An Implementation for Test-Data Generation Using Genetic Algorithms

Dawei Liu
Department of Computer Science
University of Kentucky
Lexington, KY, United States
+1 859-447-7345
dawei.liu@uky.edu

ABSTRACT

In order to reduce the high cost of manual software testing and at the same time to increase the reliability of the testing processes researchers and practitioners have tried to automate it. [1] One of the automatic test data generation is by using genetic algorithm. This paper would introduce an implementation for test-dada generation using genetic algorithm. It shows us a tool suite for genetic test-data generation with *EvoSuite*, *JUnit* and *EMMA*. This project combined the strategies of test case combination, coverage evaluation and test suite selection. The goal of this paper is to implement a genetic test suite generation tools for higher code coverage of the test suites, and to evaluate how much code coverage of unit tests could the tool suite promote.

Keywords

Genetic Test, Test Suite Generation, Test Case Combination, Test Suite Selection, Code Coverage for Unit Test

1. INTRODUCTION

When people begin the unit tests, the test data is indispensable. A straightforward way is to write the unit test codes manfully and then test them with unit test tools, e.g. *JUnit*. However, this strategy could not be efficient or scalable, and the correctness of the unit test could not be promised. Therefore, test generation tools are innovated to relieve software testers from manually coding.

1.1 Motivation

One of the popular test tools used for Java program testers is *EvoSuite*. It is a tool that automatically produces test suites for Java programs that achieve high code coverage and provide assertions. *EvoSuite* implements several novel techniques, leading to higher structural coverage and an efficient selection of assertions based on seeded defects, which is a critical feature that other Java tools miss. [2]

For the code branch coverage strategy used in *EvoSuite*, it provides us a considerable code coverage on unit test. However, the code coverage of a test suite generated by *EvoSuite* depends on the randomization seed for suite generation. This feature of *EvoSuite* makes me consider if it is feasible to promote the test code coverage by combination of the test suites.

Inspired by the paper Test-Data Generation Using Genetic Algorithms, genetic algorithm would be a considerable approach to leverage the test code coverage of test suits. The genetic algorithm conducts its search by constructing new test data from previously generated test data that are evaluated as good candidates. [3] However, the genetic test suite generation tool called *TGen* is not available for free use. Neither its source code nor binary executable files could be found on open source hubs. Another problem of *TGen* is that its target language. C is the only language it would support. It is almost impossible to apply it directly for OO languages, e.g. Java. Therefore, this paper introduce a genetic test suite generation tools for code coverage promotion of test suites generated by EvoSuite, using strategies of test case combination, coverage evaluation and test suite selection.

1.2 CONCEPTS

We use the similar concepts that has already existed in biology – chromosome, gene, and mitochondrion to help us understanding the processing of our implementation. The four categories of concepts are illustrated as Figure 1 and following:

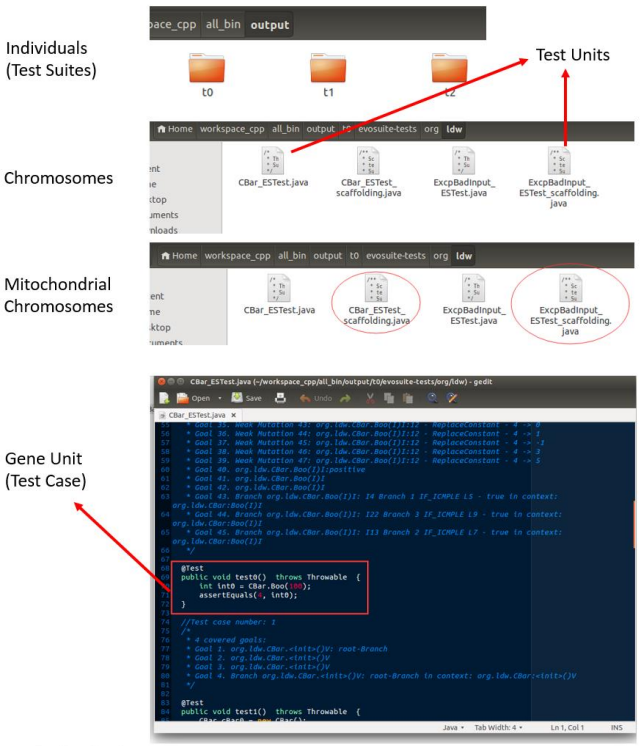

Figure 1 – Concepts for Test Suites

Individual: A test suite would be seems as an individual. Each test suite in a same group would be assumed having the same number and types of chromosomes.

Chromosome: Each java file of unit tests would be seemed as a Chromosome. The gene units (test cases) in the couple's chromosomes (except mitochondrial) with the name would exchange each other into the new generation.

Gene Unit: Each test case in a java file of unit tests would be seemed as a gene unit. The unit of genes would not be changed into the new generation, but the combination of genes would be based on the randomly choosing gene unit from its parents. The position of a gene unit would not be changed after combination.

Mitochondrial Chromosomes: As mitochondrion in animals, mitochondrial chromosomes would not exchange genes into the new generation. New generation's mitochondrion would only derivate from one of the parent. In our implementation, we consider files whose names are "*_scaffolding.java" the Mitochondrial chromosomes, since any two of them with the same file name would not be different each other.

1.3 Test Suite Generation Steps

The first step of test suite generation is to generate initial test suites. Users can use command *OrgGen* to generate their initial test suites, also they can write the *JUnit* test suites which have the similar structure of the test suites generated by *EvoSuite*. With the initial test suites, user can execute *CombGen* to make the combination of test cases in each test suites. With this test suites, users can test them directly, or they can use *DataTest* to get the code coverage information, for further case selection operation by *TRJudge*. By looping the operations of *CombGen*, *DataTest*, and *TRJudge* for several time, users can finally get the best coverage of the test suites.

In section 2, it would introduce the architecture and the implementation of the tool suite. Section 3 would shows us the evaluation on the code coverage of test suites. Section 4 would concludes the content of this paper and bring us the consideration on future study for its drawbacks.

2. ARCHITECTURE AND IMPLEMENTATION

2.1 Architecture

The architecture of this project is showed in Figure 2. The project is consisted with four main commands: *OrgGen*, *DataTest*, *CombGen*, and *TRJudge*. The four commands represents the basic functionalities of genetic test suite generation. OrgGen is used to generate initial test suites; *CombGen* is designed for data case combination; *DataTest* is to evaluate the coverage of each test suites; *TRJudge* is to ponder the average coverage of test suites and to eliminate the undesirable test suites based on the ranks of coverages.

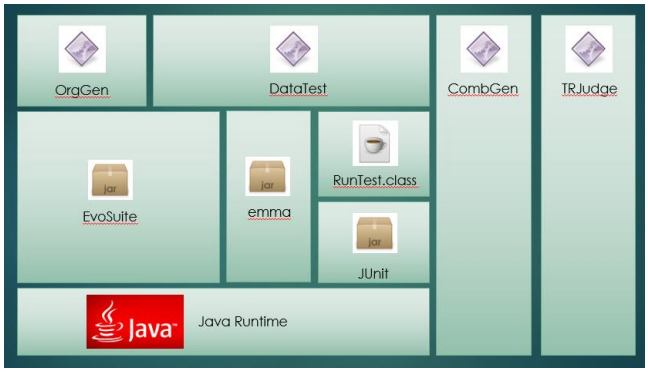

Figure 2 - Architecture

2.2 OrgGen

OrgGen is designed to generate initial test suites by using EvoSuite. It would make a group for the test suites generated by EvoSuite.

The command as following:

OrgGen <srcDir> <binDir> <number of suites> [<seed>]

Using these arguments, user can specify the project to test. <srcDir> would be assigned with the source directory of the project and <binDir> would be set with its binary file directory. <number of suites > can be used to tell OrgGen how many suites should OrgGen generates. <seed> can be specified for replaying the test suite generation. All the test suites would be randomly generated by the <seed>, and they would be placed into the directory *output.*

2.3 CombGen

Combination is implemented in the *CombGen*. It two different command formats.

1) *CombGen <suite1> <suite2> <reproduction limit> [<seed>] [<beginIndex>].*
   This command can be used to generate child test suiltes combined from *suite1* and *suite2*.
2) *CombGen -g <group dir> <mingle rate (%)> <reproduction limit> [<seed>] [<beginIndex>]*
   Randomly choose couples from group directory to generate child test suites.

The child test suites are placed into directory *output-comb*.

In *CombGen*, it would analyze the unit test java files in each test suites. It would collect the test member functions as test cases. The test cases in two test suites would be combined randomly each other when it find their file names are the same. After the combination, it would write the combined test cases into a new unit test java file. The results of combination can be controlled by specifying the argument *seed*.

2.4 DataTest

DataTest is used to generate test reports.

Command Line:

1) *DataTest <binDir> <SuiteDir>*
2) *DataTest -g <binDir> <groupDir>*

Firstly, it would call "javac" compile the unit tests of assigned test suites. After the compilation, it would execute *EMMA* to run *RunTest.java* to execute the unit tests. The test results would be placed into directory *DataTest_Report* in each test suite directory, and there would be two types of test reports: one is unit test result, which contains the running statues of *EMMA*, *JUnit*, and/or *EvoSuite* and the test results (e.g. outputs, exceptions, assertions, etc.) of unit test; another is code coverage result, which has the coverage analysis for different packages.

In the process of *DataTest*, *EMMA* is used to analysis the code coverage, and *EvoSuite* and *JUnit* are used for unit test reports.

2.5 TRJudge

To promote the code coverage of the unit tests, a coverage analyzing tool and a test suite selection tool are indispensable. *TRJudge* is designed to analyze the coverage report generated by *DataTest* which uses *EMMA* for code coverage information. The command line of TRJudge as following:

1) *TRJudge -e <group dir> -linecoverage*
2) *TRJudge -j <group dir> <dst dir> -linecoverage -top <number>*

The first command line can be used to get the average line coverage of all test suites. TRJudge would account all the line coverage information in each test suite, and bring us the final analysis report.

The second command line contains the functionality of the first line command. Moreover, it would select top ranked test suites by line coverage to copy them into the destination directory. They have the better code coverage than that were not selected by TRJudge.

3. EVALUATIONS

3.1 Test Source

In the evaluation, I use three Java files to present the project which needs unit tests. One file for declaration of an exception, another two are the units need to be tested. The codes of the three files show in Figure 3, 4 and 5.

```
package org.ldw;

public class ExcpBadInput extends Exception {

    /**
     *
     */
    private static final long serialVersionUID = 8456187775862736079L;

}
```

Figure 3 – *ExcpBadInput.java*

```
package org.ldw;

public class CBar {
    public static int Boo(int v) {
        if(v > 400)
            return 1;
        else if(v > 300)
            return 2;
        else if(v > 100)
            return 3;
        else
            return 4;
    }
}
```

Figure 4 – *CBar.java*

```
package org.ldw;

public class CFoo {
    void Bar(String a, int b) throws Exception {
        String p = "";
        if("a".equals(a))
            p = "haha";
        else if("abc".equals(a))
            p = "haha1";
        else if("add".equals(a))
        {
            assert(b != 32); // b != 32
            p = "9d2w";
        }
        else if(a.equals("adv")) // a != NULL
            p = "bbbbb";
        else if("vs".equals(a))
            p = "hhhh";
        else if("13".equals(a))
        {
            if(b < 78) // b >= 78
                throw new ExcpBadInput();
            assert(b < 132); // b < 132
            p = "1345";
        }
        else if("dv".equals(a))
            p = "vc";
        else if("gb".equals(a))
        {
            if(b > 0)
                p = "dd";
            else
                p = "ff";
        }
        int cons = 3994/(b-29); // b != 29
        p = p + cons;
        System.out.println(p);
    }
}
```

Figure 5 – *CFoo.java*

3.2 Initial Test Suites Generation

We use a script *makeInitialTestSuites.sh* to simplify the steps of initial test suites generation. This script would generate 100 test suites by using *OrgGen*. However, not all of the test suites can be used for the combination. Since the EvoSuite may crash with different seed, the test units in one test suite may not be complete. Therefore, the script would call another script *filtInitialTests.sh* to eliminate the test suites that are not complete. Also, *filtInitialTests.sh* would bring us the first code coverage reports for the remained test suites. By these reports, *filtInitialTests.sh* would call *TRJudge* to select 20 initial test suites. Therefore, the script *makeInitialTestSuites.sh* would bring us the 20 initial test suites selected from that generated by *OrgGen*.

3.3 Genetic Generation

To generate the combined unit tests, script *one_generation.sh* would call *CombGen* to combine the selected test suites. Since *CombGen* would place the combined test suites into the directory *output_comb*, the script would remove the original directory *output* and rename *output_comb* to *output*. Then the script would call *DataTest* to retrieve the code coverage of each test suite. After that, it would call *TRJudge* to select test suites for next time of combination. The average code coverage of the selected test suites from the demo shows in Table1. It tells us from the initial generation to the 4th generation the code coverage are closely increased with 2%. From the 4th generation to the 5th generation, the increasing is 0.2%, which means after 4th generation the code coverage results from combination and selection would be plain and stable.

Table 1 – Average Code Coverage in Each Generation

generation	initial	1st	2nd	3rd	4th	5th
Code Coverage (%)	78.9%	81.3%	83.5%	85.9%	87.3%	87.5%

4. Conclusion

This paper brings a tool suite for genetic test suite generation. As the result of the evaluation, it proves the tool suite can be used for code coverage promotion for unit tests with well-designed scripts. The four commands represent the four basic functionality of the suite, and they can be executed with specific arguments and orders to generate the considerable code-covering test suites.

4.1 Drawbacks for future studies

Since the initial test suite generation depends on EvoSuite, once EvoSuite crashes, some chromosomes (test units) would be absent in individuals (test suites). Since any couples with different names of chromosomes would not have their baby, this drawback would potentially impact the reproduction rate of test case generation. To ensure the completeness of test units (chromosomes) in initial test case units, this project introduce script *filtInitialTests.sh* to eliminate any test suites that has not complete test units. However,

it would guarantee the completeness of test suites only if there is at least one test suite that is complete, and it cannot solve the problem of inefficiency brought by the crashes of EvoSuites.

Mutation is not implemented yet in the project. Moreover, since the project does not use the parallel execution, it would not be more efficient on multiple-CPU platforms. Future study is needed for test suite mutation and parallel task execution.

5. REFERENCES

[1] Edvardsson, J., A Survey on Automatic Test Data Generation, staff.unak.is, 1999

[2] Fraser, G., and Arcuri, A., *EvoSuite: Automatic Test Suite Generation for Object-Oriented Software*. In *ESEC/FSE '11 Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ACM New York, NY, USA, 2011, 416-419.

[3] Pargas, R. P., Harrold, M. J., and Peck, R. R., *Test-Data Generation Using Genetic Algorithms*, Journal of Software Testing, Verification and Reliability. 1999