Impacts of Program Structures on Code Coverage of Generated Test Suites

Ryoga Watanabe, Yoshiki Higo, and Shinji Kusumoto

Graduate School of Information Science and Technology, Osaka University, Japan
{ryg-wtnb, higo, kusumoto}@ist.osaka-u.ac.jp

Abstract. Unit testing is a part of the process of developing software. In unit testing, developers verify that programs properly work as developers intend. Creating a test suite for a unit test is very time-consuming. For this reason, research is being conducted to generate a test suite for unit testing automatically, and before now, some test generation tools have been released. However, test generation tools may not be able to generate a test suite that fully covers a test target. In our research, we investigate the causes of this problem by focusing on structures of test targets to improve test generation tools. As a result, we found four patterns as the causes of this problem and proposed subsequent research directions for each pattern to solve this problem.

Keywords: Unit testing · Test generation tool · Code coverage · Program structures

1 Introduction

In software development, software testing is conducted in order to verify that programs work properly as developers intend. Software testing is conducted in several phases, depending on the granularity of the test target. Unit testing is a process of testing functions or methods, which are the smallest granular units of the test target. Bugs and problems can be identified early since unit testing is conducted early in software development. Therefore, unit testing is an essential part of software development.

Creating a test suite for a unit test is very time-consuming. For this reason, research is being conducted to generate a test suite for unit testing automatically [8]. Before now, test generation tools such as EvoSuite [2], Randoop [9], and SUSHI [1] have been released. There are several approaches to generating a test suite. For example, EvoSuite uses genetic algorithm, Randoop uses random testing, and SUSHI uses symbolic execution as well as genetic algorithm.

However, test generation tools may not be able to generate a test suite that fully covers a test target. This problem depends on various factors, such as algorithms of generation tools, parameter settings during test generation, and structures of test targets. In our research, we investigate the cause of this problem by focusing on structures of test targets to improve test generation tools.

```
1  boolean getBooleanProperty(String prop, boolean defaultValue) {
2      String val = System.getProperty(prop);
3      if ( val == null )
4          return defaultValue;
5      if ( val.equalsIgnoreCase("true") ) {
6          return true ;
7      } else {
8          return false ;
9      }
10 }
```

Fig. 1: An example of the method whose statements are not executed by a test suite generated by EvoSuite

We used EvoSuite to generate test suites from 768 Java methods and then manually picked up the methods whose program statements were not fully covered by the test suites. For these methods, we identified the non-covered program statements. We then considered why these program statements are not covered, and we then classified these methods into patterns that we had created based on the found causes of this problem. As a result, we found four patterns and proposed subsequent research directions for each pattern to solve this problem.

2 Motivating Example

Fig. 1 shows a method whose program statements are not covered by a test suite[1] generated by EvoSuite shown in Table 1. The highlighting in Fig. 1 indicates that the `false` branch of the conditional statement `val == null` in line 3 and all the program statements after line 5 are not executed.

We consider the reason why these program statements are not executed by the test suite generated by EvoSuite. The reason is that the condition of line 3 never becomes `false` with the generated test cases. Then, focusing on the `val` variable in this conditional statement, this variable is the return value

Table 1: An example of the test suite from the method shown in Fig. 1 generated by EvoSuite

Input		Assertion
`prop`	`defaultValue`	
`"Hwiz5]f"`	`true`	returns `true`
`"7G"`	`false`	returns `false`
`null`	`false`	returns `false`
`""`	`false`	throws `IllegalArgumentException`

[1] In this paper, the statements partially executed by a generated test suite are presented in yellow . Herein, partially executed means that only a `true` or `false` branch is executed by a generated test suite. The statements never executed by a generated test suite are presented in orange .

of `System.getProperty()` in line 2. This means that the condition of line 3 never becomes `false` because `System.getProperty()` always returns a `null` value. The `System.getProperty()` method is designed to return the value of a real system property name (such as `"user.dir"`) if input; otherwise, it returns a `null` value. Therefore, we can conclude that there are some non-covered program statements by the generated test cases because EvoSuite cannot generate any test case that inputs a real system property name because of the `System.getProperty()` method.

3 Investigation Settings

3.1 Dataset

We used a dataset [7] consisting of 768 Java methods as the subject of our investigation. These methods have the following characteristics: (1) they do not depend on external variables or methods, (2) they have one or more arguments and a return value, (3) they are described in the Java 8 or earlier specification, and (4) they use only the `java.lang` or `java.util` classes.

3.2 Test Generation Tool

We used EvoSuite to generate test suites for the methods in the dataset. EvoSuite generates highly covered JUnit test suites based on genetic algorithm, using techniques such as the hybrid search [6], the dynamic symbolic execution [4], and the testability transformation [5]. EvoSuite is unique among the other test generation tools for Java in its ability to generate test suites with high coverage [11].

EvoSuite has several coverage criteria as coverage targets for generating a test suite. This means that our investigation results will differ depending on which criteria we focus on. We selected to focus on line coverage and branch coverage because they are representative of code coverage and can be measured and visualized by JaCoCo[2].

3.3 Process Steps

We have taken the following steps to prepare for the investigation.

Step.1: for each method from the dataset, we made class definitions that include the method itself. To ensure that all class definition files are compilable, we inserted the import statement `import java.util.*;` at the beginning of each file.

Step.2: for each class made in Step.1, we generated a test suite using EvoSuite and measured the coverage of the test suite using JaCoCo.

Step.3: we manually picked up the methods whose line or branch coverage is less than 1.0. Out of 768 methods in the dataset, we picked up 73 methods, which is 9.5% of all the methods. In the steps, we excluded the methods from

[2] `https://www.eclemma.org/jacoco/`

```
1 Float parseFloat(String value, float defaultValue) {
2     try {
3         return Float.parseFloat(value);
4     } catch (NumberFormatException e) {
5         ...
6     }
7 }
```

Fig. 2: An example of the method that is classified into "Method parameters require specific values"

which EvoSuite cannot generate tests that fail or are incorrectly evaluated as not covered due to the bug in JaCoCo.

Step.4: for the methods we picked up in Step.3, we identified the non-covered program statements by referring to the report generated by JaCoCo. We then classified those methods into patterns that we had created based on the found causes.

4 Investigation Results

As the reasons why the program statements were not covered, we found four patterns:

- "Method parameters require specific values"
- "Method parameters require specific types"
- "Methods include infeasible program statements"
- "Methods include multithreaded program statements"

4.1 "Method parameters require specific values"

In this pattern, parameters that satisfy some conditions are required in order to cover the non-covered program statements. Fifty-eight out of 73 methods are classified into this pattern. Fig. 2 shows an example of the method that is classified into this pattern. This method converts any string representing a `float` type value (e.g., `"10.0f"`) to an actual `float` type value. In this method, the normal process of `Float.parseFloat()` is not covered by the generated test suite. In order to cover this process, a test case is required to input any string representing a `float` type value into this method.

4.2 "Method parameters require specific types"

In this pattern, parameters of derived classes of the class in the parameter definition are required in order to cover the non-covered program statements. Six out of 73 methods are classified into this pattern. Fig. 3 shows an example of the method that is classified into this pattern. This method casts the parameter of `Object` type to `int` type and returns its value. In this method, the `false` branch of the condition in line 2 and the statement in line 4 are not covered.

```
1  int uncheckedIntCast(Object x) {
2      if ( x instanceof Number )
3          return ((Number) x).intValue();
4      return ((Character) x).charValue() ;
5  }
```

Fig. 3: An example of the method that is classified into "Method parameters require specific types"

```
1  List<String> splitToList0(String str, char ch) {
2      List<String> result = new ArrayList<>();
3      int ix = 0, len = str.length();
4      for (int i = 0; i < len; i++) {
5          if (str.charAt(i) == ch) {
6              result.add(str.substring(ix, i));
7              ix = i + 1;
8          }
9      }
10     if ( ix >= 0 ) {
11         result.add(str.substring(ix));
12     }
13     return result;
14 }
```

Fig. 4: An example of the method that is classified into "Methods include infeasible program statements"

The branch in line 3 evaluates whether or not the parameter type is `Number`. In order to cover them, a test case is required to input a variable other than `Number` type.

4.3 "Methods include infeasible program statements"

In this pattern, methods have the program statements that are never executed, no matter what parameters are given. Six out of 73 methods are classified into this pattern. Fig. 4 shows an example of the method that is classified into this pattern. This method splits the `str` string by the `ch` character and returns the result as a list of strings. The variable `ix` is initialized with the value `0` in line 3, and there is no program statement to decrease `ix` below `0`, even in the program statement of line 4 to 9, where the value of `ix` may change. Therefore, the `false` branch of the `if` statement in line 10 is never executed, no matter what parameters are given.

4.4 "Methods include multithreaded program statements"

In this pattern, a test case using multithreading is required to cover the non-covered program statements. Three out of 73 methods are classified into this pattern. Fig. 5 shows an example of the method that is classified into this pattern. This method puts the currently running thread to sleep. In this method, the checked exception `InterruptedException` in line 4, thrown by `Thread.sleep()` in line 3, is not caught by the generated test suite. `InterruptedException` is

```
1  Integer apply(Integer i) {
2      try {
3          Thread.sleep(1);
4      } catch ( InterruptedException e ) {
5          e.printStackTrace() ;
6      }
7      return i;
8  }
```

Fig. 5: An example of the method that is classified into "Methods include multi-threaded program statements"

an exception thrown when a thread is in waiting, sleeping, or occupied and the thread is interrupted. In order to catch this exception, a test case is required to throw `InterruptedException` intentionally using multithreading.

5 Discussion

We propose subsequent research directions to increase coverage for the four patterns we found in our investigation.

5.1 "Method parameters require specific values" and "Method parameters require specific types"

The reason why EvoSuite cannot cover program statements classified into the patterns "Method parameters require specific values" or "Method parameters require specific types" is that EvoSuite cannot generate a test case that inputs any string (i.e., any value of `java.lang.String`) to satisfy some conditions to cover them.

EvoSuite uses constants of primitive or `String` type statically embedded in Java bytecode when EvoSuite evolves a test suite in genetic algorithm [2]. Applying this feature, modifying EvoSuite to externally provide the constants and types the user wants EvoSuite to treat while evolving a test suite would improve coverage.

As a simple exploration to demonstrate the effectiveness of this direction, consider inserting a *fake branch* to the test target. A *fake branch* is an `if` statement that does not affect the original method's functionality containing values or types to execute non-covered program statements. For example, suppose the test suite is generated again using EvoSuite for the method shown in Fig. 2 with a *fake branch* inserted, as shown in Fig. 6. In that case, the non-covered program statements (i.e., line 3 in Fig. 2) will be executed because EvoSuite can refer to the value embedded by the *fake branch* (i.e., `"10.0f"`).

5.2 "Methods include infeasible program statements"

Infeasible program statements can never be executed, regardless of the given parameters. Therefore, it is impossible to solve by adding or modifying test cases.

```
1    Float parseFloat(String value, float defaultValue) {
2  +       if (value == "10.0f") {}
3         try {
4           return Float.parseFloat(value);
5         } catch (NumberFormatException e) {
6             ...
7         }
8     }
```

Fig. 6: An example of inserting the *fake branch*

Instead, excluding the infeasible program statements from coverage goals or lower the coverage priority when generating test suites would improve coverage. EvoSuite uses an approach that optimizes the coverage criteria as a whole rather than each coverage goal of the coverage criteria. This results in a lower coverage priority for infeasible program statements. Integrated development environment features can detect some infeasible program statements. If programmers use this feature to remove infeasible program statements before generating the test suite, these statements can be excluded from the coverage goals. For example, in the method shown in Fig. 4, this corresponds to removing the `if` statement in line 10.

5.3 "Methods include multithreaded program statements"

EvoSuite is not compatible with the generation of test suites that use multithreading [3]. Therefore, from the point of improving program structures, it is impossible to solve this problem. On the other hand, the research [10] has been conducted to develop EvoSuite to generate test suites that support multithreaded processing.

6 Conclusion and Future Work

Our research investigated how program structures affect code coverage of generated test suites. As a result, we found four patterns as the reasons the program statements are not covered. We considered whether a technique exists to solve why program statements are not covered for those patterns and proposed subsequent research directions for each pattern.

Our investigation is conducted in the settings explained in Section 3. Therefore, different results may be obtained if another investigation is conducted in different settings. In future work, we will verify what results can be obtained by conducting the same investigation on a different dataset or actual Java projects and on different criteria or test generation tools.

Acknowledgments

This research was supported by JSPS KAKENHI Japan (JP20H04166, JP21K18302, JP21K11829, JP21H04877, JP22H03567, JP22K11985)

References

1. Braione, P., Denaro, G., Mattavelli, A., Pezzè, M.: SUSHI: A test generator for programs with complex structured inputs. In: Proc. International Conference on Software Engineering. pp. 21–24 (2018)
2. Fraser, G., Arcuri, A.: Evosuite: automatic test suite generation for object-oriented software. In: Proc. European Conference on Foundations of Software Engineering. pp. 416–419 (2011)
3. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Transactions on Software Engineering pp. 276–291 (2013)
4. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 213 – 223 (2005)
5. Harman, M., Hu, L., Hierons, R., Wegener, J., Sthamer, H., Baresel, A., Roper, M.: Testability transformation. IEEE Transactions on Software Engineering pp. 3–16 (2004)
6. Harman, M., McMinn, P.: A theoretical and empirical study of search-based testing: Local, global, and hybrid search. IEEE Transactions on Software Engineering pp. 226–247 (2010)
7. Higo, Y., Matsumoto, S., Kusumoto, S., Yasuda, K.: Constructing dataset of functionally equivalent java methods. In: Proc. International Conference on Mining Software Repositories. pp. 682–686 (2022)
8. McMinn, P.: Search-based software test data generation: a survey. Software Testing, Verification and Reliability pp. 105–156 (2004)
9. Pacheco, C., Ernst, M.: Randoop: feedback-directed random testing for Java. In: Proc. Object-Oriented Programming, Systems, Languages and Applications. pp. 815–816 (2007)
10. Steenbuck, S., Fraser, G.: Generating unit tests for concurrent classes. In: Proc. International Conference on Software Testing, Verification and Validation. pp. 144–153 (2013)
11. Vogl, S., Schweikl, S., Fraser, G., Arcuri, A., Campos, J., Panichella, A.: Evosuite at the sbst 2021 tool competition. In: Proc. International Workshop on Search-Based Software Testing. pp. 28–29 (2021)