Assignment 1 - Testing

CS4110 - Software Testing and Reverse Engineering

Gerard van Alphen - 4303512 - TUD
Jacco Brandt - S1423290 - UT
Wouter van der Zwan - 4019806 - TUD

Search-Based Software Engineering

Introduction

Search-based software engineering focuses on using search techniques like genetic algorithms. Such algorithms can be useful during software debugging, as it can be used for automatic test generation. These tests, for example, aim to reproduce an error and with that speed up the process of fixing a bug. An implementation of automatic test generation using a genetic algorithm is EvoSuite, which will be further explained in the following section.

EvoSuite

So how does EvoSuite generate its test cases? It does so by using genetic algorithms which are inspired by natural selection in biology and on evolution. In the process of evolution, three parts keep repeating[1]:

- Selection
- Reproduction
- Mutation

In nature this means that at the selection part, the fittest individuals survive, they then reproduce and their offspring mutates by inheriting the properties of their parents and evolving.

This can also be applied for the generation of test cases. An individual can now be viewed as a sequence of statements. The population (of individuals) is initialized at random. The next step is to pick the individuals which are fit enough to be parents. The criteria that is used for this are branch coverage and test suite length. A higher fitness means a bigger chance to be picked as parent.

With the parents picked, the offspring can be generated and mutated. This is done by combining the parents which produces new individuals (children). These children get mutated with probability 1/n (with n being the size of the test suite) by either deleting, change or insert a test case[2].

Above process will be repeated until either the perfect solution is found, or the search time is over.

[1] R.P. Pargas, M.J. Harrold and R.R. Peck, “Test data generation using genetic algorithms”, http://www.cc.gatech.edu/~harrold/6340/cs6340_fall2009/Readings/pga.pdf
[2] M. Soltani, A. Panichella and A. van Deursen, “A Guided Genetic Algorithm for Automated Crash Reproduction”, https://pure.tudelft.nl/portal/files/11926462/TUD_SERG_2017_006.pdf

EvoCrash

EvoCrash is a program built upon the EvoSuite, but it covers a different purpose than the latter. EvoSuite is meant to generate test suites for existing code, to be used as a Unit tests to check whether or not the changed code still works before committing. EvoCrash, on the other hand, is designed to help the programmer with resolving an occurring bug. It can analyse the stack trace that Java provides and tries to generate a test case which produces the exact same stack trace.

EvoCrash is invoked by providing a stack trace that can be used to generate the test cases. This stack trace provides the type of exception raised, and a list of all stack frames which have been involved in the creation of the situation leading to this crash. These provide all the necessary information, the ExceptionType, the classes and methods involved and the relevant line numbers, to attempt generating a similar test case. While in EvoSuite each class can be used to generate test cases for, EvoCrash just tries to aim to generate a test-case which triggers the provided stack trace. Therefore, EvoCrash uses the class in which the exception was thrown as main target class for the test generation [2].
In order to create a test case which reproduces the provided stack trace EvoCrash uses a genetic algorithm(s), just like EvoSuite. However, EvoCrash makes use of another, novel genetic algorithm than its predecessor, a guided genetic algorithm (GGA) [2].
The first part of this algorithm is the generation of the initial population (aka initial test cases). The creation of these test cases are guided by the stack trace, each test should at least call one of the methods present in the stack trace. This limits the available statements and expressions compared to the same process in EvoSuite [2].
The tests in the population pool are all evaluated by checking their fitness: whether the target line number is covered, the correct exception is thrown and how similar the generated stack trace is to the provided stack trace. The highest scoring test cases will be used as basis (or parents) for the generation of the new generation test cases, if there is still enough time left.
If the allocated time for EvoCrash has ran out and there is no perfect test case created, EvoCrash will create an empty method body and report that it failed to create a test case that successfully replicates the stack trace. The test generation will also end if one of the created test cases successfully produces the provided stack trace, and EvoCrash will write this test into the method body and exit successfully.
Otherwise, if there is still time left and the perfect test case has not been found yet, EvoCrash will use the previously selected parents to generate a new generation of test cases (or offspring). This is done using a variety of genetic techniques. First is the crossover technique, which swaps a statement from one parent with a statement from the other, resulting in two (new) offspring test cases. If the swapped statement (such as a method call with parameters) is dependent on other statements or variables, these too are swapped into the other test case. If the resulting offspring of this mutation does no longer contain the target method call (the one throwing the exception), this crossover will be reversed and the

mutation algorithm will be solely responsible to change the offspring into something different than its parents [2].
After the crossover algorithm, EvoCrash will apply mutations with a certain probability to all individual test cases. Every statement in a test case has exactly a probability of 1 divided by the number of statements in that test case to get mutated. A mutation can either be the addition of a statement, the removal of a statement (and all variable declarations that only this statement uses) or the change of statement. If, because of a mutation, the call to the exception throwing method is removed from the test case, this guided mutation algorithm will continue to mutate the test up until it contains this method call again [2].
From these newly generated offspring test cases EvoCrash will select new parents based on their fitness and continue this cycle until the allocated time runs out or a perfect test case has been found. Because of the random nature of the GGA-algorithm, this perfect test case probably contains a lot of statements that have nothing to do with the actual bug. That is the reason why this test case will be processed by the native EvoSuite test-optimization method, which removes all statements that do not affect the fitness function (which is how well the stack trace matches the provided stack trace), resulting in a much smaller final test [2].

Using a test case generated by EvoCrash

The experience using a test case generated by EvoCrash to debug a program differed among the team members. Two out of three people found the generated test class helpful. The EvoCrash test case helped by pinpointing the exact location of the bug. This is especially helpful in the unfamiliar source code. It meant checking fewer files and methods for the cause of the bug, which reduced the debugging time.

One of us found debugging with the EvoCrash generated test case difficult. Because the test case contained a lot of mocking and automatic variable names the test case was found to be difficult to understand.

Although EvoCrash did make it easy to find the cause of the bug this was not necessarily seen as an advantage. Getting directly deep inside the source code gave little context of the functionality of the source code. By reproducing the strack trace during debugging the developer is forced to have some understanding of the code, which eventually helps solving the bug.

It is interesting to see how the experience differed among us. Reasons for this can probably be found in the differences between the assignments we have done. EvoCrash was found valuable for assignments 2 and 4, but not so much for assignment 3. An explanation could be EvoCrash works better in some cases compared to others. Previous experience with debugging and/or someone’s existing debug workflow could also have had an influence.

Reflect on how EvoCrash could have helped you in your own programming endeavours (1 example is enough)

The main advantage of EvoCrash is getting to the root cause of the bug quicker. It might sound trivial, but this is also how it could help us in our programming endeavours the best. When working on a project we are not familiar with, EvoCrash will probably be most valuable. The generated test case will limit the amount of code to look at, thus we will be finished faster with the debugging task. However for a code base that we are familiar with, the added value of EvoCrash will probably be lower. With the generated test case and its poor readability, it will probably take more time to squash the bug.

EvoCrash can possibly become of great assistance while finding bugs in concurrent code. Unfortunately, as said during lecture, EvoCrash is not (yet) capable of this. We once encountered a bug in a tool that would fail to finish a specific function if some specific (but common) interaction happened earlier on the interface. Reproducing and finding the cause of this bug took several days. If EvoCrash was capable of testing this kind of bugs, it could have saved a lot of time.

Possible improvements to EvoCrash

Our main complaint with EvoCrash would be the readability of the generated test case. We found the test case to be quite unreadable. The strings used as parameters contain a random text, not even close to what they should be. This give a false idea of the problem causing the bug. By using more descriptive strings this confusion can possibly be prevented. This could possibly be done by using already defined class constants. Either way, the test case should become more readable.

For large constructor and methods calls a lot of parameters are either mocked out or simply a null variant. This makes reading the test case more difficult. A suggested improvement is to split the method call over multiple lines, and add a comment at the end of the line with the name of the parameter.

Reflect on whether you would apply EvoSuite/EvoCrash in your own daily programming routines. Why (not)?

We would not necessarily use EvoSuite/EvoCrash in our own daily programming routines. Right now we think the application is not polished enough to make it convenient to work with. This might be because we would need some more training in using it.

Another reason is that we do not often work on projects which we have never worked on before and this is where we think that EvoCrash would be the most useful.
As for EvoSuite, because the most type of testing we do/create is application-layer testing, this is something that these tools are not designed to do. We only see limited value in unit-testing on top of application-layer testing, and if there is added value, we will write a test ourselves to ensure that it is a complete, readable and not randomized test.

We do however see the added value of EvoSuite and EvoCrash, you just have to use it in the right situation. It can help you reduce the time finding bugs which give you more time for developing. And let’s face it, nobody likes finding bugs.