the student side).

This paper aims to share our experience with EvoSuite, illustrating specific issues we identified, such as the insufficient handling of polymorphism and the lack of efficiency and effectiveness in generating tests for specific methods. While EvoSuite provides a solid foundation, our findings suggest that more advanced engines could incorporate features like improved static analysis and dynamic adaptability to better manage these challenges. Developing these new engines would enhance coverage accuracy, reduce the overhead of test generation, and offer more precise testing capabilities, ultimately providing a more robust solution for developers and researchers. We stressed EvoSuite and found errors in its core. Throughout our work, we encountered several challenges and limitations. In this paper, we highlight the problems faced, the solutions implemented, and the findings made. These findings cannot be found in the official tutorials [16], in the StackOverflow responses related to EvoSuite [17], or in the official GitHub repository for the tool [18]. We hope our experience will be a useful guide for future developers and researchers who wish to use EvoSuite in their projects.

Section 2 outlines our discoveries and challenges. In Section 3, we conclude by summarizing our experiences with EvoSuite, highlighting solutions implemented and lessons learned.

II. Findings and Challenges

This section explains the main problems found when using EvoSuite in challenging contexts. It also describes some possible solutions to these problems.

A. *Producing tests for specific methods*

For many research and industrial tasks, e.g., to produce regression tests, it is necessary to generate unit tests for each method under study. Unfortunately, the default behavior of EvoSuite is to generate test files for each class in the application but not for each method. As a result, EvoSuite generates methods `test00, test01`… for a given class, and it is difficult to identify which specific methods are being tested by each generated test. This lack of clarity can significantly impact test coverage, hindering developers' ability to assess whether all relevant methods have been adequately tested. According to previous studies [19], well-named unit tests are essential for understanding the purpose of a test and for navigating through a suite of tests. Descriptive names help developers quickly identify gaps in coverage and ensure that critical paths are thoroughly tested. To address the problem of identifying the methods being tested, we explored two different approaches within EvoSuite that allow for more granular test generation. Each approach comes with its own set of advantages and disadvantages.

Name-based strategy. One strategy to identify the method targeted by a generated unit test is to use the `-Dtest_naming_strategy=COVERAGE` property, which applies the algorithm proposed in [19]. This allows us to identify the tested method in scenarios where a class contains methods with distinct names, as shown in Table I.

TABLE I. EvoSuite-generated tests’ names for methods with distinct names.

Method signature	Test names
boolean is9(int a)	testIs9, testIs9WithNegative
boolean is10(int a)	testIs10, testIs10ReturningTrue, testIs10WithPositive
boolean is11(int a)	testIs11, testIs11ReturningTrue

Nevertheless, our tests showed that polymorphism causes the generation of descriptive names to fail, especially when overloaded methods have the same name but different signatures. In particular, when overloaded methods have at least two parameters with different types, the name generation becomes inaccurate, making it difficult to understand what is being tested (see Table II). Therefore, while this approach improves the identification of the methods under test in many cases, there are still limitations when dealing with polymorphism, and a complementary approach is needed.

TABLE II. EVOSUITE-GENERATED TESTS' NAMES FOR OVERLOADED METHODS (PROBLEMATIC POLYMORPHISM).

Method signature	Test names
boolean is9(int a, int b)	testIs9Taking2Ints, testIs9Taking2IntsReturningTrue
boolean is9(int a, float b)	testIs9Taking1And1ReturningTrueAndIs9Taking1And1AndIs9Taking1And1AndIs9Taking1And1WithPositive0
boolean is9(int a, String b)	testIs9Taking1And1ReturningTrueAndIs9Taking1And1AndIs9Taking1And1AndIs9Taking1And1WithPositive0, testIs9Taking1And1, testIs9Taking1And1WithEmptyString

Target method. Another alternative is to use the `-Dtarget_method` property, which requires the bytecode signature of the method to be tested [20]. Unlike relying on method names, which can sometimes be ambiguous or prone to changes, specifying the target method via its bytecode signature provides a precise and unambiguous identification. EvoSuite generates a separate test file for each method under test using this property.

This approach eliminates the need to parse the method's name to understand which method is being tested, as each test file is explicitly associated with a specific method through its bytecode signature. Moreover, this method-based separation simplifies the organization and management of tests, making it easier to locate and maintain test cases for individual methods within a codebase. However, this approach also has limitations: as we show next, it can only be used under certain circumstances.

1. In EvoSuite 1.0.6, the `-Dtarget_method` property is compatible only with the `BRANCH`, `ONLYBRANCH`, and `INPUT` coverage criteria. Otherwise, it is ignored. Therefore, we can only use it by forcing these three coverage criteria using `-criterion` argument.
2. Another critical issue, reported in [21] but not resolved yet, affects EvoSuite 1.1.0 and 1.2.0 versions and produces a `NullPointerException` in a class within the library responsible for generating tests for the `WEAKMUTATION` and `STRONGMUTATION` coverage criterion. This library is invoked by the main class of the search algorithm that EvoSuite has been using since version 1.1.0, called DynaMOSA. Therefore, there are two ways to avoid this error. The first is to change EvoSuite's search algorithm using the `-Dalgorithm` property. However, it is important to note that this algorithm is the most effective for generating unit tests [22]; so the cost of using this solution is a loss of coverage, ranging from -3% to -21% with single criteria, and from -8% to -36% with multiple criteria. Another solution to this problem is to keep using DynaMOSA but avoid using the weak and strong mutation coverage criterion. This can be done by specifying the default criteria with `-Dcriterion` and skipping the `WEAKMUTATION` and `STRONGMUTATION` criteria. In this case, the cost of this solution is a loss of mutation score of 0.04 with weak mutation and 0.17 with strong mutation [23].

Our tests have revealed that another problem can appear together with the previous one: EvoSuite 1.1.0 and 1.2.0 may struggle to achieve 100% branch coverage, which prevents reaching 100% in other coverage criteria. This problem occurs when methods work with arrays or objects that implement `java.lang.Collection`, as shown in Example 1.

Example 1: Low branch coverage in the presence of collections. Consider the following method:

```
public boolean checkEmpty(java.util.List list) {
   if (list == null || list.isEmpty())
        return false;
   else return true;
}
```

EvoSuite cannot achieve 100% branch coverage if we generate test cases for this method (i.e., using the `target_method` property). The `else` branch remains uncovered, and EvoSuite times out while attempting to cover this branch. In such situations, it may be useful to consider reducing the timeout using `-Dsearch_budget`.

To analyze this case, we conducted a small experiment using the code from Part 2 of the EvoSuite's tutorial. The results are shown in Table III, where *Target* indicates whether tests are generated for each class or method. *Version* is the EvoSuite version used. *Coverage requested* is the type of coverage that EvoSuite tries to maximize, and *resulting coverage* shows the results obtained. Finally, *runtime* displays the time consumed with different timeouts for each target (15 and 60s).

TABLE III. COMPARISON OF COVERAGE AND GENERATION TIMES FOR DIFFERENT EVOSUITE CONFIGURATIONS AND VERSIONS.

Target	Version	Coverage requested	Resulting coverage		Runtime	
			Cov. Type	Cov.	(60s)	(15s)
Class (default)	Any	Default	Output	97.00%	185 s	49 s
			MethodNoEx.	93.75%		
			WeakMutation	98.25%		
			Others	100.00%		
		Branch	Branch	100.00%	7 s	7 s
Method	1.0.6	Branch	Branch	100.00%	-	179 s
	≥ 1.1.0	Branch	Branch	82.92%	-	224 s
	≥ 1.1.0	Default	Line	93.45%	-	224 s
			Branch	82.92%		
			MethodNoEx.	83.33%		
			WeakMutation	34.37%		

CBranch	82.92%
Output	68.33%
Others	100.00%

When running EvoSuite with its default configuration, we achieved 100% coverage in almost all default criteria regardless of the version. However, as we did not reach 100% in all cases, EvoSuite continues attempting to do so until the timeout expires. Reducing the timeout from 60 to 15 seconds produced the same results in less time. We achieved 100% coverage in just 7 seconds when generating tests using only the branch criterion. In tests with `target_method`, we used the default algorithm of EvoSuite 1.0.6 (MONOTONIC_GA). These tests were revealing, as EvoSuite seems not to generate tests until the timeout expires, significantly increasing the test generation time for each method. Although versions higher than 1.0.6 support various coverage criteria, achieving a good result is challenging. In contrast, focusing solely on branch coverage in version 1.0.6 may be more efficient and effective. This complements the results of [24], which showed that *Default* test case generation achieves better results (i.e., higher or same coverage) than *Branch* testing. This can be explained by the fact that in later versions, EvoSuite with the `target_method` property struggles to achieve 100% branch coverage, which it would obtain without using this property. Even if we execute EvoSuite $\geqslant$ 1.1.0 focusing only on branch coverage, version 1.0.6 achieves better results (better coverage and less runtime). This highlights the importance of considering older versions, such as 1.0.6, which, despite lacking some newer features, offer better stability and coverage performance under certain conditions.The observed challenges in achieving 100% branch coverage, particularly in more recent versions of EvoSuite when using the target_method property, point to a broader concern regarding the potential impact of reduced coverage on fault detection. Studies have shown that higher code coverage generally correlates with an increased likelihood of fault detection [12]. However, as highlighted in [13], code coverage is only a moderate indicator of fault detection across a test set, with its effectiveness being more pronounced in exceptional test cases. The drop in coverage, especially in complex scenarios like those involving collections, may lead to undetected faults, thus compromising the overall reliability of the software. This risk underscores the importance of maintaining high coverage levels where possible, while also recognizing the need for complementary testing strategies to address any gaps.

B. Controlled Environment Execution

Generating and executing unit tests in different systems is not possible by default. The cause is that EvoSuite's generated tests come with scaffolding that prepares the EvoSuite environment using `@Before/@After` methods. One such method is `setSystemProperties`, which sets properties (e.g., `user.dir`) that depend on the machine where the tests were generated and may differ from the machine where they will be executed. This can be avoided by disabling the sandboxing system with the properties `-Dsandbox=false` and `-Dfilter_sandbox_tests =true`, which, in turn, removes these dependences to the generation environment. Nevertheless, disabling the sandbox introduces security risks, as the test cases can execute potentially malicious user code without the sandbox's protection [25].

To address the security risks, we have implemented an architecture where the third component of ASys, installed on the user's machine (either teacher or student), handles the generation and execution of unit tests. For teachers, this component generates the tests using EvoSuite, ensuring they are tailored to the specific programming exercises. For students, the same component runs the tests against their solutions, including both grading and evaluating their submissions.

EvoSuite enhances security by isolating potentially harmful code through sandboxing mechanisms. However, ASys takes a different approach by performing the grading and test execution directly on the client side, specifically on the student's machine. This strategy ensures that any risks associated with executing code are confined to the local environment, thus protecting the broader system infrastructure. This client-side grading not only secures the ASys infrastructure but also enhances performance, compatibility, and flexibility in a distributed system.

III. Related Work

The generation of tests for specific methods and their execution in different environments are topics that have received little attention in the literature. While the development of EvoSuite has been supported by numerous studies highlighting its challenges [8] and identifying its ineffectiveness in certain situations [11], most of this work focuses on the execution of EvoSuite at the project level, without clearly distinguishing the tested methods. This poses a significant problem because, even if tests successfully detect faults, it becomes difficult to contextualize these issues without tests being specifically documented for each method.

One area that has been explored is the impact of parameter tuning on EvoSuite's performance. Studies like [26] have shown that appropriate parameter tuning can improve EvoSuite's performance, although, in most cases, default values are sufficient. However, these investigations do not address the granularity of test generation at the method level, leaving an important gap in the literature.

The study in [19] partially addresses this issue by introducing an algorithm that attempts to assign descriptive names to the tested methods, improving the identification and contextualization of tests. Despite this advancement, there is still work to be done to achieve more effective documentation of the generated tests.

Regarding the sandboxing employed by EvoSuite, developers have made significant efforts to use bytecode instrumentation to automatically separate code from its environmental dependencies and to set the state of the environment as part of the generated call sequences [27]. However, EvoSuite also implements a custom Security Manager that restricts many dangerous interactions with the environment, while still allowing specific system configurations, such as user.dir, to ensure that tests execute consistently [9]. This explains why certain system properties remain set in the automatically generated tests, despite efforts to isolate the environment.

Although there are autograding solutions in the literature that employ various security techniques, such as those mentioned in [25], there is no documented use of these techniques in combination with EvoSuite, particularly focusing on client-side security. This highlights a gap that our work addresses by implementing security at the client side in ASys.

IV. Conclusions

Our experience with EvoSuite has been instrumental in identifying various challenges and solutions in configuring and generating automated unit tests. We have thoroughly explored the wide range of configurable parameters offered by EvoSuite, providing guidance on how to find the right values to solve problems and optimize test generation.

One significant challenge we encountered was the generation of specific tests for individual methods. EvoSuite's default behavior of producing non-descriptive test names (e.g., test00, test01, etc.) complicates the identification of which specific methods are being tested, which can significantly impact test coverage. To address this, we explored two distinct approaches: a name-based strategy, which is a valid option when there is no method overloading. However, this approach is limited by issues related to polymorphism, particularly when overloaded methods are involved, leading to inaccurate or unclear test names. The second approach involves the use of the `target_method` parameter, but we also encountered errors and limitations with this option, such as compatibility issues and difficulties in achieving full branch coverage, especially when methods involve `java.lang.Collection`.

Moreover, while newer versions of EvoSuite offer additional features, our tests revealed that these versions sometimes struggle with issues like reduced branch coverage when using the `target_method` property with data structures like `java.lang.Collection`. In contrast, older versions, such as 1.0.6, demonstrated better stability and coverage performance under certain conditions. This highlights the importance of carefully selecting the version of EvoSuite based on the project's specific needs, even if it means foregoing some of the newer features.

We also addressed the risk of dependencies produced in the generated test cases with the environment in which they were generated. This was particularly challenging in distributed environments where tests needed to be executed on multiple machines. By disabling EvoSuite's sandboxing system, we mitigated environment-specific dependencies, but this introduced security risks, as it allowed potentially malicious code to execute without the sandbox's protection. To solve this, we implemented an architecture in ASys that allows tests to be generated on the teacher's machine and executed on the student's machine, thereby confining any risks to the local environment.

In conclusion, our practical experience with EvoSuite provides useful knowledge for identifying common challenges in generating automated unit tests and offering practical solutions to overcome them. We are confident that our findings will benefit other development teams looking to leverage the capabilities of EvoSuite to the fullest in their software projects.

Looking ahead, we plan to expand our experiments by applying the `target_method` parameter of EvoSuite to the SF100 benchmark, a statistically sound collection of Java projects from SourceForge [28]. This will allow us to evaluate our solutions in a more diverse and realistic environment, identifying opportunities for improving coverage and effectiveness in more complex contexts. Additionally, we aim to explore the generation of tests for scenarios involving inheritance and method overriding, addressing the challenges EvoSuite faces in these situations. This exploration will help us determine whether the issues encountered with overloaded methods also apply to inherited and overridden methods, ensuring a more comprehensive understanding of EvoSuite's capabilities and limitations in object-oriented programming contexts. By enhancing the tool's ability to manage these complexities, we hope to ensure more comprehensive and accurate testing across a wider range of software projects.

Acknowledgement

This work has been partially supported by the Spanish *MCIN/AEI* under grant PID2019-104735RB-C41 and by *Generalitat Valenciana* under grant CIPROM/2022/6 (Fasslow). Carlos Galindo was partially supported by the Spanish *Ministerio de Universidades* under grant FPU20/03861.

References

[1] OpenAI, “Introducing ChatGPT.” Accessed: May 26, 2024. [Online]. Available: https://openai.com/chatgpt/

[2] Z. Yuan *et al.*, “No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation,” 2024.

[3] M. Harman, S. A. Mansouri, and Y. Zhang, “Search-based software engineering: Trends, techniques and applications,” *ACM Comput. Surv.*, vol. 45, no. 1, Dec. 2012, doi: 10.1145/2379776.2379787.

[4] Y. Tang, Z. Liu, Z. Zhou, and X. Luo, “ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation,” 2023.

[5] G. Fraser and A. Arcuri, “EvoSuite: Automatic test suite generation for object-oriented software,” in *SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering*, Oct. 2011, pp. 416–419. doi: 10.1145/2025113.2025179.

[6] A. Gambi, G. Jahangirova, V. Riccio, and F. Zampetti, “SBST Tool Competition 2022,” in *2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST)*, 2022, pp. 25–32. doi: 10.1145/3526072.3527538.

[7] G. Jahangirova and V. Terragni, “SBFT Tool Competition 2023 - Java Test Case Generation Track,” in *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, IEEE, May 2023, pp. 61–64. doi: 10.1109/SBFT59156.2023.00025.

[8] G. Fraser and A. Arcuri, “Evosuite: On the challenges of test case generation in the real world,” in *2013 IEEE sixth international conference on software testing, verification and validation*, 2013, pp. 362–369.

[9] G. Fraser and A. Arcuri, “A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite,” *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 2, Dec. 2014, doi: 10.1145/2685612.

[10] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, “Does automated white-box test generation really help software testers?,” in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, in ISSTA 2013. New York, NY, USA: Association for Computing Machinery, 2013, pp. 291–301. doi: 10.1145/2483760.2483774.

[11] Z. Fan, “A Systematic Evaluation of Problematic Tests Generated by EvoSuite,” in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2019, pp. 165–167. doi: 10.1109/ICSE-Companion.2019.00068.

[12] 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. doi: 10.1109/ASE.2015.86.

[13] X. Cai and M. R. Lyu, “The effect of code coverage on fault detection under different testing profiles,” *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–7, May 2005, doi: 10.1145/1082983.1083288.

[14] F. Del Frate, P. Garg, A. P. Mathur, and A. Pasquini, “On the correlation between code coverage and software reliability,” in *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE’95*, 1995, pp. 124–132. doi: 10.1109/ISSRE.1995.497650.

[15] D. Insa, S. Pérez, J. Silva, and S. Tamarit, “Semiautomatic generation and assessment of Java exercises in engineering education,” *Computer Applications in Engineering Education*, 2020, doi: 10.1002/cae.22356.

[16] G. Fraser, “A Tutorial on Using and Extending the EvoSuite Search-Based Test Generator,” in *Search-Based Software Engineering*, P. Colanzi Thelma Elita and McMinn, Ed., Cham: Springer International Publishing, 2018, pp. 106–130.

[17] “StackOverflow - EvoSuite questions.” Accessed: May 26, 2024. [Online]. Available: https://stackoverflow.com/questions/tagged/evosuite

[18] “EvoSuite GitHub repo.” Accessed: Jan. 01, 2024. [Online]. Available: https://github.com/EvoSuite/evosuite

[19] E. Daka, J. M. Rojas, and G. Fraser, “Generating unit tests with descriptive names or: would you name your children thing1 and thing2?,” in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, in ISSTA 2017. New York, NY, USA: Association for Computing Machinery, 2017, pp. 57–67. doi: 10.1145/3092703.3092727.

[20] “JNI Types and Data Structures.” Accessed: Jun. 02, 2024. [Online]. Available: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/types.html#wp276

[21] “EvoSuite Issues - Using EvoSuite target_method.” Accessed: Jun. 02, 2024. [Online]. Available: https://github.com/EvoSuite/evosuite/issues/439

[22] J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler, and A. Arcuri, “An empirical evaluation of evolutionary algorithms for unit test suite generation,” *Inf Softw Technol*, vol. 104, pp. 207–235, 2018, doi: https://doi.org/10.1016/j.infsof.2018.08.010.

[23] G. Fraser and A. Arcuri, “Achieving scalable mutation-based generation of whole test suites,” *Empir Softw Eng*, vol. 20, no. 3, pp. 783–812, 2015, doi: 10.1007/s10664-013-9299-z.

[24] G. Fraser and A. Arcuri, “Whole Test Suite Generation,” *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013, doi: 10.1109/TSE.2012.14.

[25] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, “Review of recent systems for automatic assessment of programming assignments,” in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, New York, NY, USA: ACM, Oct. 2010, pp. 86–93. doi: 10.1145/1930464.1930480.

[26] A. Arcuri and G. Fraser, “Parameter tuning or default values? An empirical investigation in search-based software engineering,” *Empir Softw Eng*, vol. 18, no. 3, pp. 594–623, 2013, doi: 10.1007/s10664-013-9249-9.

[27] A. Arcuri, G. Fraser, and J. P. Galeotti, “Automated unit test generation for classes with environment dependencies,” in *Proceedings of the 29th ACM/IEEE International Conference on Automated*

Software Engineering, in ASE ’14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 79–90. doi: 10.1145/2642937.2642986.

[28] G. Fraser and A. Arcuri, “Sound empirical evidence in software testing,” in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 178–188. doi: 10.1109/ICSE.2012.6227195.