Software Testing and Reverse Engineering

Testing Assignment

Arshad Ali
a.ali-1@student.utwente.nl
s1869485 - CS CybSec

Kasper Wijburg
k.m.wijburg@student.utwente.nl
s1322524 - CS CybSec

March 20, 2017

1 Introduction

As society advances, software plays an increasingly important role. Naturally so does the validation of this software. Errors in software can be very costly: In 2002 a report by the NIST (National Institute of Standards and Technology) reported that software bugs cost the U.S.A. $59.5 billion annually[1]. In addition to financial impact, in the past faulty software has also led to a number of fatal accidents[2]. It is therefore important that software is rigorously tested before it is put to use. An issue however is that the complexity of testing software tends to grow much faster than the complexity of the software itself; in the worst cases this growth difference is exponential[3]. This results in software which can not be exhaustively tested for each edge case and the need for an alternate form of testing arises.

Search-based software engineering (SBSE) can be used to convert software engineering problems into computational search problems, to which a metaheuristic can then be applied. Metaheuristic search techniques are high-level frameworks that utilise heuristics to seek solutions for combinatorial problems at a reasonable computational cost[4]. These metaheuristic search techniques can be used to automate test data generation, both for functional and structural testing. They have been used to test grey-box properties such as safety contraints, and to test non-functional properties, such as worst-case execution time[4].

2 Auto Generate Test

Not all bugs result in program crashes, and for the verification of the correctness of software, test outcomes do not always have any formal specification. As the difficulty of this task a small set of test sets production is important, and its representativeness can measure using code coverage [5]. The common approach having one coverage goal has fundamental problem of targeting one coverage goal [5].

Java Programing language target by the tools and techniques for automated test generation due it's popularity. Enterprise applications such as Java Enterprise Edition (JEE) or Spring frameworks are the reasons of its popularity. Web containers e.g WildFly and GlassFish handles the code unit [6]. Valid scenarios would not be represent by automatically generated unit without considering how these beans initialize by web container.

Test cases can automatically be generated by EvoSuite tools. A novel hybrid approach used by EvoSuite for optimizing and satisfying a coverage criterion of whole test suite[5].EvoSuite works on byte-code level and for the test cluster it collects all necessary information from byte-code via Java Reflection. Which means it does not required the source code of SUT and is also applicable to other language that compile to Java byte-code. All language construct can be handled by EvoSuite. Top level class consider by EvoSuite at a time during test generation. During execution, all class anonymous and its member classes at byte code level instrumented to keep track of called methods[5].

2.1 Generating tests

For generating tests in EvoSuite prerequisites are required to be installed:

1. The first requirement is a java 8 JDK and a command-line prompt, with java and javac.
2. Obtain and install EvoSuite

The jar file will contains all dependencies libraries with EvoSuite that is needed to generate tests. For generating test, we need to specify two things: Class under test and Class Path where the bytecode and dependencies can be found.

3 EvoCrash

EvoCrash is a search-based approach to crash replication based on using data from crash stack traces[7]. It is based on EvoSuite[5], a well-known automatic test suite generation tool for Java. In its search for a test case that can trigger the target crash, thereby revealing the buggy frame in the crash stack trace, it employs a Guided Genetic Algorithm (GGA) and a smart fitness function.

EvoCrash is given a log file of a crash as input. It parses this file and extracts the crash stack frames of interest. The last of these frames is where the exception was thrown, however the cause of the exception can be in any of the frames or even outside the stack trace. EvoCrash targets this last frame as it attempts to generate a stack trace as close as possible to the original trace.

3.1 Fitness function

A fitness function can be used to summarise how close a given design solution is to achieving the set aims, in this case how close the generated stack trace is to the original. The fitness function employed by EvoCrash is formulated considering following three main conditions: the statement where the exception is thrown must be covered, the target exception must be thrown and has to be covered, the generated stack trace must be as similar to the original as possible. This fitness function is defined by Soltani et al.[7] as follows:

$$f(t) = 3 \cdot d_s(t) + 2 \cdot d_{except}(t) + d_{trace}(t) \tag{1}$$

We identify the following variables: $d_s(t)$ denotes the distance between t and the target statement; $d_{except}(t)$ indicates whether the target exception is thrown or not and is therefore a binary value; lastly $d_{trace}(t)$ denotes the distance between the generated stack trace and the original trace.

In order to determine the line distance $d_s(t)$, two well-known heuristics are used: approach level, and branch distance. For the branch distance, the rules established by Soltain et al.[7] are used in order to determine its scores.

If $d_{except}(t) = 1$, the target exception is not thrown, in this situation the trace distance, $d_{trace}(t)$ is set to its maximum value: 1.0. In the situation where $d_{except}(t) = 0$ and therefore the target exception is thrown, the value of $d_{trace}(t)$ is calculated using stack trace distance which is defined as follows[7]:

$$D(S^*, S) = \sum_{i=1}^{n} min\{diff(e_i^*, e_j) : e_j \in S\} \quad (2)$$

In this function S^* is the expected trace; S is the actual stack trace triggered by the given event t; $diff(e_i^n, e_j)$ is the distance between trace element e_i^* in S^* and trace element e_j in S.

Equation (2) is then used in the following definition[7] for the calculation of the trace distance:

$$d_{trace}(t) = \varphi(D(S^*, S)) = D(S^*, S)/(D(S^*, S) + 1) \quad (3)$$

3.2 Guided Genetic Algorithm

GGA contains the three main steps of traditional genetic algorithms[7]. First the creation of an initial population of random tests. These tests are then evolved over subsequent generations using crossover and mutation. Lastly The fittest tests of each generation are selected using the fitness function. However unlike traditional genetic algorithms in coverage-based unit testing tools, GGA gives a higher priority to methods involved in the target failure.

3.2.1 Initial population

The routine with which the initial population is generated plays a paramount role since it performs sampling of the search space[8]. Traditional coverage-based tools often use routines which are designed to generate a population which calls as many methods as possible[5]. However this is not the main goal for crash replication. The routine used by GGA gives higher importance to methods contained in the crash stack frames, thereby greatly reducing the size of the search space.

3.2.2 Crossover and mutation

Even when each of the tests created by the initial population routine contain one or more of the methods from the stack trace, these methods can be lost during the crossover process performed by genetic algorithms. To prevent this occurrence, GGA employs a guided single-point crossover operator. This method of crossover applies a correction procedure which inserts all required objects and primitive variables into non well-formed offspring.

After performing crossover, GGA may mutate the tests by adding, changing and removing some statements. Using traditional mutation, this may result in the removal of the methods from the stack trace. Therefore GGA employs a guided-mutation operator. This version of mutation checks whether the test still contains at least one of the target methods. Should the test contain none of the target methods, it continues to mutate the test untill at least one of these methods has been inserted.

3.2.3 Post processing

At the end of the search process, GGA uses the fitness function as defined in Equation (1) to determine the fittest test cases. However due to the random mutations during the search process, the final test t_{best} can contain methods which are irrelevant to the replication of the test. Therefore, GGA post-processes t_{best} by applying the test optimization routines available in EvoSuite[5].

4 Assignment

In the assignment, we received 2 debugging task to be analyse the crash report and try to fix the crash, or suggest the solution to rectify these causes of crash based on debugging. For this task we need to install some prerequisite tools and plug-ins for debugging:

(i) Git

(ii) Maven

(iii) Eclipse

(iv) Maylen

Following the instruction for debugging Task 1 using Eclipse, where JUnit test applied on the classes to find the crash, Some classes do not show any crash but in TestSuite.class where Test Class was not found in the project that cause the crash. While, In Task 2 the Null pointer exception found to cause the crash with applying the Junit test on classes. With the lack of java language knowledge, It was not possible to test and find the possible causes of crash program, Also the manual process is not feasible and time taking for such kind of debugging and finding the bugs in the code that was written by someone else. But using Eclipse with the given EvoCrash stack trace and apply the JUnit test on the project help to identify the bugs, and the causes that can crash the program, With precise information regarding bugs can help to find the solution to mitigate the causes of crash.

5 Reflection

5.1 Previous programming endeavours

We have not had many programming endeavours in Java that warranted the use of extensive test case generation. The largest project in Java was a simple game for a course in the first year of the bachelor *technische informatica* at the University of Twente. However part of that assignment was to create your own JUnit tests and therefore we feel that using EvoCrash would not have spared us any time there.

We can however envision a different situation in which not EvoCrash, but a similar tool meant for other languages than Java would have been useful. During another course of the bachelor *technische informatica* we had to design a bridge weight distribution simulation in Haskell. The first three calculations were be performed by the program correctly, the fourth and later calculations gave output that was complete nonsense. After several hours of attempting to fix the issue we were still unable to discover the error in the code. We therefore asked the professor for assistance in locating the bug, however he to was unable to find it within reasonable time. In situations like this a tool like EvoCrash for Haskell would have been a godsend.

5.2 Improving EvoCrash

The usability aspect of EvoCrash has a lot of room for improvement. Figuring out why it wouldn't run and getting it to run was an issue which consumed nearly as much time as EvoCrash saved compared to manual testing. Additional output and clarification when EvoCrash is run could greatly improve this aspect. An example of this is while setting up EvoCrash running the command "*java -jar evocrash*-1.0.0.*jar*" gave no output at all. This makes it hard to determine whether the program gives no output in the command line when run successfully, or if it simply did not do anything.

5.3 Using EvoCrash

It is highly unlikely we will use Evocrash in our own programming endeavours. The main reason for this being that we never use Java unless it is mandatory. Another reason is that we rarely write code which is intended for use by others than ourselves and therefore rarely create tests for the code, as the code we write for personal use often is not complex or crucial enough to warrant extensive testing.

References

[1] RTI, "The economic impacts of inadequate infrastructure for software testing," National Institute of Standards and Technology, Tech. Rep. 7007.011, 2002. [Online]. Available: `http://www.nist.gov/director/planning/upload/report02-3.pdf`.

[2] M. Zhivich and R. K. Cunningham, "The real cost of software errors," *IEEE Security Privacy*, vol. 7, no. 2, pp. 87–90, 2009, ISSN: 1540-7993. DOI: `10.1109/MSP.2009.56`.

[3] J. Tretmans, "Formal methods and testing: An outcome of the fortest network, revised selected papers," in, R. M. Hierons, J. P. Bowen, and M. Harman, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, ch. Model Based Testing with Labelled Transition Systems, pp. 1–38, ISBN: 978-3-540-78917-8. DOI: `10.1007/978-3-540-78917-8_1`. [Online]. Available: `http://dx.doi.org/10.1007/978-3-540-78917-8_1`.

[4] P. McMinn, "Search-based software test data generation: A survey," *Software Testing Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[5] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[6] A. Arcuri and G. Fraser, "Java enterprise edition support in search-based junit test generation," in *International Symposium on Search Based Software Engineering*, Springer, 2016, pp. 3–17.

[7] M. Soltani, A. Panichella, and A. van Deursen, "A guided genetic algorithm for automated crash reproduction," in *Proceedings of the 39th International Conference on Software Engineering (ICSE 2017)*, ACM, 2017.

[8] A. Panichella, R. Oliveto, M. Di Penta, and A. De Lucia, "Improving multi-objective test case selection by injecting diversity in genetic algorithms," *IEEE Transactions on Software Engineering*, vol. 41, no. 4, pp. 358–383, 2015.