Software Testing and Reverse Engineering
Assignment 1

Sander Bosma (4512324)
Jente Hidskes (4335732)

March 2017

1 Introduction

Since writing software tests is a time consuming task, it desirable to automate this process by generating a test suite which contains a minimal set of tests that maximizes branch coverage. However, the number of possible tests and the limited time available make this a challenging problem: even for small classes it is infeasible to examine all possible tests. One of the possible solutions is to apply search techniques such as random search, hill climbing, simulated annealing or genetic algorithms. The practice of applying search techniques on software engineering problems is called Search-Based Software Engineering, or SBSE [2]. SBSE can not only be applied for generating test cases to maximize the branch coverage, but can also be used for crash replication; i.e. generating a test case to reproduce a given stack trac.

In Section 2, the usage of SBSE in test generation is briefly explained, while in Section 3 a more detailed description of the algorithm used by EvoCrash is given. We discuss our experience with the usage of a test generated by EvoCrash in Section 4 and consider some points for possible improvements in Section 5. Finally, in Section 6 we reflect upon the usefulness of the EvoCrash and EvoSuite tools.

2 Test Generation

The goal of adding test cases to a program is twofold: 1) to verify that the current build is correct and 2) to ensure that future changes do not inadvertently change the behaviour of the program ('regression testing'). Automatic test case generation can be used without developer intervention for generating regression tests. However, human verification of the generated tests is required in order to verify that their output is correct. Test cases can contain any number of assertions; in order to minimize the workload for the developers, it is desirable to minimize this number of assertions. In this section, we highlight one specific tool to generate test cases: EvoSuite.

EvoSuite produces test suites for individual classes or complete projects without user intervention, using whole test suite generation and mutation-based assertion generation [1]. Whole test suite generation does not produce individual test cases targeting individual coverage goals; instead, multiple test cases are generated simultaneously to target all branches at the same time. This has the benefit that the result is neither dependent on the order of fulfilling individual coverage goals, nor by their difficulty (or even infeasability) [1].

Generated test cases still require an *oracle* to verify that their output is indeed capturing the correct program behaviour. In tests, these are typically captured using assertions. As mentioned, these oracles are still verified for correctness by humans. In order to minimize the developer workload, it is essential that the number of assertions is minimized. EvoSuite uses mutation testing to determine which assertions capture mutant branches [1]. In mutation testing, defects (*mutants*) are inserted into a program and test cases are executed to determine how many of these mutants they capture. Mutants that are not captured illustrate a deficiency in the generated test suite and guide EvoSuite towards creating a new test case or improving an existing test case. After a test suite has been generated, EvoSuite minimizes the number of assertions by only retaining those assertions that kill mutant branches [1].

3 EvoCrash

EvoCrash is post-failure approach to automatic failure reproduction. It takes a stack trace and attempts to generate a test case that reproduces it [3]. EvoCrash builds on top of EvoSuite; it uses the same

general approach of using an evolutionary algorithm to reach its goal. In the case of EvoCrash, the goal is to find an execution path that is as similar to the stack trace as possible, while still producing the exception. To reach this goal with a genetic algorithm a fitness function is defined, which is a weighted sum of the distance between the path and the target statement (based on the approach level and branch distance), the distance between the given stack trace and the generated path, and a binary value which has a value of one if and only if the target exception is thrown [3].

3.1 Guided Genetic Algorithm

EvoCrash uses a slightly modified version of a genetic algorithm, which is called a *guided* genetic algorithm (GGA). This algorithm is briefly discussed here. As in a normal genetic algorithm, an initial population is generated and multiple generations are generated by modifying the previous generation. Crossovers and mutations are applied on the population in each generation, after which the samples that perform best (i.e. have the best fitness score) are used as initial population for the next generation. The population is a set of samples; each sample is a generated test case. A test case consists of one or more statements, and each statement is used as a chromosome in the genetic algorithm.

3.1.1 Initial population

The requirements of the initial population are slightly different for crash reproduction: rather than generating a well-distributed set of tests, a smaller set is required because it performs the sampling of the search space [3]. To generate the initial population, EvoCrash needs the class under test, the set of failing methods from the stack trace and the population size. For each test, a random number is generated that determines the amount of statements in the test. EvoCrash will keep adding method calls until the target size is reached. When a method call requires parameters, EvoCrash will generate random objects to use as parameters. To speed up the algorithm, methods that are in the stack trace are prioritized over those that are not. If a method is private, EvoCrash will attempt to call non-private methods that call the private one.

3.1.2 Guided Crossovers

A crossover is executed upon two test cases. The idea is to split both test cases into two parts and to exchange the second parts of these tests to form two new test cases. Using traditional single-point crossover, one of the resulting tests may contain all the calls from the stack trace while the second contains none. This impedes the progress of the algorithm. EvoCrash uses *guided* single-point crossover to solve this [3]. Guided single-point crossover exchanges parts of the parents, just like traditional single-point crossover. However, after recombining the parts it checks whether one of the resulting tests does not contain any method calls that are in the stack trace. When this happens, the sample is discarded and replaced by a copy of one of the original two samples.

Crossover potentially results in test cases that are not well formed, for example because a method is called while the supplied parameters are not declared in the test case. These cases are automatically corrected by EvoCrash by declaring and initializing the required variables [3].

3.1.3 Guided Mutation

With a low probability, tests are mutated after crossover. When a test is mutated, an iteration over all statements in the test is started. Each statement can by removed or changed, and a new method call can be inserted at the current position. This means that all the relevant method calls may be removed. Therefore, EvoCrash uses *guided* mutation: if a method call is removed and no method calls remain that call any of the targeted methods, then the mutation continues until such a method call is added [3]. When a method call statement is changed, it is changed into a call of a method with compatible return and parameter types. The parameters are taken from previously declared variables, set to null, or randomly generated. Guided mutation is thus different from a standard genetic algorithm mutation in that it ensures that the mutated test cases call one or more of the targeted methods.

3.1.4 Post processing

EvoCrash stops the genetic algorithm when either the stack trace is perfectly replicated in a test case or a timeout occurs. To maximize the readability of the generated text, any irrelevant statements are

removed (i.e. statements whose removal does not deteriorate the fitness score). Furthermore, a value simplification step is performed where the used values (such as strings and integers) are simplified [3].

4 Empirical Validation of EvoCrash

For an empirical validation of EvoCrash usage, we were given two tasks. In each of these tasks, we were given an existing project and a stack trace of an exception produced by that code that we had to fix, without introducing other bugs (regression tests were available to check this). In the first task we were given only this stack trace and we had to write our own test case to verify whether the bug was fixed correctly, while on the second task a testcase generated by EvoCrash was given. Both team members were given the same two projects (and stack traces), named "IAE" and "NPE". However, one was given a test case for "IAE" while the other was given a test case for "NPE". Since the experiences of both team members are different, they are discussed separately.

Sander was first given the "NPE" project, without a test case. The first step taken after looking at the code surrounding the error was to write a test case in order to reproduce the exception. This proved to be not quite straightforward, as the context of the project was unknown, and the proper way to initialize variables in order to reproduce the exception took a few minutes. Once the test case was working, it was relatively easy to use the debugger to step through the code to identify the problem and to write a fix for it.

The second task already had a test case, which helped to quickly identify the problem, after which the problem was easily fixed. In other words, using EvoCrash helped speedup the debugging process by taking away the need to write a new test case. The perceived difficulty of the second task was much lower, but this can not be solely attributed to EvoCrash; the second bug simply required less contextual information to solve.

Jente was first given the "IAE" project, without a test case. The first step was to inspect the method calls in the stack trace in order to write a test case that triggered the exception. This was rather straightforward due to the simplicity of the code. In this process, the cause of the exception was identified and quite easily resolved. However, the fix combats only the symptom on the path where it occurred; I later identified another path that can lead to the same exception being thrown.

The second problem (the "NPE" project) had a test case included, so the issue was reproducable. Consequently, I did not have to spend time on tracking the issue down and writing a fix. The stack trace resulting from the failing test also allowed quicker navigation through the code, which certainly helped with understanding the context. The codebase was significantly more complex, so this extra assistance was appreciated. Again, however, I am unsure if the fix is combating a symptom or the root cause.

5 Possible Improvements

The testcase generated for the NPE project contained a call to a method which required a string argument. The string used by EvoCrash was `", true) call failed."`, which could be misleading to the programmer, since this exact string was not required to reproduce the exception. EvoCrash should have tried to shorten the used string, by for example trying whether the string `""` or `"a"` would be sufficient to reproduce the exception. Even though string simplification is part of EvoCrash [3], it could use some improvements.

Compared to hand written tests, the test generated by EvoCrash is more difficult to read (compare Listing 1 and Listing 2). An important reason for this is naming of variables. Whereas programmers can intelligently choose variable names (for example, based on its function), EvoCrash names variables only based on their type. It might be a good idea to look into ways to improve the naming of the variables, even though this is likely a very difficult problem.

It is also worth mentioning that the test produced by EvoCrash for the "NPE" project did match the given stack trace, even though the hand written test did (see Listing 1). EvoCrash therefore must have received a timeout before it was able to find this test. Unless a very short timeout period was used, the performance of the search algorithm would be an important point of investigation.

```
public void testLog() {
    Logger logger = Logger.getLogger("org.example.foo");
    logger.setLevel(Level.ERROR);
    SyslogAppender appender = new SyslogAppender();
    appender.setSyslogHost("test");
```

```
    logger.addAppender(appender);
    logger.log("test", Priority.FATAL, "Hello, World", null);
  }
```

Listing 1: A (readable, hand-written) test case triggering the exact stack trace

```
public void testCrash() throws Exception{
    SyslogAppender syslogAppender0 = new SyslogAppender();
    syslogAppender0.setSyslogHost(", true) call failed.");
    Object mockMinguoDate0 = new Object();
    Exception mockException0 = new Exception();
    ThrowableInformation throwableInformation0 =
        new ThrowableInformation((Throwable) mockException0);
    Throwable mockThrowable0 = new Throwable(",3U");
    LocationInfo locationInfo0 =
        new LocationInfo((Throwable) mockThrowable0,
        ", true) call failed.");
    LoggingEvent loggingEvent0 =
        new LoggingEvent((String) null, (Category) null, 0L, (Level) null,
        (Object) mockMinguoDate0, "    ", throwableInformation0,
        (String) null, locationInfo0, (Map) null);
    syslogAppender0.append(loggingEvent0);
}
```

Listing 2: The test case as generated by EvoCrash

Another idea to improve the usability of EvoCrash (though not directly related to the EvoCrash code) would be the possibility to integrate EvoCrash in an online bug tracker, such that whenever a stack trace is reported by a user, EvoCrash automatically tries to reproduce it. This would greatly simplify the ease of use for developers. We suppose in some way this is already possible by using EvoCrash's command line interface and some scripting.

6 Reflection

From the experiences with the EvoCrash tool in the tasks, EvoCrash certainly looks promising, though it would be most effective if were automatically run upon receiving a bug report. It should be noted that care must be taken by the developer to not only fix the generated test case, but also to verify that the root cause of the problem is fixed, rather than only a symptom of it (as mentioned in Section 4); by giving the developer a test case, it becomes tempting to rely solely on that test case to verify the fix.

Whether or not EvoSuite would be used is an interesting question. It is a very useful tool to generate regression tests. However, there are two main problems that inhibit the usage. The first is that the generated tests are not very descriptive, i.e. it is not immediately clear what the test case tests for. Another problem is that it does not combine well with existing tests. Whereas hand-written tests are usually categorized by functionality, EvoSuite might generate a test suite where related tests are spread all over the class; a test for a best-case scenario of a method might be placed far away from a test of the worst-case scenario of the same method. This, together with the fact that the test cases are usually less readable, makes it difficult to use the test suite for documenting the expected behaviour of (parts of) the class under test.

References

[1] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM, 2011.

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

[3] Soltani, Mozhan, and Panichella, Annibale and van Deursen, Arie. A Guided Genetic Algorithm for Automated Crash Reproduction. Unpublished.