STRE Assignments 1 - Testing with EvoCrash

C.Q. du Crest de Villeneuve s1874659
Ines Duits s1876171
M.R. Zamani s1869590

March 20, 2017

1 Introduction

The use of tests during the software development process is essential to create functioning code. The test can provide an input for the program and look whether the output is the expected output or a different result, such as a crash. Tests can also execute some specific parts of the program to find a failure [SP]. In this way, tests help finding flaws and errors present in the software. Test inputs could be created manually or could be generated automatically [AA10]. Creating test inputs manually has disadvantages: the cost of manually creating test inputs is high [RR06]. Moreover, developers would overlook certain test cases thus not uncovering all the flaws [SP] or even not all execution path. Therefore, manually testing is not very desirable for real life, large scale software projects. Automatic generation of test can help overcome these disadvantages but it is not currently widespread amongst developers that find it hard to understand and maintain the tests [SP].

In this paper we will begin with explaining how automatic test inputs can be generated thanks to search-based software engineering. Then we present EvoSuite, a Java automatic unit test generator. Finally, in Section 4 we will introduce the tool EvoCrash, which can be used to automatically generate tests cases. We will explain how we used EvoCrash on an assignment provided by the University of Twente. In section 5 we will reflect on how EvoCrash helped us and possibly could help us in the future.

2 Search-based software engineering

For further reference, this introduction to search-based software engineering and its use for automatic test input generation was written using mostly the following papers [McM04], [GF13] and [SP]

Search-based software engineering (SBSE) is the use of a metaheuristic search technique to solve software engineering problems. Metaheuristic search techniques find close but not exact solutions to optimization problems by finding, creating or optimizing a heuristics at an acceptable computation cost. One of the main applications of these search techniques is automatic generation of test inputs.

To tackle the software engineering problem using SBSE, it must first be translated to a computational search based problem. For software testing, that implies defining the space of test cases or inputs for our software. This space is obviously too large to use a random approach, instead using a metaheuristic to find important test cases is more efficient. For optimization, one needs to associate a value to the different test cases based on their fitness. This value is assigned by an objective or fitness function which could be defined to either assign higher values for better test cases then you try to maximize it or assign lower values for better test cases, in which case you would want to minimize it. The optimization will be done through a metaheuristic.

In order to use this objective function and a metaheuristic to manipulate this function, a way to map the test input to a value understandable by the algorithm needs to be chosen. A good mapping will ensure that similar test cases will have a close value once mapped.

A simple example of an optimization algorithm, or metaheuristic, is *Hill Climbing*, in which you begin with picking a random point in your data set. During the search you inspect the neighborhood of this point and look for points higher than the current one; if one is found, the highest point is chosen as the new base point and the search for the highest point continue with the neighborhood of the new point. The algorithm then loops until the maximum is believed to be found or the allocated time has expired. The problem with this is that the optimal solution can get stuck in a local maximum, and never reach the actual highest point. There are different methods which can be used for SBSE. An

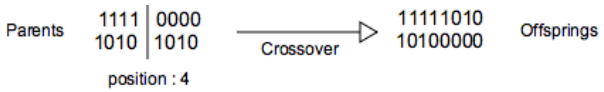
important method employed by EvoCrash is the *Genetic Algorithm* method which we will explain, for more clarity, in its application for EvoSuite in section 3.

3 EvoSuite

EvoSuite is a tool that automatically generates JUnit test cases for Java code using the genetic algorithm metaheuristic. We define a test suite as a set of test cases and a test case as a sequence of Java statements. Examples of possible statements are defining variables, constructing instances of classes, methods calling and assigning values. Both the amount of test cases in a test suite and the number of statements in a test case may vary depending on the software that is tested, the optimal length and number of these needs to be found during the test generation while finding the most optimal test suites.

The genetic algorithm begins with an initial population of test suites randomly generated. Based on their fitness values, given by the already defined fitness function, two test suites called the parents are selected. The parents will generate offspring through a crossover as can be seen in Figure 1.

Figure 1: An example of a crossover of two parents at position 4



In this procedure a random position x is selected and the offspring are generated by taking the first x test cases from one parent and the remaining test cases from the other parent. The two offspring are then mutated: with a probability of $\frac{1}{3}$ removal, changing or insertion of test cases will occur. Removal is done by randomly deleting test cases from the offspring. When changing occurs random values used by certain test cases are modified or when no value is used by the test case, it is swapped for a similar one. Finally, insertion will add one or more test cases in a random position of the test suite. Ultimately, either the offspring or the parents are kept and reinserted into our population of test suites depending on their fitness values, thus generating more optimized test suites. Subsequently, two new parents are selected and the algorithm loops until the best solution is believed to be found or the allocated resource is exhausted. Note that every time the test suites are modified while generating offspring, we have to make sure that the test suits are still acceptable Java input. Acceptable in the sense that Java knows how to read and use the new generated test input, but of course the input still can create errors.

This algorithm allows the selection of the best test suites while integrating the possibility to worsen the intermediate solutions, thus avoiding local extremum that are problematic for *Hill Climbing*. Moreover, instead of the common approach where test inputs are generated and evolved for each goal separately and are afterward combined to a single test suite, EvoSuite will evolve all the test inputs in a test suite at the same time. This is called *whole test suite generation* and is empirically proven to achieve 188 times higher branch coverage with 62 percents smaller test suites as found in the study by Gordon Fraser and Andrea Arcuri [GF13].

The fitness function in EvoSuite considers branch coverage and test suites length as main test criterion. A program consists of different branches when the code has a control structure, for example an `if` statements or a `for loop` statement. In these cases, depending on condition of the control structure, a new code branch is created. Depending on the condition, some branches can require specific input values and are therefore hard to reach, while other branches may be unreachable. The optimal solution for a test suite in this case is a test suite that covers the maximal number of feasible branches. In the case of two test suites with the same amount of feasible branches, the fitness function will consider the test suite with the least number of statements as the preferred test suite.

A more detailed explanation of how EvoSuite work and in particular the exact probabilities for mutations can be found in the paper *Whole Test Suite Generation* by Gordon Fraser and and Andrea Arcuri [GF13].

4 EvoCrash

4.1 Crash replication

In the previous sections, we have emphasized on importance of testing in general and explained the concept of automated test case generation and its benefits. Software testing in general is a technique

prior to the failure occurrence. Post-failure approaches try to replicate the crash by exploiting data available after the failure [MSed].

In order to solve these failures, developers need to know the states and/or the input that caused the crash. Having this knowledge, a developer is able to trace the execution path of the software and follows the call trace to identify the root of the crash. However, this procedure is not always straightforward. In many case the information provided to developers does not suffice. For example tracing a segmentation fault in a multi threaded C software could be hazardous, as the line that crashes the software may not have anything to do with the flaw directly. In the worse case scenario, developers could receive segmentation fault from different locations in the code for the same flaw. The issue worsen when external dependencies and code complexity is added to the mix. In these cases developers could spend substantial amount of resources in order to replicate the crash and identify the cause, because trying to manually replicate the crash is very labor intensive [MSed].

Once developers are able to confirm that the crash occurs and identify the triggering conditions, they need to implement a fix. They also need to verify that the fix is valid and the modifications have not created any new anomaly. This is done by testing the fix and the whole software system against the new fix.

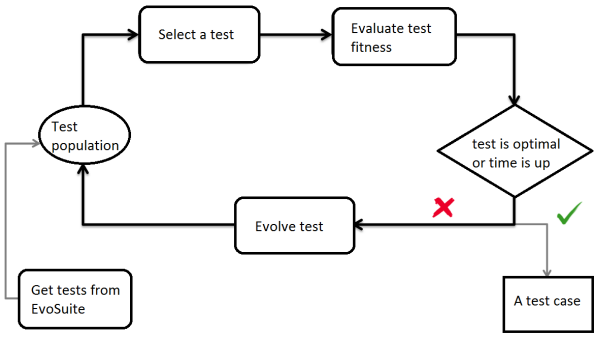
4.2 Crash replication with EvoCrash

In this section of the document we present a Java crash replication tool built on top of EvoSuite, namely EvoCrash. EvoCrash is a post-failure approach which uses a novel *Guided Genetic Algorithm* (GGA) and a fitness function that evaluates the competence of these candidates for replicating the same failure, to generate test case candidates [MSed].

Existing crash replication tools employ coverage oriented genetic algorithms. Coverage oriented approaches are not suitable for crash replication as they cover all the methods and code elements in the class under test (CUT). Hence they suffer from large search space which decreases the probability of generating satisfactory test data that results in the same failure. EvoCrash, on the other hand, "exploits single execution path and object states that characterize software failures" [MSed]. EvoCrash evolutionary search-based approach adopts a novel GGA which uses the crash stack trace to direct the search. Therefore resulting in a smaller search space.

The goal of EvoCrash is to generate a test case that crashes the software in the same location and results in the most similar stack trace to the original crash. Therefore, EvoCrash initially parses the crash stack trace to identify the types of exception thrown and the list of the methods involved in the crash, their class names and line numbers. All this information could be used as inputs for test generating tools like EvoSuite, however, EvoCrash only uses the last stack frame, that is the class that the exception was thrown from as an input for EvoSuite. This will result in satisfying the goal of generating similar crash stack frame as the original crash. Once EvoSuite generates candidate test cases based on the input, fitness function would select the fittest candidates among the test population to regenerate the target crash. Figure 2 shows the whole process of EvoCrash for crash replication. In the coming subsections, we delve into the key steps of this figure.

Figure 2: Diagram for EvoCrash



4.3 EvoCrash's fitness function

EvoCrash's fitness function evaluates test cases based on the fact that they have covered the line where the exception is thrown, whether they trowed the target exception, and if the generated crash stack is

the most similar to the original crash. Formula 1 demonstrates EvoCrash's fitness function.

Formula 1: The EvoCrash fitness function

$$f(t) = 3xds(t) + 2xd - except(t) + d - trace(t) \tag{1}$$

As is shown in the EvoCrash fitness function in Formula 1, the fitness function is consist of three elements: (i) *ds* is the distance of the execution path of the test case with the target statement i.e. the location of the crash, (ii) *d-expect* is a binary value which represent whether the exact exception has been thrown and (iii) *d-trace* calculate the difference between the stack trace generated by the test case and the original crash. The lower the value of the fitness function, the more desirable the test case is. Therefore, if the target exception is thrown *d-except* would be assigned zero, otherwise it would be a one. In case the exception is not thrown, then calculation of the distance between stack traces is not applicable anymore, thus *d-trace* would be set to the maximum value it could have: one.

As it can be observed from the above formula and the fact that *ds* has the highest factor amongst other, it can be deduced that the highest priority for the fitness function is the difference between execution path of the test case and the target crash. In order to measure this difference, EvoCrash adopts two heuristics in the distance function (*ds*), namely approach level and branch distance which will guide the search for branch and statement coverage [MSed]. Approach level calculates the "distance between the path of the code executed by the test case and the target statement" while the branch distance employ a number of well-established rules "to score how close the test case is to satisfying the branch condition for the branch on which the target statement is directly control dependent".

In case the target crash has occurred, the fitness function will assess the differences between the stack trace generated by the test case and the original crash. This technique for measuring the difference between the stack traces will result in better match between stack traces comparing to the prior technique. In the previous technique If two stack traces have a shared element in two different position e.g. 3rd element in one stack trace and 4th element in the other one, the stack traces would be ignored. On the other hand, the new technique considers the closest (shared) element and compares the similarity between generated stack. Fitness function will return zero if and only if the stack traces are identical.

4.4 EvoCrash's Guided Genetic Algorithm

As discussed earlier in this document, EvoCrash adopts Guided Genetic Algorithm (GGA) which focuses on the target failure i.e. last element of stack trace, instead of coverage-based algorithms that consider all the methods in the CUT. GGA creates and evolve test cases which always obtain at least one of the method contained in the stack trace to increase the probability of triggering a similar stack trace to the original crash.

Picking the initial population for genetic algorithms is a crucial step as it samples the search space. Unlike EvoSuite or other coverage-based tools which generate a well-distributed population of all methods in the CUT, initial populator algorithm used in GGA prefers the methods in the crash stack frames over others for its initial population. If the methods in the stack frames are private, the algorithm guarantees invoking them by calling their public invoker. In order to invoke some of these method an object should be instantiated from the class to perform proper method call. For this the algorithm uses `INSERT-METHOD-CALL` which sets the input parameter by re-using objects and variable already defined in the test case, setting some input values to null or randomly generating new objects and primitive values [MSed].

The initial population starts with assigning a low probability $\frac{1}{sizeSet}$ to the crash stack frame and increases the probability in each iteration to ensure the existence of at least one method from crash stack frames. However, during the evolution process, tests might lose the inserted target call. A reason for this could be a single-point cross-over where despite the fact that parents have a method from the crash stack frame, the child will have no method from the crash stack frame. Since the cut-point is random, it is possible that the offspring does not contain the target method. The difference between guided single-point crossover and the standard one described in Section 2 relies on the fact that guided single-point cross-over will monitor the child after recombination to make sure the target methods are not missing, otherwise it would reverse the changes and redefines both offspring as pure copy of the parents and then mutation would be used as a technique for evolution.

After crossover the generated test case might not be well-formed. The offspring, for example, might not contain the constructing methods for calling the target function. The guided single-point crossover

performs correction procedure on the child so all the necessary objects and primitive variables are inserted to the test case thus creating a proper method call.

GGA employs guided mutation as another genetic operator to add, remove, or change some statements in the test case with a low probability after crossover. The algorithm iterates over the statements in the test case and mutates them with the probability of $\frac{1}{n}$. For insertion of a statement, a new method call at a random point i would be added. If a statement is added to the test case, the algorithm makes sure necessary objects are instantiated and required declaration and initialization of primitive variable occurs. In case of modifying a statement, if it is declaration of a primitive variable, the value would be randomly changed. When modifying a method or constructor call, the guided mutation algorithm substitute the method with another public method/constructor which return the same type. The input for this new method could be taken either from the previous in the test case, null, or randomly generated. At last, for removing a statement, the algorithm remove all the corresponding variable and objects as well. In case the test case loses the target method call because of the mutation, the algorithm would iterate until at least one of the targeted method is invoked in the test cases. GGA makes sure that during deletion and modification, relevant calls to the methods in the crash stack trace would not be removed. As the methods were inserted and modified randomly, test cases might contain many irrelevant statement to the crash. Finally, GGA will use test optimization routines used in EvoSuite to reduce the element which does not participate in the fitness function.

4.5 Working with EvoCrash

EvoCrash was available to us as an executable JAR file. The tool should be used by a number of mandatory parameters. Table 1 presents these parameters and their description. The description for most of these parameters are trivial. Those worth mentioning are `Dtarget_frame` and `class`. `Dtarget_frame` specifies the level the tool deepens in the crash stack trace frames. If the stack trace has more than one frame, assigning this value to the level you want to include from your stack trace in the test case. As discussed earlier, During the mutation procedure, the guided approach will guarantee the target methods would exist in the test case. Therefore, you could select which frame(s) to be the target. `Class` specifies the CUT for EvoSuite. As mentioned earlier, it is better to use class of the last frame in the crash stack trace to generate a more similar stack trace.

Table 1: Parameters for EvoCrash

Parameter	Description	Value
generateTests	single test generation mode in EvoSuite	N/A
Dcriterion	Crash replication optimization criterion	CRASH
Dtest_dir	Output folder for generated test cases	Folder location
Drandom_tests	Number of test to produce	INT [e.g 1]
Dminimize	Post optimization (minimizing the test)	TRUE/FALSE
Dtarget_frame	The frame level up to which parse the stack trace	INT
Dvirtual_fs	virtual file system for all File I/O operations	TRUEFALSE
DEXP	Location of crash stack trace	Folder location
Dtarget_exception_crash	target exception	String
Dreplace_calls	Replace nondeterministic calls and System.exit	TRUEFALSE
Dsandbox	use sandbox environment	TRUEFALSE
class	target class for Evosuite	Folder location
projectCP	project dependencies	File location

Once you execute the tool, the tool start printing the information. The result for each iteration is printed out as depicted in Figure 3.

Figure 3: EvoCrash printing out the fitness function result for each children.

```
INFO  evo_logger - lineFitness: 0.9375
INFO  evo_logger - currentFitness: 5.8125
```

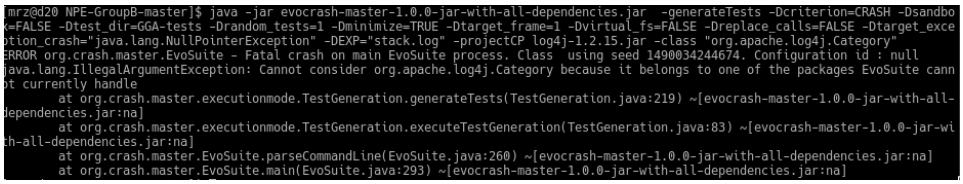
Once the tool finishes, in case it could successfully replicates the crash, it prints out the stack trace and the test case. You can find the test case and run it against your could in the directory you have

specified.

5 Conclusion

This document was focused on crash replication through automated testing. In this paper we described in general how search-based software engineering works and how test automation tools like EvoSuite benefit from it. Then explained in detail what is crash replication and how EvoCrash works. We tried to use EvoCrash for our survey projects. The jar file to be provided for one of the project crashed EvoSuite. The error we got was that the class belongs to one of the packages EvoSuite cannot handle, as seen in Figure 4.

Figure 4: Error for the assignment.



For the other one, the jar dependency file was not provided. We tried to use to maven to build the project, yet unfortunately due to insufficient knowledge about Java, we were not successful. However, we used EvoCrash on the hand on project. We observed the result and the generated crash. The tool provides very useful information to developers. The test case and input/state which caused the crash is vital information to developers for implementing a fix as it is the cause for the failure. Besides, once they have implement a patch, the generated test cases could be used to make sure the fix has solve the issue. These two facts could become easily hidden in big industrial projects. Adopting this tool could save a lot of resources and result in a better quality of testing.

The usage and benefits of the EvoCrash tools has been discussed throughout this document. Our impression of the tool was that it is still under development. It was not very user friendly, and the output did not provide useful and enough information. This issue worsen when the tool itself crashes and there is no way of knowing why. There were some incident where the tool ran with printing nothing at all and showing no output.

One of the main reasons why automatic testing tools are not widely adopted is because developers do not want to waste time learning how to use the complicate tools for testing. Unfortunately it was hard for us to learn the tool in detail. To make the tool widespread, it has to be as easy to use as possible. A graphical user interface could be added and a set of default options could help a beginner user to run the program. A simple tutorial on how to use the EvoCrash tool will also be a good addition.

References

[AA10] Lionel Briand Andrea Arcuri, Muhammad Zohiaib Iqbal. Formal analysis of the effectiveness and predictability of random testing. 2010.

[GF13] Andrea Arcuri Gordon Fraser. Whole test suite generation. *IEEE Transactions on software engineering*, 39(2), 2013.

[McM04] Phil McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability.*, 14(2):105–156, 2004.

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

[RR06] Klaus Wolfmaier Rudolf Ramler. Economic perspectives in test automation: Balancing automated and manual testing with opportunity cost. *Automation of software test*, pages 85–91, 2006.

[SP] M. Beller A. Zaidman H. Gall S. Panichella, A. Panichella. The impact of test case summaries on bug fixing performance: An empirical investigation.