Testing assignment 1

Chris Langhout 4281705, Gijs Weterings 4272587

1 Introduction

The idea of search-based software engineering is not new. In 1976, Webb Miller and David Spooner wrote a paper[1] about generating floating-point test data automatically. This set up the concept for search-based software engineering, where software engineers use tools such as genetic algorithms to solve a wide range of challenges in software engineering.

One of the most notable uses for this technology is the generation of testing data or test cases. Because exhaustive testing is rarely a good idea, a fitness function for test suites can be defined, where generated test cases are tested against a set of criteria. For EvoSuite for example, unit tests are minimised, so less instructions per functional test causes a higher fitness score.

Search-based software engineering is a broadly applicable technology, that can be used for applications where the use of humans would be costly or infeasible. Writing unit tests by hand is important for staying aware of the testability of the system, to gain knowledge of the system and to ensure quality. However, for large systems, especially those that have a big testing debt, automatic test case generation can be a great tool.

Another good use case for search-based software engineering is automatic crash replication. Here, a genetic algorithm can simulate input in the system, to try to get a repeatable crash replication. It then can minimise the scope, until a minimal crash replication is found. This greatly simplifies the manual work needed to fix bugs and crashes.

2 Generating tests with EvoSuite

When EvoSuite starts generating test cases, it takes a hybrid, search-based approach. It creates a test suite of generated tests for each class, and measures its fitness level. This fitness level is determined by a number of factors, such as different measures of coverage (lines, branches, outputs and mutation

testing[1]), and test method size (less instructions per test is better). To improve the fitness function's initial result, an evolutionary algorithm is used, which uses crossover and mutation to change test cases, and then tests if that change resulted in a better fitness.

This evolutionary algorithm is a good first basis, but EvoSuite ran into a problem where the genetic approach took a long time, if ever, to find some specific conditions for satisfying a branch. To make sure EvoSuite could find these specific corner cases, the genetic algorithm has been altered to work together with a dynamic symbolic execution[2] (DSE) approach. In this approach, bytecode that is executed while running the test cases is analysed symbolically, and from that, some strategic (non-random) mutations can be applied to the genetic algorithm.

As an example, think of an if statement in a method, that states:

```
if (x == "hello") {
   return 1;
else {
   return 3;
}
```

Getting this method to return the integer 3 is trivial for a genetic algorithm. Apart from one corner case, every value of `x` will result in the `return 3` instruction. However, the one case where a different result is returned, may be critical to the application and needs to be tested. Here, DSE can help, with analysing the bytecode, extracting the literal `"hello"` as a symbol, and applying it to x for the next round of the genetic algorithm.

3 Detailed description of EvoCrash

EvoCrash is built on top of EvoSuite. EvoCrash takes a stacktrace as input, together with a test population, generated by EvoSuite[3]. A test from the population is chosen, then EvoCrash checks the value of the test given a fitness function. The test will be more valuable if the given Exception is thrown and a similar stacktrace is given as a result. If this value is below the

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

threshold, the test is changed by an evolutionary algorithm, using crossover and mutation and put back in the test population. Then a new loop for EvoCrash is started.

Using this process EvoCrash generates a test matching the given stacktrace as closely as possible. By mutating the test cases EvoCrash strives to return a test case that represents the global maximum of the fitness function.

4 Our experience in debugging with EvoCrash

Working on the assignments we discovered that EvoCrash can be really useful when debugging if you know how to use it. Providing test cases based on the provided stacktrace gives a high probability of test cases touching the source of this exception. This gives a hint where to look but is no automatic solution. Given a failing test case, discovering the source of the failure can still be hard. Furthermore, when this test is resolved, it does not necessarily mean that the entire bug is fixed. Since the tests are not written by a person it is possible that some conditions are not tested. On the other hand, in the second assignment, a lot of test cases were presented, but none of them were failing. With only the stack trace present for finding the source of the failure, more manual effort is required to find the source of the bug. Depending on the context of the failure this can sometimes be faster, as a test may point to the wrong direction, but most of the time it will take longer to find the source of the failure. EvoCrash is certainly helpful in the debugging process, but intuition is still the most important factor for finding failures in source code.

5 Reflection on EvoCrash for own programming endeavours

A few years ago, Gijs worked on a back-end system for a gaming community website. Because of the size of the codebase and the volunteer position Gijs was in, he was mostly dependent on the existing codebase. Much of the codebase was untested due to time constraints, so if somebody wanted

to work on crash reports, they either needed to write tests on the spot, or manually try to crash the system. One specific crash turned out to be platform related, so manual triggering of the crash on a local system did not work properly. Only after spending a few hours on writing tests for this system, and running that on the staging environment (which doubled as a CI server at the time), the crash reoccurred. By using that test case as a guide, the fix for the crash was trivial and done in minutes.

Had we been able to use EvoCrash at the time, we might have saved hours on debugging and writing our own exploratory tests. Although honestly, spending more time to fix our technical debt would certainly have helped as well.

6 Improving EvoCrash

The biggest improvement we see for EvoCrash (and this translates to EvoSuite as well), is to make the tests more human readable. Variables are named via a naming schema `<classname><incrementingNumber>`. In some test cases, this can turn very confusing very fast. We propose a heuristic that uses the parameter names of the function calls the object is passed to, as a more contextually aware way of naming.

For example, say a test would say

```
Object object0 = new Object();
classUnderTest0.doMethod(object0);
```

EvoCrash could check what the first parameter of the `doMethod` method is called, and use that to rename `object0` to some thing more semantically relevant.

Besides that, Many of the generated test cases still have unneeded instructions. To improve this, a post-processing step that determines what instructions affected the crash itself could be implemented. This probably does need some proper testing to see if it doesn't have unexpected side effects.

7 Conclusion

EvoSuite and EvoCrash are novel ways to automate the time-intensive parts of working on a maintainable code base. By generating tests, either to improve coverage or to replicate crashes, the codebase can quickly be assessed.

In our view, these tools could be great for some projects, but for others it would be less useful. Projects with a lot of complex structures could really benefit from a tool that can easily comb trough all branches.

However, small projects, with relatively simple code, can easily be maintained by hand. In our opinion, writing tests can be tremendously useful in gaining more understanding of a system, certainly if the programmer has not worked on the product since the initial commit.

Therefore, on sizable projects, we would propose keeping a manually written test suite, and amending that with a second test suite that is generated by EvoSuite/EvoCrash.

References

[1] W. Miller and D. L. Spooner, "Automatic generation of floating-point test data," *IEEE Transactions on Software Engineering*, no. 3, pp. 223–226, 1976.

[2] J. P. Galeotti, G. Fraser, and A. Arcuri, "Improving search-based test suite generation with dynamic symbolic execution," in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pp. 360–369, IEEE, 2013.

[3] 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.