SBFT Tool Competition 2023 - Java Test Case Generation Track

Gunel Jahangirova
King's College London, United Kingdom
gunel.jahangirova@kcl.ac.uk

Valerio Terragni
University of Auckland, New Zealand
v.terragni@auckland.ac.nz

ABSTRACT

This paper details the eleventh edition of Java Unit Testing Competition, covering its setup, challenges, and findings. The competition featured five Java test case generation tools: EvoSuite, Kex-concolic, Kex-symbolic, Utbot-concolic, and Utbot-fuzzer, all of which were evaluated on a benchmark of 100 classes taken from 5 open-source Java Projects. We assessed the generated test cases based on code and mutation coverage, as well as human understandability - a metric introduced in this edition of the competition.

ACM Reference Format:
Gunel Jahangirova and Valerio Terragni. 2023. SBFT Tool Competition 2023 - Java Test Case Generation Track. In *The 16th Search-Based and Fuzz Testing Workshop (SBFT'23), May 14, 2023, Melbourne, Australia.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3526072.3527538

1 INTRODUCTION

The eleventh edition of the Java Testing Tool Competition received five submitted tools, namely EvoSuite [8], Kex-Concolic [5], Kex-Symbolic [5], UTBot-Concolic [3, 4] and UTBot-Fuzzer [3, 4]. Furthermore, similarly to previous editions, we used Randoop [11] as a baseline for comparison. This tool competition is conducted along with the competition for cyber-physical systems[6] and for [10]. Each tool has been executed on 100 classes under test (CUTs) sampled from five different Java projects. The competing tools have been compared by using line, branch and mutant coverage metrics, for two different time budgets, i.e., 30 and 120 seconds. Moreover, we have conducted a study to measure how understandable are the generated test cases for the human participants.

In order to guarantee a fair comparison among the competing tools, the execution of the tools for generating test suites and their evaluation has been carried out by using a dockerized infrastructure [7]. The results show that EvoSuite achieves higher code coverage, while Utbot-concolic generates more human-understandable test cases.

2 THE BENCHMARK SUBJECTS OF THE JUNIT TESTING COMPETITION

The selection of the projects and classes under test (CUTs) to use as a benchmark for test case generation has been done by considering the following criteria: (i) the projects must belong to different application domains [8]; (ii) projects must be open-source for replicability purposes. To select the subjects for the competition we relied on the curated list of popular Java frameworks, libraries and software[1]. We selected five subjects each of which belong to a different category. We focused on projects that rely on Maven as a build framework, and have developer-written JUnit[2] test suites. Specifically, we picked:

- *Apache Commons Collections* (https://commons.apache.org/proper/commons-collections/) is a library that extends the Java Collections Framework by adding many powerful data structures that accelerate the development of most significant Java applications.
- *JSoup* (https://github.com/jhy/jsoup) is a Java library for working with real-world HTML.
- *ta4j* (https://github.com/ta4j/ta4j) is a Java library for technical analysis. It provides the basic components for the creation, evaluation and execution of trading strategies.
- *Spatial4j* (https://github.com/locationtech/spatial4j) is a general-purpose spatial/ geospatial open-source Java library.
- *threeten-extra* (https://github.com/ThreeTen/threeten-extra) provides additional date-time classes that complement those in Java SE 8 and is curated by the primary author of the Java 8 date and time library.

Among all 1,145 classes across five projects, we only kept the ones that (i) have at least 2 branches (ii) have at least one method with McCabe's cyclomatic complexity higher than five. To calculate the number of branches in the class and cyclomatic complexity of each method in that class we used the JaCoCo [1] code coverage tool. Then, we filtered out all the classes that are non-testable. We verified testability by running the Randoop test case generation tool for 10 seconds and checking whether any test cases were generated with this budget. If that was not the case, the class was classified as non-testable. The applied filtering step left us with a list of 298 classes. Based on the time and resources available for running the competition, as well as, taking into account the high number of competing tools, and the fact that we used two different time budgets, we randomly sampled 100 classes to use as our benchmark (as shown in Table 1). It should be noted that this is the largest benchmark set used in the history of the competition.

Table 1: Description of the benchmark.

Project	#CUTs	# Filtered CUTs	# Sampled CUTs
Collections	473	98	26
JSoup	246	38	14
ta4j	256	81	30
Spatial4j	92	30	13
threeten-extra	77	52	17
Total	**1,145**	**298**	**100**

[1] https://github.com/akullpp/awesome-java
[2] https://github.com/junit-team/junit4

3 COMPETING TOOLS

Five tools have competed in the eleventh edition: EvoSuite [8], Kex-concolic [5], Kex-symbolic [5], Utbot-concolic [4], Utbot-fuzzer [4].

EvoSuite [8] uses evolutionary search to automatically generate test suites that aim to maximise various code coverage criteria. The current default evolutionary algorithm of EvoSuite is Dynamic Many-Objective Sorting Algorithm (DynaMOSA) [12].

Kex-symbolic and **Kex-concolic** [5]. Kex is a platform for analysis of JVM programs, which mainly focuses on automatic test generation with the aim to maximize branch coverage criterion. Kex can generate tests in fully static mode without running any actual code (Kex-symbolic) and in concolic mode (Kex-concolic) which combines symbolic and concrete executions.

Utbot-concolic and **Utbot-fuzzer** [3, 4]. UTBot Java is a part of the UnitTestBot tool lineup [3, 4] for automated unit test generation. This year, UTBot Java is implemented as Utbot-fuzzer, which is a pure greybox fuzzer, and Utbot-concolic, which is based on dynamic symbolic execution paired with fuzzing. The Utbot-fuzzer gathers constant values from the code under test to generate inputs faster. Utbot-concolic also generates human-readable test descriptions.

Randoop [11], used as a baseline in the context of the competition, generates unit tests using a feedback-directed random test generation, which collects information from the execution of the tests as they are generated to reduce the number of redundant and illegal tests [11].

4 METHODOLOGY OF THE TESTING COMPETITION

4.1 Calculation of the structural coverage criteria

We considered only two different time budgets: 30 and 120 seconds due to time and resource constraints, the high number of participating tools (five plus Randoop as baseline) and the large size of our benchmark. To account for the randomness associated with certain tools (such as search-based or random approaches), we executed each tool 10 times for each CUT. This resulted in 12,000 executions in total, which we used for statistical analysis: i.e., 100 CUTs × 6 tools × 2 time budgets × 10 repetitions. For all the competing tools, we were able to complete the planned number of executions.

We ran each tool on virtual machines with the same architecture i.e., Google Cloud e2-highmem-8 virtual machine instances equipped with 8 vCPUs, 64 GB of RAM and 50 GB of memory. We used a dedicated instance for the combination of each tool and time budget, employing 12 virtual machines instances overall.

Metrics computation. We used line, branch, and mutation coverage metrics to measure the performance of the tools. We utilized JaCoCo [1] to compute the line and branch coverage metrics and PITest [2] for the mutation analysis. To ensure the feasibility of our experiments, we allocated a maximum of five minutes for mutation analysis per CUT and a timeout of one minute for each generated mutant. For classes with more than 200 mutants, we randomly sampled only 33% of them, while for classes with more than 400 mutants, we sampled 50%.

This year we have ensured to deliver the results of the competition way ahead of the final deadline so that in case any problems are found with the runs can be repeated. This proved useful for the Utbot-concolic tool, as its runs on the JSoup subject were affected by the fact the JSoup is also one of its internal dependencies.

Statistical analysis. To support the obtained results, we performed statistical tests in the exact way as in the previous edition of the competition [9].

4.2 Measuring Test Case Understandability

While high coverage and fault detection capabilities are essential indicators of test suite quality, the adoption of automatic test case generation tools in practice heavily relies on the understandability of their outputs to developers. To emphasize this often-overlooked aspect of automatically generated test cases, we introduced a new metric in this edition of the competition that quantifies their understandability. Unlike coverage metrics, understandability cannot be measured by third-party tools at runtime, but must be assessed by humans. To facilitate this assessment, we conducted a human study in which participants were provided with test cases generated by the competing tools and asked to rank them based on their understandability.

Test case selection. As part of the competition, we have generated thousands of test cases running each participant tool. Assessing each such test case in terms of understandability is infeasible due to constraints in time and the finances required to compensate the human participants. We, therefore, conducted a small study with a limited number of test cases getting evaluated. Our goal was to select one class from each subject program and compare the test cases that are testing one of the methods of this class. For each project, we selected one class and its method which we believe performs the most generic and well-known functionality, so that it does not require a lot of effort from the human participants to understand what the class and the method do. However, Kex-concolic and Kex-symbolic tools did not generate any test cases for the ta4j subject. In contrast, for JSoup there was not a single class for which all 5 competing tools generated test cases. Therefore, we had to exclude ta4j and JSoup from this study. For the classes and their method for the remaining three subjects, we detected all the test cases that have a call on the method and selected randomly only one of them. We gave preference to the test cases that have an assertion that predicates on the return value of the method. However, not all tools have generated such test cases.

The final task. The task presented to the human participant consisted of (i) three Java classes their source codes provided (ii) five test cases each from one of the five competing tools (iii) question(s) asking the participant to rank the test cases for each class in terms of understandability from the most understandable to the least understandable (iv) question(s) asking the participant to describe in natural text the behaviour of the test case they have rated the most understandable (v) question(s) asking the participant to explain why the test case they ranked the least understandable is hard to understand (vi) the final questions asking the participant whether the task had clear instructions, was easy to perform and whether an hour of time provided for the task was enough. The questions in points (iv) and (v) were used as attention questions, to ensure that

Table 2: Statistics on number of test cases generation for each tool and each time budget.

tool	timeBudget	Total # gen. tests	Median # of gen. tests
randoop	30	630,425	204
	120	2,070,129	485
kex-symbolic	30	71,395	82.5
	120	294,839	262.5
kex-concolic	30	30,021	24
	120	96,388	80
utbot-fuzzer	30	20,392	9
	120	23,281	10
utbot-concolic	30	37,302	29
	120	49,701	35
evosuite	30	54,965	35
	120	43,443	23.5

the participant understands what the test cases are doing and can justify the decision on ranking a test case at the lowest position. In cases when these explanations were not satisfactory, the data points of the participant were removed from the final dataset.

Participant Recruitment. We posted our study on Prolific Academic[3] which is a specialised crowdsourcing platform to collect research data. We have indicated the knowledge of Java and JUnit as the required skills to be allowed to participate in our study. We offered a payment of 10 GBP to each participant (Prolific Academic recommends at least 8 GBP per hour of work).

5 RESULTS OF THE JUNIT TESTING COMPETITION

5.1 Results for Structural Coverage Criteria

Table 2 shows the total and average number of generated test cases for each tool and time budget. As expected, increasing the time budget generally leads to an increase in the number of generated test cases for all tools, except for EvoSuite. This could be due to EvoSuite's minimization process, which aims to reduce the number of generated tests while maintaining coverage. With a higher time budget, EvoSuite has more time to explore the search space of possible test cases to identify those that maximize coverage. It is worth noting that both Kex-concolic and Kex-symbolic failed to generate test cases for subject *ta4j*. Figures 1, 2, and 3 report the distribution of line, branch, and mutation coverage of the tools, for all CUTs for each specific time budget. Note that we considered the median value for each CUTs, as we had 10 runs for each CUT.

Line Coverage. The tool with the lowest median line coverage is the baseline Randoop (= 38.20%). This is an expected result as the test case generation of Randoop is not guided by coverage. Interestingly, increasing the time budget from 30 to 120 seconds has no effect on the median line coverage, although it generates more test cases (see Table 2). For all the other tools the increase of time budget lead to an increase of line coverage. The tool that performed the best is EvoSuite with a median of 97.00%.

Branch Coverage. As for line coverage, the tool with the lowest median branch coverage (see Table 2) is Randoop (= 4.8%). Similarly, the increase in the test budget leads to an increase in branch

[3] https://www.prolific.co/

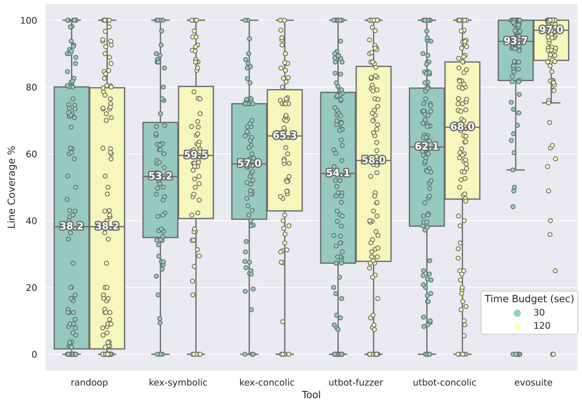

Figure 1: Line Coverage Ratio for 30 and 120 seconds.

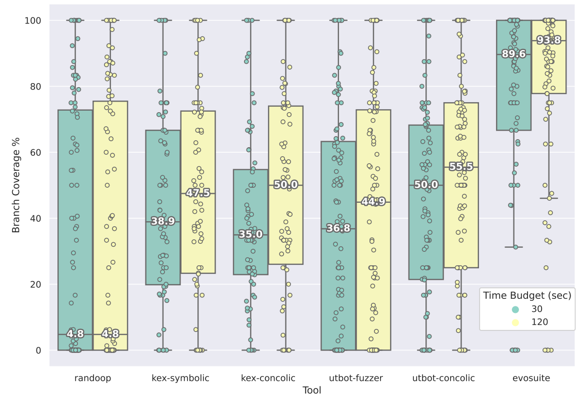

Figure 2: Branch Coverage Ratio for 30 and 120 seconds.

coverage for all tools except Randoop. The tools that achieve a median branch coverage greater or equal to 50% for at least one of the time budget are Utbot-concolic, Kex-concolic and EvoSuite.

Mutation Coverage. The mutation coverage is the ratio between the number of mutants that were killed by at least one test and the total number of mutants being generated. The performance of the tools drops in a noticeable manner when it comes to the mutation score with the median being equal to zero for all tools. Moreover, for Kex-concolic and Kex-symbolic the mutation score is zero for all subjects. Overall, EvoSuite achieves the highest mutation score, followed by Utbot-concolic.

Scores and Rankings. The formula for the score [7] has been created and improved during the previous editions of the tool competition and takes into account the line and branch coverage, the mutation score, and the time budget used by the generator. Moreover, it applies a penalty for flaky and non-compiling tests. We observed a final score of 678.12 for EvoSuite, 530.71 for Utbot-concolic, 426.19 for Randoop, 381.48 for Utbot-fuzzer, 195.09 for Kex-concolic and 128.80 for Kex-symbolic. The rankings of the tools are reported in the column *CoverageR* of Table 4.

5.2 Results for the Test Case Understandability

Our goal for the test case understandability study was to obtain rankings for the test cases from 20 participants. We evaluated each submitted response manually and checked whether the responses to the questions that require textual descriptions were properly

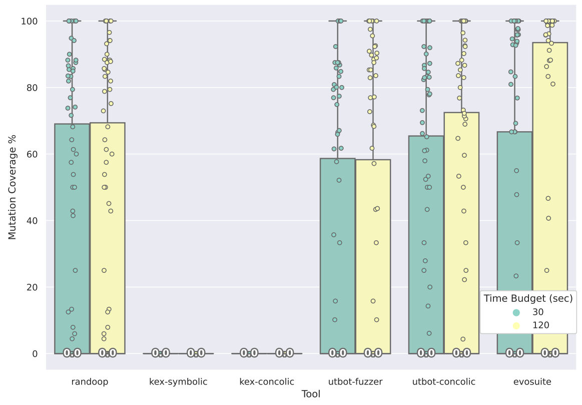

Figure 3: Mutation Coverage Ratio for 30 and 120 seconds.

Table 3: Result for Test Case Understandability Study.

Tool	LMap	Months	DUtils	Average
EvoSuite	2.38	2.08	2.23	2.23
Utbot-concolic	1.92	2.23	2.23	2.13
Utbot-fuzzer	3.54	2.77	2.69	3.00
Kex-symbolic	3.62	4.00	4.23	3.95
Kex-concolic	3.54	3.92	3.62	3.69

filled. If this was not the case, we excluded the entry from our final dataset. In our attempt to replace the rejected entries with high-quality responses, we hired 30 human participants overall. However, only 13 of the entries had the required quality of the responses and were used in our final dataset.

Table 3 reports the results of the study. Columns *LMap*, *Months*, *DUtils* report the average rankings the human participants assigned for each tool to the test cases for `LinkedMap`, `Months` and `DistanceUtils` classes accordingly. Column *Average* reports the average across the test cases for the three classes. As the results show, the tool with the highest understandability of the selected test cases is Utbot-concolic, followed by EvoSuite.

5.3 Overall Results

Given the small number of test cases being evaluated and the small number of human participants in the study to measure test case understandability, we gave the understandability score a low weight of 10%. Therefore, our final score is measured as a weighted sum between the final rankings for the coverage metrics and the understandability rankings, with the former having a weight of 0.9 and the latter having a weight of 0.1. Table 4 reports the ranking obtained based on the coverage metrics, ranking based on the understandability and the ranking based on the combination of the two. As we can see, the final ordering of the participant tools is EvoSuite, followed by Utbot-concolic, Utbot-fuzzer, Kex-concolic and Kex-symbolic.

6 CONCLUSIONS AND FINAL REMARKS

This year marks the eleventh edition of the Java Test Case Generation Competition which had 5 competing tools. In this edition, we

Table 4: Final Rankings.

Tool	CoverageR	UnderstandabilityR	OverallR
EvoSuite	1.79	2.23	1.83
Utbot-concolic	2.61	2.13	2.56
Utbot-fuzzer	3.76	3.00	3.68
Kex-symbolic	4.995	3.95	4.89
Kex-concolic	3.95	3.69	3.92

used the largest benchmark dataset and introduced a new qualitative metric to measure the understandability of the test cases. While running the tools, we encountered an issue with JaCoCo when it would produce an error if it had to instrument two classes with the same name, but coming from different dependencies. We handled this error via manual intervention this year, we plan to apply a fix that would avoid it automatically in the upcoming editions. Moreover, for the next editions, we aim to improve the current design of the understandability study, to ensure the higher quality of the responses, by coming up with more meaningful criteria to select the test cases and stricter criteria to select the participants.

ACKNOWLEDGEMENTS

We would like to thank the Google Open Source Security Team for sponsoring credits on the Google Cloud platform, which were used to run all the experiments of this competition. We would also like to thank the IEEE Technical Community on Software Engineering (TCSE) and ACM Special Interest Group on Software Engineering (SIGSOFT) for sponsoring SBFT'23.

REFERENCES

[1] 2021. JaCoCo. https://www.jacoco.org/. [Online; accessed 23-02-2021].
[2] 2021. PiTest. http://pitest.org/. [Online; accessed 23-02-2021].
[3] 2023. UnitTestBot GitHub Repo. https://github.com/UnitTestBot.
[4] 2023. UnitTestBot Web Page. https://www.utbot.org/.
[5] Azat Abdullin and Vladimir Itsykson. 2022. Kex: A platform for analysis of JVM programs. *Information and Control Systems* 1 (2022), 30–43. http://www.i-us.ru/index.php/ius/article/view/15201
[6] Matteo Biagiola, Stefan Klikovits, Jarkko Peltomaki, and Vincenzo Riccio. [n.d.]. SBFT Tool Competition 2023 - Cyber-Physical Systems Track. In *16th IEEE/ACM International Workshop on Search-Based And Fuzz Testing, SBFT 2023, Melbourne, Australia, May 14, 2023.*
[7] Xavier Devroey, Alessio Gambi, Juan Pablo Galeotti, René Just, Fitsum Kifetew, Annibale Panichella, and Sebastiano Panichella. 2021. JUGE: An Infrastructure for Benchmarking Java Unit Test Generators. https://doi.org/10.48550/arXiv.2106.07520
[8] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Transactions on Software Engineering and Methodology* 24, 2 (dec 2014), 1–42. https://doi.org/10.1145/2685612
[9] Alessio Gambi, Gunel Jahangirova, Vincenzo Riccio, and Fiorella Zampetti. 2022. SBST tool competition 2022. In *2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 25–32.
[10] Dongge Liu, Jonathan Metzman, Marcel Böhme, Oliver Chang, and Abhishek Arya. [n.d.]. SBFT Tool Competition 2023 - Fuzzing Track. In *16th IEEE/ACM International Workshop on Search-Based And Fuzz Testing, SBFT 2023, Melbourne, Australia, May 14, 2023.*
[11] Carlos Pacheco and Michael D Ernst. 2007. Randoop: Feedback-Directed Random Testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion - OOPSLA '07*, Vol. 2. ACM Press, 815. https://doi.org/10.1145/1297846.1297902
[12] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering* 44, 2 (2018), 122–158. https://doi.org/10.1109/TSE.2017.2663435