CS4110 Software Testing & Reverse Engineering
Testing assignment 1

Owen Huang (4317459)
Kin Lok Chow (4509447)

March 20, 2017

1 Introduction

Search Based Software Engineering (SBSE) was introduced to make the jump from human-based searching to machine-based searching for software engineering problems. The main problem in software engineering is the optimization problem. The problem with humans was that they can only work at a certain speed and are error prone. For manually creating tests it is therefore infeasible. So with SBSE it not only take the manual work away for test generation and crash reproduction, it also removes any human error problems. To experience crash replication first hand, we have used debugged two large projects with and without tests generated by EvoCrash. During a short time of 45 minutes per large project, we were tasked to identify the root cause of the system failure by merely a stack trace of the failure and to provide a fix. In this document, we explore first what SBSE means for automatic test code generation in Section 2 and discuss the most prominent methods found in extant literature. After, we dive into specifications and the internals of EvoCrash in Section 3. We then follow with a discussion on our experiences with EvoCrash when we had to debug two large projects in Section 4 and finally discuss possible improvements to EvoCrash in Section 5.

2 Automated test generation

The automated generation of tests all started with the most basic method, Exhaustive Testing (ET). With ET all probable input values are tried, to find out if a sequence of input exists which will break the system. Will small test files this is still feasible to run, but with many inputs and a large amount of possibilities this method becomes infeasible in terms of computation. As a result, instead of searching the entire search space, it would make sense to try to first reduce this search space into something more manageable.

2.1 Metaheuristic algorithms

To optimize the search space, the application of metaheuristic algorithms was investigated throughout the years. Metaheuristic algorithms are problem-independent, so they do not require to know how the exact context of the problem and how the heuristic exactly works. One of the easiest but most well known methods is the Hill Climbing method, which is a local search algorithm. It starts with a random test set and randomly changes one element in the set. This could by either adding, deleting or changing that element in the set. If this newly random made test set has a better fitness score than its previous version, it will continue with the new one otherwise it is discard. This fitness score is computed by a separately designed fitness function. This process will continue until no better options are found as result of changing one statement. Unfortunately, this approach will likely get stuck in a local optimum. This relates to the situation in which the algorithm thinks no better solution is possible, however looking at the big picture this is surely not the case. To overcome getting stuck

in a local optimum there are many different methods published, that avoid this by using mutation [1] or temperature [2].

This last method is better known as Simulated Annealing (SA). Basically, it works the same as the hill climbing method, but it does not always disregard test cases that are less fit than the fittest test. It introduces the notion of temperature, which accepts these worse cases with a certain probability. The higher the temperature, the higher this probability is. Similarly, through an iterative approach, each iteration the temperature will decrease and as a result will be less likely to move to a "different" hill.

2.1.1 EvoSuite - Genetic Algorithm

EvoSuite is a tool that can automatically generate test cases for Java applications [1]. To be able to do this, EvoSuite creates several assertion test cases to cover large portions of the code. But how does EvoSuite build all these cases?

To produce these test cases with high code coverage, EvoSuite uses a particular search-based algorithm: the Genetic Algorithm (GA). GAs are inspired by the process of natural selection, which consists of many different processes. EvoSuite uses a combination of selection, mutation and crossover. At the start of the GA, an initial population of random initial test suites is generated. A test suite consists of a sequence of statements, whereas each statement is either a numeric variable (e.g. `int a = 4`), a class initialization (e.g. `Color green = new Color()`), a call to a public member variable of an object (e.g. `int a = Color.opacity`) or a method statement (e.g. `int a = Color.giveOpacity()`). With this initial population, an iterative process follows until one of the following two conditions is met:

- The GA has found a test suite which satisfies its coverage criterion
- The GA has used all of its the allotted resources. These resources are usually often taken to be time or a maximum amount of iterations. Each iteration there are several steps that will occur, we will describe them shortly below.

The first step is the selection, it uses a rank based approach which is measured based on their fitness scores. This fitness score is computed through a fitness function eq. (1). The fitness score is not only calculated on the obtained coverage, but also the branch distance which is calculated with the method by McMinn[3].

$$fitness(T) = |M| - |M_T| + \sum_{b_k \in B} d(b_k, T) \tag{1}$$

To elaborate on eq. (1), the function computes the fitness score for a test suite T, where $|M|$ is the total amount of methods it has and $|M_T|$ is the amount of methods the test suite T actually executes. Finally we calculate the branch distance for each branch with $\sum_{b_k \in B} d(b_k, T)$. For each of the test suites, a fitness score is obtained, where lower scores rank better than higher. Moreover, the higher ranked test suites have a higher probability to be chosen for the next step, than the lower rank ones.

The second step is crossover, with the two test suites obtained from the selection, we can create two offspring. EvoSuite uses a single-point crossover, where the chosen point depends on a random variable α, the cut-point, which is a continuous variable between 0 and 1. The cut-point is then an α amount of the test cases for the test suite and $(1 - \alpha)$ of the other test suite, so the amount of test cases in a test suite do not matter.

The third step is mutation, which modifies the affected component. For each of the two offspring there is for each test case a chance of $1/N$ that they get mutated, N is here the amount of test cases in the test suite. When a mutation occurs, there are three different operations that can happen:

1. **Remove**: A statement is removed from the test case.
2. **Change**: Changing the statement value, which fits the current statement type.
3. **Insert**: A new statement is added at a random place.

[1] `http://www.evosuite.org/`

The three operations also happen in the above order, so we do not remove operations we just modified. Each of the three operations have an equal chance of 1/3 to occur, so when a mutation takes place the amount of changes stays low. Furthermore the probability for each statement in the remove and change operations is $1/M$, where M is the amount of statements. For insert, it keeps adding statements with a probability of σ until it fails.

$f_P \leftarrow min(fitness(P_1),\ fitness(P_2))$
$f_O \leftarrow min(fitness(O_1),\ fitness(O_2))$
$l_P \leftarrow length(P_1) + length(P_2)$
$l_O \leftarrow length(O_1) + length(O_2)$
$T_B \leftarrow Best\ individual\ of\ current\_population$
if $f_P \vee (f_O = f_P \wedge l_O \leq l_P)$ **then**
    for O *in* $\{O_1,\ O_2\}$ **do**
        if $length(O) \leq 2 \times length(T_B)$ **then**
            $Z \leftarrow Z \vee \{O\}$
        else
            $Z \leftarrow Z \vee \{P_1\ or\ P_2\}$
    else
        $Z \leftarrow Z \vee \{P_1, P_2\}$
$current\_population \leftarrow Z$

Finally, the selection to decide whether the offspring gets included in the next generation or not, depends on their length (the amount of statements in all test suites) and fitness score. In the code above [4], we have a snippet of the GA. In this snippet we can see how the algorithm decides which test suites get included in the next population (`current_generation`). Following, `P` and `O` are related to the chosen parent and offspring test suites respectively. Depending on the two criteria, two of the four test suites are included in `Z`. After this, the next generation gets ready to start a new iteration, if the conditions are not met.

2.2 Test input methods

The goal to automate the creation of good test cases, which cover a wide variety of code can happen based on certain methods.

- The first case is when you have access to the internal structure, which we call white-box testing.
- For black-box you test on the logical behavior of the system.
- Grey-box tests which uses both the logical behavior and the internal structure of the system.
- Lastly we have non-functional test, which focuses on they way the system operates, like load testing or stress testing.

In the case of EvoSuite, they use a form of white-box automatic testing, because of the internal codes they require to create their initial population.

3 Crash reproduction with EvoCrash

To manually reproduce software failures as a result of reported bugs or crash reports by users, can be a daunting task. These reports do not often make it apparent what the location is, where the system failure is initiated. As a result, EvoCrash allows for automated crash reproduction by analysing a stack trace of the crash [5]. EvoCrash is a search-based evolutionary algorithm and follows the same core concepts as EvoSuite. Similar to EvoSuite, EvoCrash consists of two main elements that identify the core of the evolutionary algorithm:

1. A fitness function, which will be used to evaluate the quality of the generated test case.

2. A novel Guided Genetic Algorithm (GGA), which guides the Genetic Algorithm to focus more on methods of the Class Under Test (CUT), which in EvoCrash corresponds to the class where the exception in the crash is thrown.

EvoCrash is built in Java and allows for generating JUnit test cases that can target specific types of crashes by specifying the type of exception to be thrown, such as a `NullPointerException` or `InvalidArgumentException`, etc. To provide an overview how EvoCrash works internally and to understand in what parts it excels and what its limitations are, we investigate the fitness function in Section 3.1 and the GGA in Section 3.2 in more depth.

3.1 Fitness function

EvoCrash's fitness function makes three main considerations and formulates them as conditions, which are listed in order of importance:

1. The line where the exception is thrown *must* be covered by the test case.
2. The target exception *must* also be thrown.
3. The stack traces of the original crash report and the one of the generated test case *should* be as similar as possible.

The fitness function taken directly from [5] is denoted by:

$$f(t) = [3 \cdot d_s(t) + 2 \cdot d_{except}(t) + d_{trace}(t)] \in [0, 6] \tag{2}$$

The fitness function f takes as input a test t and uses fixed weights for each consideration accordingly. Following, $d_s(t)$ relates to a distance computed by measuring how 'far' the test t is off the target line. This distance is computed using two heuristics *approach level* and *branch distance* [5, 3].

The *approach level* is computed as the minimum amount of control dependencies, which relates to the minimum amount of lines that need to be executed before the target line is evaluated or in other words a line's 'nesting level'.

The *branch distance* is computed by measuring the distance of t to the branch condition of which target line is contained in. Next the $d_{except}(t)$ is a binary value indicating if the exception was really thrown. This distance is assigned 0 when the target exception is thrown and 1 otherwise.

Finally $d_{trace}(t)$ measures the distance between the original and test generated stack trace for evaluating the similarity of the two stack traces. This is computed by taking the normalized (i.e. dividing by the *numerator* + 1) sum over all the *minimum* distances over *elements* of the stack traces. These elements are triplets containing *class name*, *method name* and *line number*. The distance is computed between the elements of the stack traces through a simple mapping. In case the class names do not match, a value of 3 is assigned. Similarly, when both class names and method names do not match, a 2 is assigned. Finally if all do not match, then the normalized difference between the line numbers of the stack traces is taken. By definition, for a test t, $f(t) = 0$ is only possible when all three major conditions are fulfilled. Similarly, a value $0 < f(t) \leq 6$ will be proportionally assigned to $f(t)$, if this is not the case.

3.2 Guided Genetic Algorithm

In contrast to EvoSuite, EvoCrash uses a GGA rather than a regular GA. This GGA aims to prioritize methods that are directly involved in the crash higher than in the general setting of a GA, in which this type of weighting is not performed. Similar to a GA, the GGA follows the first three steps of forming the *initial population*, *crossover*, and *mutation*. Moreover, the GGA also performs *post processing* at the very end to make the generated test cases more readable for the developers. To show how the GGA works in detail, we discuss each of the steps in order.

As briefly mentioned before, the first step of any GA is forming the *initial population* which will be used as the sampling space for the following steps of the algorithm. The GGA of EvoCrash uses random test generation to form its pool of tests. The GGA requires as input how large the initial

population should be (i.e. the amount of tests). Furthermore, it uses the stack trace to extract which public method calls in what classes were causing the system failure. Through an iterative approach, public method calls that are part of this stack trace are inserted with a probability $1/size$, where *size* is a random integer and denotes the maximum amount of statements to be added in a test t. This probability grows in each iteration in case this procedure fails and no statement is added. The procedure stops when the target amount of statements *size* is reached.

The next step is the *crossover*, in which the GGA uses EvoSuite's GA single-point crossover as its first steps. Similarly, it generates two 'child' tests, using the first α statements obtained through a defined random cut-point α from a parent test p_1 and $|p_2| - \alpha$ statements from the second parent test p_2. The remaining statements from both parent tests also form a second child. These operations however may sometimes cause the target method calls causing the crash to be lost in several of the child tests. The GGA simply discards these particular test cases and leaves the process of evolution over to the next step: the *mutation*.

This step is also an iterative process that keeps repeating itself as long as the test t does not contain any of the target methods (which may be the cause of the mutation 'accidentally' removing or changing the statements corresponding to the target methods). Similar to EvoSuite, the same three different types of mutation are also present in EvoCrash, and apply to the level of statements: *insertion*, *deletion*, and *change*. The GGA will loop over every statement in t and with equal probability mutate the statement. The probability for every type of mutation is defined beforehand.

Firstly, mutation by insertion is defined as inserting primitive variables or instantiating new objects. Next, mutation by deletion is defined as deleting a method call, which will also delete all input parameters for this method call. Finally, mutation by change differs in changing primitive variables, which will be simply given a new value corresponding to its type, and method or constructors. If the statement to be changed is a method call or object instantiation, this statement will be changed to some other method or constructor call of similar return type. All necessary input parameters are taken from previously mutated statements or randomly generated.

The GGA concludes with a *post processing* step. After all previous steps, the GGA has returned the fittest test by definition of (2). Due to all these previous steps adding in possible redundant statements, which do not affect the test case, these are removed in a greedy fashion using the available functions in EvoSuite. Furthermore, randomly generated input variables are given sensible naming (also using EvoSuite functions) to make the test more readable for the developer.

4 Debugging with EvoCrash

For one of us, who is a beginner in Java, it was hard to probe what I was doing in the beginning. To generate the tests and obtain the stack trace of EvoCrash took some time the first time. Nonetheless, it does make it easier to find the source of the problem, by pinpointing me to a stack of potential problematic methods. Instead of clueless searching manually, it did help me to get some place to start. EvoCrash also makes it easier for beginners, since you do not have to learn how to write a test case, but only have to understand the Java syntax.

For the other, who has more experience in Java, it was very nice to already have a test case at my disposal to reproduce the crash. Compared to manually having to create a test case and debugging mean while, using EvoCrash saved a lot of time and also helped tremendously in identifying where the system falls short. This also meant that implementing a fix was very easily verified, since the fix would remedy the failing test. Without EvoCrash, this would mean having to combine both manual debugging and iteratively improving my written test case to ensure that the fix is truly targeting the original crash of the program found in the stack trace.

As a result, experiencing debugging with EvoCrash via the survey, we conclude that this tool could have helped identifying errors in many of our previous programming endeavours, where code was documented very badly and difficult to understand. Letting EvoCrash generate the test cases reproducing crashes would help making code more robust and allows to more easily understand where the system's weaknesses lie. However, this does require a priori knowing what type of exception should be targeted.

5 Evaluation of EvoCrash

At the moment, one of us has no intention to code in Java and if the need arises it will likely be only basic applications. As a result, for me it is then not that important to test all cases if it breaks with such small programs. For the reproduction of my crashes it will likely not be hard to find, while it is likely on the lines I just worked on. If I am going to work on larger Java projects in the future it could be an interesting option, because it would make my debugging life a bit easier. But by then we have to see if there are more interesting option, which cover a larger amount of the crash cases.

For the other, EvoCrash is definitely something that could be useful sometimes during programming. However, it would not be the direct go-to option when I would get stuck. This is because it would also take some time to set all parameters up, which for small issues may be more time consuming than just using the regular IDE debugger. Furthermore, EvoCrash targets specific types of exceptions, but sometimes these similar type of exceptions can be thrown in the trace. As a result, I see EvoCrash being very useful for trace depth level 1 crashes, since the probabilistic nature of EvoCrash is kept at a minimum for these types of crashes. However, for more complex bugs EvoCrash can still provide some general indications where things could have gone wrong. Moreover, I would absolutely consider using EvoCrash for testing code that is difficult to understand and just needs to be extremely robust. Since this would allow me to just immediately add additional fixes and test them.

To conclude, EvoCrash is definitely useful when you are at wit's end on why your application is causing system failures. EvoCrash generates tests through a more strict policy for generating the initial test pool and evolving them than previous GAs. EvoCrash's GGA is more optimized for the application of crash reproduction compared to e.g. EvoSuite's GA, which targets test generation rather than crash reproduction. However, EvoCrash is still a GA which means that it is also highly probabilistic. The developed fitness function in eq. (2) uses fixed weighting of terms, but the impact of these weights is still insufficiently explored. Especially, in the case a developer is specifically looking for a very specific instance of a crash, the similarity of the stack trace may be deemed much more important. A suggestion would be to use a adaptive fitness function, through an iterative process or perhaps by applying techniques from different domains such as machine learning to train a fitness function. Moreover, EvoCrash is still only accessible through the command line interface. To help developers and make it more user friendly, a GUI could help to set the parameters (including dependencies, target exceptions, etc.) which would allow the developer to save more time to focus on remedying any crash EvoCrash reproduces.

References

[1] F. C. M. Souza, M. Papadakis, Y. Le Traon, and M. E. Delamaro, "Strong mutation-based test data generation using hill climbing," in *Proceedings of the 9th International Workshop on Search-Based Software Testing*, SBST '16, (New York, NY, USA), pp. 45–54, ACM, 2016.

[2] H. Waeselynck, P. Thévenod-Fosse, and O. Abdellatif-Kaddour, "Simulated annealing applied to test generation: Landscape characterization and stopping criteria," *Empirical Softw. Engg.*, vol. 12, pp. 35–63, Feb. 2007.

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

[4] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Softw. Eng.*, vol. 39, pp. 276–291, Feb. 2013.

[5] M. Soltani, A. Panichella, and A. van Deursen, "A guided genetic algorithm for automated crash reproduction.".