Software Testing and Reverse Engineering (CS4110)

Assignment 1

Pouya Omid Khoda	4625323
Mohammad Ahmadinia	4205227

Table of Contents

1. Introduction

Search-Based Software Engineering (SBSE)

Software testing is an important component of any successful software development process. A software test consists of an input that executes the program and a definition of the expected outcome. Many techniques to automatically produce inputs have been proposed over the years, and today are able to produce test suites with high code coverage. The main goal of Search Based Software Engineering (SBSE) is to change a way of solving software engineering problems with machine-based search instead of human-based search. In Software Engineering (SE) a Search Based Optimization (SBO) algorithms are used to address problems in SE [1].
In order to demonstrate how conceptually is possible to reformulating software engineering to search based software engineering, we would like to explain how to apply mathematic algorithm search to a large body of software engineering problems. In order to reformulate software engineering as a search problem, it will be necessary to first define a representation of the problem which is dependent to symbolic manipulation, second a fitness function which is defined in terms of this representation and finally trying set of different search algorithms and analyzing and comparing the results.

- **Representation**
  The representation of a candidate solution is critical to shaping the nature of the search problem, and also should be accurate enough to show the difference among individuals. As an example floating point numbers and binary code are used frequently for problem parameters in existing applications.

- **Fitness**
  The fitness function is defined as a function, which takes a possible solution to the problem as input and produces as output, how "fit" our how "good" the solution is with respect to the problem in consideration. A fitness function should possess the following characteristics: Firstly, the fitness function should be enough fast to compute and secondly it must quantitatively measure how fit is the solution or how fit individuals can be produced from the given solution. In some cases that the problem is too complex, it is possible to do fitness approximation to suit our needs.

- **Search algorithms**

  Search algorithm has been designed such as :
  - Exhaustive search (try all possibilities)
  - Random search (try random solutions)
  - Heuristic search (pick a random spot, assess its fitness, and explore the search space accordingly)

  One of the main barriers to the use of a search algorithm in SBSE is tuning. A search algorithm can have many parameters that need to be set. There are different metaheuristic search algorithms that have been applied such as Hill Climbing, Simulated Annealing, and Genetic Algorithms [2].

Hill Climbing is a local search, which picks a random spot in the search space and after finding the neighbors, it will measure the fitness of the spots and its neighbors. As e next step, either it will pick the best neighbors (if any) and iterate the finding a new neighbors and so on. OR if there is not better neighbor because the fitness is not ideal yet, it will re-start from the beginning.

Genetic Algorithms are closely related to the concept of survival of the fittest [2]. Solutions in the search space are referred to as "individuals" or "Chromosomes", which collectively form a "population". One of the main barriers to the use of a search algorithm in SBSE is tuning. A search algorithm can have many parameters that need to be set. For example in this case, to use a genetic algorithm, one has to specify the population size, type of selection mechanism (roulette wheel, tournament, rank-based, etc.), type of crossover (single point, multi-point, etc.), crossover probability, type and probability of mutation, type and rate of elitism, etc. The choice of all these parameters might have a large impact on the performance of a search algorithm.
A genetic algorithm is a global search which is inspired by the natural selection in biology. This method has been designed in three steps:

First, fitting individuals will be selected. Each individual is made up of components called "chromosomes", that could be a vector of binaries or characters or even sequence of statements in test case. In this step randomly initial population P (or seed some individuals) will be generated and loop until budget is consumed or the one is found! Then will be evaluated the fitness of each individual in P and it will select its parents.
To be continued, Second step is reproduction, it means they reproduce offspring or called doing crossover, in this step, two parents will be used for one-point crossover. It means a position will be randomly picked, and then elements will be swapped up to that selected position. For every parents a new child will be generated. A generated child needs to be mutated in the next step.

And finally third step is mutations. In this step, a generated child after crossover and offspring will be used. In order to mutate an individual with a probability of 1/n (n is a length of the individual), some chromosomes need to be modified. This modification could be based on adding value, changing values or removing variables to/from the individuals.

EvoSuite

The Evosuite tool automatically produce JUnit tests for Java software [3]. Given a class under test (CUT), Evosuite makes sequences of calls that maximize testing criteria such as line and branch coverage, while at the same time produce JUnit assertions to receive the current behavior of the CUT. Experiments on open source projects and industrial systems [4] have shown that Evosuite can successfully achieve good code coverage. In order to understand how to use EvoSuite for automated JUnit test generation during software development, we have read this article [5] and we tried to explain it in abstract. This paper [5] discusses the technical challenges that a testing tool like Evosuite needs to address when handling Java classes coming from real-world open source projects, and when producing JUnit test suites intended for real users.

Evosuite is a search-based testing (SBST) tool that uses a Genetic Algorithm (GA) to produce test suites that could be sets of test cases. Evosuite achieves highest possible code coverage. The GA (explained in introduction) starts with an initial population of randomly generated test suites, and successively evolves these test suites by applying selection, crossover and mutation, until a solution has been found or a stopping condition is reached for example in case of timeout.
Based on the comparison between Evosuite and previous SBST tools [6], it is obvious that Evosuit does not try to generate one test case at a time for different coverage goals, but that it evolves whole test suites targeting all coverage goals at the same time.
This has some advantages: impossible coverage goals do not impair the search, there is no coincidental coverage, and there is no question on the order in which coverage goals should be addressed.

EvoCrash

The actions that are needed to take for debugging are as following, first, it is needed to reproduce a crash to confirm that crash is going to happen. Second, what would be a defect after crash and identify the crash trigging conditions? Third, finding a valid fix which does not introduce a new defects, and finally, check that the crash is not reproducible anymore and adding the crash reproducing test to the test suit. EvoCrash has been designed to support the first step, which is crash replication.
There are two main requirement for a prosperous application of search-based techniques [7][8]. First we need to have an appropriate fitness function to guide the search to encompass to the target. Second, there must be a proper search algorithm to boost tests in a closer fashion to resemble the crash while condemning the tests that carry poor fitness values.
In Evocrash, we need to use a single test from Evosuite, it means we need to pick a target class for which initial test are produces by Evosuite. Then the fitness of the test should be evaluated. In this step for running the candidate test, we need to check if is the target line number covered in the test? Is the target exception thrown? How similar is the generated stack trace to the target trace? For answering those questions, EvoCrash first parse the log file given as input in order to extract the crash stack frames of interest. It’s worth mentioning that a standard java stack has the type of the thrown exception and a generated list of the stack frames at the time of the crash. Every single stack represents a method that has been part of the failure. Therefore, it has the method name, the class name and the line numbers where the crash happened. Plus, last frame also represents the location where an exception has been thrown, the source problem could be within any frame though.

In next step, the test will be evaluated with applying two standard GA operators, which are crossover and mutation (possible mutations on a test case).
If the search time is over, and the search was not successful, the empty test will be written to the target test directory.

2. Experiment

In case of debugging assignments, since none of us could actually the finish what was required during the 45 minutes survey and ran out of time, the following would be a speculation of what could have happened if we could ideally apply EvoCrash to those tasks it's useful in finding errors and difficulties and helps optimizing the code. Specifically EvoCrash can be used to replicate crashes and help programmers to evaluate the situation using those produced idea test cases

Having the ability to automatically generate test cases and replicating crashes is an obvious advantage programmers would like to have in their attempt to test their desired software, which in this case EvoCrash comes in handy.

In case of future improvement, earlier there was a discussion regarding guiding the EvoCrash into favoring method calls over constructor calls and if EvoCrash is capable of doing such thing. Later on, there was another discussion that led to a discovery that EvoCrash still needs to make improvement regarding a smarter log parser (which did not happen during our tasks). In the end, there is still a need to find an algorithm to decrease the mismatch between the tests and specifications.

There is definitely a potential EvoCrash which makes it helpful to identify what is the cause of the failure in a better fashion than using manual/random testing generation, plus it improves the identification of how to reproduce the failure. Even though a crash can be reproduced using unit testing, it takes too much time and effort. This is the place where EvoCrash comes to the rescue as EvoCrash has an approach based on an automatic crash reproduction and it has been proven that EvoCrash is capable of iterating the crashes successfully so yes we would definitely try to apply EvoSuite/EvoCrash in our daily programming routines.

Conclusion

EvoCrash is a exquisite Guided Genetic Algorithm (GGA) which features a fitness function, to search for a test case that can trigger the target overhelm and demonstrate the buggy frame in the crash stack trace. Experimental assessments illustrate that EvoCrash addresses the primary difficulties that were encountered by three cutting-edge approaches, and therefore beat them in automated crash reproduction. In this assignment we tried to apply EvoCrash into total of four different tasks and filled various surveys in order to help the developer team behind EvoCrash to evaluate the challenges we encountered while performing the tasks and handed out the data we collected using Mylyn plus other artefacts for a further review.

References

1- Harman, M., McMinn, P., De Souza, J.T. and Yoo, S., 2012. Search based software engineering: Techniques, taxonomy, tutorial. In Empirical software engineering and verification (pp. 1-59). Springer Berlin Heidelberg.

2- P. McMinn. Search-based software test data generation: a survey. Software testing, Verification and Reliability, 14(2):105–156, 2004.

3- 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 , ser. ESEC/FSE '11, 2011, pp. 416–419.

4- A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite," ACM Transactions on Software Engineering and Methodology , vol. 24, no. 2, pp. 8:1–8:42, Dec. 2014.

5- EvoSuite: On the challenges of test case generation in the real world (tool paper)," in IEEE Int. Conference on Software Testing, Verification and Validation (ICST) , 2013.

6- Whole test suite generation," IEEE Transactions on Software Engineering (TSE) , 2012

7-] M. Harman, P. McMinn, J. De Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in Empirical software engineering and verification. Springer, 2012, pp. 1–59

8- M. Harman, S. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," ACM Comput. Surv., vol. 45, no. 1, p. 11, 2012. [Online]. Available: http: //doi.acm.org/10.1145/2379776.2379787