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.