# Testing and Reverse Engineering

Tim van der Lippe (4289439) & Chak Shun Yu (4302567)
& Thomas Smith (4316320)

March 20, 2017

## 1 Search Based Software Engineering

Search Based Software Engineering (SBSE) attempts to solve software engineering problems using metaheuristics search techniques. An essential bridging process to perform in SBSE, is to redefine the software engineering problem into a search problem [1]. The aim of SBSE is to move software engineering problems away from human-based searching and putting the focus on machine-based searching [1]. This way, humans can put the focus on guiding this automated search, instead of performing the search manually [1].

The concept of automatic test code generation and crash replication is to provide and validate input to a program. Manual performance of this task by an individual is extremely costly, difficult and laborious [2]. Automation of an exhaustive enumeration of a program's input have proven to be unsuccessful due to scaling in size and complexity of software. This is where SBSE comes in. Metaheuristics search techniques are designed to search for an (near) optimal solution in a finite search space at a reasonable computational cost. All statements and branches in a software program can be translated to constraints. Based on these constraints, it is possible to define an objective (or fitness) function. This function represents a finite area of the program's input, which ensures the search space needed for applications of search algorithms.

## 2 EvoSuite core concepts

EvoSuite [3] uses a genetic algorithm to generate a population of test suites. A test suite contains a set of tests. Each test consequently consists of a set of statements.

The genetic algorithm applies transformation to each test suite in the population. It has two options: applying crossover between two test suites and/or mutate a test. During crossover, a random position in testsuite A is chosen. All tests up to that position are swapped with the corresponding tests on these positions in testsuite B. The other option is mutation, in which one statement of a test is mutated into a different statement.

EvoSuite starts with a random set of tests to initialize the genetic algorithm. Based on these tests, EvoSuite will run the tests on the source code and obtain the corresponding coverage. Based on the coverage distance function, EvoSuite will apply crossover and mutation as described above. The coverage distance function is defined as the distance between a test suite and a given branch

b. If the branch is covered, the distance is equal to 0. If the branch is not covered, the distance is equal to 1. EvoSuite also tries to minimize the addition of redundant tests, in which it increases the distance the more times a branch is covered. Therefore, if a branch is covered twice, it is better than being covered five times. All in all, the genetic algorithm aims to minimize the total distance function of a test suite and all available branches.

# 3 EvoCrash description

EvoCrash[4] is a tool built on top of EvoSuite to provide developers a means of effortlessly reproducing a crash scenario after a crash has been reported. The tool aims to generate a test case that reproduces a crash with only the stack trace of that crash as input. Preliminary research shows that a genetic algorithm with a simple fitness function can sometimes be effective in replicating a failure. In general however, this approach is not ideal because existing testing frameworks are aimed at maximizing coverage instead of reproducing very specific crash scenarios.

To combat this issue, EvoCrash uses a *guided* genetic algorithm to narrow down the search space. A key step in reducing the search space is to define a better initial population of tests. Instead of aiming for a uniform sample of the search space, the selection algorithm in EvoCrash will prioritize method calls that appear in the input stack trace. The fitness function must ensure that the search is guided towards a suitable candidate test that reproduces the desired failure. To this end, the fitness is evaluated based on whether:

- the statement where the exception is thrown is covered
- the target exception was thrown
- generated stack trace is similar to the original one

Lastly, the guided algorithm, used to find a candidate test while maximizing fitness, employs specialized crossover and mutation operators to ensure that at least one method in the stack trace is covered. After the search terminated, the test with the highest fitness value is post-processed to minimize the amount of statements in the test and make it more suitable for developer consumption.

# 4 How EvoCrash did/didn't help with debugging

EvoCrash can save time by generating a test case that reproduces a crash. However, this test will lack things like documentation and sensible names that help developers understand the root cause of a crash. Understanding is arguably the most important feature of a test that demonstrates a fault, since it is used mainly to communicate between the person that finds a bug and the person that has to fix the bug. While EvoCrash does attempt to generate more readable tests by post-processing, it is still impossible to generate good names and documentation. This makes that developers still need to spend a significant amount of time to:

- understand how the test triggers the fault
- check whether the test correctly triggers the intended fault
- rewrite and document the test to make it suitable for consumption by other developers

Time spent on these three things can significantly be reduced if the reporter of a bug provides a well-documented test case. This is not something that EvoCrash facilitates or encourages.

On the other hand, the test case can be directly executed and used in combination with a debugger. A developer can set a breakpoint on the line that throws the exception and directly inspect the stack-trace, local variables and their state. Therefore, EvoCrash saves the developer time coming up with a reproducible test case and allows the developer to more quickly inspect the issue with a debugger.

# 5 EvoCrash help during programming

We have tried to use EvoCrash on stack traces in two projects: Netty and RxJava. Sadly, in both projects we were unable to get EvoCrash to produce a useful result. For RxJava, the problem turned out to be twofold. The first problem is that RxJava heavily relies on inner classes in their design. EvoSuite seems to be unable to deal with this. The second problem appeared after manually extracting the involved member class to a separate file. EvoCrash was able to generate the test listed below. It seems like the generated test was not post-processed properly as only line 6, 25 and 26 are useful for replicating the crash. Another observation is that the generated statements are very verbose and repetitive. Most of the qualified calls could have been avoided by an import, for example.

```
1  Consumer<Object> consumer0 = null;
2  ObservableReplay.ReplayObserver<String> observableReplay_ReplayObserver0 = null;
3  ObservableReplay.InnerDisposable<String> observableReplay_InnerDisposable0 = new
       ObservableReplay.InnerDisposable<String>(observableReplay_ReplayObserver0,
       observableReplay_ReplayObserver0);
4  observableReplay_InnerDisposable0.isDisposed();
5  int int0 = 0;
6  ObservableReplay.SizeBoundReplayBuffer<ObservableInternalHelper.
       ErrorMapperFilter> observableReplay_SizeBoundReplayBuffer0 = new
       ObservableReplay.SizeBoundReplayBuffer<ObservableInternalHelper.
       ErrorMapperFilter>(int0);
7  ObservableReplay.ReplayObserver<ObservableInternalHelper.ErrorMapperFilter>
       observableReplay_ReplayObserver1 = new ObservableReplay.ReplayObserver<
       ObservableInternalHelper.ErrorMapperFilter>(
       observableReplay_SizeBoundReplayBuffer0);
8  ObservableReplay.InnerDisposable<ObservableInternalHelper.ErrorMapperFilter>
       observableReplay_InnerDisposable1 = new ObservableReplay.InnerDisposable<
       ObservableInternalHelper.ErrorMapperFilter>(observableReplay_ReplayObserver1,
       observableReplay_ReplayObserver1);
9  observableReplay_InnerDisposable1.isDisposed();
10 int int1 = 1906;
11 TimeUnit timeUnit0 = TimeUnit.HOURS;
12 TestScheduler testScheduler0 = new TestScheduler();
13 ObservableReplay.SizeAndTimeBoundReplayBuffer<Object>
       observableReplay_SizeAndTimeBoundReplayBuffer0 = new ObservableReplay.
       SizeAndTimeBoundReplayBuffer<Object>(int1, int0, timeUnit0, testScheduler0);
14 ObservableReplay.ReplayObserver<Object> observableReplay_ReplayObserver2 = new
       ObservableReplay.ReplayObserver<Object>(
       observableReplay_SizeAndTimeBoundReplayBuffer0);
15 observableReplay_ReplayObserver2.onComplete();
16 int int2 = 0;
17 ObservableReplay.UnboundedReplayBuffer<ObservableInternalHelper.MapToInt>
       observableReplay_UnboundedReplayBuffer0 = new ObservableReplay.
       UnboundedReplayBuffer<ObservableInternalHelper.MapToInt>(int2);
18 ObservableReplay.ReplayObserver<ObservableInternalHelper.MapToInt>
       observableReplay_ReplayObserver3 = new ObservableReplay.ReplayObserver<
       ObservableInternalHelper.MapToInt>(observableReplay_UnboundedReplayBuffer0);
19 ObservableReplay.InnerDisposable<ObservableInternalHelper.MapToInt>
       observableReplay_InnerDisposable2 = new ObservableReplay.InnerDisposable<
       ObservableInternalHelper.MapToInt>(observableReplay_ReplayObserver3,
       observableReplay_ReplayObserver2);
20 observableReplay_InnerDisposable2.isDisposed();
21 int int3 = 0;
22 ObservableReplay.SizeBoundReplayBuffer<ObservableInternalHelper.
       ErrorMapperFilter> observableReplay_SizeBoundReplayBuffer1 = new
       ObservableReplay.SizeBoundReplayBuffer<ObservableInternalHelper.
       ErrorMapperFilter>(int3);
23 SQLIntegrityConstraintViolationException
       sQLIntegrityConstraintViolationException0 = new
```

```
        SQLIntegrityConstraintViolationException();
24  observableReplay_SizeBoundReplayBuffer1.error(
        sQLIntegrityConstraintViolationException0);
25  ObservableReplay.ReplayObserver<ObservableInternalHelper.ErrorMapperFilter>
        observableReplay_ReplayObserver4 = new ObservableReplay.ReplayObserver<
        ObservableInternalHelper.ErrorMapperFilter>(
        observableReplay_SizeBoundReplayBuffer0);
26  observableReplay_ReplayObserver4.dispose();
```

For Netty, we did not manage to get EvoSuite working at all. We tried running the tool with command line options:

```
1  java -jar evocrash-master-1.0.0-jar-with-all-dependencies.jar \
2      -generateTests -Dcriterion=CRASH -Dsandbox=FALSE -Dtest_dir=GGA-tests \
3      -Drandom_tests=1 -Dminimize=TRUE -Dtarget_frame=1 -Dvirtual_fs=FALSE \
4      -Dreplace_calls=FALSE -Dtarget_exception_crash="java.lang.
           NullPointerException" \
5      -DEXP="trace.txt" -class "io.netty.handler.codec.DecoderException" \
6      -projectCP "netty/all/target/netty-all-4.1.9.Final-SNAPSHOT.jar"
```

This gave the following output:

```
1  INFO  evo_logger - * Starting client
2  INFO  evo_logger - * Connecting to master process on port 12172
3  INFO  evo_logger - * Analyzing classpath:
4  INFO  evo_logger -   - netty/all/target/netty-all-4.1.9.Final-SNAPSHOT.jar
5  INFO  evo_logger - * Finished analyzing classpath
6  INFO  evo_logger - * Generating tests for class io.netty.handler.codec.
       DecoderException
7  INFO  evo_logger - * Test criterion:
8  INFO  evo_logger -   - Crash Coverage Testing
9  INFO  evo_logger - * Setting up search algorithm for individual test generation
10 INFO  evo_logger - * Passed the target level 1, so stopping the parser!
11 ERROR org.crash.master.statistics.SearchStatistics - No statistics has been
       saved because EvoSuite failed to generate any test case
```

The program seems to skip the search entirely and exits with a very general error.

# 6 How can EvoCrash be improved?

While the direct reproducible test case is very welcome, reading the code is challenging. The direct cause for lack of readability is the generation of non-descriptive variable names.

In practice, tests are not only to verify the correctness of the program, they also serve as teaching new developers joining the project. A new developer can inspect the existing tests to get a good grasp of the program behavior, its API usage and code-documentation of bugs that were originally in the program. To be able to have clear code-documentation, tests must have descriptive names and reasonably named variables. At the moment, EvoCrash automatically generates the variable names, which convey no extra information. For developers, reading

such a test case is hard and might be even harder than without a test case. E.g. you are trying to debug/reason about the generated test case, which might take more time than reproducing the exception yourself.

The two experiences described in section 5 have lead us to the conclusion that the usability of EvoCrash could be improved upon before it is ready to be used during daily development. The two main improvements that we would like to see are:

- More readable and minimal test cases
- Clear error messages and documentation

# 7 Would we use EvoCrash/EvoSuite daily?

In Section 6, we have described our issue with the non-descriptiveness of the test created by EvoCrash. In case this would be fixed, we would very well consider to integrate EvoCrash into our programming routines, as it aids in the documentation of the source code, the communication between developers and debugging.

We are also of the opinion that EvoCrash is more useful for developers who are less familiar with the project. An developer more familiar with the project often already has an idea where to look in the source code and how to correctly debug the error. This does often not hold for less familiar developers. For them, a generated test can serve as a very helpful starting point to understand the error. This also helps them in understanding the project.

Another reason why we think EvoCrash might not serve very useful, as will only be able aid in fixing bugs caused by exceptions. Unfortunately, bugs are not always caused by exception, thus diminishing the usefulness of EvoCrash.

# References


[1] Mark Harman. *Search Based Software Engineering*, pages 740–747. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

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

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

[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.