On the Difficulties Behind Automated Test Case Generation

Valentine Mairet 4141784
Martin Rogalla 4173635
Aleksandra Taneva 4510488

April 10, 2016

Abstract

Automated test case generation was introduced to make the life of software testers easier. However, some problems regarding the generation of test cases still remain. In this report, the EvoSuite tool for automated Java JUnit test case generation is run on 12 different Java classes to investigate the difficulties behind this automated process. Each class is chosen for its size, complexity, and how difficult it seems to test at first glance. Additionally, this report provides an appendix with a short piece on the usability of EvoSuite and snippets of classes EvoSuite was run on. By correlating cyclomatic complexity with branch coverage, and with test readability, the hope is to find a reason behind the difficulty to test a specific branch, and to generate readable tests to help the developer generate oracles easier. The correlation coefficients found were insignificant, with 0.1753 and 0.3044 for cyclomatic complexity with branch coverage and with test readability respectively. Therefore, cyclomatic complexity does not have an influence on neither code coverage nor generated test readability. However, it seems that the cause for low coverage lies deeper. When tests require specific input that needs conscious intervention, EvoSuite does not seem to be able to generate such test. The tool cannot reason about the source code as a human would. EvoSuite also fails at boundary testing. Moreover, when dealing with specific Java Generics input, EvoSuite cannot properly generate tests either. The results show that EvoSuite is highly optimized for branch coverage, but when it fails at doing so, the reason mostly lies within the reasoning needed to deal with the source code.

Contents

1 Introduction

Software testing is an essential component of software development that allows developers to assess the reliability and correctness of a software product. Over the years, researchers have designed techniques to automatically produce test cases, where the code coverage criterion is the most commonly used guiding medium for the generation. Now, with the newest tools available, it has become possible to generate entire test suites with high coverage. One of these tools for Java software is the EvoSuite framework [6].

1.1 EvoSuite

EvoSuite is a tool that automatically generates JUnit tests with an evolutionary algorithm [3, 6]. This tool can be directly run from the command line or through the Eclipse or IntelliJ IDEA IDEs, with the options to either generate tests according to the *whole-test-suite* or *single-target* approach [6]. It can also be configured to either generate tests class per class or for the entire project at once [2].
EvoSuite works at the bytecode level and collects all necessary information for test case generation from the bytecode via Java Reflection [6]. This means that EvoSuite can also be used for other programming languages that compile to Java bytecode (such as Scala or Groovy) [5, 11].
There is, however, one problem associated with automatic test case generation, known as the *oracle problem* [10]. Automatic oracles that assert what the outcome of a test should be are possible when it comes to explicit events such as 'the program should not crash'. But when dealing with more complex outcomes, human intervention is necessary to specify oracles [6]. For this to be feasible, EvoSuite aims not only at high code coverage but also at small test suites that make manual oracle generation as easy as possible.

1.2 Limitations of EvoSuite

But EvoSuite does not always get it right, and some coverage goals are occasionally not fulfilled [6]. This can be explained by a few language-related problems that are encountered by the EvoSuite tool during test case generation.
In particular, classes using Java Generics are problematic, as type erasure removes much of the useful information during compilation and all generic parameters are considered as Object. To overcome this problem, EvoSuite always inserts an Integer object into container classes, in order to cast returned Object instances back to Integer [6]. However, for a method with Generic input that is not compatible with an Integer object, things can go wrong.
Additionally, other factors may contribute to explaining why EvoSuite does not always get it right. There may be methods with inputs for which conscious human intervention is necessary, and there is no way for EvoSuite to know what the input is supposed to be [6]. An instance of this problem is a method that takes an integer and returns *true* if that integer is prime. For a test to cover the *true* branch and check the correctness of the code, the input must be a prime number.

1.3 This Research

This paper poses the question of difficulty behind automated test case generation with EvoSuite and dives into the reasons why such difficulty exists. Why are certain pieces of code hard to generate tests for? Does EvoSuite fail at generating tests because the code is bad or too complex? Does EvoSuite always generate readable tests to ease the generation of oracles by developers?
After a recap of previous work done on and with EvoSuite in section 2, the methods for this research are explained in detail in section 3. Then, the results are presented in section 4. Lastly, this paper concludes the research executed and brings up a discussion on the implication of the results found.
On a side note, an analysis of the usability of the EvoSuite tool is given. This analysis provides a few details on the problems encountered when using EvoSuite and what difficulties regarding the usage of the framework possibly hindered the progress of this research.

2 Previous Work

Previous work on and with EvoSuite has been done to prove automated test case generation could result in test suites with high code coverage and little test cases [6]. The same team who presented EvoSuite additionally inquired upon the effectiveness of EvoSuite and the *whole-test-suite* approach [7].

2.1 Making of EvoSuite

Making EvoSuite with the *whole-test-suite* approach aimed at high code coverage and the generation of smaller test suites to make the specification of assertions as easy as possible [6].
EvoSuite achieves this by selecting branch coverage as a coverage criterion and executing a genetic algorithm that selects a random population of test suites and evolves this population. It stops the evolution process when a solution is found which fulfills the coverage criterion or when the set of resources (such as time or number of fitness evaluations) have been used up [6].
A solution is defined as a *test suite*, which is represented as a set of test cases. A test case is represented by a sequence of statements. The fitness function estimates how close a solution is to covering *all* branches by taking into account *branch distance* for each branch, and the number of branches left to cover. It also takes into account the size of the solution, to make sure the generated test suite does not grow to large. It was shown that this approach fulfilled large coverage goals and succeeded at generating small enough test suites for easy oracle generation [6]. But what about the performance of the *whole-test-suite* approach compared to *single-target*?

2.2 Whole Test Suite vs Single Target

By comparing the coverage achieved by the *whole-test-suite* and *single-target* approach, it was establish that, in general, the *whole* approach performs better [7, 8].
Existing research has shown that the *whole-test-suite* approach lead to better coverage results than the traditional *single-target* approach. There was reasonable doubt on whether this would be the case when targeting difficult goals. An in-depth analysis was performed to study if this doubt could be confirmed in practice [7].
EvoSuite was run on 100 different Java classes respectively for both techniques. It was found that indeed, the Whole approach had more difficulties when targeting difficult goals. However, these cases are very few compared to the cases for which Whole performed significantly better. This thus confirmed that overall, *whole-test-suite* is better.

3 Methods

For automated test case generation to be optimal, the test suite needs to achieve high coverage while remaining relatively small for oracle generation [6]. But remaining small does not necessary mean the generated test cases are readable for developers. The target of our research is the difficulties regarding automated test case generation when using EvoSuite. Hence, this paper focuses mainly on the difficulty to achieve certain coverage goals and difficulty to generate readable tests. The two questions that arise from this focus are: what is the reason behind unfulfilled coverage goals and what can make EvoSuite generate illegible tests?

3.1 Experiment

In an attempt to provide answers, we execute an experiment, running the EvoSuite tool on 12 Java classes, each picked out of different projects, from open source software to our own. These classes were chosen according to size and complexity, and our own opinion on the difficulty to test such class. We estimated this difficulty based on whether a test would need a highly specific input to cover certain branches. We also designed or modified a few classes especially for this experiment. The chosen classes are reported in Table 1.
For our experiment, the *whole-test-suite* approach is used, because it is proved to be better than *single-target* [7, 8]. This means that the *whole* approach will most likely be used more often, so it is more interesting to execute this research with this approach.

Class	Project	Source
PrimeChecker	EvoTest	Own
IntStack	EvoTest	Own
EnglishNumberToWords	NumberToWords	Real's HowTo
ColourPicker	EvoTest	Own
org.joda.time.chrono.ISOChronology	Joda-Time	GitHub
org.joda.time.DateTimeUtils	Joda-Time	GitHub
com.google.common.base.Optional	Guava	GitHub
com.google.common.base.Strings	Guava	GitHub
com.google.common.math.IntMath	Guava	GitHub
com.google.common.graph.ImmutableGraph	Guava	GitHub

Table 1. For each class, the table reports which project the class belongs to, and where the project was found. *Own* indicates the provenance is our own project.

Rating	Description
1	Diving into the code is required and some parts are still unclear.
2	Diving into the code is required to actually understand the tests.
3	Reading the tests a few times is required to understand them.
4	The tests are explicit overall, but some parts require more reading.
5	The tests are explicit from the first read.

Table 2. For each generated set of test for a specific branch, the team gives a grade from 1 to 5 as according to this table.

After test suite generation with EvoSuite, we pursue the experiment by taking a look at three problems:

1. Does cyclomatic complexity of the class under test (CUT) have an influence on code coverage [4]?
2. Does cyclomatic complexity of the CUT have an influence on test readability?
3. Are there more specific reasons why EvoSuite fails at covering a branch?

3.2 Measurements

Using Cobertura, the cyclomatic complexity of the CUT is evaluated and reported along with the corresponding code coverage results [1]. With these numbers, the correlation between cyclomatic complexity and code coverage can be calculated to determine whether code complexity has an influence on achieving coverage goals for that piece of code.
Pursuing with Cobertura and the cyclomatic complexity of the CUT, each member of our team rates the generated test on readability, with a grade from 1 to 5. The rating is described in table 2. The test score is then averaged and correlated with the complexity, to see if code complexity has an influence on test readability. Averaging each score given by team members shows a more reliable result on readability because the opinion of multiple people is taken into account. Statistically speaking, this gives a better indication than just one person's opinion.

Class	Coverage
PrimeChecker.isPrime	83%
IntStack	100%
EnglishNumberToWords	73%
ColourPicker	85%
org.joda.time.chrono.ISOChronology	83%
org.joda.time.DateTimeUtils	56%
com.google.common.base.Absent	50%
com.google.common.base.Optional	33%
com.google.common.base.Present	100%
com.google.common.base.Strings	84%
com.google.common.math.IntMath	81%
com.google.common.graph.ImmutableGraph	100%

Table 3. For each class, the table reports the branch coverage achieved by the EvoSuite framework.

For each method with uncovered code, we take a deeper look into the code itself in an attempt to find a reason behind the failure. We report possible explanations and motivate how those reasons may hold.

4 Results

EvoSuite was run on 12 different Java classes. The results regarding branch coverage of each class are reported in Table 3. In the following sections, the cyclomatic complexity of the CUT is reported respectively with corresponding branch coverage and test code readability. The correlations between the results are calculated according to Pearson's correlation coefficient formula [9]:

$$\mathrm{r} = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2}\sqrt{\sum_i (y_i - \bar{y})^2}}$$

4.1 Cyclomatic Complexity and Code Coverage

The results for the cyclomatic complexity of the CUT and branch coverage are reported in Table 4. With Pearson's formula, the resulting correlation coefficient is 0.1753. Despite this result being positive, the relationship between cyclomatic complexity and branch coverage is weak. This value is too close to 0 to establish any rule on the influence of code complexity on code coverage.
Although the number of classes being relatively low, the types of classes were variate. From data classes to single-method classes with high complexity, it seems these characteristics had no influence on the total coverage achieved by EvoSuite. It is possible assume that **cyclomatic complexity of the CUT has no influence on the total branch coverage reached**, but given the small dataset of classes, it is unwise to draw final conclusions.

4.2 Cyclomatic Complexity and Test Readability

The results for the cyclomatic complexity of the CUT and average test readability score given by the team members are reported in Table 5. With Pearson's formula, the resulting correlation coefficient is

Method	Cyclomatic Complexity	Coverage
PrimeChecker.isPrime	6	83%
IntStack	2.143	100%
EnglishNumberToWords	4.667	73%
ColourPicker	18	85%
org.joda.time.chrono.ISOChronology	1.8	83%
org.joda.time.DateTimeUtils	2.226	56%
com.google.common.base.Absent	1.071	50%
com.google.common.base.Optional	1.222	33%
com.google.common.base.Present	1.167	100%
com.google.common.base.Strings	3.6	84%
com.google.common.math.IntMath	6.6	81%
com.google.common.graph.ImmutableGraph	1.727	100%
Correlation		0.1753

Table 4. For each class, the table reports the average cyclomatic complexity of that class and the total branch coverage achieved by EvoSuite.

0.3044. This coefficient is weak, which shows little relationship between the results.
The ColourPicker class was orchestrated to hold a very high cyclomatic complexity, but the branches were relatively simple to cover and tests were easy to generate. When removing the ColourPicker results from the calculation, a correlation coefficient of 0.5655 is obtained. This is a moderate positive correlation, which indicates that there is a chance that high cyclomatic complexity influences test readability.
It is possible assume that **cyclomatic complexity of the CUT has a small influence on test readability**, but given the small dataset of classes, it is unwise to draw final conclusions regarding this hypothesis.

4.3 Specific Reasons for Unfulfilled Goals

Since cyclomatic complexity does not seem to have an influence on neither code coverage nor generated test readability, there must be other specific reasons why some coverage goals were not fulfilled. In this section, parts of the uncovered code are analyzed in order to provide an explanation for absence coverage of certain branches.
In Figure 1, the `isPrime(int p)` method checks if the input p is a prime number. This is an instance of the problem mentioned earlier in section 1.1. The method was designed such that it only performs the check when the input is bigger than 1,000,000. Since EvoSuite could not find an input bigger than 1,000,000 that was prime (the probability of such finding being quite low), it was not able to generate a test that covered the *true* branch.
It seems that part of the code shown in Figure 2 is not covered because of the influence of a random number. Test input has no influence on how this random number behaves. Therefore, EvoSuite cannot generate an input that satisfies the condition to cover that branch.
In Figure 3, the `convert(long number)` function accepts a parameter of type *long*. EvoSuite fails to generate larger numbers to cover the red highlighted lines. Surprisingly, EvoSuite does not test the boundaries of the long parameter. Doing so would have resulted in the coverage of at least one of the uncovered branches.
In order to cover the two branches that were not covered by EvoSuite-generated tests in Figure 4, a tester would need to input specific strings with a high or low surrogate. EvoSuite does not know what

Method	Cyclomatic Complexity	Test Readability
PrimeChecker	6	5
IntStack	2.143	5
EnglishNumberToWords	4.667	5
ColourPicker	18	4.5
org.joda.time.chrono.ISOChronology	2	4
org.joda.time.DateTimeUtils	2.226	4.5
com.google.common.base.Optional	1.222	2.5
com.google.common.base.Strings	3.6	5
com.google.common.math.IntMath	6.6	4.5
com.google.common.graph.ImmutableGraph	1.727	1.5
Correlation		0.3044

Table 5. For each method, the table reports the average cyclomatic complexity of that method and the average score given by the team on code readability.

```
1  6  public class PrimeChecker {
2
3        public boolean isPrime(int p) {
4  6         if (p > 1000000000) {
5  2             int max = (int) Math.ceil(Math.sqrt((double) p));
6
7  44            for (int i = 2; i < max; i++) {
8  44                if ((p % i) == 0) {
9  2                     return false;
10                    }
11                }
12
13 0              return true;
14           } else {
15 4              return false;
16           }
17       }
18 }
```

Figure 1. Representative of uncovered branches in PrimeChecker.

```
41
42       public Color pickRGB() {
43 4         Random rnd = new Random();
44 4         int i = rnd.nextInt() * 10;
45
46 4         if (i < 4) {
47 2             return Color.RED;
48 2         } else if (i < 10) {
49 0             return Color.GREEN;
50          } else {
51 2             return Color.BLUE;
52          }
53       }
54 }
```

Figure 2. Representative of uncovered branches in ColourPicker.

```
67 4            String snumber = Long.toString(number);
68
69              // pad with "0"
70 4            String mask = "000000000000";
71 4            DecimalFormat df = new DecimalFormat(mask);
72 4            snumber = df.format(number);
73
74              // XXXnnnnnnnnn
75 4            int billions = Integer.parseInt(snumber.substring(0,3));
76              // nnnXXXnnnnnn
77 4            int millions  = Integer.parseInt(snumber.substring(3,6));
78              // nnnnnnXXXnnn
79 4            int hundredThousands = Integer.parseInt(snumber.substring(6,9));
80              // nnnnnnnnnXXX
81 4            int thousands = Integer.parseInt(snumber.substring(9,12));
82
83              String tradBillions;
84 4            switch (billions) {
85                      case 0:
86 4                           tradBillions = "";
87 4                           break;
88                      case 1 :
89 0                         tradBillions = convertLessThanOneThousand(billions)
90                                          + " billion ";
91 0                         break;
92                      default :
93 0                         tradBillions = convertLessThanOneThousand(billions)
94                                          + " billion ";
95              }
96 4            String result =  tradBillions;
97
98              String tradMillions;
99 4            switch (millions) {
100                     case 0:
101 4                        tradMillions = "";
102 4                        break;
103                     case 1 :
104 0                       tradMillions = convertLessThanOneThousand(millions)
105                                         + " million ";
106 0                       break;
107                     default :
108 0                       tradMillions = convertLessThanOneThousand(millions)
109                                         + " million ";
110             }
111 4           result =  result + tradMillions;
112
```

Figure 3. Representative of uncovered branches in EnglishNumberToWords.

high and low surrogate mean and as it does not fall under standard String input, it cannot cover these two branches.
In Figure 5, a simple optimization in the case that a left-shift in the power function can be used in not covered. Since EvoSuite cannot directly reason about the source code, it does not know that such a branch exists and thus there is a possibility that it goes undetected. This is one of the issues that can arise when using a black-box approach.

5 Conclusion and Discussions

EvoSuite is a very promising automated test case generation tool but it encounters some difficulties during the process. The generated tests can be obscure to read and various branches may remain uncovered. The main reason found to have unfulfilled goals comes from a lack of knowledge and ability to reason regarding the internal code structure to generate specific input values required to cover a branch. If there is a condition for which a specific input value is necessary, EvoSuite occasionally does not find the

```
216             /**
217              * True when a valid surrogate pair starts at the given {@code index} in the given {@code string}.
218              * Out-of-range indexes return false.
219              */
220             @VisibleForTesting
221             static boolean validSurrogatePairAt(CharSequence string, int index) {
222 68            return index >= 0
223 44              && index <= (string.length() - 2)
224 8               && Character.isHighSurrogate(string.charAt(index))
225 0               && Character.isLowSurrogate(string.charAt(index + 1));
226           }
227         }
```

Figure 4. Representative of uncovered branches in Strings.

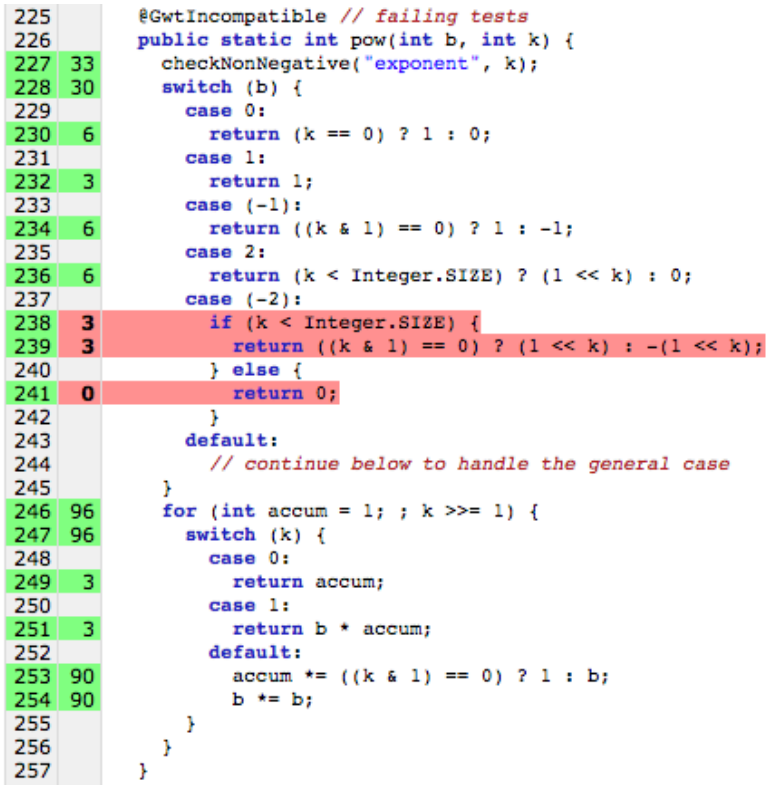

Figure 5. Representative of uncovered branches in IntMath.

input value to reach that branch.
This is a limitation of any black-box testing tool, which could be improved by analyzing the source code of the system under test. These improvements would shift the scope of EvoSuite being a black-box testing tool to a more white-box testing tendency.
But a white-box approach does not solve the problems regarding automated test case generation. The PrimeChecker CUT shows another difficulty, even when applying white-testing techniques to EvoSuite: it requires a significant amount of resources and time to find the required input value to cover a specific branch, e.g. finding a very large prime number.
The experiment for this research performed an exploratory analysis on the correlations between cyclomatic complexity, code coverage and test readability. Findings indicate that the cyclomatic complexity of the CUT has no apparent influence on the total branch coverage reached. Additionally, a weak positive correlation of 0.3044 between cyclomatic complexity and readability of the automatically generated tests is found. Due to the limited size of the dataset used, no definite conclusion can be drawn from these findings. However, it does indicate that it might be interesting to pursue an in-depth study on a larger scale that will look at the correlation between cyclomatic complexity and the readability of automatically generated tests. This target of research could have meaningful implication for the design of test oracles after automated test case generation.

References

[1] Cobertura - a code coverage utility for java. `http://cobertura.github.io/cobertura`. Accessed: 2016-04-09.

[2] Evosuite - documentation - commandline. `http://www.evosuite.org/documentation/commandline`. Accessed: 2016-04-09.

[3] Junit - about. `http://junit.org`. Accessed: 2016-04-09.

[4] What is cyclomatic complexity? `http://www.tutorialspoint.com/software_testing_dictionary/cyclomatic_complexity.htm`. Accessed: 2016-04-10.

[5] Venners B, Spoon L, and Odersky M. *Programming in Scala*. Artima, 2008.

[6] Fraser G and Arcuri A. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[7] Fraser G and Arcuri A. On the effectiveness of whole test suite generation. *Search-Based Software Engineering*, 8636:1–15, 2014.

[8] Fraser G and Arcuri A. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering (EMSE)*, 20(3):783–812, 2015.

[9] Dekking F M, Kraaikamp C, Lopuhaa H P, and Meester L E. *A Modern Introduction to Probability and Statistics*. Springer, 2005.

[10] Barr E T, Harman M, McMinn P, Shahbaz M, and Yoo S. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[11] Subramaniam V. *Programming Groovy*. O'Reilly - O'Reilly Media, 2008.

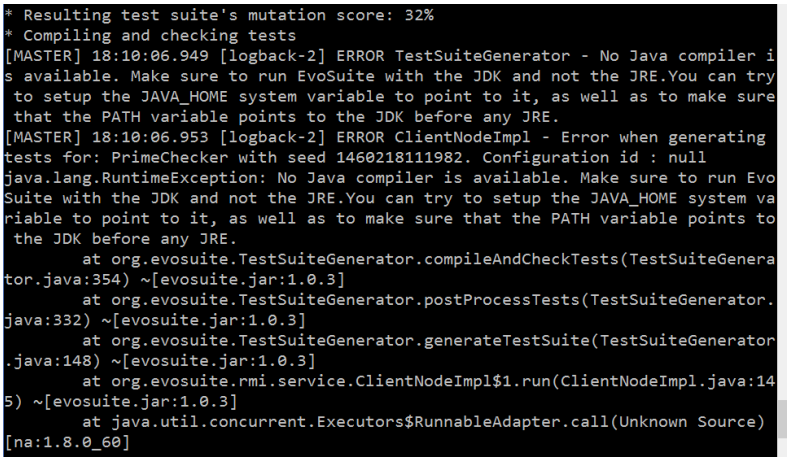

Figure 6. EvoSuite cannot find the Java compiler.

A Usability of EvoSuite

In this section, we briefly explore the usability of EvoSuite. Automated test case generation is a real solution for the burden software testing can be. But if a tool like EvoSuite that provides such automation is difficult to use, it does not add much ease to the process of testing. We provide with a report on problems encountered when using the tool on Microsoft Windows 10, and on further issues concerning Maven and EvoSuite.

A.1 Problems Encountered

In this part, we report the issues that we encountered during installation and working with automated test generation tool EvoSuite. The operating system is Microsoft Windows 10, 64bit. After a proper installation, according to the guidelines from the official site `http://www.evosuite.org/` and choosing one of the different ways to execute a test generation tool, namely run EvoSuite from the command line. The tool is assembled and compiled in an independent executable jar (e.g., `evosuite.jar`). The CUT that we have chosen is Joda-Time that provides API that replaces the standard Java date and time classes. However, we discovered a problem on one of the working stations after invoking the given command:
`$ java -jar evosuite.jar -generateSuite -projectCP <class_path> -class <class_name>`,
where the option class is used to define the CUT, and the option -projectCP is used to indicate the classpath. The error message is shown in figure 6.
Theoretically, if the correct path is provided, this approach is supposed to work fine if EvoSuite is specified in a "static" context, e.g., when the class path does not change, and a user only changes the CUT's file name. The installed version of JDK is `1.8.0_60`. The system and user variables of the system are defined as follows: `JAVA_HOME` variable is set at `jdk1.8.0_60` and `PATH` at `jdk1.8.0_60/bin`. However, one of the possible reasons for triggering that error was the assumption that the User Environment variables were not properly set. The `PATH` and `JAVA_HOME` variable were excluded from the system variables list but apparently that did not lead to a positive outcome and the possibility to run the test suite to the end. Before executing the command that is supposed to generate tests, the following command was executed to make sure the system could refer to the JDK properly: `PATH=%JAVA_HOME%:%PATH%`. Unfortunately, that did not change anything and the error message still appeared. Bottom line, even though the similar set of system variables that did not result in desired output.
To conclude, a certain set of the environment variables and JDK options did not provide the opportunity for properly set characteristics of the operating system in order to provide the needed scenario to run EvoSuite. The absence of detailed documentation and lack of experience resulted in the impossibility of automated test to be generated on the Windows 10 system. Nevertheless, when EvoSuite works, it is an effective tool for generating tests to simplify the testing process for developers. Is a stated solution

that will definitely expand its role in the process of automated test generation which will be adopted by software engineers.

A.2 Maven and EvoSuite

Generating tests is not all - we need to get these tests to run. Joda-Time and our own projects made use of Maven as a software project management and comprehension tool. Getting EvoSuite dependencies right in the Maven setup is relatively trivial, but it did not work well with other plugins used by the projects.
For example, we integrated Cobertura in each project to calculate branch coverage and cyclomatic complexity. Cobertura uses bytecode instrumentation to perform code coverage checks within a class and creates a new class file for the CUT. EvoSuite generates tests that are run inside an EvoSuite-dependent sandbox, meaning EvoSuite has its own type of runner. When running `mvn corbertura:cobertura` to generate Coberura reports, the tests need to be executed within this sandbox. The runner attempts to find the CUT but instead runs into two classes: the original CUT, and the class instrumented by Cobertura. This confusion causes the JVM to throw a `ClassNotFoundException` and the tests do not get executed.
To circumvent this problem, it was necessary to manually remove the EvoSuite sandboxing and any EvoSuite dependency within the generated tests. Some generated tests used specific EvoSuite assertions to see if an exception was thrown. These assertions, unfortunately, had to be removed to generate Cobertura reports.

B Code Snippets

In this section, we provide code snippets from classes we ran EvoSuite on.

B.1 PrimeChecker

```
1  public class PrimeChecker {
2
3      public boolean isPrime(int p) {
4          if (p > 1000000000) {
5              int max = (int) Math.ceil(Math.sqrt((double) p));
6
7              for (int i = 2; i < max; i++) {
8                  if ((p % i) == 0) {
9                      return false;
10                 }
11             }
12
13             return true;
14         } else {
15             return false;
16         }
17     }
18 }
```

Listing 1. PrimeChecker

B.2 IntStack

```
1  public class IntStack {
2      private int[] stack;
3      private int top_index;
4
5      public IntStack(int capacity) {
6          stack = new int[capacity];
7          top_index = -1;
8      }
9
10     public void push(int i) throws Exception {
11         if (!isFull()) {
```

```
            int new_top_index = top_index + 1;
            stack[new_top_index] = i;
            top_index = new_top_index;
        } else {
            throw new Exception("Stack is full");
        }
    }

    public int pop() throws Exception {
        if(!isEmpty()) {
            int i = peek();
            top_index = top_index - 1;
            return i;
        } else {
            throw new Exception("Stack is empty");
        }
    }

    public int peek() throws Exception {
        if (!isEmpty()) {
            return stack[top_index];
        } else {
            throw new Exception("Stack is empty");
        }
    }

    public boolean isEmpty() {
        return top_index == -1;
    }

    public boolean isFull() {
        return top_index == capacity() - 1;
    }

    public int capacity() {
        return stack.length;
    }
}
```

Listing 2. IntStack

B.3 ColourPicker

```
public class ColourPicker {

    public Color toColor(boolean number, int i, String name) {
        if (number) {
            if (i == 0) {
                return Color.WHITE;
            } else if (i == 1) {
                return Color.RED;
            } else if (i == 2) {
                return Color.GREEN;
            } else if (i == 3) {
                return Color.BLUE;
            } else if (i == 4) {
                return Color.MAGENTA;
            } else if (i == 5) {
                return Color.YELLOW;
            } else if (i == 6) {
                return Color.CYAN;
            } else {
                return Color.BLACK;
            }
        } else {
            switch (name) {
                case "red" : return Color.RED;
                case "green" : return Color.GREEN;
                case "blue" : return Color.BLUE;
                case "yellow" : return Color.MAGENTA;
```

```
                case "magenta" : return Color.YELLOW;
                case "cyan" : return Color.CYAN;
                default : if(i == 0) {
                    return Color.WHITE;
                } else {
                    return Color.BLACK;
                }
            }
        }
    }

    public Color pickRGB() {
        Random rnd = new Random();
        int i = rnd.nextInt() * 10;

        if (i < 4) {
            return Color.RED;
        } else if (i < 7) {
            return Color.GREEN;
        } else {
            return Color.BLUE;
        }
    }
}
```

Listing 3. ColourPicker

B.4 EnglishNumberToWords

Source: `http://www.rgagnon.com/javadetails/java-0426.html`.

```
public class EnglishNumberToWords {
  private static final String[] tensNames = {
      "",
      " ten",
      " twenty",
      " thirty",
      " forty",
      " fifty",
      " sixty",
      " seventy",
      " eighty",
      " ninety"
  };

  private static final String[] numNames = {
      "",
      " one",
      " two",
      " three",
      " four",
      " five",
      " six",
      " seven",
      " eight",
      " nine",
      " ten",
      " eleven",
      " twelve",
      " thirteen",
      " fourteen",
      " fifteen",
      " sixteen",
      " seventeen",
      " eighteen",
      " nineteen"
  };

  private EnglishNumberToWords() {}

  private static String convertLessThanOneThousand(int number) {
```

```
41      String soFar;
42
43      if (number % 100 < 20){
44        soFar = numNames[number % 100];
45        number /= 100;
46      }
47      else {
48        soFar = numNames[number % 10];
49        number /= 10;
50
51        soFar = tensNames[number % 10] + soFar;
52        number /= 10;
53      }
54      if (number == 0) return soFar;
55      return numNames[number] + " hundred" + soFar;
56    }
57
58
59    public static String convert(long number) {
60      // 0 to 999 999 999 999
61      if (number == 0) { return "zero"; }
62
63      String snumber = Long.toString(number);
64
65      // pad with "0"
66      String mask = "000000000000";
67      DecimalFormat df = new DecimalFormat(mask);
68      snumber = df.format(number);
69
70      // XXXnnnnnnnnn
71      int billions = Integer.parseInt(snumber.substring(0,3));
72      // nnnXXXnnnnnn
73      int millions  = Integer.parseInt(snumber.substring(3,6));
74      // nnnnnnXXXnnn
75      int hundredThousands = Integer.parseInt(snumber.substring(6,9));
76      // nnnnnnnnnXXX
77      int thousands = Integer.parseInt(snumber.substring(9,12));
78
79      String tradBillions;
80      switch (billions) {
81        case 0:
82          tradBillions = "";
83          break;
84        case 1 :
85          tradBillions = convertLessThanOneThousand(billions)
86              + " billion ";
87          break;
88        default :
89          tradBillions = convertLessThanOneThousand(billions)
90              + " billion ";
91      }
92      String result =  tradBillions;
93
94      String tradMillions;
95      switch (millions) {
96        case 0:
97          tradMillions = "";
98          break;
99        case 1 :
100         tradMillions = convertLessThanOneThousand(millions)
101             + " million ";
102         break;
103       default :
104         tradMillions = convertLessThanOneThousand(millions)
105             + " million ";
106     }
107     result =  result + tradMillions;
108
109     String tradHundredThousands;
110     switch (hundredThousands) {
```

```
111         case 0:
112           tradHundredThousands = "";
113           break;
114         case 1 :
115           tradHundredThousands = "one thousand ";
116           break;
117         default :
118           tradHundredThousands = convertLessThanOneThousand(hundredThousands)
119               + " thousand ";
120       }
121       result =  result + tradHundredThousands;
122
123       String tradThousand;
124       tradThousand = convertLessThanOneThousand(thousands);
125       result =  result + tradThousand;
126
127       // remove extra spaces!
128       return result.replaceAll("^\\s+", "").replaceAll("\\b\\s{2,}\\b", " ");
129     }
130 }
```

Listing 4. EnglishNumberToWords