UtBot at the SBST2021 Tool Competition

1st Dmitry Ivanov
Software Analysis Team
Huawei, Russian Research Institute
St. Petersburg, Russia
dmitry.ivanov@huawei.com

2nd Nikolay Bukharev
Software Analysis Team
Huawei, Russian Research Institute
St. Petersburg, Russia
bukharev.nikolay@huawei.com

3rd Alexey Menshutin
Software Analysis Team
Huawei, Russian Research Institute
St. Petersburg, Russia
menshutin.alexey@huawei.com

4th Arsen Nagdalian
Software Analysis Team
Huawei, Russian Research Institute
St. Petersburg, Russia
nagdalian.arsen@huawei.com

5th Gleb Stromov
Software Analysis Team
Huawei, Russian Research Institute
St. Petersburg, Russia
stromov.gleb@huawei.com

6th Artem Ustinov
Software Analysis Team
Huawei, Russian Research Institute
St. Petersburg, Russia
ustinov.artem1@huawei.com

***Abstract*—UtBot is an automatic test generator for Java programs developed by Huawei and based on symbolic execution. It tries to cover as many branches as possible using the program's bytecode. To do that UtBot analyzes paths in the control flow graph of a given method, constructing constraints for them, and tries to find satisfying input values using SMT-solver to cover corresponding branches. In this paper, we report the results of UtBot at the ninth edition of the SBST 2021 tool competition.**

I. Introduction

Regression testing is a well-known and commonly used technique that allows developers to make sure their changes don't break the existing logic. To simplify the process of writing tests, we are developing an automatic test generator named UtBot, which produces tests for a given Java program. It uses the program's control flow graph to collect constraints for each possible execution path and tries to cover them with test cases. As we can see from the results of the SBST 2021 tool competition [3], UtBot gives the best coverage and mutation score of all symbolic (and concolic) engines that have participated in the contest.

II. UtBot description

UtBot is a symbolic execution based test generation tool. It automatically produces a set of JUnit tests for a given class or its methods. To do that, the tool requires a Java classpath with all compiled dependencies. UtBot provides an IntelliJ plugin along with a command-line interface. Table I contains the main information about the generator.

TABLE I
Main information about UtBot

Prerequisites	
Static or dynamic	Static
Software type	Java classes
Lifecycle phase	Unit testing for Java program
Environment	Java 8/Java 11
Knowledge required	JUnit 4
Experience required	No specific experience required
Input and output of the tool	
Input	Classpath with all dependencies
Output	Set of test cases
Operation	
Interaction	CLI, IntelliJ plugin
User guidance	—
Source of information	—
Maturity	Being developed since September 2020
Technology behind the tool	Symbolic execution
Obtaining the tool and information	
License	Proprietary prototype
Cost	—
Support	None
Empirical evidence about the tool	
Effectiveness, Scalability	—

III. Implementation details

Since UtBot has a symbolic execution engine at its core, it has to analyze multiple execution paths extracted from the program's bytecode. We use Jimple [2] provided by Soot to get a more simple representation of bytecode significantly decreasing the number of instructions. Having obtained a control flow graph (CFG) from the bytecode, we use it to get all the possible execution paths. For each of them, we build a set of constraints that are used by z3 (SMT-solver by Microsoft [1]) to determine if our path conditions are satisfiable. If they are, we can construct input values from the found model and create a test case that covers the desired branch of the program.

A few words about our memory and type systems. We heavily rely on the array and bit-vectors theories to represent symbolic memory. Every Java object in our program is represented by its address and type. Class fields are stored in z3 arrays that contain either addresses or values of objects' fields depending on whether a field is a reference value or not. It allows us to process cases where some objects are equal by reference (so-called memory aliasing). All objects

TABLE II
RESULTS OF UTBOT ON THE SBST BENCHMARK CLASSES

Benchmark	Java Class	Line Coverage			Branch Coverage			Mutants Coverage			Mutants Killed		
		60s	120s	300s	60s	120s	300s	60s	120s	300s	60s	120s	300s
GUAVA-46	com.google.common.collect.MinMaxPriorityQueue	24.7%	30.2%	30.2%	25.8%	30.2%	30.2%	38.3%	42.7%	42.7%	38.3%	42.7%	42.7%
GUAVA-134	com.google.common.collect.Ordering	19.0%	23.0%	23.0%	12.0%	25.0%	25.0%	22.3%	29.9%	29.9%	22.3%	29.9%	29.9%
GUAVA-181	com.google.common.cache.CacheStats	81.0%	81.0%	81.0%	100.0%	100.0%	100.0%	96.1%	96.1%	96.1%	96.1%	96.1%	96.1%
GUAVA-128	com.google.common.net.MediaType	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
GUAVA-71	com.google.common.escape.Escapers	78.6%	78.6%	78.6%	66.7%	66.7%	66.7%	90.9%	90.9%	90.9%	90.9%	90.9%	90.9%
GUAVA-273	com.google.common.collect.TreeRangeMap	8.6%	13.7%	13.0%	3.7%	8.0%	7.7%	9.5%	14.8%	14.6%	9.5%	14.8%	14.6%
GUAVA-254	com.google.common.hash.HashCode	65.1%	65.1%	65.1%	50.0%	50.0%	50.0%	56.1%	56.1%	56.1%	56.1%	56.1%	56.1%
GUAVA-237	com.google.common.hash.Hashing	23.9%	34.1%	34.1%	21.0%	30.0%	30.0%	29.6%	37.5%	37.5%	29.6%	37.5%	37.5%
GUAVA-192	com.google.common.reflect.TypeToken	5.9%	7.0%	7.0%	2.6%	2.6%	2.6%	7.1%	8.2%	8.2%	7.1%	8.2%	8.2%
GUAVA-200	com.google.common.util.concurrent.MoreExecutors	5.2%	6.2%	13.0%	2.2%	3.2%	5.1%	6.3%	8.0%	13.7%	6.3%	8.0%	13.7%
GUAVA-108	com.google.common.collect.Range	38.0%	46.3%	48.1%	21.7%	42.3%	47.6%	43.7%	58.4%	62.1%	43.7%	58.4%	62.1%
GUAVA-11	com.google.common.collect.MapMaker	31.9%	31.9%	31.9%	50.0%	50.0%	50.0%	56.1%	56.1%	56.1%	56.1%	56.1%	56.1%
Average		31.8%	34.8%	35.4%	29.6%	34.0%	34.6%	38.0%	41.6%	42.3%	38.0%	41.5%	42.3%

are represented in the form of constraints on the cells of arrays of symbolic memory corresponding to their fields. Those constraints are stored in the SMT-solver. The type system also uses the described symbolic memory.

We strive to cover only reachable instructions of the source code. That is why we have to run our tests to make sure we have generated test cases with valid assertions. Due to a similar reason, we have decided not to use mocks. Tests that employ mocking may potentially cover unreachable code, which would increase code coverage but does not make any sense in terms of software testing.

IV. BENCHMARK RESULTS

Table II demonstrates the results of UtBot achieved on the competition's benchmarks [3] for each of the given time budgets. The produced coverage values vary for different classes. In some cases, coverage reaches the level of $80-100\%$. In other cases, the results are lower. In the following sections, we describe the issues that could lead to the low coverage values for some of the classes.

A. Time limitations

An important factor is the limitations imposed by the time budgets. The symbolic execution technique is inherently time-consuming and that is why in many cases it requires more time in order to achieve better results. To mitigate this problem we have decided to split time budgets into two equal parts. The first part is equally distributed between all methods so that each of them gets some portion of the time. The second part is used to run tests for methods in an arbitrary order.

B. Technical issues

First of all, the results table states that all coverage metrics for the class `com.google.common.net.MediaType` are zero. However, we could not reproduce this result. In our local launches, we have achieved 25% branch coverage for this class. To calculate coverage we applied the JaCoCo Gradle plugin and used it to automatically generate HTML coverage reports each time our test classes are launched.

One other aspect that affects the time consumption of our engine is the requests to the SMT-solver. In some cases they can take significant time, so we have made a temporary decision to use timeouts for each request.

Another issue that could have caused some problems involves static fields. Although it will be fixed once the engine starts providing the code generator with certain static fields-specific information.

There are several other bugs in our tool that we are aware of. Some of them have already been fixed and the rest will also be corrected shortly.

V. CONCLUSION

Although UtBot has shown the best result of all participants using symbolic execution technique, there is still a vast amount of opportunities to improve our tool in the future. Right now the bottleneck of the engine is its performance and our primary goal is to speed up its work. To do it, we can include concrete execution into our analysis to make our engine concolic.

REFERENCES

[1] Getting started with z3: A guide. https://rise4fun.com/z3/tutorial/guide. Accessed: 2021-03-10.

[2] Arni Einarsson and Janus Dam Nielsen. A survivor's guide to java program analysis with soot. 2008.

[3] Sebastiano Panichella, Alessio Gambi, Fiorella Zampetti, and Vincenzo Riccio. Sbst tool competition 2021. In *International Conference on Software Engineering, Workshops, Madrid, Spain, 2021*. ACM, 2021.