Assignment 1 – Testing

Jan-Gerrit Harms - 4163400

Jeffrey Steen - s1193074

Priyanka Radja - 4615026

Search Based Software Testing and Crash Replication

The term Search Based Software Engineering (SBSE) describes a field of work in which Software Engineering problems are reformulated as optimization tasks that can be solved using meta-heuristic search techniques. The general idea is that the development of software consists of two types of tasks, namely those that require creative thinking to discover a solution and those that require tedious work to implement and extensively test the solution. In the original human-centred approach to Software Engineering, developers perform both while SBSE tries to utilize the reliability and speed of machines for the latter task, to reduce errors and time, and thereby the costs of software projects.

This is done by defining an optimization problem with an initial solution space and a fitness function which determines how close a solution is to the optimum that we are looking for. Then a general-purpose algorithm can be used to find that optimum of the fitness function, which should correspond to the optimal solution of the Software Engineering problem. The experienced reader will realize the challenges which this approach introduces. Firstly, large production-ready software is extremely complex so the solution space for any type of problem will be huge and non-deterministic. And secondly, the fitness function needs to model a solution which is hard for a non-mathematical design task. The first problem led to the use of meta-heuristic search techniques for the optimization such as hill climbing, simulated annealing and evolutionary algorithms. The latter problem is largely a design task which is discussed in various scientific works.

The success of SBSE varies with the subtask of the development process that is targeted to be automated, the most successful of which are requirements, design and testing. This report will be mostly covering the Search Based Testing techniques and more specifically crash replication.

Testing is one of the more work-intensive tasks in software development and can rather easily be converted to an optimization problem such as “What is the smallest set of tests to cover all branches in the code?” That is why this field has gained a lot of focus from the scientific community and progress is made quickly. Especially, the data generation for test cases and reproducing crashes is a time-consuming task that seems to be a good candidate for machine support. Subject of this analysis will be the two software projects, EvoSuite and EvoCrash, which target test-suite generation and crash reproduction, respectively, using genetic algorithms.

The rest of this report is structured as follows. First we will introduce the two software projects and explain their main characteristics. Next we will describe our personal experience with EvoCrash, and then reflect on the usefulness and usability of the software. Finally conclude the report and provide some recommendations.

Test case generation

The design of good tests for software is a difficult task even for experienced developers. There is an almost infinite solution space, i.e. the amount of possible cases to test is almost infinite for large software, and decisions have to be made what to test. Luckily, some metrics have been defined which help the developer assess the quality of the test suite. These metrics are usually coverage goals such as line coverage or branch coverage. Once the developer knows what to test in order to achieve a certain coverage goal, he also needs to specify the expected behaviour which is also called an oracle. There are some oracles which are universally true like “the program should not crash” but in general such oracles need to be manually added.

In software supported testing, the generation of inputs is done using an algorithm that optimizes some fitness function which consists of one or more coverage goals. However the developer still needs to add the oracle to finish the test cases. That is why there needs to be a trade-off between maximizing the coverage and having a small amount of test cases to reduce manual work.

The automated test data generation is in the focus of this section and the available approaches differ in two parts, the search algorithm used and the type of testing which is done.

Search algorithms

Hill Climbing - This algorithm starts with an arbitrary solution to a problem and incrementally changes this solution to check if the modified solution produces a better result. If so, the change is added to the solution and this process is repeated until no further improvements could be found.

Simulated Annealing - This algorithm, like hill climbing, starts with an arbitrary solution, and keeps changing it. The difference however, is that in simulated annealing, changes that do not improve the result, but allow for other changes that could not be reached by normal means are accepted. The negative allowed impact of changes is then continuously reduced until a single potential solution remains.

Evolutionary algorithms - This algorithm starts with a number of potential solutions. By applying a quality function, the least “fit” solutions are eliminated, and from the remaining solutions, additional solutions are generated. This process is repeated several times, until a certain fitness is reached.

Types of testing

Structural (White-Box)Testing - This testing approach involves knowing the code that you are going to be testing, and devising tests specifically for that code. This approach aims to test if all pieces of code do what they are meant to do.

Functional (Black-Box) Testing - This testing approach involves writing tests that see if the program has the intended functionality. This is not to see if the code is correct, but if the program meets all the requirements set up.

Grey-Box Testing - This testing approach is a middle ground between white-box testing and black-box testing. For grey-box testing, the structure of the code that needs to be tested is partially known, but not completely. This means the tests will be based on the internal code to some extent, but still involve the functionality approach of black-box testing.

EvoSuite

EvoSuite is one example of a software for test-suite generation which has been developed as a research project by Gordon Fraser and Andrea Arcuri. It implements a search-based approach to generate JUnit tests for existing software projects and was tested on 1741 classes with a higher coverage outcomes and smaller test suites than comparable existing software. Fraser and Arcuri realized that the existing tools were only considering one coverage goal at a time which lead to problems because goals are not independent and sometimes not feasible, thus covering one goal will sometimes cover another goal as well but not if the order was reversed. Also, some branches are harder to cover than others which makes the solution of these software unpredictable and unstable.

They came up with a solution which is to cover “all coverage goals at the same time while keeping the number of tests cases as small as possible.”(Fraser et.al.). The technique they use starts with an initial population of test suites and uses a Genetic algorithm to find an optimal test of test cases which is the result of the algorithm. We can therefore conclude that the software uses Evolutionary algorithms and structural White-Box Testing as their underlying concepts.

EvoCrash

EvoCrash is a post-failure approach used for crash debugging which employs Guided Genetic Algorithm(GGA) to generate JUnit tests that Java developers can use for debugging. It is built on top of the EvoSuite and helps in finding an ideal test case which crashes at the same location as the original crash and provides a similar stack trace [1]. Manual crash replication is simply too tedious and also the data available to the user for the replication is insufficient. Although many existing approaches are available for crash replication, they suffer from performance overhead [2] and privacy issues [3]. EvoCrash being a post-failure approach employs only the stack trace files for creation of the ideal test case as other types of information gathered after the failure may not be available to the developers. Since the stack trace is used as guidance in the search for a test case, the search space is reduced. An initial population of tests is created which addresses at least one of the methods given in the stack trace files. Crossover and mutation operators are used to avoid creating tests that do not call the failing methods. EvoCrash also uses a fitness function which assesses the different candidate test cases.

EvoCrash works by using the stack trace as input and parses it into the type of exception and the list of stack frames indicated by the method name, class name and the line number where the failure occurred. The last stack frame denotes where the failure occurred and is used as the class under test (CUT) while synthesizing the ideal test case. In order for a test case to pass as optimal by the fitness function, it should cover the line where the exception occurred, throw the same target exception and generate stack trace similar to the original [1]. The fitness function used by EvoCrash is given below:

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

where,
$d_s(t)$ - location of crash

$d_{except}(t)$ - binary value (0 or 1) indicating if target exception is thrown or not
$d_{trace}(t)$ - distance between generated stack trace and expected stack trace (if any).

The Guided Genetic Algorithm (GGA) used by EvoCrash gives high priority to the methods in the CUT that were involved in the target failure with the help of three genetic operators. Therefore, the steps of the genetic algorithm are i) initial population (comprising of random tests), ii) evolution of the tests using crossover and mutation and iii) selection of fittest test using the fitness function (post-processing).

Personal experience with EvoCrash

Before starting the work on this report we had a chance to test EvoCrash ourselves on real-world exception stack-traces. We were given two debugging tasks in sequence; one of them included the stack-trace and a test case generated by EvoCrash to reproduce the stack-trace while the other only had the stack-trace. Our tasks were to fix the bug and record the time to finish it.

Because each of our team-members was given a separate task we will share our experiences in the following three sections.

Assignment 1 (Jan)

I have little experience with JUnit and Java itself and was unsure if I would be able to fix the bug itself. However by having the test to reproduce the bug, I was sure which argument resulted in the exception. All that I needed to do was read through the stack trace and go frame by frame, each time checking what should happen with the given input. This helped me to quickly find the point where the input caused a problem. A theoretical fix was easily implemented, whether the fix is correct remained questionable. The good thing about having a test case to reproduce the bug is that it gives you confidence that the solution is correct. Especially in combination with a large test-suite which tests if the fix introduced other unfavorable behavior. That was not the case.

At first I did not realize the advantage that the EvoCrash test-case would give me so I started on the second task and this time I was quite confident that I would be able to fix the bug. Without any way to reproduce the bug I first started with reading through the frames of the stack-trace however this is harder if you don’t know which input parameters caused the crash. So, I started writing my own test-case to reproduce the bug, however it succeeded with all the parameters I tried and without reproducing the crash I ran out of time.

Therefore I can conclude that the generated test case definitely helped with fixing the bug in the software because it speeds up reproducing the crash and gives confidence in the fix that you provide.

Assignment 1 (Jeffrey)

Having the test that created the error, along with the Stack trace made it easy to locate the origin of the error. By analyzing the code, using the eclipse debugger it became clear rather quickly what was causing the error to occur. By testing for that situation this error could be avoided. Both the EvoSuite test, as well as the Maven test did not report any errors after implementing this fix. For the second task, there was no test case provided that showed the origin of the error. Neither was Maven able to find the error shown. This meant that only the provided stack trace could be used. When looking at the code this trace pointed to, as well as the type of error that was happening, I had a good idea of what could be causing the problem. Expanding one of the provided test cases with a test of my own allowed me to recreate the error quite quickly. This error was easily caught and prevented. However, because I was unable to reproduce the problem, I am unsure if this is the only location the error can originate from. Also, I am not completely sure the functionality of the code remained intact. No problems were found by Maven after making the change, but that does not guarantee there aren't any.

Assignment 1 (Priyanka)

The first task was debugged solely by looking at the stack trace as no test case was provided. Having a look at the function with the line of error as denoted by the stack trace provided a clear picture of what had went wrong. The bugs were fixed by modifying and adding some parameters. Example: the declaration of the map type was not parameterized and the error was fixed by adding an appropriate parameter <Object, Object>. Note that some of the errors denoted by the stack trace for the first task were undetected by Maven test.

The second task had a test case provided and it took quite some time to understand the test case and its functionality. The error was identified as caused by an incorrect definition of the LoggingEvent which was later fixed with the help of the test case.

How EvoCrash could have helped our own programming endeavours

Fixing errors and crashes manually with the help of a stack trace can be time consuming and may not always be successful. This is because stack frames in the trace show where the exception or the crash occurred but the root cause of such a failure could be in any of the frames or even outside the stack trace [1]. Moreover, the information provided by the stack trace is at times insufficient to understand what could have gone wrong to cause such an error. In such cases, EvoCrash helps in replicating a similar test case with the same error and same stack trace solving which can help identify and solve the problem which led to the original crash. Example: While working on a previous java project, an UnknownHostException thrown with the line number in the stack trace did not necessarily indicate what has caused such an exception. Manual debugging was cumbersome as the code was complex and had multiple authors. Therefore the context of the variables in the class under test (CUT) remained unclear. If we were aware of EvoCrash and if the same had been used, a test case replicating the same crash would have been generated. The said test case would have helped resolve the UnknownHostException in the original program as solving the test case would have provided more information on the exact cause of the failure. This would have saved time by speeding up the debugging process.

Improvements for EvoCrash

Proper documentation of the test cases with tutorials, provided as part of the EvoCrash jar file would be an improvement we would suggest. The documentation of the test cases covering the exceptions that the test case may throw leading to its crash is important because knowing what causes the generated test case to break will help understand the original crash better. Another improvement would be to allow EvoCrash to qualify a test as fit i.e. as passing the fitness function when the target line or exception is covered even though the stack trace similarity of the test case with that of the original stack trace is not achieved [1]. EvoCrash could also be developed further to work on any programming language other than java. As it employs the stack trace of the programs post run, the stack traces can be parsed according to the specification of the programming language used and the same steps like i) initial population, ii) evolution of the tests using crossover and mutation and iii) selection of fittest test using the fitness function (post-processing) can be applied.

Using EvoCrash in our daily programming

When programming, occasionally you run into errors of which you are not sure what caused them. The error will point to the part of the code where it was thrown, but sometimes the cause of this error is not apparent. It can be quite time consuming to try and find out under which conditions the program fails, especially if that failure depends on the state of variables that are not obviously related. For these situations we would use EvoCrash to generate a test case which can reproduce the error. Knowing what steps produce the error can greatly speed up the process of finding the fault in the code. Especially for certain fringe cases this can be difficult manually. It may not be a tool that you will need to use daily, but it is a handy resource to have available to you when the easy solutions prove insufficient. Since you do not need to change the way you are programming, there is no extra effort involved to have this option. For this reason we would use EvoCrash for our programming whenever the origin of the crash is not readily apparent.

References

1]Soltani, M., Panichella, A., & van Deursen, A. (2017). A Guided Genetic Algorithm for Automated Crash Reproduction. In Proceedings of the 39th International Conference on Software Engineering (ICSE 2017). ACM.
2]N. Chen and S. Kim, “Star: Stack trace based automatic crash reproduction via symbolic execution,” IEEE Tr. on Sw. Eng., vol. 41, no. 2, pp. 198–220, 2015.
3]M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson, “Jcharming: A bug reproduction approach using crash traces and directed model checking,” in 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER). IEEE, 2015, pp. 101–110.