Search-Based Testing and Crash Replication with EvoSuite and EvoCrash

CS4110 Testing Assignment 1

David Bergvelt - 4642457
Tom Peeters - 4176510

March 20, 2017

1 Introduction

Software engineering is a complex discipline, whose diverse challenges require a variety of approaches. For example, creativity is needed to choose the correct algorithm or data structure for a particular task, while brute force is needed to consider the many edge cases of a problem and create solutions for all of them.

As it happens, there is a significant subset of the activities of software engineering that lend themselves to automation. In particular, these activities can be thought of as search problems, where an optimal (or near-optimal) solution is chosen from the set of all possible solutions based on some fitness metric. This is the basis for *search-based software engineering* (SBSE), which attempts to view many of the problems of software engineering as search problems to be solved computationally, rather than manually [2].

The SBSE process can be generalized as a process consisting of four main steps. First, an abstract representation of the problem must be formulated. This gives us a domain to search over and introduces a mathematically precise description of the given problem and possible solutions. Next, some fitness function must be defined. This function will serve as a means to distinguish different candidate solutions by providing a metric by which to rank them. Third, a search over the set of possible solutions is performed, using the fitness function to evaluate solution quality and converge towards an ideal solution. Finally, the efficacy of the results of the search is analyzed, potentially repeating step three with different search techniques in order to improve the solution quality [2][5].

Software engineering problems that benefit the most from SBSE techniques tend to be ones that are both challenging to solve manually and easy to represent as search problems (i.e. can easily be represented abstractly and have their solutions ranked by a fitness function). Two promising applications of SBSE which fit these criteria are automated test generation and crash replication. In this paper, we will examine how SBSE helps us solve these problems, specifically looking at two pieces of software, EvoSuite and EvoCrash, which apply SBSE techniques in order to automate test generation and crash replication.

2 Automated test generation

Testing is an essential part of the software engineering process, offering a method by which code correctness may be evaluated and increasing confidence in program stability and reliability. However, manually defining test cases can be time consuming and difficult, especially in large, complex projects. This is because tests should ideally be designed according to both breadth and depth of coverage: they should cover both all regions of code (breadth) and all expected code behaviors (depth) [2]. For example, a non-ideal test suite might contain tests for all classes in a project, but fail to account for some edge cases that produce exceptions or other unwanted behavior. Conversely, a test suite might thoroughly test some of the classes in a project, but fail to account for others and thus be liable to miss bugs. Furthermore, test suites should be as small as possible while still satisfying the breadth and depth criteria, so as to reduce time spent on testing. These numerous requirements for an ideal test suite make manual test generation especially challenging [2] [3].

Luckily for software engineers, test generation may be automated with SBSE techniques. To illustrate how SBSE may be applied to this problem, we will take a look at EvoSuite, a tool which uses evolutionary algorithms to search for ideal test suites.

2.1 EvoSuite

Due to the size and complexity of the domain of candidate solutions for the problem of test generation, *metaheuristic* techniques may be employed to increase the efficiency of the search process. One example of such a metaheuristic technique is the use of evolutionary algorithms, which is the method by which EvoSuite generates test suites [3] [5].

Evolutionary algorithms take inspiration from the process of natural selection in order to iteratively search for an ideal solution to a given problem [3]. A typical evolutionary algorithm will proceed according to the following process: First, an initial population P is chosen. This may either be done randomly or may be manually seeded (this may be done to, among other reasons, avoid getting stuck in local minima of the fitness function). Next, all candidates in P are evaluated according to a fitness function, and ranked accordingly. This allows us to perform the next step, selecting the "fittest" individuals from P to serve as the parents of the next generation. This set of parents will be used to generate a set of offspring by a process known as *crossover*. Next, the set P' of the parents and their offspring is *mutated*, randomly introducing small variations to the candidates. Finally, P' is inserted into P, and these steps (sans the first step of choosing an initial population) are repeated until the search budget is consumed or a sufficiently "fit" solution is found [5].

Now that we have a general idea of how evolutionary algorithms function, let's examine how they are used within EvoSuite. There are several areas in the description of evolutionary algorithms above where characteristics of the technique are left abstract as they are highly implementation dependent. These are namely: solution representation, fitness evaluation, crossover method, and mutation method.

Solution Representation

Solutions in EvoSuite are represented as whole test suites. EvoSuite uses an approach known as *whole suite test generation* to generate tests, which means that an individual in the population that is evolved by the evolutionary algorithm is defined as a suite of tests (instead of a single test case). This has been shown experimentally to produce test suites with a higher ratio of coverage to suite size when compared to the traditional approach of targeting a single test goal at a time [1].

Fitness Evaluation

In order to rank the quality of the test suites it searches through, EvoSuite considers a criterion known as *branch coverage*. Branch coverage refers to the requirement that a test suite to reason about all possible paths of control flow that may occur during program execution. For example, if there exists an `if` statement in the code being tested, a test suite with full branch coverage should test the behavior of both the branch where the conditional is true and the branch where it is false. In the event of a tie, i.e. when multiple test suites have the same branch coverage, test suite size is used as a secondary criterion to rank candidates, prioritizing smaller test suites [1].

Crossover

Once EvoSuite has used its fitness function to identify the best test suites in the current population, it begins the crossover process, using these test suites as parents to to produce new offspring test suites. In EvoSuite, this process specifically works by producing two offspring for each two parents every time the crossover operation is invoked. The test cases of the offspring are determined by first choosing a random number α in the range [0,1], then selecting the first $\alpha|P_1|$ test cases from the first parent, and the last $(1-\alpha)|P_2|$ test cases from the second parent to form the test cases for the first child test suite (where $|P_1|$ and $|P_2|$ refer to the number of test cases in the first and second parent test suites, respectively). The second child's test cases will be the opposite of this, taking the first $\alpha|P_2|$ test cases from the second parent and the last $(1-\alpha)|P_1|$ test cases from the first parent [5] [1].

Mutation

In addition to crossover, test suites in EvoSuite are evolved through random mutations. These mutations occur both on the level of individual test cases and on the level of test suites. On average, one test case is mutated per test suite, where a test case mutation consists of some combination of removing statements, adding statements, or changing statements. A test suite mutation consists of adding a test case, where the probability of adding each subsequent test case decreases exponentially [5] [1].

3 Automated crash replication

Although impressive test suites can be generated using EvoSuite, bugs can and will continue to slip through. When a bug does slip through and cause a crash in the system, it is often a difficult and tedious process to find and resolve the issue, as well as confirm that the fix applied works correctly without introducing new bugs. However, SBSE techniques can be applied to this domain as well, generally known as crash replication or post-failure techniques. Work has been done on this topic in [4], producing the aptly named EvoCrash.

3.1 EvoCrash

EvoCrash is a program that, given a stack trace of a crash, utilizes SBSE techniques and genetic algorithms to produce a test case which replicates the given stack trace. By doing so, it not only makes it simpler to discover the fault of the crash, but also provides developers with a test case that can be used to confirm that the bug has been properly dealt with.

Being based on SBSE techniques and genetic algorithms, two basic components of genetic algorithms are present in EvoCrash as well: a fitness function and an evolution function. However, as EvoCrash only needs to find a specific, known crash, it becomes possible to apply additional techniques to speed up the search. These techniques start by analyzing the stack trace of the crash which EvoCrash must replicate.

The stack trace provides a great deal of information to EvoCrash on how to replicate the crash, as it contains the exception thrown as well as the list of stack frames generated at the time of the crash. EvoCrash can utilize this information in order to guide both its fitness function and evolution function. This makes it so that EvoCrash does not utilize a standard genetic algorithm, but a *guided* genetic algorithm.

3.2 Guided Genetic Algorithm

The fitness function of EvoCrash is based on the given stack trace. In order to attain maximum fitness, the following conditions should be met: the same exception must be thrown, the error-producing statement should be as close as possible, and the list of stack frames should be as similar as possible.

However, the guided aspect truly shows in the test case evolution. It is evident in all evolutionary aspects of EvoCrash: the initial population, crossover and mutation. As the stack trace provides EvoCrash with a list of stack frames that are involved in producing the crash, it can use this knowledge to promote the usage of these methods. Next, we will explain how each aspect utilizes this information.

Guided Initial Population

The initial population of EvoCrash tests are heavily biased towards the methods involved in the crash, ensuring that each method present in the list of stack frames is present at least once.

Guided Crossover

Crossover functions in much the same way as standard crossover. However, if, after crossover, no method that is present in the stack trace is present in the test, then we know that the generated test case cannot cause the desired crash. If this happens, the changes made are reversed and the mutation operator is used instead.

Guided Mutation

Just like the initial population, mutation is biased towards methods involved in the stack trace. Additionally, just like crossover, it will revert and attempt different changes if no required method call remains in the test case after mutation.

Finally, once a satisfactory test case has been generated, a post-processing step is applied. This is done because the genetic algorithm may have included statements that are not required for the crash to occur and will make the test harder to understand for humans. Fortunately, the test minimisation already present in EvoSuite can be used here.

4 Personal experience and reflection

As part of our course "Software Testing and Reverse Engineering", both of us participated in a survey for EvoCrash. In this survey, we were given two debugging exercises for crashes that had been found for a particular program. In the first exercise, an EvoCrash test case was supplied to assist in debugging, where as in the second exercise no such test case was given.

These exercises did an excellent job in showing the power of EvoCrash. The first exercise was simple to solve, as EvoCrash pointed us in the right direction. For example, when the crash is a NullPointerException, it becomes immediately obvious which value was null thanks to EvoCrash clearly initializing a certain variable as null.

This is on stark contrast to debugging without EvoCrash. Even with a simple bug such as a NullPointerException, the cause can be difficult to track down as you cannot always be sure which variable in a statement was null. Even if you do know, it is not always obvious why it was null; which begs for further research before a fix can be made. Furthermore, the ability to immediately test the correctness of your test without having to write any code yourself is a great boon.

4.1 Areas for Improvement

We believe that EvoCrash has already shown remarkable results. EvoCrash test cases correctly reproduce the desired crash, and are furthermore reasonably clear and concise.

However, EvoCrash leaves much to be desired in terms of usability when actually generating the test case. EvoCrash is started using a single, large command line statement that can be rather tough to swallow. It furthermore provides no indicator as to its progress.

We believe that EvoCrash's main focus should lie in increasing usability. Rather than being started with a single command line statement, a custom-built GUI or Eclipse plugin

would be ideal. Another alternative is to have EvoCrash itself request/confirm (missing) arguments through the command line, and if need be re-request invalid arguments. In other words, EvoCrash should work on guiding the user more when generating the test case. Furthermore, a simple indicator of EvoCrash's progress would go a long way – for example, by displaying the fitness of the currently most-optimal test case.

5 Conclusion

EvoSuite and EvoCrash represent an important step forward in the evolution of the discipline of software engineering. By applying SBSE techniques to tasks that are difficult or tedious for programmers to do manually, these tools enable software engineers to focus their attention on other parts of the development process, ultimately boosting the efficiency and quality of their work.

However, there are still hurdles that must be overcome before these tools may enjoy widespread use. As we have noted, a critical challenge lies in the design of the user experience of these tools. In order to truly make the lives of software engineers easier, EvoSuite and EvoCrash must be easy to use, with a low enough barrier of entry that the majority of software engineers will be able to efficiently use them with little to no initial coaching. Once this issue is addressed, we believe that EvoSuite and EvoCrash will be a great addition to any Java developer's workflow, and we look forward to the future progress of these projects.

References

[1] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, Feb 2013.

[2] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. *Search Based Software Engineering: Techniques, Taxonomy, Tutorial*, pages 1–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[3] Phil McMinn. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.

[4] Arie van Deursen Mozhan Soltani, Annibale Panichella. A guided genetic algorithm for automated crash reproduction. 2017.

[5] Mozhan Soltani. Evocrash: Evolutionary-based crash reproduction. University Lecture, 2017.