UTBot Java at the SBST2022 Tool Competition

Dmitry Ivanov
Huawei Research
St. Petersburg, Russia
dmitry.ivanov@huawei.com

Alexey Menshutin
Huawei Research
St. Petersburg, Russia
alexey.menshutin@huawei.com

Denis Fokin
Huawei Research
St. Petersburg, Russia
denis.fokin@huawei.com

Yury Kamenev
Huawei Research
St. Petersburg, Russia
kamenev.yury@huawei.com

Sergey Pospelov
Huawei Research
St. Petersburg, Russia
pospelov.sergey@huawei.com

Egor Kulikov
Huawei Research
St. Petersburg, Russia
kulikov.egor@huawei.com

Nikita Stroganov
Huawei Research
St. Petersburg, Russia
stroganov.nikita@huawei.com

ABSTRACT

UTBotCpp and UTBot Java [3] are automatic white-box test generators for C/C++ and Java programs correspondingly. The tools were developed by Huawei and are based on symbolic and concrete execution. They try to cover as many branches as possible using program bytecode. For this purpose, UTBot tools analyze paths in the control flow graph of a given method, construct constraints for them, and try to find satisfying input values using SMT-solver to cover corresponding branches. In this paper, we report the results of UTBot Java at the tenth edition of the SBST 2022 tool competition.

ACM Reference Format:
Dmitry Ivanov, Alexey Menshutin, Denis Fokin, Yury Kamenev, Sergey Pospelov, Egor Kulikov, and Nikita Stroganov. 2022. UTBot Java at the SBST2022 Tool Competition. In *The 15th Search-Based Software Testing Workshop (SBST'22), May 9, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3526072.3527529

1 INTRODUCTION

The main concepts behind UTBot were described in the previous article [4]. This year, there were presented two versions of UTBot – with and without mocks, that took third and sixth places correspondingly. The most important features developed throughout the year and the problems we found during the preparation for SBST2022 will be described in the sections below. The full results of the competition can be found in the main article [1].

2 UTBOT DESCRIPTION

UTBot is a symbolic execution based test generation tool. It analyzes paths in the control flow graph of the Method Under Test (MUT), collects encountered path constraints and constructs appropriate test methods, using information from a model calculated by STM-solver. There are important improvements compared to last year's version, such as concrete execution, symbolic mocks and other improvements, that helps UTBot to achieve better results.

SBST'22 , May 9, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9318-8/22/05…$15.00
https://doi.org/10.1145/3526072.3527529

3 IMPLEMENTATION IMPROVEMENTS

Two versions of the product were sent to the competition: with and without mocks, **UTBot-concrete** and **UTBot-mocks** correspondingly. The first one *does not mock anything* except several predefined classes and runs concrete execution at the end. Meanwhile, the second one *mocks everything* that does not belong to the Class Under Test (CUT), but it does not run concrete execution. Now there will be a short description of the concepts behind these versions.

3.1 Symbolic mocks

During the analysis, the engine mocks objects and functions depending on the mock strategy and remembers information, needed to restore required behaviour later during the test generation using the Mockito framework. At the competition, UTBot-mocks used the strongest mock strategy – mock all the objects and methods outside of the CUT, while UTBot-concrete mocked several predefined classes. These classes either have flaky behaviour (i.e., Random), or there is no reason to analyze them since they do not affect the behaviour of the MUT (for example, as Logger).

3.2 Concrete execution

Concrete execution is a very important technique, that accompanies symbolic execution. It runs MUT concretely in a child process. In UTBot it is used in many places: it generates information for assertions using values extracted from a calculated result to eliminate possible path divergence[1] at the end of the analysis, runs whenever symbolic engine gets stuck, minimizes the resulting set of tests. Unfortunately, concrete execution was disabled in UTBot-mocks during the competition due to performance problems. We

[1] when the result of the symbolic analysis is different from the real one

already fixed them and the concrete execution is enabled in the latest version of the engine.

3.3 Default case

An important goal is to prevent situations when some method does not have any generated test cases for it, which might happen because of insufficient time for symbolic analysis. For this purpose, at the beginning of the test generation, concrete execution runs each method with default parameters values and produces corresponding test methods.

3.4 Wrappers

Since classes of the standard library are quite complicated for analysis (such as Map, Set, List, String, etc.), so-called *wrappers* were implemented – classes' approximations with the same behaviour, that contains special constructions to make analysis easier.

4 BENCHMARK RESULTS

At the competition, UTBot-concrete took third place and UTBot-mocks took sixth [1]. However, there was a problem during collecting the results – UTBot-mocks, as the name suggests, produces code with mocks, but many classes (45 out of 65) were marked as "uncompilable" because of their presence. This happened because there was no Mockito-inline framework in the classpath, and we did not discover the problem till it was too late to fix it. Moreover, the problem with dependencies affected the results of UTBot-concrete as well, because it mocks some classes like Random regardless of the mock strategy – there were 19 classes with mocks that got zero coverage in 106 runs (out of 650), six of which have never been compiled. Their average line and branch coverage are up to 28.8% and 19.6% correspondingly, that increased overall line coverage to 46.8% and branch coverage to 39.2% with a 120*s* time limit in case of providing a valid classpath.

Since many correct classes have zero mean coverage in the results from the main article [1], UTBot-mocks' results were recalculated using files generated at the competition. These classes can be found at [2] alongside with coverage report (per class and average). The final results (from the article [1] for UTBot-concrete and from the reports in github repository [2] for UTBot-mocks) are presented in the Table 1. Note that every class with compilation error has zero coverage and tests that might cause JVM crashes were removed.

5 PROBLEMS

During the preparation, four main problems had revealed.

5.1 Time and other limitations management

For many classes, the engine stops generations long before the given time limit. At the beginning of the analysis, the time budget is divided equally for each MUT. When method analysis is finished, the left part of the time budget returns to the common pool and the time limit for each method recalculates using this additional time. If there are complicated methods at the beginning and easy methods at the end, there will be huge leftovers of time when the queue of the methods to analyze become empty.

Table 1: UTBot Java results

	Line coverage		Branch coverage		Mutant coverage	
	30s	120s	30s	120s	30s	120s
Concrete	42.3%	44.1%	33.6%	37.4%	23.9%	25.8%
Mocks	43.2%	43.6%	33.2%	34.8%	14.7%[a]	15.6%[a]

[a]Mutant coverage for UTBot-mocks was not recalculated with required dependencies, so the results in the table might be lower than the real one

5.2 Bugs in concrete execution

There were bugs in the concrete executor that had been fixed before the competition but a problem with performance. Sometimes concrete execution for mocks took too much time, so it has been decided to send two separate tools – with concrete execution and without mocks and vice versa. Currently, this problem has been already fixed.

5.3 JVM Crashes

Because of the bugs in the concrete execution, if it fails, a result generated by the symbolic virtual machine is used for the test generation. Sometimes concrete execution produces tests that might crash JVM (for example, methods for file stream processing or work with files). Now there is no need for such fallback, moreover tests that crash JVM are detected by the concrete executor and form a separate test suit.

5.4 Compilation errors

Generating compilable code is a serious challenge. Some errors have been found only at the competition: our internal classes in the code, wrong names for an object of Null type, ambiguous calls and others. Because of such errors, the generated tests did not produce any coverage for a lot of classes and the total score is less than UTBot might achieve.

6 CONCLUSION

UTBot has shown a big growth since last year's competition, not only because of the results but also because of the greatly increased stability. For the second year in a row, it has the best result among the tools using symbolic execution. There are clear ways for improvement, such as transformation into a concolic engine, combining concrete and mocks strategies, path selector improvements and many others. Coupled with the result obtained at the competition it gives confidence in the great prospects of the UTBot.

REFERENCES

[1] Alessio Gambi, Gunel Jahangirova, Vincenzo Riccio, and Fiorella Zampetti. 2022. SBST Tool Competition 2022. In *15th IEEE/ACM International Workshop on Search-Based Software Testing, SBST 2022, Pittsburgh, PA, USA, May 9, 2022.*
[2] Huawei. 2022. UTBot Java SBST2022. https://github.com/Software-Analysis-Team/UTBotJava-SBST-2022.
[3] Huawei. 2022. UTBot main page. https://github.com/UnitTestBot.
[4] Dmitry Ivanov, Nikolay Bukharev, Alexey Menshutin, Arsen Nagdalian, Gleb Stromov, and Artem Ustinov. 2021. UtBot at the SBST2021 Tool Competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST).* 34–35. https://doi.org/10.1109/SBST52555.2021.00015