Search-based Software Testing and Automated Crash Replication

André Santos (4643313), Azqa Nadeem (4542606)
CS4110 Software Testing and Reverse Engineering
Delft University of Technology

March 20, 2017

1 Introduction

In industry, software engineering problems are generally solved manually, which is an error-prone, impractical and costly process. With the latest advances in technology, this process can, to some extent, be automated, mitigating some of the issues of being solved manually.

A software engineering problem can often be seen as an optimization or a search problem, i.e. finding the best solution from a space of all possible solutions. Even in the simplest of real-world problems, the search space is typically too large to search exhaustively. Therefore, a near optimal solution is often acceptable. These are the features that justify the use of a meta-heuristic approach, such as, random search, hill climbing, simulated annealing and genetic algorithms.

2 Search-based Software Engineering

SBSE [3] is the application of meta-heuristic search-based techniques to software engineering problems in order to find a near optimal solution. A meta-heuristic is a strategy that guides the search process towards an optimal solution, thus avoiding to search the whole solution space.

In order to apply the said strategy to a software engineering problem, the latter must first be reformulated as a computational search problem. This generally entails the definition of a solution space and a representation for each of the solutions, otherwise we would not be able to apply these strategies to the problem. Furthermore, we will need to define an objective function. An objective function[1] is a problem specific function, $f : S \to \mathbb{R}$, used by the search algorithm to evaluate the quality of a solution, it maps the solution space onto the space of real numbers. Since this function is problem specific, its output can mean anything, i.e. a cost, a gain, a measure of similarity, a distance, etc.

Finally, the definition of operators is also problem specific because different approaches require different operators, e.g. in the context of genetic algorithms, a problem might benefit from single-point crossovers but might not from bit flips.

2.1 Remarks on industry

In software development industry, nearly every developer acknowledges the importance of tests. However, not every developer goes trough the trouble of writing them. There was a study [2] that evaluated the time spent on writing tests versus writing production code. In that study, it was observed that over 50% of the subjects did not write any tests in a 3 month observation period and among the subjects that claimed they wrote tests, only 47% actually wrote them. Furthermore, among the group of subjects that wrote tests during those 3 months it was also asked how much time they spent writing tests in comparison to the time writing production code, and the responses, on average, were respectively 48% and 52%. Although, in reality, it was, on average, respectively 25% and 75%. In conclusion, most developers overestimated their time spent on testing.

To tackle both the lack of test cases, and the poor effectiveness of some of the written ones, we can automatically generate tests. EvoSuite is a tool that does exactly that, as described in section 3.

It is also relevant to address the inefficiency regarding fixing bugs that is prevalent in the software development industry. The usual resolution of these bugs generally entails the discovery of the bug by someone, following a submission of a bug report, usually containing crash data of the program, which is later used by the developers to fix the bug. The developer will, in turn, try to reproduce this issue with the end goal of finding the crash preconditions in order to fix the code and write a test for this situation for future verification purposes. This process is often very long and tedious. Therefore, it is not the most optimal allocation of resources for whomever

[1]In the context of genetic algorithms, it is usually called fitness function.

is employing these developers. As a result, various tools have been developed that, to some extent, automate this process. EvoCrash is one of such tools, that will be described in detail in section 4.

3 Automated test suite generation - EvoSuite

EvoSuite is an automated unit test generator which generates JUnit compatible test suites. It uses a genetic algorithm as a meta-heuristic, which follows natural selection. A possible solution [2] is a test suite, which is a collection of test cases. A random sample of test suites is evolved overtime until a solution fulfilling the fitness function is found or the search resources e.g. time, memory, etc. are up. During the evolution, this algorithm follows the principle of elitism or survival-of-the-fittest which states that only the best individuals from last generation are selected for further evolution.

EvoSuite operates on byte-code level so it does not need the actual source code to make any decisions. Operating on byte-code level has its perks, for example, complex conditional statements can be broken down into more, but simpler statements, that make the branch coverage evaluation easier; and potentially being able to support multiple languages that generate byte-code like Java does (e.g. clojure).

An avid reader might have noticed that it is rather strange how an automated test suite generator can know the expected outcome of a test with only access to the program's byte-code. In fact, it does not. Rather, it generates a small, yet effective, set of assertions that concisely depict the current behavior of the system, allowing the developer to detect unexpected behavior. More on this problem is described in section 7.1.

3.1 Algorithm specifications

- The algorithm begins with a randomly generated sample of test suites.
- Each test suite contains N number of test cases and each test case has length L. Both N and L have a fixed upper bound beforehand for bloat control (a common problem with genetic algorithms that causes successive solutions to be bigger but no better than the previous ones). The optimal value for N and L depends on the problem at hand, and hence, must be searched for in the search space.
- Best individuals from the previous generation are selected for the next generation based on rank selection, which uses the fitness function. Let two of the samples be P_1 and P_2.
- The offsprings are created based on cross-over of the two parent samples P_1 and P_2. Let O_1 and O_2 be the two offsprings. A random single point crossover is defined as swapping elements between parents with a probability of α, such that:
  - O_1 is αP_1 and $(1-\alpha)P_2$
  - O_2 is αP_2 and $(1-\alpha)P_1$.
- Next, O_1 and O_2 are mutated. With a probability of $\beta = \frac{1}{T}$, where T is the length of the test suite, either something is added, removed or modified in the offspring. In the case of entire test suite generation, test cases are added, removed or modified. In case of test case generation, statements are added, removed or modified. When adding or removing a statement, all dependent statements are added or removed along with the statement under consideration. When modifying a primitive statement, the value is changed by some random number, and when modifying a method call, a method with same return type is chosen at random.
- Finally, each of the mutated samples are evaluated using a fitness function and only the best ones (among both parents and offsprings) are added to the next generation of samples. This process is repeated until either the resources are spent or an optimal solution is found.

3.2 Fitness function

EvoSuite can support any type of fitness function. In the paper [1], the authors have used a combination of branch coverage, branch distance and length of the test suite in their fitness function.

In control structures, such as `if()` or `switch-case`, one of the several code branches can be executed based on the input parameter. *Branch coverage* will be maximum if test cases that trigger most branches are included in the test suite. *Branch distance* is a measure of how far away a solution is from the optimal one. The search is guided towards solutions that minimize this distance. Lastly, a test suite that has less test cases but has the same branch coverage as another one, is preferred.

[2]Also called a chromosome in genetic algorithm terminology.

Since we have multiple goals for the fitness function, these goals must be prioritized, otherwise conflicting solutions can exist. In case of [1], having higher branch coverage is more rewarded than the length of the test suite.

3.3 Optimizations

Once the genetic algorithm is done, the final test suite is further optimized by running a greedy minimization algorithm that successively removes redundant statements. By the end of this step, the smallest possible test suite remains where each statement and/or test case contributes towards the coverage criteria.

Real-world software can be very complex containing several branches. When generating a test suite, all these branches can rarely be covered since, for some branches, no input value exists that can trigger that particular branch. Such branches are marked as infeasible goals. If EvoSuite encounters such a goal, the search can wander off. Furthermore, since the search space is so vast, evolving one test case at a time will degrade the performance drastically.

Therefore, EvoSuite evolves whole test suites simultaneously so that, even if an infeasible goal is reached or the algorithm is taking longer to find the solution for a certain test case, other more promising candidates are evolving in the meantime. Compared to evolving one test at a time, when EvoSuite performs poorly for some complex goal, its performance is degraded by a small factor. However, when it does better, the increase in performance is significant.

4 Automated crash replication - EvoCrash

The test cases that are generated by EvoSuite are evaluated based on the fitness function. So, if the fitness function is specialized for crash replication, the tests that replicate the crash, will sometimes be generated. However, since the search space is so huge, the probability of finding that one test case that replicates a crash is like searching for a needle in a haystack. Therefore, the search needs to be guided.

Moreover, EvoSuite is a coverage-based tool. It tries to cover as many branches as possible, without considering the exact input value used to cover a branch. It is possible that all branches are covered but the crash is still not replicated because of the absence of a specific input value for a branch. In this case, EvoSuite will not generate another test case for already covered code.

EvoCrash is an extension of EvoSuite for automated crash replication based on an improved guidance algorithm and fitness function [4]. EvoCrash uses the post-failure approach, where it looks at the crash stack trace in order to generate a test case that would replicate that crash. In other words, the search for a solution is guided by the crash log itself.

In addition to guiding the genetic algorithm, a crash-replication-oriented fitness function is introduced as well. It tries to find a stack trace t^* that is the most similar to the one given initially t. The new fitness function has the following requirements ordered in terms of priority:

1. *The line that throws the exception must be covered* D_{line} *:* It use two heuristics to guide the search closer to the target line. The *approach level* measures the distance between the path of code executed by the current test and the target line. The *branch distance* measures how close the current test is to satisfying the target branch condition.

2. *The target exception type must be same as that in the crash log* D_{excp} *:* It uses a boolean to denote whether the exception was thrown or not. In case the exception was thrown, the trace distance is measured as explained in the next step. Otherwise, it is set to 1, since there is no need to calculate the distance for an irrelevant trace.

3. *The generated stack stack must be similar to the target crash stack trace* D_{trace} *:* The distance between the two stack traces is measured by comparing each line (containing class name, method name, and line number) with the lines of the target stack trace and measuring the difference. A value of 0 is assigned iff the two lines in the stack traces are identical. Otherwise, a score of 1 to 6 is assigned based on their differences.

$$f(t) = 3D_{line}(t) + 2D_{excp}(t) + D_{trace}(t) \tag{1}$$

Equation 1 describes the fitness function mathematically. EvoCrash uses the following three improved algorithms to guide the search that build on top of EvoSuite:

4.1 Initial population selection

While EvoSuite starts off with a uniformly distributed random selection of test cases, EvoCrash is more focused towards test cases that contain at least one of the method calls from the crash stack trace. The purpose is to further reduce the search space. In this algorithm, method calls are iteratively added to the test cases, but before a method can be correctly called, the relevant object is initialized with either existing input parameters, *null*, or generating them randomly.

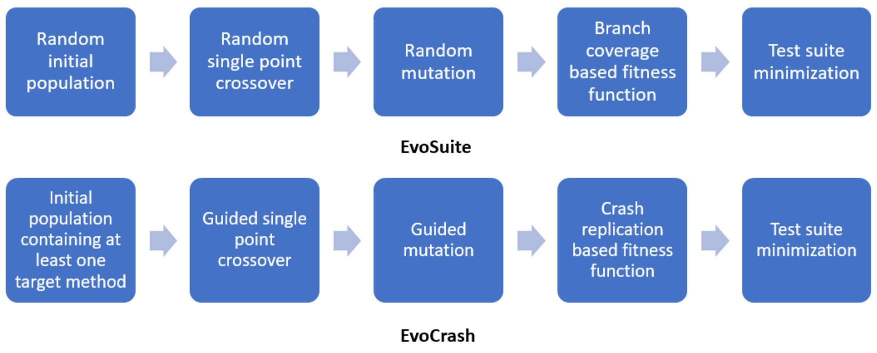
4.2 Guided crossover

The single point crossover algorithm from EvoSuite randomly shuffles statements from the parents. In automated crash replication, we are interested in specific statements from the crash stack trace. These statements have already been added in the initial population selection phase, but the random shuffling can cause offsprings to lose these method calls that we are interested in. To avoid this problem, EvoCrash uses a smarter single point crossover algorithm, where if the method call we are interested in is not present in the offspring, the offspring will take on the value of the parent itself. Otherwise, it works like the standard algorithm.

4.3 Guided mutation

As mentioned in the previous section, EvoSuite can either add, delete or modify statements as part of the mutation phase. However, when modifying or deleting statements, the risk of losing the method calls that we are interested in is still present. Therefore, EvoCrash modifies the algorithm and performs the following tasks with the probability of $\frac{1}{n}$, where n is the length of the test case:

- Addition: Adding random statements just as the default behavior
- Modification: If a primitive value is to be modified, the new value is chosen at random. If a method call is to be modified, then a public method call of the same return type is chosen with input parameters either from the previous call, *null*, or randomly generated.
- Removal: If the mutation causes the offspring to lose the method call we are interested in, the loop is iterated till the addition step causes one of those method calls to be re-inserted.



Random initial population
Random single point crossover
Random mutation
Branch coverage based fitness function
Test suite minimization
EvoSuite
Initial population containing at least one target method
Guided single point crossover
Guided mutation
Crash replication based fitness function
Test suite minimization
EvoCrash

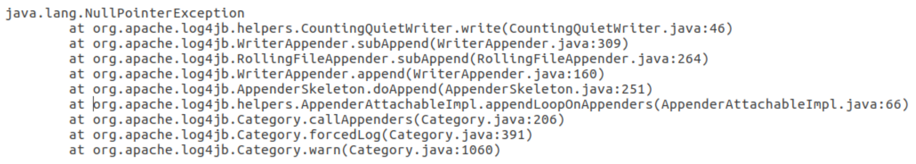
Figure 1: EvoSuite versus EvoCrash

5 EvoCrash in our debugging endeavors

While we did not explicitly use EvoCrash to generate test cases, in the second part of the assignment, the given test case (which we assume was generated by EvoCrash) helped us in figuring out where the crash had occurred. The source code was more complex and the stack trace was relatively longer than that for the first task. However, in the first task, the source code was fairly straight forward and simple to find the reason for the crash, and to fix it. In such a scenario, one of the authors wrote a test herself to verify if her fix really solved the bug. Setting up tools like EvoCrash for such simple tasks would only slow down the debugging process.

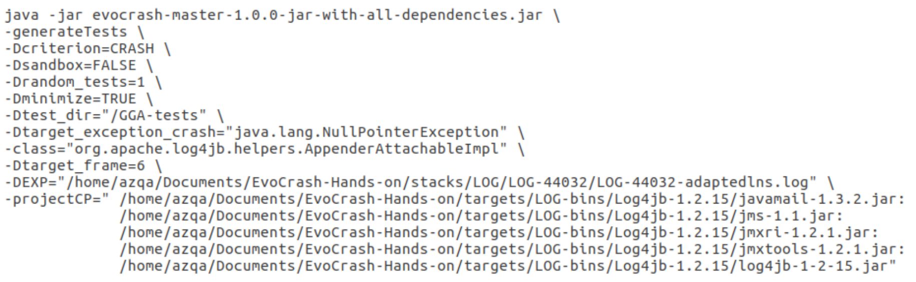
5.1 Test ride on a Apache Log4J

We were interested in applying EvoCrash to a sample project to see how it behaved. We used Apache-Log4J version 1.2.15 which had reported a crash due to `NullPointerException`. The following section presents our observations. Figure 2 shows the crash log we used and Figure 3 shows the command setup. We ran EvoCrash for all the 9 stack frames. EvoCrash generated test cases for 2 frame levels, where one of the test case was empty. For other frames, either the search does not begin or if it begins, it stops abruptly.



```
java.lang.NullPointerException
        at org.apache.log4jb.helpers.CountingQuietWriter.write(CountingQuietWriter.java:46)
        at org.apache.log4jb.WriterAppender.subAppend(WriterAppender.java:309)
        at org.apache.log4jb.RollingFileAppender.subAppend(RollingFileAppender.java:264)
        at org.apache.log4jb.WriterAppender.append(WriterAppender.java:160)
        at org.apache.log4jb.AppenderSkeleton.doAppend(AppenderSkeleton.java:251)
        at org.apache.log4jb.helpers.AppenderAttachableImpl.appendLoopOnAppenders(AppenderAttachableImpl.java:66)
        at org.apache.log4jb.Category.callAppenders(Category.java:206)
        at org.apache.log4jb.Category.forcedLog(Category.java:391)
        at org.apache.log4jb.Category.warn(Category.java:1060)
```

Figure 2: Log-44032-adaptedlns.log



```
java -jar evocrash-master-1.0.0-jar-with-all-dependencies.jar \
-generateTests \
-Dcriterion=CRASH \
-Dsandbox=FALSE \
-Drandom_tests=1 \
-Dminimize=TRUE \
-Dtest_dir="/GGA-tests" \
-Dtarget_exception_crash="java.lang.NullPointerException" \
-class="org.apache.log4jb.helpers.AppenderAttachableImpl" \
-Dtarget_frame=6 \
-DEXP="/home/azqa/Documents/EvoCrash-Hands-on/stacks/LOG/LOG-44032/LOG-44032-adaptedlns.log" \
-projectCP=" /home/azqa/Documents/EvoCrash-Hands-on/targets/LOG-bins/Log4jb-1.2.15/javamail-1.3.2.jar:
             /home/azqa/Documents/EvoCrash-Hands-on/targets/LOG-bins/Log4jb-1.2.15/jms-1.1.jar:
             /home/azqa/Documents/EvoCrash-Hands-on/targets/LOG-bins/Log4jb-1.2.15/jmxri-1.2.1.jar:
             /home/azqa/Documents/EvoCrash-Hands-on/targets/LOG-bins/Log4jb-1.2.15/jmxtools-1.2.1.jar:
             /home/azqa/Documents/EvoCrash-Hands-on/targets/LOG-bins/Log4jb-1.2.15/log4jb-1-2-15.jar"
```

Figure 3: EvoCrash command

- **Frame 9,8,7,2:** The search starts but stops abruptly. For frame 9 and 7, we think that no exception can be generated at that level. Intuitively, in such a case, EvoCrash should not even begin the search, or at least print out a message stating so. Frames 8 and 2 refer to protected methods.
- **Frame 6:** An empty test case is generated. With default settings, the search is unable to find a solution and the timeout is reached. We then played around with additional options and their probabilities as given in `Properties.java` file, but were still unable to guide the search to a test case. One major reason behind this result was our lack of understanding of what each option meant exactly for the search as we found the description itself to be inadequate.
- **Frame 5,4,3:** Search does not even begin. Frame 5 refers to a synchronized method call which is not yet supported by EvoCrash. No exception can be thrown at frame level 4. Frame 3 refers to a protected method, but we see inconsistent behavior from EvoCrash because search does begin for protected methods as in frame level 8 and 2.
- **Frame 1:** A valid test case is written which follows our intuition of sending a null string to the write method.

6 Where EvoCrash could have really helped

One of the authors recall a personal account about the company where she had previously worked. It was a software development company, revolving around one main software suite. The senior developers would write the code for new features, but give the task of writing unit tests to the junior developers in an attempt to make them understand the logic. Their idea was that this method would work perfectly since the developer would need to go through quite a lot of source code before writing an effective test. However, in practice, it didnt work because the junior developers had so much difficulty understanding the source code that they came up with a hack. They would copy existing tests and change some parameters without thinking about their effectiveness. Popular IDEs like IntelliJ would give hints about the code coverage to the developer, which they used as the primary driving force for writing tests. Such tests merely provided good code coverage, but were poor in terms of verifying the correctness of the underlying code. Therefore, near the release of the project, they saw a lot of bugs coming in even after all the tests had passed. Moreover, for some crashes, the developers only knew the work flow but never wrote a test to replicate it. So once the crash was fixed, they never tried that work flow again after making other changes. This is a prime example of where EvoSuite could have helped in generation of a test suite, and when a crash was reported, EvoCrash could have generated test cases automatically to replicate the crash. This would not only have saved a lot of man hours writing useless tests, but would have allowed the junior developers to work on new feature sets.

7 Limitations, underlying assumptions and possible improvements

While EvoCrash might seem like a dream come true for application developers, it has some limitations that can make it infeasible to use in the real-world. Some of these limitations have been carried over from EvoSuite. Hence, they are enumerated in a separate section:

7.1 EvoSuite

- EvoSuite assumes that the code being tested is deterministic, i.e. running a test case twice would generate the same result.
- The generated test cases are in line with the logic of the code under test. This implies that if the logic of the code is flawed, the generated test cases will be testing wrong assertions. While there is no easy way to automate the logic checking behavior, the developer using this tool must be aware of this limitation.
- Each test runs in its own independent thread. In case of a timeout, the test might exit but the thread might still survive, hence blocking memory that can be used to run other processes.
- Currently, EvoSuite can only cover source code that does not have any environmental dependencies, or in other words, does not perform unsafe operations e.g. writing to memory in random fashion.
- Since EvoSuite operates on byte-level code, type information becomes very important for container classes. Currently, such classes have to be manually marked beforehand.
- There are languages that can be compiled to Java byte code, either by default or by using some intermediate language, such as LLVM. However, EvoSuite only generates Java tests. The developers can add support for other languages.

7.2 EvoCrash

- Synchronized method calls are not yet supported, which means EvoCrash cannot be used with multi-threaded applications, which are generally the ones where manual debugging is difficult.
- Currently, a crash stack trace right out of an execution sequence cannot be used with EvoCrash - several modifications have to be made to adhere to the format it requires. A smarter log parser can be used that operates on crash logs as generated by the framework, so that the process can be automated.
- Currently, EvoCrash works with one frame at a time. However, we want to be able to generate a test case for each frame level to know exactly when the crash has been fixed. Therefore, multiple frames could be handled by EvoCrash *in parallel* which will also reduce the latency of generating multiple test cases.
- Currently, EvoCrash has some inconsistent behavior for protected calls in different projects. For example, in certain cases the search begins but stops abruptly (e.g. Log4J), and in other cases the search does not even begin (e.g. Java.Collections). Moreover, if a frame with, for example, a method call is present that only forwards calls to other methods, no crash can occur at this level. Hence, the search does not even need to begin. These edge cases can be looked at and improved upon.
- When the search fails, a more verbose output would be helpful for debugging purposes.
- The tool only supports Java stack traces. If EvoSuite were to support other languages, EvoCrash would also need to extend its parsing capabilities to parse other languages' stack traces.

Considering the current limitations, and the fact that we generally do not write production code for university assignments, using EvoCrash seems like too much of a hassle. However, there is no denying that it can prove to be very useful tool under specific conditions. Unfortunately, as students, we do not really come across those situations often.

Bibliography

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

[2] I Levaja. Watchdog for intellij: An ide plugin to analyze software testing practices. 2016.

[3] Phil McMinn. Search-based software test data generation: A survey. *Software Testing Verification and Reliability*, 14(2):105–156, 2004.

[4] Mozhan Soltani, Annibale Panichella, and Arie van Deursen. A guided genetic algorithm for automated crash reproduction. In *Proceedings of the 39th International Conference on Software Engineering (ICSE 2017)*. ACM, 2017.