Search Based Software Engineering and Automated Test Suite Generation

Jens de Waard
4009215

Bart Heemskerk
4143469

March 20, 2017

1 Introduction

Imagine that you're given a parabolic curve, such as $y = x^2$ and are tasked with finding the minimum. One way to approach this start with a random place on the x-axis and checking the corresponding value for y. Then, do the same for the neighboring values of x. If you've found a decrease in the value of y, you keep heading in that direction on the x-axis, until you've found the lowest possible value of y and the corresponding value of x.

This approach is called "hill climbing", and it is an example of search techniques. Other techniques include Simulated Annealing and Genetic Algorithms. Such search techniques lie at the heart of Search Based Software Engineering.

In the example, the function $y = x^2$ was the *objective function*, which the described algorithm sought to minimize. It is also possible to choose an objective function which needs to be maximized.

Search Based Software Engineering (SBSE) applies these techniques to various problems from Computer Science. On such problem is finding a good set of tests for a software project. In SBSE it is crucial to define a proper objective function. An example of such an function is translating a test suite[1] into the percentage of code it covers.

Search Techniques like Hill Climbing can assist in automatically generating test suites. By slightly changing the test suite, like adding statements to a case or adding entirely new test cases, the algorithm can close in on a test suite that maximizes the objective function.

2 Test Case Generation Using EvoSuite

There already exist tools that use SBSE to generate test suites. One such tool is EvoSuite. As the name implies, EvoSuite uses an evolutionary approach to generate entire test suites.

Initially, EvoSuite generates random test cases. From this starting point it utilizes a Genetic Algorithm to evolve this suite into one that satisfies a chosen criterion.

Genetic Algorithms are inspired by natural evolution. Properties of the population (in this case, the test cases) are encoded in *chromosomes*. Members of the population are measured by their *fitness*, which is computed via a function akin to the earlier mentioned objective function.

Fitter members of the population are used to create offspring through *crossover*, which combines properties from a pair of items to create new members of the population, much like how children are a combination of their parents. Exploration of the search space is enhanced through mutation. There exists a small chance that properties of the resulting offspring are changed randomly.

There are several ways to select the parents used for creating offspring. Simply selecting the items with the highest fitness will mean the algorithm quickly converges on a local optimum while leaving a large part of the search space unexplored. EvoSuite uses a technique called *rank selection*. The

[1] A set of test cases

items are sorted based on their fitness, and then their rank is used to select the parents. The item on rank 1 has a higher chance of being selected than the one on rank 2, but these odds are irrespective of their actual fitness.

To compute the fitness of a test suite, EvoSuite utilizes a concept called *branch distance*. In essence, this is a quantifier that describes how close a program comes to execute a branch. When evaluating an if-statement like *if(x = 0)* with a value of $x = 5$, the program has a branch distance of 5. The entire fitness function is

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

M is the number of methods in the System Under Test, M_T is the number of methods that were executed by the test suite T and the last part of the sum of all the branch distances of all the branches in the program. In practice, EvoSuite normalizes the branch distances to values between 0 and 1.

3 How Does EvoCrash work?

EvoCrash is a tool designed to replicate computer crashes. It is built on EvoSuite and also utilizes genetic algorithms. Given a stack trace of a program crash, EvoCrash generates a test case that reproduces that crash. This helps developers fix those bugs.

EvoCrash uses search-based techniques to find the test that is most suitable test to replicate the error described in the stack trace. To do this, EvoCrash uses two elements: A fitness function to guide the search and a proper search algorithm that uses the fitness function to find the most suitable test-case.

A Java exception stack trace contain two elements: the type of exception and a list of stack frames generated at the time of the exception. Each frame corresponds to a function call within a Java class, containing the method name, the class name and the line number of where the exception was thrown. The last stack frame is of most importance, because EvoCrash uses the class in that frame as the main class to test.

3.1 Fitness Function

The fitness function describes the distance between a stack trace created by the generated test and the stack trace that needs to reproduced. The distance is zero only if:

i the exception is thrown at the same line

ii the exception is the same type of exception, and

iii the stack traces are as similar as possible.

EvoCrash has the following fitness function for a test t:

$$f(t) = 3 * d_s(t) + 2 * d_{expected}(t) + d_{trace}(t)$$

The line distance $d_s(t)$ uses *approach level* and *branch distance* between the last stack frame of the generated stack trace and the stack trace to be replicated. The $d_{expected}(t)$ is 0 when the type of exceptions match, and 1 otherwise. The trace distance d_{trace} is calculated by comparing the full stack traces. The generated stack trace is $S^* = \{e_1^*, ..., e_n^*\}$, where each element equals $e_i^* = (C_i^*, m_i^*, l_i^*)$ with C_i^* as the class, m_i^* as the method and l_i^* as the line number. The difference between each elements of each stack is

$$dif(e_i^*, e_i) = \begin{cases} 3 & C*_i \neq C_i \\ 2 & C*_i = C_i \text{and } m_i^* \neq m_i \\ \varphi(|l_i^* - l_i|) & \text{Otherwise} \end{cases}$$

where $\varphi(x) \in [0, 1]$ is a normalization function. Using this formula, the generated stack trace S^* and the stack trace to match S, the stack trace distance is calculated as follows:

$$D(S^*, S) = \sum_{i=1}^{n} min\{diff(e_i^*, e_j) : e_j \in S\}$$

This will result in zero if the two stack traces match exactly. EvoCrash uses the normalized version of this function:

$$d_{trace}(t) = \varphi(D(S^*, S)) = D(S^*, S)/(D(S^*, S) + 1)$$

As a result of this, the function $f(t)$ will be a value between 0 and 6, where 0 is a test that creates the desired stack trace.

3.2 Guided Genetic Algorithm

EvoCrash uses a novel genetic algorithm named GGA (Guided Genetic Algorithm) to create tests that reproduce the test as closely as possible. GGA can be divided broadly into three parts:

i Creating the initial population,

ii Mutating the population until a test replicate the crash or time runs out, and

iii Clean the best test from unnecessary method calls.

EvoCrash takes a population size N as one of the inputs when called, which is used as the size of the initial population. The initial population is created by creating N empty tests and filling them with function calls either from the stack trace or function calls from the class to be tested, based on a certain probability. To prevent that a tests is filled with only function calls from the class to be tested, the probability of picking a stack trace function call increases every time a class function call is added, eventually leading to a probability of 1. This initial population is tested with the fitness function to check the best possible score of the population.

While no test has been created that reproduces the desired stack trace, EvoCrash produces new sets of test cases by evolving the previous set of test cases. EvoCrash uses two mechanisms to generate these new test cases: Guided Crossover and Guided Mutation.

Guided Crossover splits two test cases into two parts on a randomly picked line number. It then proceeds to put the last part of a test case after the first part of the other test case and vice versa, creating two new, offspring test cases. The difference with regular crossover is that when an offspring test case is generated with no method from the crash, that offspring is replaced with one of the parents. To avoid non well-formed tests, guided crossover applies a correction procedure that add all required objects and primitive variables into the offspring tests.

Guided Mutation looks at each statement of a test case and decides based on random chance if that statement will be mutated. This mutation can be:

i deleting the statement

ii changing the statement

iii inserting a new method call

When deleting a statement, if that call was a method, values used as input are deleted if they are not used anywhere else in the test case. Changing the statement can vary in its outcome. Changing a declaration will change the value of that declaration by another random value. Changing a method or constructor call will change its input parameters. Because statements can be deleted, it may delete the function call that is part of the stack trace. If that happens, guided mutation will mutate the test case until a new method call is introduced that is part of the desired stack trace.

If a test is created that reproduces the desired stack trace exactly (thus has a fitness function result of 0) or if the upper threshold for time EvoCrash may use to mutate is reached, the test case with the lowest fitness function result is picked from the latest generation of offspring tests. This test may include some statements that are not necessary to reproduce the crash, due to the randomness of the mutations. EvoCrash uses the following test optimization routines of EvoSuite to reduce

```
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);
}
```

Figure 1: The failing test case

the size of the generated test: test minimization and values minimization. The test minimization removes all unnecessary statements using a greedy algorithm and values minimization shortens the identified numbers and simplifies the randomly generated strings.

4 Using EvoCrash for Debugging

We investigated four different crashes with two different exceptions; two Null Pointer Exceptions (NPE) and two Illegal Argument Exceptions (IAE). We were asked to solve the bugs that caused these exceptions. In two of the cases, a test generated by EvoCrash was provided.

An example of a given stack trace is given in Figure 2. What happened was that the Collections API gives the power to transform different kinds of mappings to another. When doing so, it uses the current size of the original map as the initial capacity of the new type. If a map was empty, this meant the initial capacity would be 0, which would be illegal.

As this is a rather simple error to reproduce, EvoCrash would certainly help with providing a test that helps to check if the error was solved. Considering that this error was not caught by any of the provided tests, this is a boon. However, EvoCrash would do little to offer insight into the reason of the crash, which is knowledge that a real developer would need in order to find a solution that does not impact the program in other places.

One of the NPE did come with a test case generated by EvoCrash. However, this test was considerably less helpful. The test case is shown in figure 1. The crash resulted from a member field of the SyslogAppender being null, so EvoCrash dutifully replicated the crash by creating such a SyslogAppender and then trying to append a logging event to it. However, this test would still fail, even when the earlier problem (the *layout* member variable being null) was solved, for example by choosing a default value for it. This violates the rule that tests should test only one thing per test case.

```
java.lang.IllegalArgumentException: Initial capacity must be greater than 0
    at org.apache.commons.collections.map.AbstractHashedMap.<init>(AbstractHashedMap.java:142)
    at org.apache.commons.collections.map.AbstractHashedMap.<init>(AbstractHashedMap.java:127)
    at org.apache.commons.collections.map.AbstractLinkedMap.<init>(AbstractLinkedMap.java:95)
    at org.apache.commons.collections.map.LinkedMap.<init>(LinkedMap.java:78)
    at org.apache.commons.collections.map.TransformedMap.transformMap(TransformedMap.java:153)
    at org.apache.commons.collections.map.TransformedMap.putAll(TransformedMap.java:190)
```

Figure 2: An example stack trace

5 Improving EvoCrash

Figure 1 shows a test case that was generated by EvoCrash. Even if you disregard the usual verbosity of Java, the test is still hard to read. Redundant symbols, like the 0 appended to all variable names, require additional mental effort to be parsed. This may be helpful when multiple objects of the same type are needed, but unless that is actually the case it only serves to clutter the test.

Additionally EvoCrash appears to have a preference for using null values. In this case, that did reproduce the crash, but only because the NPE it was intended to trigger occurred before the NPEs that would otherwise have been triggered by this design choice. It probably won't be trivial, but EvoCrash might produce better tests if it tried a top down approach.

Instead of creating objects when only when really necessary, create objects with zero values (empty strings, 0 for integers, etc.) and replace those objects with null values to try and replicate the null pointer exception. In this specific case, EvoCrash might then have created a test that succeeded after solving the original problem with the code.

6 Conclusions

The generation of test cases through tools such as EvoSuite and EvoCrash has much to offer to programmers. However, generating test suites that fully cover all combinations of inputs and all possible paths throughout the program is infeasible. Using EvoSuite to generate a suite that sufficiently covers the program is a good start. Should any crashes still occur, EvoCrash can generate a test case to be added to the suite.

Using EvoCrash as a kind of jumping board to supply a crash-reproducing test case is a good idea. This test case can then be cleaned by a human developer to meet other, non-functional requirements, like variable names, length and only testing a single error. However, we find it unlikely that we will utilize EvoCrash in this way without the improvements discussed in section 5.