Delft University of Technology

TestSpark: IntelliJ IDEA's Ultimate Test Generation Companion

Sapozhnikov, Arkadii ; Olsthoorn, Mitchell; Panichella, A.; Kovalenko, V.V.; Derakhshanfar, P.

DOI
10.1145/3639478.3640024

Publication date
2024

Document Version
Accepted author manuscript

Published in
Proceedings - 2024 ACM/IEEE 46th International Conference on Software Engineering

Citation (APA)
Sapozhnikov, A., Olsthoorn, M., Panichella, A., Kovalenko, V. V., & Derakhshanfar, P. (2024). TestSpark: IntelliJ IDEA's Ultimate Test Generation Companion. In *Proceedings - 2024 ACM/IEEE 46th International Conference on Software Engineering: Companion, ICSE-Companion 2024* (pp. 30-34). (Proceedings - International Conference on Software Engineering). IEEE / ACM. https://doi.org/10.1145/3639478.3640024

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

TestSpark: IntelliJ IDEA's Ultimate Test Generation Companion

Arkadii Sapozhnikov
arkadii.sapozhnikov@jetbrains.com
JetBrains Research
Berlin, Germany

Mitchell Olsthoorn
M.J.G.Olsthoorn@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Annibale Panichella
A.Panichella@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Vladimir Kovalenko
vladimir.kovalenko@jetbrains.com
JetBrains Research
Amsterdam, The Netherlands

Pouria Derakhshanfar
pouria.derakhshanfar@jetbrains.com
JetBrains Research
Amsterdam, The Netherlands

ABSTRACT

Writing software tests is laborious and time-consuming. To address this, prior studies introduced various automated test-generation techniques. A well-explored research direction in this field is unit test generation, wherein artificial intelligence (AI) techniques create tests for a method/class under test. While many of these techniques have primarily found applications in a research context, existing tools (*e.g., EvoSuite*, Randoop, and AthenaTest) are not user-friendly and are tailored to a single technique. This paper introduces *TestSpark*, a plugin for IntelliJ IDEA that enables users to generate unit tests with only a few clicks directly within their Integrated Development Environment (IDE). Furthermore, *TestSpark* also allows users to easily modify and run each generated test and integrate them into the project workflow. *TestSpark* leverages the advances of search-based test generation tools, and it introduces a technique to generate unit tests using Large Language Models (LLMs) by creating a feedback cycle between the IDE and the LLM. Since *TestSpark* is an open-source (https://github.com/JetBrains-Research/TestSpark), extendable, and well-documented tool, it is possible to add new test generation methods into the plugin with the minimum effort. This paper also explains our future studies related to *TestSpark* and our preliminary results. **Demo video:** https://youtu.be/0F4PrxWfiXo

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Unit Test Generation, IntelliJ IDEA Plugin, Large Language Models

ACM Reference Format:
Arkadii Sapozhnikov, Mitchell Olsthoorn, Annibale Panichella, Vladimir Kovalenko, and Pouria Derakhshanfar. 2024. TestSpark: IntelliJ IDEA's Ultimate Test Generation Companion. In *Proceedings of 46th International Conference on Software Engineering (ICSE '24 Demonstrations).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '24 Demonstrations, April 14–20, 2024, Lisbon, Portugal
© 2024 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Software testing is essential in the software development process, yet it can be time-consuming and costly [6]. Developers need to manually craft tests that cover various behaviors of their projects. As a result, numerous studies [15] propose a range of techniques to automatically generate tests for different testing levels, including unit [2, 5, 7, 9, 11, 14, 18], integration [10], and system level [4]. These studies confirm that the generated tests not only achieve high code coverage [17] but also proved valuable for error detection [19] and debugging [8]. However, most of these unit test generation tools were primarily designed and used for research studies. Consequently, openly available test generation tools specialize in one technique and often rely on a command-line interface, making them less user-friendly within development environments like IDEs. As a result, users need to interact with each tool separately outside the IDE and, later, integrate tests into their projects manually.

This paper introduces *TestSpark*, an open-source, extendable, and well-documented IntelliJ IDEA plugin for unit-level test generation of Java programs. It is designed to ease unit test generation with various techniques. Currently, *TestSpark* supports two technologies: Search-based software test generation (SBST) and Large Language Model (LLM)-based test generation. The former is one of the most effective unit test generation techniques [12]. The latter has shown potential in helping developers in their software engineering tasks, including software testing [13]. However, a recent study [20] shows that a large portion (> 50%) of tests generated by LLMs are malformed and non-compiling. *TestSpark* introduces an approach (Section 3.2) to ensure all tests generated by LLMs are compilable by proposing a feedback loop between the LLM and the IDE. *TestSpark* is designed to let contributors easily integrate other test generation tools by following our documentation[1].

Within the IDE, users can seamlessly generate, analyze, modify, and integrate unit tests using *TestSpark*. Our tool is capable of generating tests at various granularity levels, such as for a class, method, or even a single line of code. Once tests are generated, *TestSpark* offers a visual representation along with a detailed coverage report, encompassing covered lines and killed mutants. Users have the flexibility to fine-tune (manually or by LLM) and select the generated tests, while also providing feedback to enhance each test case. Finally, users can easily integrate the tests into their projects.

[1]https://github.com/JetBrains-Research/TestSpark/blob/main/CONTRIBUTING.md

We have released this plugin in the JetBrains Marketplace[2]. Despite the recent release, the plugin has garnered positive feedback and more than 1K downloads. We designed studies to assess the usability of our plugin and the usefulness of the generated tests (explained in Section 4). Moreover, the results of our preliminary study show that test generation techniques and features implemented in *TestSpark* assist developers in their unit testing tasks. *TestSpark* is helpful for both software developers and researchers, acting as a bridge between these two communities, all with a singular aim: improving test generation techniques for practical usage. Developers can readily use different test generation approaches to assist them in writing unit tests. Also, researchers can implement their novel strategies within this framework, evaluating them in a development environment and collecting invaluable user feedback.

2 RELATED WORK

Java Unit Test Generation tools. In the past few years, researchers and developers implemented multiple unit test generation tools for Java programs leveraging various techniques (*e.g.,* search-based [9, 11], LLM-based [5, 14, 18], and symbolic execution [1–3, 7]). Among these tools, *EvoSuite* [11], which uses an SBST approach [16], outperforms other tools in terms of structural coverage and fault detection [12]. Most of these Java unit test generation tools use a command-line interface and output their result in one or multiple files (*e.g.,* CSV report file and a Java file containing the generated tests). Hence, they are not fully integrated into IDEs, where developers write tests. We note that *EvoSuite* also has an IntelliJ IDEA plugin [3], supporting only outdated versions of the IDE (*i.e.,* their compatibility range is 15.0 — 2019.3.5). Moreover, this plugin requires installing *EvoSuite* separately. Also, after test generation, the plugin directly saves the generated tests and their report in a folder. In contrast, *TestSpark* provides a user-friendly interface in IntelliJ IDEA to generate tests for Java code with only one click and visualizes the generated tests and their coverage. Users can interact with each test and use LLMs to enhance the tests generated by each technique. *TestSpark* is designed to be extendable and supports the integration of multiple test generation tools. Lastly, *TestSpark* is a standalone plugin (*i.e.,* does not need any other installation) and supports the latest versions of the IntelliJ IDEA.

Java Unit Test Generation plugins in IntelliJ IDEA. By searching relevant keywords (*e.g.,* unit test) in the JetBrains Marketplace, we found five Java unit test generation plugins that are under active development (*i.e.,* have at least one yearly release[4]). *Kex* [1, 2] and *UnitTestBot* [3] use symbolic execution for test generation. A prior study [12] confirms that they cannot achieve the structural coverage and fault detection capability of *EvoSuite*. In contrast to *TestSpark*, these plugins do not provide any visualization for reporting the execution of the generated tests. *DiffBlue*[5] and *SquareTest*[6] are paid plugins that generate Java unit tests. There is little information about the techniques these plugins use, as their code is not openly available. These plugins also do not visualize

[2]https://plugins.jetbrains.com/plugin/21024-testspark
[3]https://plugins.jetbrains.com/plugin/7985-evosuite-plugin
[4]Date of query: *Sep 12, 2023*
[5]https://plugins.jetbrains.com/plugin/14946-diffblue-cover--ai-for-code
[6]https://plugins.jetbrains.com/plugin/10405-squaretest

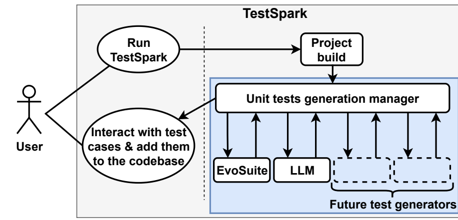

Figure 1: *TestSpark* workflow

the execution of the generated tests. Finally, *Chat Unit Test*[7] is an LLM-based test generation plugin that uses *OpenAI*'s *ChatGPT* to generate tests for a Java class and save the response directly into the code base. *TestSpark* also supports LLM-based test generation; however, it can additionally generate tests for individual lines of code. Instead of directly saving all tests in a project file, *TestSpark* lets users i) analyze each test using the visualized coverage, ii) ask an LLM to apply further enhancement on each test, and iii) select the interesting tests and save them as a new test suite or integrate them into existing test files. *TestSpark* is not only an LLM-based test generation framework. Since *TestSpark* has an extendable architecture, new test generators can easily be integrated into this pipeline. Tests generated by any technique can be improved by LLM upon the user's request. E.g., users can ask LLM to improve the readability of a test case generated by *EvoSuite*.

3 *TESTSPARK*

TestSpark is an IntelliJ IDEA plugin for generating unit tests that leverages two techniques to bring test generation into the development environment: SBT (via the state-of-the-art tool *EvoSuite*) and LLM-based test generation (Section 3.2). Fig. 1 illustrates *TestSpark*'s workflow. Users can initiate the test generation process for a unit under test (referred to as *UUT*) by simply right-clicking on a unit and selecting the `TestSpark` option. Upon selecting the desired test generation technique, the process starts. First, the plugin builds the project within IntelliJ IDEA. This step is crucial as *EvoSuite* requires the compiled code for code instrumentation [11]. Similarly, LLM-based test generation requires compilation for test execution and validation based on the model's generative outcome. Following code compilation, the **unit test generation manager** employs the chosen technique to initiate the generation process. Once the tests are collected, they are seamlessly transmitted to the visualization service, where the results are presented to the user (see Section 3.3).

3.1 Interaction with *EvoSuite*

To integrate *EvoSuite* within *TestSpark*, we pursued two main goals: (i) receiving the tests generated by *EvoSuite* and its coverage information inside the plugin instead of reading it from files and (ii) adding a feature to *EvoSuite* that tests individual lines. This empowers users to focus the search process on covering specific lines, enhancing the flexibility of the test generator. This enhanced version of *EvoSuite* is available on GitHub [8]. In contrast to the *EvoSuite*

[7]https://plugins.jetbrains.com/plugin/22522-chatunitest
[8]https://github.com/ciselab/evosuite/tree/thunderdome

IntelliJ IDEA plugin, *TestSpark* automatically includes the latest release of an enhanced version of *EvoSuite* during the build process.

To accomplish the first goal, we implemented the *Compact Reporter* module inside *EvoSuite* that generates a serialized report. This report contains all information regarding the tests generated by *EvoSuite*, including test cases and the coverage achieved by each.

For the second goal, we modified the default test generation algorithm (DynaMOSA [16]) in *EvoSuite*. DynaMOSA generates tests to cover lines and branches in a selected class or method. This algorithm **1)** analyzes the control flow graphs (CFG) of class/method under tests; **2)** it selectively adds branches and lines that are not control-dependent on any other branches into the *active search objectives*; **3)** it generates a new set of tests and identifies newly covered goals (*i.e.,* branches and lines) in the *active search objectives*; **4)** it saves the tests reaching previously uncovered goals into the archive; **5)** it updates the *active search objectives* by removing newly covered goals and adding lines and branches that are control-dependent on these; **6)** repeats steps 3 to 6. In the single-line coverage mode, which we implemented, we made specific modifications to steps 2 and 6. In these steps, we exclusively add lines and branches that, when covered, provide the test with the opportunity to cover the selected target line (as per the CFG). In other words, this algorithm excludes irrelevant lines and branches to make the search process more focused on covering the target line.

The plugin generates a command to invoke `evosuite.jar` for test generation. This command passes the *UUT* and the path to the compiled project as parameters.

3.2 LLM-based test generation

After selecting the preferred LLM platform (*i.e.,* the *OpenAI* platform and an internal service API in JetBrains) and providing the authentication token, users should select the specific LLM model to be used for test generation. With these configurations in place, users can proceed to request test generation for the *UUT*. The plugin then generates a corresponding prompt tailored for the LLM.

3.2.1 Prompt generation. The prompt should provide sufficient details for the LLM to generate executable tests. However, LLMs have a maximum prompt size, so it is necessary to balance the size and the information provided. To achieve this balance, the prompt generated by *TestSpark* should contain the following information: (i) the problem description; (ii) *UUT*'s code; (iii) the signatures of methods and objects passed to *UUT* as input parameters; (iv) the polymorphism relations of the classes used by the *UUT*. Users can modify the prompt template in the settings and add additional information (*e.g.,* customized testing objectives or number of test cases). Also, the depth of input parameters (used for listing the method signatures) and polymorphism depth are set by the user in *TestSpark* settings. If the prompt generated with the given values exceeds the maximum permitted prompt size, *TestSpark* decreases these values iteratively until the prompt is smaller than the permitted size.

3.2.2 Receiving the LLM response. After prompt generation (① in Fig. 2), *TestSpark* sends it to the selected LLM (② - ③). Once the LLM's response is received, *TestSpark* parses it, extracting the test code written in Java ④. Following this, *TestSpark* checks each test case individually to identify and save the ones that compile

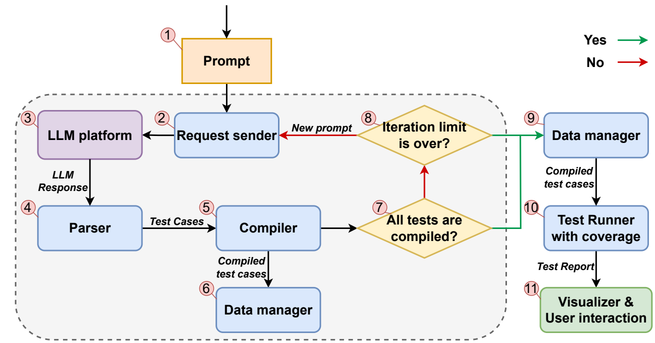

Figure 2: LLM-based test generation

(⑤-⑥). In case the compilation of the test cases (⑦) is successful, *TestSpark* gathers the code coverage information using *JaCoCo*[9] (⑨-⑪) and visualizes the results (see Section 3.3). However, if the compilation of the test cases fails and the iteration limit is not reached (⑧), *TestSpark* generates a new prompt containing the compilation error and sends a request to the LLM to fix the detected error. The maximum number of iterations of such a loopback is set by the user in the plugin settings. If the number of iterations is exhausted, the plugin continues the process with saved (compilable) tests, if any. Otherwise, it shows an error message and asks to try generating tests for a smaller *UUT* (*i.e.,* method or line).

3.3 Visualization

After receiving the result, the user can interact with the test cases in various ways and view their coverage. Fig. 3 illustrates an example of such a result. The *Coverage* tab shows the structural coverage and mutation score achieved by all generated tests (① in Fig. 3). The table adjusts dynamically, only calculating the statistics for the selected tests. In the *Generated Test* tab, each generated test is presented. At the top of this panel, *TestSpark* provides the count of passed and selected test cases, along with buttons to select, unselect, and delete all test cases (②-⑥ in Fig. 3, respectively).

Users have the option to copy, like, dislike, and select each test case (⑧-⑪, respectively). Also, the border color of each test block indicates whether it passed (green) or failed (red). In case of failure, users can hover over the error symbol ⑱ to view the error message ⑲. Moreover, the user can modify each test in the code field ⑫. After modification, the "run" ⑬, "reset to the initial code" ⑭, and "reset to the last run" ⑮ buttons become available, allowing users to execute the updated test and undo their actions. Unwanted tests can also be removed ⑯. Additionally, each test case has a text field where users can directly send a modification request to the LLM ⑰, *e.g.,* adding comments. This option triggers a new request to the LLM and shows the updated test in a new code window. The user can switch between the old and updated tests ⑦.

Each covered line contains green square on its left. Clicking on this square opens a list of test cases that cover this line ⑳. If the user uses *EvoSuite* as generator, *TestSpark* shows the list of mutants

[9] https://www.jacoco.org/jacoco/

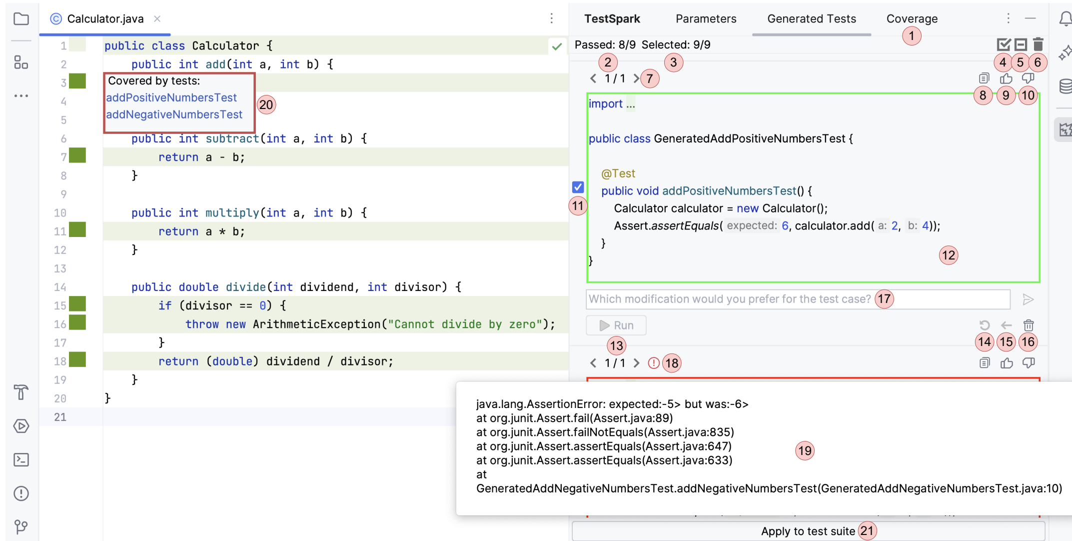

Figure 3: *TestSpark*'s tests visualization

in this line. Clicking on a test case name or a mutant highlights the test cases in the "Generated Tests" that cover or kill it.

Finally, users can integrate the selected/modified tests into their project by clicking the "Apply to test suite" button ㉑. The user can put the tests either in a new or existing file. In the first case, the user needs to select a folder and then enter the name of the new test class. In the second case, they can select a Java file to which the new test cases will be added. In both cases, our plugin adds the required imports, and thereby after integration, the saved/updated test file is compilable and ready to execute. Moreover, *TestSpark* offers the option to change the colors of covered lines for color-blind users.

4 EVALUATION

This section explains the designs of our planned studies. Since our goal for implementing *TestSpark* is to aid developers in development and unit testing tasks using test generation, our research questions are focused on users' interactions with this plugin and how useful they find the test generators and features in *TestSpark*: **RQ1:** *How useful are the generated tests for developers in their day-to-day testing practices?* and **RQ2:** *How usable is TestSpark?*

Analyzing features usage statistics (FUS): We are currently implementing collectors to gather anonymized FUS from our 1K+ users. These collectors are designed to answer the following questions: (1) How often do users use each test generation technique? (2) For which *UUT* do users prefer to generate tests (*e.g.,* class, or line)? (3) How often is the test generation successful for each technique? (4) How long does it take to generate tests by each test generation technique? (5) How many tests generated by each technique are integrated into the code base? (6) which part of the tests are modified by users (*i.e.,* test data, method calls, or assertions)? (7) How often do users provide feedback to the LLM (using ⑰ in Fig. 3)?

Questionnaire: First, we provide a list of tasks to participants (who have practical experience in unit testing) to use *TestSpark* for testing their Java projects. Tasks include generating tests for lines, methods, and classes. Then, we ask participants to analyze the tests and select/integrate the ones that they find useful for their project. After performing these tasks, the participants should answer a questionnaire about each test generator and feature in *TestSpark*. We have conducted a preliminary study based on this questionnaire with two participants (a colleague with unit testing experience and an academic researcher who actively studies software testing). Participants confirmed the usefulness of coverage achieved by generated tests and appreciated the ability to interact with test cases before integrating them into their projects. Moreover, they noted that generating tests for methods and lines is often more convenient because classes are sometimes too coarse-grained. This confirms the usefulness of the single-line test generation feature.

5 CONCLUSION AND FUTURE STEPS

In this paper, we have introduced *TestSpark*, an open-source unit test generation plugin for IntelliJ IDEA. Leveraging cutting-edge techniques, *TestSpark* empowers users to effortlessly generate tests for a *UUT*, all within the familiar IDE environment. With a user-friendly interface and a range of features, developers can efficiently manage their testing tasks without switching contexts. *TestSpark* also provides an infrastructure for researchers to implement new test generation approaches and assess these techniques according to user feedback. Preliminary observations and feedback indicate the usefulness of *TestSpark* in accelerating unit testing tasks. Additionally, we discussed the designs of our planned studies. As future steps, we aim to expand *TestSpark*'s language support. Also, we aim to combine test generation methods to generate better tests (*e.g.,* improve LLM-based technique using SBT).

ACKNOWLEDGMENTS

We thank the students who helped us with the initial prototype of this plugin: Jegor Zelenjak, Martin Mladenov, Kiril Vasilev, Lyuben Todorov, Sergey Datskiv, and Bolek Khodakov.

REFERENCES

[1] 2021. Kex. https://github.com/vorpal-research/kex/tree/sbst-contest.

[2] 2022. Kex-Reflection. https://github.com/vorpal-research/kex/tree/sbst2022-reflection.

[3] 2022. UTBot. https://github.com/UnitTestBot.

[4] Andrea Arcuri. 2019. RESTful API automated test case generation with EvoMaster. *ACM Transactions on Soft. Eng. and Methodology (TOSEM)* 28, 1 (2019), 1–37.

[5] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 2022. Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code. CoRR abs/2206.01335 (2022).

[6] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 179–190.

[7] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. 2018. SUSHI: a test generator for programs with complex structured inputs. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 21–24.

[8] Mariano Ceccato, Alessandro Marchetto, Leonardo Mariani, Cu D Nguyen, and Paolo Tonella. 2015. Do automatically generated test cases make debugging easier? an experimental assessment of debugging effectiveness and efficiency. *ACM Transactions on Soft. Eng. and Methodology (TOSEM)* 25, 1 (2015), 1–38.

[9] Pouria Derakhshanfar and Xavier Devroey. 2022. Basic block coverage for unit test generation at the SBST 2022 tool competition. In *Proceedings of the 15th Workshop on Search-Based Software Testing*. 37–38.

[10] Pouria Derakhshanfar, Xavier Devroey, Annibale Panichella, Andy Zaidman, and Arie van Deursen. 2022. Generating Class-Level Integration Tests Using Call Site Information. *IEEE Transactions on Software Engineering* 49, 4 (2022), 2069–2087.

[11] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[12] Gunel Jahangirova and Valerio Terragni. 2023. SBFT tool competition 2023-Java test case generation track. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. IEEE, 61–64.

[13] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2312–2323.

[14] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. CODAMOSA: Escaping coverage plateaus in test generation with pre-trained large language models. In *International conference on software engineering (ICSE)*.

[15] Phil McMinn. 2011. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 153–163.

[16] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2017. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* 44, 2 (2017), 122–158.

[17] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. A large scale empirical comparison of state-of-the-art search-based test case generators. *Information and Software Technology* 104 (2018), 236–256.

[18] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* (2023).

[19] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *2015 30th IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. IEEE, 201–211.

[20] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020).