TU Delft

Delft University of Technology

Beyond Unit-Testing in Search-based Test Case Generation: Challenges and Opportunities

Panichella, Annibale

DOI
10.1109/SBST.2019.00010

Publication date
2019

Published in
2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)

Citation (APA)
Panichella, A. (2019). Beyond Unit-Testing in Search-based Test Case Generation: Challenges and Opportunities. In *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)* (pp. 7-8). Article 8812175 IEEE. https://doi.org/10.1109/SBST.2019.00010

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.

Beyond Unit-Testing in Search-based Test Case Generation: Challenges and Opportunities

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

***Abstract*—Over the last decades, white-box search-based techniques have been applied to automate the design and the execution of test cases. While most of the research effort has been devoted to unit-level testing, integration-level test case generation requires to solve several open challenges, such as the combinatorial explosion of conditions or pre-condition failures. This paper summarizes these challenges in white-box testing and highlights possible research directions to overcome them.**

***Index Terms*—Test Case Generation, Search-Based Software Testing, Integration Testing, White-box Testing**

I. Introduction

Search algorithms have been widely applied to automate the process of generating and executing unit-level tests. The maturity of the field is proved by surveys (e.g., [14]), large-scale studies (e.g., [9], [10]), and tool competitions (e.g., [15], [16]). Prior studies showed that unit-test generation allows achieving high code coverage [9], [15], [16], detect real-bugs (e.g., [17]) and reduce the cost of debugging (e.g., [8], [18]) compared to manually-written tests. Despite these noteworthy results in unit-testing, existing search-based approaches mostly rely on black-box strategies (e.g., model-driven testing, input/output diversity) when moving toward integration and system-level testing. This is due to technical and research challenges related to white-box testing when integrating single units. In this paper, we discuss the most prominent yet open challenges as well as viable solutions to overcome them.

II. Open Challenges

A. Challenge 1: The Path and Condition Explosion Problem

When integrating multiple components, software testers have to decide in which order the classes should be integrated and tested. This problem is also known as *integration test order*: find the order that reduces the test stub cost. Search algorithms have been applied to solve this problem (e.g., [1]). Generating tests at integration level in a white-box fashion require (i) to solve the integration test order problem; (ii) to generate input data that satisfies the precondition of the first class in the order; (iii) to generate assertions for the output of the last class in the order (oracle problem). Another important step is to decide when to stop testing, i.e., when the integrated classes are adequately tested. Zhenyi and Offutt [3] introduced structural criteria for integration testing, such as *all-coupling uses* and *all-coupling paths*. The state of one path executed in a class `A` can impacts the state of a called class `B`. Therefore, the number of paths to consider grows (explodes) with the number and size of the classes to integrate. This poses a challenge to the scalability of test case generation.

***Direction**: focusing on a subset of all-coupling paths.*

To overall idea is to focus on specific paths of interests rather than all-coupling paths. For example, let us assume we want to integrate and test the classes `A` and `B`, where the former calls some methods of the latter. We may test the case `A` calls `B` by satisfying its preconditions, and the case where such pre-conditions are not satisfied. An interesting example is represented by testing Application Programming Interface (API) uses. APIs misuses [4] are commons as clients may invoke the APIs violating its implicit preconditions. Interesting paths then are those where the client class invokes the APIs without verifying whether the data passed to the APIs satisfy or not its implicit precondition. Static analysis would play a relevant role in the identification of these paths of interests.

B. Challenge 2: Integration with DataBases

Databases are commonly used to manage and store data (e.g., medical data) in modern applications. It is very common to have SQL queries within the traditional code. In the example shown in Listing 1, an SQL query is used within Java code to retrieve the credential of a user from an SQL database. First, the connection to the database is established; then a prepared statement is filled with the `username` and `password`; finally, the output of the query (variable `res`) is used within the Java code. Reaching 100% of branch coverage in the example of Listing 1 requires to initialize the database with valid and useful data such as the results of the query execution satisfy the conditions in the underneath Java code. However, generating test data for both Java code and SQL databases remains an open problem [2]. On the other hand, recent work focused on testing SQL queries in isolation [5] using search algorithms and with the goal of satisfying structural criteria for query coverage.

***Direction**: Unifying the coverage criteria for database queries and traditional (e.g., Java) code.*

Branch and decision coverage criteria have been defined for both traditional code and database queries. When queries appear within Java code, the corresponding control flow graph (CFG) can be enhanced by including the CFG of the query as well. Then, the search should be guided by combining the coverage heuristics for the Java code (e.g., approach level and branch distance) with the heuristics for query coverage (step level and step distance [5]). To speed the search, in-memory

database engines could be used (as done in [5]) by mocking the connection to SQL databases, which are slower and more expensive to set up.

Listing 1. SQL query within Java code

```
protected boolean login(String username,
                        String password) {
...
PreparedStatement stmt =
   connection.prepareStatement("SELECT * "+
      FROM User where userId=? AND psw=?");
stmt.setString(1, username);
stmt.setString(2, password);
stmt.executeQuery();
ResultSet res = stmt.getResultSet();
if (res !=null) {...}
}
```

C. Challenge 3: External Files with Content

Data can also be stored in external files, e.g., XML or JSON files. EvoSuite uses functional mocks when the class under test contains environmental calls to read/write files [7]. In these cases, calls to external files can be replaced with virtual calls that mimic the behavior of the environment [7]. However, the class under test might need files with specific content to satisfy some branch conditions. In these cases, a simple mocking does not help to cover the CUT [8].

***Direction**: Inferring the content of external files from the constants in the source code.*

XML and JSON files are characterized by a well-established and documented open standards. Generating files with XML (or JSON) format can be viewed as a grammar inference problem where the grammar rules are related to the standard format while the terminals (strings) can be potentially extracted from the source code of the CUT through static analysis. Indeed, strings appearing in the source code should be likely contained in the external files (constant seeding). However, the generated files must be well formatted, and this represents a critical constraint to the file generation problem. Rather than creating external files, more advanced functional mocking should be used to virtualize not only file system calls but also the content of such files.

D. Challenge 4: Resource Usage

Integration-level tests are usually more expensive to run than unit-level tests. Therefore, an important angle to consider is to generate test cases that consume fewer resources (CPU, memory and energy usage) at the same level of code coverage. Early attempts to reduce the resource usage of generated unit-tests have been discussed and investigated in the related literature [9], [10]. However, new follow-up studies are needed as search-algorithms in test case generation have become more sophisticated and more effective in the last decade [9], [10]. We also need further studies in the context of integration-level test generation where the problem of efficient resource usage is more critical.

***Direction**: using performance testing techniques within the search process.*

Prediction methods and risk analysis techniques have been used in unit (e.g., [19] and performance testing (e.g., [13]). They represent viable solutions for test case generation tools where direct measurements are infeasible due to their massive overhead.

III. Conclusion

Noteworthy progress has been made in search-based unit test generation research. However, moving from unit-level to integration-level test generation requires to solve several open research and technical challenges. In this paper, we discuss some of the most critical challenges and highlights possible research directions to address such challenges.

References

[1] L. Briand et al. “An investigation of graph-based class integration test order strategies.” IEEE Transactions on Software Engineering, vol. 29, issue 7, 2003, pp. 594–607.

[2] A. Arcuri.“RESTful API Automated Test Case Generation with EvoMaster.” ACM Transactions on Software Engineering and Methodology (TOSEM) vol. 28, issue 1, 2019, pp. 1–37.

[3] J. Zhenyi, and J. Offutt, “Coupling-based criteria for integration testing.” Software Testing, Verification and Reliability, vol. 8, issue 3, 1998, pp. 133–154.

[4] A. Amann et al. “A Systematic Evaluation of Static API-Misuse Detectors.” IEEE Transactions on Software Engineering, 2018.

[5] J. Castelein et al. “Search-based test data generation for SQL queries.” In Proceedings of the 40th International Conference on Software Engineering (ICSE), 218, pp. 1220–1230.

[6] M. Emmi et al. “Dynamic test input generation for database applications”. Proceedings of the international symposium on Software testing and analysis, 2007.

[7] A. Arcuri et al. “Private api access and functional mocking in automated unit test generation.” IEEE international conference on software testing, verification and validation (ICST), 2017.

[8] M. Soltani et al. “Search-Based Crash Reproduction and Its Impact on Debugging.” IEEE Transactions on Software Engineering, 2018.

[9] A. Panichella et al. “A large scale empirical comparison of state-of-the-art search-based test case generators.” Information and Software Technology, vol. 104, 2018, pp.236–256.

[10] J. Campos et al. “An empirical evaluation of evolutionary algorithms for unit test suite generation”. Information and Software Technology, vol. 104, 2018, pp. 207–235.

[11] K. Lakhotia et al. “A multi-objective approach to search-based test data generation,” The 9th International Conference on Genetic an Evolutionary Computation (GECCO), 2007.

[12] J. Ferrer et al. “Evolutionary algorithms for the multi-objective test data generation problem.” Software: Practice and Experience, vol. 42, issue 11, 2012, pp. 1331–1362,

[13] P. Huang et al. “Performance Regression Testing Target Prioritization via Performance Risk Analysis,” in Proceedings of the 36th International Conference on Software Engineering (ICSE), 2014.

[14] P. McMinn. “Search-based software test data generation: a survey” Software testing, Verification and reliability, vol. 4, issue 2, 2004, pp. 105–156.

[15] U. R. Molina et al. “Java unit testing tool competition-sixth round.” IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST), 2018.

[16] A. Panichella, U. R, Molina. “Java unit testing tool competition-fifth round.” IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST), 2017.

[17] A. M. Moein et al. “An industrial evaluation of unit test generation: Finding real faults in a financial application.” The 39th International Conference on Software Engineering: Software Engineering in Practice Track, pp. 263–272, 2017.

[18] S. Panichella et al. “The impact of test case summaries on bug fixing performance: An empirical investigation.” The 38th International Conference on Software Engineering (ICSE), pp. 547–558, 2016.

[19] G. Grano et al. “Branch coverage prediction in automated testing.”, Journal of Software: Evolution and Process, 2019.