Software Testing & Reverse Engineering Lab Assignment

Jos Winter - 4290356
Robin Sveningson - 4586123
Lars Krombeen - 4280709

March 2017

1 Introduction

In a world where software becomes more and more important, the damage caused by software increases. The question rises if these failures could have been prevented. If a system fails it should be repaired as soon as possible. Would the failure have occurred if the system was tested more manually, or was the crash caused by a class which was not tested at all because the developer was confident it worked perfectly.

Software engineering is a field full of trade-offs between quality and time in which problems can be solved with a near optimal solution or with a solution that gives results within a specified acceptable tolerance. Examples are that test coverage for a class should be higher than 80% or that the solution must be a factor of the optimal solution. Another example is when a stack trace of a crash is given and a developer has to fix it. A developer can temporarily fix the crash for that specific case or solve the underlying problem which will take longer.

Metaheuristic search-based optimisation is a procedure or heuristic which finds, generates or selects a heuristic which provides a solution that does not guarantee to be the optimal solution but falls within the tolerance[1]. The factors of software engineering make metaheuristic search-based optimisation techniques suitable in solving the problems in reasonable computational cost [1, 2]. Examples of metaheuristic algorithms are genetic algorithms [3], simulated annealing [4], hill climbing [5] and tabu search [6].

When the use of Search Based Optimisation if applied to Software Engineering it is called Software Based Software Engineering (SBSE) [7, 2]. The use of SBSE has been proved to be successful in various fields of Software Engineering [1, 2].

SBSE can also be applied in the field of automated test code generation and crash replication [2, 8, 9] to help developers save time on writing tests and debugging.

Testing is an undecidable problem and thus SBSE in test suite generation can be used to generate small tests which cover all coverage goals [9]. Using SBSE the problem of having to know the expected outcome and input is solved

[1]https://en.wikipedia.org/wiki/Metaheuristic

[9] and can greatly reduce the amount of time developers spend on testing while making sure that the testing criteria are met.

EvoCrash is an example of SBSE in which a test case is generated based on a stack trace [8]. If a system were to fail one could use the stack trace of the failure to generate a test case. This will save valuable time and moreover, allow the developer to analyse the state while debugging which improves the quality and development time of the fix.

This paper will first explain how SBSE is used to automatically generate test code. Secondly, an analysis of how EvoCrash works is given. Finally, the personal experience of using EvoCrash is given which discusses how it affected the way of debugging, how it could be applied to our programing routines, how it could be improved and if we would use it in our programming routines.

2 Automatic test generation

Automated test generation is a technique used to generate unit tests for specified code files. S. Shamshiri, R. Just, J.M. Rojas, et al, describes it in the following way.

“Rather than tediously writing unit tests manually, tools can be used to generate them automatically — sometimes even resulting in higher code coverage than manual testing“ [10].

So the purpose of automated test generation is to automatically generate the unit tests for an application, which can be a time consuming and tedious task for developers. Often, the logic in the unit tests is not necessarily very advanced, and the work is mostly about making sure all possible combinations of input is tested for each method. This is something that instead can be generated, so that the developers can spend time on other tasks instead.

Relevant to automatically generated tests is the Oracle testing method. It introduces an “Oracle“ to the test case, which is the mechanism responsible of determining whether or not the result of a test is successful or not. An oracle contains a generator, a comparator and an evaluator. They are responsible of generating the “correct answers“ to a test, the way of comparing the actual answer to the expected answer and the way of determining if the comparison result should be counted as a pass or not respectively [11].

EvoSuite is an example of an SBSE tool that can be used for automatically generated unit tests. It is made for Java applications, and its purpose is to find issues in the application code. EvoSuite uses the Oracle method, and it generates oracles for the different classes. On EvoSuite's website, they explain how the tests are generated in the following way.

“To achieve this, EvoSuite applies a novel hybrid approach that generates and optimizes whole test suites towards satisfying a coverage criterion. For the produced test suites, EvoSuite suggests possible oracles by adding small and effective sets of assertions that concisely summarize the current behavior; these assertions allow the developer to detect deviations from expected behavior, and to capture the current behavior in order to protect against future defects breaking this behaviour“ [12]

EvoSuite is an Open Source project, meaning that the source code is available for view and can be extended by the public. It is ran on the command line, but it is also available as a plugin for many platforms, such as Eclipse, IntelliJ IDEA and maven. The main features of EvoSuite, include generation of JUnit 4 tests, optimizations of different coverage criteria, minimized tests, generation of JUnit asserts, virtual file system and virtual network. [12]

However, even if automatic test generation tools like EvoSuite seem like replacements for manual test creation, the automatic tests have some flaws. S. Shamshiri, R. Just, J.M. Rojas, et al, show that the efficiency of tools like EvoSuite is very low; “the test generation tools find 55.7% of faults, but no tool alone finds more than 40.6% of faults.“ [10]. They also show that general code coverage of these type of tools is an issue as well, meaning that not all of the code is tested by the automated methods.

3 How EvoCrash works

EvoCrash or Evolutionary-Based Crash Reproduction helps in the first step of the debugging process which is crash reproduction. EvoCrash reproduces the crash first by requiring a stack trace as input and a test suite generated by EvoSuite by using a genetic algorithm [8]. The user has to configure the maximum computation cost EvoCrash can use for the genetic algorithm in the worst case. The inner workings of EvoCrash on an abstract level can be described using the following steps:

1. EvoSuite generates a testing suite. The generated test suite functions as an initial testing population.

2. A test is selected from the testing population. This could be any test but there is some optimization in EvoCrash which selects the test which has overlap with the given stack trace.

3. The test is ran and is evaluated using the test fitness function. The test fitness function evaluates the score of the test with respect to the stack trace. For example it indicates whether the target line number is covered by the test, whether the target exception is thrown and how similar the given stack trace is to the stack trace generated by this test.

4. When the test is ideal then the optimal test is returned as output. Intuitively and in EvoCrash the optimal test scores the maximum value in the test fitness function because each condition in the test fitness function is fulfilled.

5. The currently used computation cost is determined and when the total computation cost has been reached then EvoCrash returns no test and stops searching.

6. When neither of these conditions have been reached then the test is evolved using the test evolution techniques. The evolved test replaces the original test in the testing population and the algorithm is continued from the second step.

To evolve the selected test there are two genetic algorithm techniques which can be used. The first technique that can be used is the crossover technique. This implies that test cases or statement of two tests are swapped. So in the context of EvoCrash the selected test is coupled with another test and based on both of these parent tests an offspring test is created which contains a random sequence of test statements from both parents. The second technique that can be used is the mutation technique. When mutating a test several mutations can happen. An additional method statement could be added, the input of an argument could be changed or a method statement could be removed.

One advantage of the input that EvoCrash uses is the fact that the optimal solution is known. Given the input stack trace an optimal solution or an optimal test reproduces the stack trace identically, when a test produces a similar stack trace the test fitness function will indicate a non-optimal solution and the genetic algorithm will continue searching for an optimal solution in the search space. This ensures that the genetic algorithm can detect whether a local maximum has been found instead of a global maximum in the search space. EvoCrash only returns a test when the global maximum in the search space has been discovered.

4 EvoCrash in the assignment

All members in our team participated in the EvoCrash experiment in which we were tasked with fixing two bugs in two different scenarios. For each member the scenario as well as the bug were different. When evaluating our joint experience we made the observation that we all executed the following two assignments in different orders:

- A crash/bug had to be fixed and only a Java stack trace was provided describing how the error in the code occurred.
- A crash/bug had to be fixed and a Java stack trace describing how the error in the code occurred and a test generated by EvoCrash using the stack trace which replicates the stack trace.

Independently of the order of the scenarios we have the opinion that having the test which has been generated by EvoCrash which replicated the stack trace causes an significant decrease in the time it takes to fix the bug, therefore we have the opinion that EvoCrash has a positive effect on the debugging experience. EvoCrash helped in the assignment by generating a test which replicates the stack trace. This ensures that you can start right off with debugging the program and finding what causes the actual bug. While doing the assignment without the test which replicates the stack trace it is intuitive to write such a test so the bug can be found.

We also experienced that it was easier to navigate though the code by using the debugger then by following the path taken in the stack trace. This significantly shortened the time needed to fix the bug/crash. We have the opinion that there is some bias in the EvoCrash experiment in which we participated based on our joint experience. We observed that the crash/bug with only a stack trace was in general a much easier bug to fix with respect to the crash/bug with a stack trace a generated test. This might skew the results of the experiment in favor.

5 EvoCrash in our own programming endeavours

One of us works at a company which makes plugins for Revit. The user can model a building and the model is used to determine if the user can go from room A to room B if he would sit in a wheelchair.

The plugins also send information to Google Analytics about warnings and errors. If an exception occurs which we did not catch or which is caused by a bug the stack trace is sent to Google Analytics. Having only a stack trace it is near impossible to create a proper test case which simulates the user behaviour and Revit model which caused the exception. The current procedure is looking at the stack trace, guessing what caused it, apply it to your model and launch the plugin.

EvoCrash could help us out by generating a test case based on the stack trace on Google Analytics. This will save a lot of time because we no longer have to guess what caused the crash and we can find out why the exception was thrown more easily.

6 Improvements to EvoCrash

EvoCrash could use a more formatted way of creating the test cases. It was fairly hard to read and understand the generated test case, because the format in the test was hard to read. An improved formatting for the generated test cases could help increasing readability of the tests and make them easier to comprehend.

Another improvement to EvoCrash could be to make it easier to use. In its current state it is a very complicated process to use EvoCrash, and it requires a lot of usage time. So a simpler way of using it would make the experience of using EvoCrash better. One way of doing this could for instance be to integrate EvoCrash with different IDEs. One example is the Eclipse IDE for Java, which allows the community to develop their own plugins that can be run from inside of the IDE. If an EvoCrash plugin for Eclipse was created, it could be a good experience to use it on a Java project. The plugin could have all of the settings required as default preferences, made to work with the most common use case of EvoCrash. Then a functionality for "Generate EvoCrash test case" for a given stack trace could be added, which would solve the complexity and time consumption issue.

7 EvoCrash in our programming routines

In the current version and installation of EvoCrash we have the opinion that it still is too difficult to use EvoCrash and takes too much time to execute a test and therefore would not use it in our current daily routines. However we would certainly be open to using EvoCrash if it would be easier to use it without too much configuration. Each time you use EvoCrash to successfully replicate a test from a stack trace it could easily save 10 minutes of testing time. So for example, if EvoCrash could be easily used in a IDE integration it would be very time efficient to use it.

8 Conclusion

Search based testing and automated test generation are interesting ideas that deserve some spotlight by the community. A lot of developers spend a huge part of their time writing unit tests, or other types of tests, and perhaps a lot of that time can be saved by allowing a computer generating the tests instead. EvoSuite is an interesting tool that can be used for these purposes, even though it does not always cover all of the bugs in software.

EvoCrash is a tool that we think is needed and can be very helpful in automating the testing process. The stack traces produced in order to reproduce crashes in an application can save a lot of time for the developer, and together with other automated methods, such as EvoSuite, the testing process can become much easier and much less time consuming.

In this paper we shared our own experience and thoughts with the EvoCrash application, and we provided some ideas on how it can be improved in the future. When the automatic generation tools have been improved a lot, we strongly believe that they can be very valuable to the development process of a lot of people and companies. However, our belief is that automated test generation should be an addition to the existing testing process, and not a replacement. We believe that the people developing the code always have to be a part of the testing process, since they know a lot more about the system than we are ever going to be able to teach a computer. Therefore a mixture of manual and automated test cases could be beneficial.

References

[1] M. Harman and B. F. Jones, "Search-based software engineering," *Information and software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

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

[3] D. E. Goldberg and J. H. Holland, "Genetic algorithms and machine learning," *Machine learning*, vol. 3, no. 2, pp. 95–99, 1988.

[4] P. J. Van Laarhoven and E. H. Aarts, "Simulated annealing," in *Simulated Annealing: Theory and Applications*. Springer, 1987, pp. 7–15.

[5] M. Soltani, A. Panichella, and A. van Deursen, "Evolutionary testing for crash reproduction," in *Proceedings of the 9th International Workshop on Search-Based Software Testing*. ACM, 2016, pp. 1–4.

[6] F. Glover, "Tabu search: A tutorial," *Interfaces*, vol. 20, no. 4, pp. 74–94, 1990.

[7] M. Harman, P. McMinn, J. T. De Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *Empirical software engineering and verification*. Springer, 2012, pp. 1–59.

[8] 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.

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

[10] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness

and challenges (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on.* IEEE, 2015, pp. 201–211.

[11] C. Kaner, "A course in black box software testing examples of test oracles," http://www.testingeducation.org/k04/OracleExamples.htm, 2004, [Online: accessed 19-March-2017].

[12] E. team, "About," http://www.evosuite.org/evosuite/, [Online: accessed 19-March-2017].