**EvoSuite at the SBST 2019 Tool Competition**

José Campos  
University of Washington  
Seattle, USA  
jmcampos@uw.edu

Annibale Panichella  
Delft University of Technology  
Delft, Netherlands  
a.panichella@tudelft.nl

Gordon Fraser  
Chair of Software Engineering II,  
University of Passau  
Passau, Germany  
gordon.fraser@uni-passau.de

**ABSTRACT**

EvoSuite is a search-based tool that automatically generates executable unit tests for Java code (JUnit tests). This paper summarises the results and experiences of EvoSuite’s participation at the seventh unit testing competition at SBST 2019, where EvoSuite achieved the highest overall score (255.43 points) for the sixth time in seven editions of the competition.

**1 INTRODUCTION**

The annual unit test generation competition aims to drive and evaluate progress on unit test generation tools. In the 7th instance of the competition at the International Workshop on Search-Based Software Testing (SBST) 2019, several JUnit test generation tools competed on a set of 38 open-source Java classes. This paper describes the results obtained by the EvoSuite test generation tool [8] in this competition. Details about the procedure of the competition, the technical framework, and the benchmark classes can be found in [21]. In this competition, EvoSuite achieved an overall score of 255.43, which was the highest among the competing and baseline tools.

**2 ABOUT EVOSUITE**

EvoSuite [8] is a search-based tool [12] that uses a genetic algorithm to automatically generate test suites for Java classes. Given the name of a target class and the full Java classpath (i.e., where to find the compiled bytecode of the class under test and all its dependencies), EvoSuite automatically produces a set of JUnit test cases aimed at maximizing code coverage. EvoSuite can be used on the command line, or through plugins for popular development tools such as IntelliJ, Eclipse, or Maven [2].

For this edition of the competition, EvoSuite was configured with Dynamic Many-Objective Sorting Algorithm (DynaMOSA) as a search algorithm. DynaMOSA [23, 24] is a recently developed many-objective algorithm that incrementally optimizes multiple coverage criteria at the same time. More specifically, different testing coverage requirements (e.g., branches) are treated as distinct, contrasting search objectives in a many-objective paradigm. Search objectives are computed using standard heuristics for code coverage, such as the branch distance and the approach level (see [12] for more details). Coverage requirements are prioritized during the search according to their structural dependencies in the control dependency graph. The search is initialized with the coverage requirements (e.g., branches) positioned higher in the hierarchy; then, the remaining requirements are incrementally inserted in later generations when their dominator requirements are covered.

**Table 1: Classification of the EvoSuite unit test generation tool**

<table>
<thead>
<tr>
<th>Prerequisites</th>
<th>Dynamic testing at the Java class level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Type</td>
<td>Java classes</td>
</tr>
<tr>
<td>Lifecycle phase</td>
<td>Unit testing for Java programs</td>
</tr>
<tr>
<td>Environment</td>
<td>All Java development environments</td>
</tr>
<tr>
<td>Knowledge required</td>
<td>JUnit unit testing for Java</td>
</tr>
<tr>
<td>Experience required</td>
<td>Basic unit testing knowledge</td>
</tr>
<tr>
<td>Input and Output of the tool</td>
<td>Bytecode of the target class and dependencies</td>
</tr>
<tr>
<td>Output</td>
<td>JUnit 4 test cases</td>
</tr>
<tr>
<td>Operation</td>
<td>Through the command line, and plugins for IntelliJ, Maven and Eclipse</td>
</tr>
<tr>
<td>User guidance</td>
<td>Manual verification of assertions for functional faults</td>
</tr>
<tr>
<td>Source of information</td>
<td><a href="http://www.evosuite.org">http://www.evosuite.org</a></td>
</tr>
<tr>
<td>Maturity</td>
<td>Mature research prototype, under development</td>
</tr>
<tr>
<td>Technology behind the tool</td>
<td>Search-based testing / many-objective optimization</td>
</tr>
<tr>
<td>Obtaining the tool</td>
<td>License: Lesser GPL V.3</td>
</tr>
<tr>
<td></td>
<td>Cost: Open source</td>
</tr>
<tr>
<td></td>
<td>Support: None</td>
</tr>
</tbody>
</table>

Does there exist empirical evidence about Effectiveness and Scalability: See [12, 13]
DynaMOSA evolves test cases [23], which correspond to the chromosomes for the search. Each test case consists of variable-length sequences of Java statements (e.g., primitive statements and calls on the class under test). A population of randomly generated test cases is evolved using search operators (e.g., selection, crossover and mutation) to iteratively produce test cases that are closer to satisfy the target coverage requirements. The final test suite is formed by the shorter test cases encountered during the search, and that satisfy the coverage requirements. These tests are stored in an external data structure, called archive [22, 23].

EvoSuite can be configured to satisfy for multiple coverage criteria at the same time, and the default configuration combines branch coverage with mutation testing [14] and other basic criteria [25]. Once the search is completed, EvoSuite applies various post-search optimizations aimed to improve the readability of the generated tests. For example, tests are minimized and test assertions are selected using mutation analysis [18]. For more details on post-process optimization we refer to [8, 9].

In the past, the effectiveness of EvoSuite has been evaluated on open source as well as industrial software in terms of code coverage [7, 13, 23, 27], fault finding effectiveness [1, 29], and effects on developer productivity [17, 26] and software maintenance [30]. EvoSuite has a longstanding record of success at the unit testing tool competition, having ranked second in the third edition of the competition [15] and first in all the other editions [10, 11, 16, 19, 20].

3 COMPETITION SETUP

The configuration of EvoSuite for the 2019 competition is largely based on its default values since these have been tuned extensively [4]. We used the default set of coverage criteria [25] (e.g., line coverage, branch coverage, branch coverage by direct method invocations, weak mutation testing, output coverage, exception coverage). EvoSuite uses an archive of solutions [27] to keep the search focused on uncovered goals, iteratively discarding covered goals and storing the tests that covered them. After a certain percentage of the search budget has passed and with a certain probability, EvoSuite starts using mock objects (relying on Mockito) instead of actual class instances [3] for dependencies; only branches that cannot be covered without mocks lead to tests with mock objects. As in the previous instance of the competition, we continue to use frequency-based weighted constants for seeding [28] and support Java Enterprise Edition features [5]. EvoSuite is actively maintained, therefore several bug fixes and minor improvements have been applied since the last instance of the competition, in particular regarding non-determinism and flaky tests [6], as well as new types of test assertions (e.g., related to arrays and standard container classes).

Like in previous instances of the competition, we enabled the post-processing step of test minimization—not for efficiency reasons, but because minimized tests are less likely to break. To reduce the overall time of test generation we included all assertions rather than filtering them with mutation analysis [18], which is computationally expensive. The use of all assertions has a negative impact on readability, but this is not evaluated as part of the SBST contest.

Four time budgets were used to call each tool: 10, 60, 120 and 240 seconds. We used the same strategy used in previous competition (e.g., [16]) to distribute the overall time budget onto the different phases of EvoSuite (e.g., initialisation, search, minimization, assertion generation, compilation check, removal of flaky tests). That is, 50% of the time was allocated to the search, and the rest was distributed equally to the remaining phases.

4 BENCHMARK RESULTS

On the whole, the performance of EvoSuite was in line with previous results, although EvoSuite seemed to fail on a notably large number of classes. In particular, EvoSuite failed to produce any test suites for benchmarks DUBBO-2, SPOON-169, SPOON-25, and WEBMAGIC-4, and generally struggled with most AUTHZFORCE benchmarks (highlighted in grey in Table 2).

A closer look at the benchmarks DUBBO-2 and WEBMAGIC-4 reveals that there were missing dependencies in the competition setup [21] (similar to cases in previous years [20]), which explains why EvoSuite was incapable of generating any test: When a dependency of the class under test is missing, then EvoSuite intentionally aborts with an error to inform the user of the configuration error. While this is desirable behavior during regular usage, arguably in the scope of the competition, EvoSuite could try to ignore such errors and try to generate tests nevertheless, as it might still be possible to cover code even with some missing dependencies. For the benchmark FASTJSON-3 and a 10s time budget, EvoSuite generated some test suites, but due to a bug in the competition infrastructure, no data was collected.

For the benchmarks SPOON-169 and SPOON-25, EvoSuite either failed to generate a compilable test suite or crashed due to a bug in DynaMOSA. Although EvoSuite’s minimization could help at reducing the number of un compilable / flaky test cases, for SPOON-169 all minimizations attempts ran out of time.

For the AUTHZFORCE benchmarks, although EvoSuite managed to generate some test suites, most executions for data collection ended with a failure. Specifically, for these classes, an exception was raised for test execution for all tools in the competition. However, while this exception was not problematic for other tools, it caused an inconsistent state of the JVM which led to the @Before setup methods of EvoSuite to fail (the @Before method was not able to configure EvoSuite’s custom security manager because it was never correctly removed). As a consequence, all test executions failed. We were not able to reproduce the error in the competition infrastructure, and thus cannot conjecture about the causes. The same problem also occurred for SPOON-253. While in principle it would be possible to extend EvoSuite to try to ignore such errors and keep on executing tests, as explained it is EvoSuite’s philosophy that the user should be made aware of configuration errors, and hence such an extension would only serve to tune EvoSuite for the competition.

In general, EvoSuite often struggles with a time budget as low as 10 seconds. For example, Table 2 reports “-” for benchmarks FASTJSON-1, FASTJSON-3, FESCAR-41, FESCAR-42, and SPOON-105. Surprisingly, for benchmarks SPOON-155, SPOON-16, SPOON-20, and SPOON-211, EvoSuite successfully generated test suites for small-time budgets, but it failed to do it for large time budgets. For all, EvoSuite crashed due to a bug in the DynaMOSA algorithm which had not been revealed by our previous experiments.
Consequently, even though DynaMOSA is known to increase the coverage, EvoSuite achieves substantially in general, this increase was countered with some missing data points. As always, the competition was helpful in improving the robustness of the tool, by revealing this and other bugs.

5 CONCLUSIONS

This paper reports on the participation of the EvoSuite test generation tool in the 7th SBST Java Unit Testing Tool Contest. With an overall score of 255.43 points, EvoSuite achieved the highest score of all tools in the competition.

To learn more about EvoSuite, visit our Web site:

http://www.evosuite.org

Acknowledgments: Many thanks to all the contributors to EvoSuite. This project has been funded by the EPSRC project “GREATEST” (EP/N023978/2), and by the National Research Fund, Luxembourg (FNR/P10/03).

REFERENCES


