Compressing Automatically Generated Unit Test Suites Through Test Parameterization

Aidin Azamnouri, Samad Paydar

Compressing Automatically Generated Unit Test Suites through Test Parameterization

Aidin Azamnouri and Samad Paydar

Computer Engineering Dept., Ferdowsi University of Mashhad, Mashhad, Iran
aidin.noori@mail.um.ac.ir
s-paydar@um.ac.ir

Abstract. Test maintenance has recently gained increasing attention from the software testing research community. When using automated unit test generation tools, the tests are typically created by random test generation or search-based algorithms. Although these tools produce a large number of tests quickly, they mostly seek to improve test coverage; overlooking other quality attributes like understandability and readability. As a result, maintaining a large and automatically generated test suite is quite challenging. In this paper, by utilizing a high level of similarity among the automatically generated tests, we propose a technique for automatically abstracting similar tests through transforming them into parameterized tests. This approach leads to the improvement of readability and understandability by reducing the size of the test suite and also by separating data and logic of the tests. We have implemented this technique as a plugin for IntelliJ IDEA and have evaluated its performance over the test suites produced by the Randoop test generation tool. The results have demonstrated that the proposed approach is able to effectively reduce the size of the test suites between 11% and 96%, with an average of 66%.

Keywords: automated test generation · maintainability · readability · unit test · parameterized unit tests · test suites · Randoop

1 Introduction

Unit testing is one of the test levels that has received considerable attention both from the academia and the industry practitioners. However, since each unit test targets a small scope of the software under test (SUT), it is required to generate a large number of unit tests to have the SUT appropriately tested. Furthermore, while the source code of the SUT is evolving, unit tests need to be maintained; otherwise, they may become obsolete or broken. These factors contribute to increasing the cost and complexity of unit testing, emphasizing the need for the development of automated techniques for unit test generation and maintenance.

Automated unit test generation has been an active field of research in the domain of software testing during the last decades. In this regard, different techniques and approaches have been introduced, one of the main characteristics of

which is employing a predefined iterative algorithm rooted in random testing techniques, either directly or indirectly, i.e. through search-based techniques, where the main focus is on achieving a high level of test coverage. This tendency towards improving test coverage has set the stage for overlooking some other important quality attributes, one of which is maintainability of the tests. This can be attributed to the following observations: 1) automated unit test generation techniques are capable of generating a large number of tests within a short time budget. The larger the test suite, the more challenging it is to read, understand and maintain the tests, 2) these techniques use a very naive schema for naming test methods, e.g. `test01`, `test02`, and test classes, and as a result they fail to communicate the semantic of the test. Hence, reading the body of a test method is the only way to understand what it does, which increases the cost of test maintenance, and 3) the body of the test methods generated by these techniques are usually very long, causing the readability and understandability to be reduced, and the maintainability of the tests to be affected as a result.

Based on these observations, in this paper, we propose a technique that automatically factorizes the similarities in a JUnit test suite generated by an automated unit test generation technique and transforms those tests into parameterized ones. This way, a group of similar tests that only differ in terms of their test data can be replaced by a single parameterized test with a clear specification of the different test data. The parameterized test suite would be much smaller than the original test suite, and by using the constructs provided by the JUnit framework, the logic and the data of the tests can be separated from each other. This makes it possible for the human testers to more easily and effectively read and understand the tests.

The rest of the paper is organized as follows. Section 2 briefly reviews the related work. The proposed technique is introduced in Section 3, and evaluated in Section 4. Finally, Section 5 concludes the paper.

2 Related Work

The importance of test maintenance, and in particular, the need for readable and understandable tests is acknowledged in different works. For instance, in [6], the authors have investigated test case quality and what good test cases are. One of the conclusions many of the interviewees have arrived at, is that, for a test case to be good, it is necessary for it to be readable and understandable.

The costs and importance of test maintenance are also discussed in [9], where the authors discuss that test maintenance can be incredibly costly and time-consuming, emphasizing the need for automated techniques. In another study, [5] the authors have compared the readability of manually written tests with those of the automatically generated ones. The findings of this study include two important points: 1) readability of the tests is usually overlooked by developers, and 2) generally, automatically generated tests are less readable than the manually written ones.

Another line of research for improving test maintenance is focused on producing natural language summaries for unit tests. For instance, in [10], a technique is proposed to automatically generate natural language summaries as descriptions for JUnit tests. A similar work is presented in [7], which employs natural language processing and code summarization techniques to automatically generate accurate and readable descriptions for unit tests.

A different approach to improving test maintenance is followed by the techniques that seek to automatically generate descriptive names for test methods in unit tests. For instance, in [2], the idea of coverage goals is introduced, meaning that the statements in a test method are intended to serve different testing goals, and therefore, it is possible to generate appropriate, i.e. short and descriptive, names for the test methods by identifying their coverage goals.

Parameterizing unit tests is another approach taken into consideration for improving test understandability. For instance, [4] proposes a technique for generating parameterized unit tests by generalizing pre- and post-conditions of the test methods. Moreover, in [12], the authors introduce a semi-automatic methodology called test generalization, which helps in systematically retrofitting existing concrete unit tests (CUT) as parameterized unit tests (PUT).

Recently, a fully automated CUT-PUT retrofitting technique, called AutoPUT, was introduced in [13], which seeks to identify similar concrete unit tests as candidates to become parameterized, so that the parameterized version can be produced without any decrease in the code coverage. The findings of this study illustrated that AutoPUT can be of help in developing a reliable software by augmenting the maintainability of manually-written test suites. Compared to AutoPUT, the focus of our work is on the automatically generated test suites, which we believe are of more importance and have more potential for getting parameterized.

3 Proposed Approach

In this section, the proposed approach for generating parameterized test suites is introduced. The input of this process is a JUnit test suite and the output is the parameterized version of the input test suite. This process includes the following steps:

- **Pre-Processing**: The purpose of this step is to identify *flaky* tests in the input test suite. A flaky test is a test that its pass/fail result varies in different executions [11]. For this purpose, the input test suite is executed `n` times, and any test that exhibits flaky behavior is flagged to be excluded from further processing. This is because in the post-processing phase of the proposed process, a decision is made based on the execution result of each test, and since a flaky test exhibits a non-deterministic behavior, it can reduce the reliability of the decisions made. It is worth noting that in the experiments reported in Section 4, we have used `n=5` for detecting flaky tests, which is a reasonable setting considered in the related works, for instance [1].

- **Test Clone Detection**: This step is intended to find the test methods that have identical statements, but different parameter values. For this purpose, we have developed a test clone detection component that automatically identifies, through static analysis, the *maximal test clone sets* in the input test suite. A maximal test clone set `s` is a set of test methods so that 1) for each pair of the test methods $\texttt{tm}_1$ and $\texttt{tm}_2$ in `s`, the only difference in the statements of the body of $\texttt{tm}_1$ and $\texttt{tm}_2$, would be the literal values, and 2) for every test method $\texttt{tm}_3$ that does not belong to `s`, and every test method `tm` in `s`, the difference between $\texttt{tm}_3$ and `tm` does not satisfy the above condition. The sets of test methods identified in this step are then parameterized through the next steps.
- **Literal Detection**: In this step, each maximal set of test clones `S` is processed and for each test method `m` in `s`, the abstract syntax tree (AST) is analyzed to identify the literal values used in `m`, e.g., as arguments to method calls or as the right hand side (RHS) expression in an assignment statement.
- **Test Parameterization**: The literals identified in the previous step are used to generate the `@DataPoints` annotation fragment for the parameterized version of the test. In addition, through replacing those literal values with placeholder variables, the logic of the test method is abstracted and used to generate the `@Theory` annotation fragment for the parameterized test method. This has the effect that the logic and the data of the test methods are separated, which result in improving the readability and understandability of the test.
- **Post-Processing**: Finally, the parameterized test methods generated in the previous step, are compiled and run to see if they have any errors, in which case they will be excluded. Otherwise, the coverage of the initial and parameterized versions of the test suite is evaluated to see if there is any difference. If the test coverage of the parameterized version is lower, it will be rejected, otherwise, it will be outputted as the final result of the process.

To provide an example, two test methods `test01` and `test02` generated by Randoop [8] for the class `Primes` from Apache Commons Math project are shown in Listing 1.1, and the parameterized test method generated by the proposed approach is shown in Listing 1.2, with some minor modifications for the sake of brevity.

4 Evaluation

We have implemented the proposed technique as a plugin for the IntelliJ IDEA integrated development environment and have conducted an experiment using a dataset of regression test suites generated by Randoop for 50 Java classes selected for 15 open-source Apache Commons projects. The source code and the data of the experiment are made publicly available on GitHub[1] for reproducibility purposes and further research.

[1] https://github.com/AidinProgrammer/CU2PT-Plugin

In order to evaluate the performance of our proposed technique, we considered three metrics: 1) compression ratio: the percentage of the reduction in the size (physical source lines of code) of the test suite after it is parameterized by the proposed technique, 2) applicability ratio: the percentage of the tests in a test suite that are successfully converted to the parameterized format, and 3) execution time: the time budget needed for the parameterization of a test suite generated for a single class under test.

The results of the experiment demonstrated that the proposed technique is able to provide an average compression ratio of 66% over the whole dataset. Additionally, the lowest and the highest compression ratio are respectively 11% and 96%. Furthermore, in more than 56% of the classes, the compression ratio is at least 80%, meaning that the proposed technique is very likely to cause a significant decrease in the size of a given test suite. Albeit, the reason for the large differences in the compression ratios is the fact that Randoop generates different test suites based on its method sequence generation and extension algorithm. Some of the test suites include quite similar test methods and statements and some of them include more diversity in this regard.

In some of the test suites generated by Randoop, the rate of the presence of code clones is very high; the characteristic which is utilized by our proposed method, resulting in a higher compression ratio. For instance, when a class has just a few public methods, considering the speed with which Randoop generates the test sequences, there would be a large set of method sequences generated with a high level of similarity; hence, the probability of the existence of test clones among them is increased. However, if a class has too many public methods, Randoop can generate various sequences and the similarity of the method sequences is decreased, and as a result, the number of test clones would be decreased. For the test suites with a smaller test clone ratio, there would be less potential for our proposed method to compress the test suite, resulting in a smaller compression ratio.

Regarding the applicability ratio, the highest and the lowest values are respectively 100% and 15%, with an average of 75% over the whole dataset. Moreover, in more than 62% of the classes, the applicability ratio is at least 80%, evidencing the effectiveness of the proposed technique. It should be noted that during the experiment, the proposed technique converted a total of 408,000 test clones into 23,000 parameterized tests in about 7.5 hours; resulting in an average execution time of 1.17 seconds for each parameterized test, which shows the efficiency of the proposed technique.

Listing 1.1. Example test clones generated by Randoop

```
@Test
public void test01() throws Throwable {
  if (debug) System.out.format("%n%s%n","test01");
  int i1 = Primes.nextPrime(0);
  org.junit.Assert.assertTrue(i1 == 2);
}
@Test
```

```
public void test02() throws Throwable {
  if (debug) System.out.format("%n%s%n","test02");
  int i1 = Primes.nextPrime(1);
  org.junit.Assert.assertTrue(i1 == 2);
}
```

Listing 1.2. Parameterized test for the test methods in Listing 1.1

```
@DataPoints("testData")
public static Object[][] getArguments() {
  return new Object[][]{{"test01", 0, 2}, {"test02", 1, 2}};
}
@Theory
public void test(@FromDataPoints("testData") Object[] arguments) {
  if (debug) System.out.format("%n%s%n", arguments[0]);
  int i1 = Primes.nextPrime((int) arguments[1]);
  Assert.assertTrue(i1 == (int) arguments[2]);
}
```

5 Conclusion

In this paper, an automated technique is introduced that takes as input a unit test suite generated by an automated unit test generation tool, and through static analysis and automatic code generation, compresses the given test suite by converting its constituent tests into parameterized tests. During the evaluations conducted, for more than 56% of the input classes, the proposed technique has been able to provide a compression ratio of at least 80%, providing a significant decrease in the size of the test suite while preserving the test coverage. In addition, this technique is considered efficient as it has successfully transformed over 408,000 test clones into nearly 23,000 parameterized tests in less than eight hours, which means an average execution time of 1.17 seconds for each parameterized test.

While the experiment discussed in this paper acknowledges the effectiveness and efficiency of the proposed technique, it is interesting to conduct more extensive experiments, including a larger dataset of test suites generated by Randoop, and also other automated unit test generation tools like EvoSuite [3]. This will be the main direction of our future work. Furthermore, we plan to investigate the performance of our technique on the test suites generated manually, as from the technical point of view, the proposed technique is not limited to test suites generated by automated tools.

References

1. Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T., Marinov, D.: Deflaker: Automatically detecting flaky tests. In: 2018 IEEE/ACM 40th International Con-

ference on Software Engineering (ICSE). pp. 433–444. IEEE (2018)
2. Daka, E., Rojas, J.M., Fraser, G.: Generating unit tests with descriptive names or: Would you name your children thing1 and thing2? In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 57–67 (2017)
3. Fraser, G., Arcuri, A.: Evosuite: automatic test suite generation for object-oriented software. In: Gyimóthy, T., Zeller, A. (eds.) SIGSOFT-/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011. pp. 416–419. ACM (2011). https://doi.org/10.1145/2025113.2025179, https://doi.org/10.1145/2025113.2025179
4. Fraser, G., Zeller, A.: Generating parameterized unit tests. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. pp. 364–374 (2011)
5. Grano, G., Scalabrino, S., Gall, H.C., Oliveto, R.: An empirical investigation on the readability of manual and generated test cases. In: 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC). pp. 348–3483. IEEE (2018)
6. Kochhar, P.S., Xia, X., Lo, D.: Practitioners' views on good software testing practices. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). pp. 61–70. IEEE (2019)
7. Li, B., Vendome, C., Linares-Vásquez, M., Poshyvanyk, D., Kraft, N.A.: Automatically documenting unit test cases. In: 2016 IEEE international conference on software testing, verification and validation (ICST). pp. 341–352. IEEE (2016)
8. Pacheco, C., Ernst, M.D.: Randoop: feedback-directed random testing for java. In: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion. pp. 815–816 (2007)
9. Panichella, S.: Summarization techniques for code, change, testing, and user feedback. In: 2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST). pp. 1–5. IEEE (2018)
10. Panichella, S., Panichella, A., Beller, M., Zaidman, A., Gall, H.C.: The impact of test case summaries on bug fixing performance: An empirical investigation. In: Proceedings of the 38th International Conference on Software Engineering. pp. 547–558 (2016)
11. Paydar, S., Azamnouri, A.: An experimental study on flakiness and fragility of randoop regression test suites. In: International Conference on Fundamentals of Software Engineering. pp. 111–126. Springer (2019)
12. Thummalapenta, S., Marri, M.R., Xie, T., Tillmann, N., De Halleux, J.: Retrofitting unit tests for parameterized unit testing. In: International Conference on Fundamental Approaches to Software Engineering. pp. 294–309. Springer (2011)
13. Tsukamoto, K., Maezawa, Y., Honiden, S.: Autoput: an automated technique for retrofitting closed unit tests into parameterized unit tests. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing. pp. 1944–1951 (2018)