TU Delft

CS4110 Software Testing and Reverse Engineering

Group 4 Testing

Software Testing Assignment 1

Group Members:
Maria Gatou, 4631811
Papakonstantinopoulos Ioannis, 4631838
Touloumis Konstantinos, 4620666

March 19, 2017

Introduction

Search-based software engineering (SBSE) applies metaheuristic search techniques such as genetic algorithms, simulated annealing and tabu search to software engineering problems. Many activities in software engineering can be stated as optimization problems. Optimization techniques of operations research such as linear programming or dynamic programming are mostly impractical for large scale software engineering problems because of their computational complexity. Researchers and practitioners use metaheuristic search techniques to find near-optimal or "good-enough" solutions [1]. SBSE problems can be divided into two types:

- black-box optimization problems, for example, assigning people to tasks (a typical combinatorial optimization problem).
- white-box problems where operations on source code need to be considered.

Definition: SBSE converts a software engineering problem into a computational search problem that can be tackled with a metaheuristic. This involves defining a search space, or the set of possible solutions. This space is typically too large to be explored exhaustively, suggesting a metaheuristic approach. A metric (also called a fitness function, cost function, objective function or quality measure) is then used to measure the quality of potential solutions. Many software engineering problems can be reformulated as a computational search problem[1]. The term "search-based application", in contrast, refers to using search engine technology, rather than search techniques, in another industrial application.

Metaheuristic search techniques have been applied to automate test data generation and crash replication in the following areas[2]:

- the coverage of specific program structures, as part of a structural, or white-box testing strategy
- the exercising of some specific program feature, as described by a specification;
- attempting to automatically disprove certain grey-box properties regarding the operation of a piece of software, for example trying to stimulate error conditions, or falsify assertions relating to the software's safety;
- to verify non-functional properties, for example the worst-case execution time of a segment of code.
- to formulate and optimize a proper fitness function to guide the search toward reaching the test goal as well as for the promotion of tests closer to cover the target goal and penalize tests with weak fitness values.

EvoSuite: Automatic Test Suite Generation for Object-Oriented Software

EvoSuite[3] is a tool that automatically generates unit tests with assertions for classes written in Java code. EvoSuite uses an evolutionary algorithm to generate JUnit tests. EvoSuite can be run from command line, and it also has plugins to integrate it in Maven, IntelliJ and Eclipse. EvoSuite has been used on more than an hundred open-source software and several industrial systems, finding thousands of potential bugs.

To achieve this, EvoSuite applies a novel hybrid approach that generates and optimizes whole test suites towards satisfying a coverage criterion. Optimizing with respect to a coverage criterion rather than individual coverage goals achieves that the result is neither adversely influenced by the order nor by the difficulty or infeasibility of individual coverage goals.

For the produced test suites, EvoSuite suggests possible oracles by adding small and effective sets of assertions, using mutation testing, that concisely summarize the current behavior. In mutation testing, artificial defects (mutants) are seeded into a program, and test cases are evaluated with respect to how many of these seeded defects they can distinguish from the original program. A mutant that is not detected shows a deficiency in the test suite and indicates in most cases that either a new test case should be added, or that an existing test case needs a better test oracle. These assertions allow the developer to detect deviations from expected behavior, and to capture the current behavior in order to protect against future defects breaking this behavior.

EvoSuite currently supports branch coverage and mutation testing as test objectives, but some new criteria are examined,based on data flow, as well as further inspection is made on producing more readable test cases.

EvoCrash: A Guided Genetic Algorithm for Automated Crash Reproduction

EvoCrash[4] is a search based approach, for crash reproduction. EvoCrash is built on top of EvoSuite, the well-known automatic test suite generation tool for Java.

Features that make EvoCrash special are the fitness function which is used to guide the search toward reaching the target, and applying a proper search algorithm to "reward" tests closer to mimicking the crash, while "punish" tests with poor fitness values. These two features are the basic principles of EvoCrash.

Crash Stack trace processing: To be a test case optimal for crash reproduction, has to crash at the same location as the original crash and the stack trace produced close to the original one. In order to achieve that, the target class is always targeted where the exception is thrown. Therefore,the log file given as input is parsed in order to extract the crash stack frames.

Fitness function: The fitness function is formulated to consider three main conditions that must hold so that a test case would be evaluated as optimal and have zero distance: (i) the line (statement) where the exception is thrown has to be covered, (ii) the target exception has to be thrown, and (iii) the generated stack trace must be as similar to the original one as possible.

Genetic Algorithm: In EvoCrash a novel genetic algorithm is used, namely GGA(Guided Genetic Algorithm). GGA gives higher priority to those methods involved in the target failure, in contrast with traditional search algorithms. Moreover GGA uses three novel genetic operators that create and evolve test cases that always exercise at least one method contained in the crash stack trace, increasing the overall probability of triggering the target crash. The steps are the following:

- an initial population of random tests is created
- it evolves such tests over subsequent generations using crossover and mutation
- at each generation, it selects the fittest tests according to the fitness function

What makes EvoCrash different is that it uses:

- a novel routine for generating the initial population
- a new crossover operator
- a new mutation operator

Finally, the fittest test obtained at the end of the search is optimized by post-processing

Initial population: Concerning the initial population, search algorithms focus more on generating a well-distributed population whereas EvoCrash uses a routine that gives higher importance to methods contained in crash stack frames.

Crossover: GGA leverages a novel guided single-point crossover operator in order to avoid generates two offsprings by randomly exchanging statements between two parent tests p1 and p2. The first steps are the same with the standard single-point crossover, however an offspring is defined as a pure copy of its parents. At the end of this stage a correction algorithm is applied because moving method calls from one test to another may result in non well-formed tests

Guided Mutations: After crossover, new tests are usually mutated (with a low probability) by adding, changing and removing some statements. While adding statements will not affect the type of method calls contained in a test, the statement deletion/change procedures may remove relevant calls to methods in the crash stack frame. When changing a statement at position i, the mutation operator has to handle two different cases:

- if the statement s_i is the declaration of a primitive variable, then its primitive value is changed with another random value.
- if s_i contains a method or a constructor call m, then the mutation is applied by replacing m with another public method/constructor having the same return type while its input parameters (objects or primitive values) are taken from the previous $i-1$ statements in t, set to null (for objects only), or randomly generated.

Post processing: Since method calls are randomly inserted/changed during the search process, the final test can contain statements not useful to replicate the crash. For this reason, GGA post-processes that test to make it more concise and understandable. For this post-processing, optimization routines available in EvoSuite are reused.

- test minimization
- values minimization

Test minimization applies a simple greedy algorithm: it iteratively removes all statements that do not affect the final fitness value. Finally, randomly generated input values can be hard to interpret for developers. Therefore, the values minimization from EvoSuite shortens the identified numbers and simplifies the randomly generated strings.

Test generation and Crash Fix

After participating in the survey all group members gathered the following results which involved:

- Fixing a crash that occured from a test case (first part).
- Generating a test case for replicating a crash and provide a solution (second part).

The first source code that we had to deal with is NPE code with the following stack trace:

```
java.lang.NullPointerException:
    at org.apache.log4j.net.SyslogAppender.append(SyslogAppender.java:312)
    at org.apache.log4j.AppenderSkeleton.doAppend(AppenderSkeleton.java:251)
    at org.apache.log4j.helpers.AppenderAttachableImpl.appendLoopOnAppenders(AppenderAttachableImpl.java:66)
    at org.apache.log4j.Category.callAppenders(Category.java:206)
    at org.apache.log4j.Category.forcedLog(Category.java:391)
    at org.apache.log4j.Category.log(Category.java:856)
```

Figure 1: Original Stack Trace

Following the stack trace from the reverse order starting from the log function in Category class we end up in function append in the SyslogAppender class.We can see that in line 312: *String packet = layout.format(event);* the case of the event being empty(null) is not covered and that may lead to a potential crash. The test case that replicates the crash is shown below:

```
public void testCrash() throws Exception{
SyslogAppender syslogAppender0 = new SyslogAppender();
      syslogAppender0.setSyslogHost(", true) call failed.");
      Object mockMinguoDate0 = new Object();
      Exception mockException0 = new Exception();
      ThrowableInformation throwableInformation0 = new ThrowableInformation((Throwable) mockException0);
      Throwable mockThrowable0 = new Throwable(",3U");
      LocationInfo locationInfo0 = new LocationInfo((Throwable) mockThrowable0, ", true) call failed.");
      LoggingEvent loggingEvent0 = new LoggingEvent((String) null, (Category) null, 0L, (Level) null,
                    (Object) mockMinguoDate0, "    ", throwableInformation0, (String) null, locationInfo0, (Map) null);
      syslogAppender0.append(loggingEvent0);
  }
```

Figure 2: Testing the crash

The append function called by passing as argument a LoggingEvent item with all its components being empty(null) that how the crash occurs. To fix the crash we add the following branch criterion:

```
if(event.timeStamp==0) return;
```

Figure 3: Fixing the crash

The logic behind this intervention is that a null event will have its timeStamp initialised with 0. That's how we can tell the difference between an empty and a non empty LoggingEvent. When such an empty event is found we force the function to stop and return to the previous function in the stack trace.

The second source code that we had to deal with is IAE code with the following stack trace:

```
1 java.lang.IllegalArgumentException: Initial capacity must be greater than 0
2     at org.apache.commons.collections.map.AbstractHashedMap.<init>(AbstractHashedMap.java:142)
3     at org.apache.commons.collections.map.AbstractHashedMap.<init>(AbstractHashedMap.java:127)
4     at org.apache.commons.collections.map.AbstractLinkedMap.<init>(AbstractLinkedMap.java:95)
5     at org.apache.commons.collections.map.LinkedMap.<init>(LinkedMap.java:78)
6     at org.apache.commons.collections.map.TransformedMap.transformMap(TransformedMap.java:153)
7     at org.apache.commons.collections.map.TransformedMap.putAll(TransformedMap.java:190)
```

Figure 4: Original Stack Trace

By following the Stack trace from the reverse order starting from putAll function of the TransformedMap class, we can see that an illegal argument

exception occurs in AbstractHashedMap in line 142: *if (initialCapacity < 1)* throw new IllegalArgumentException("Initial capacity must be greater than 0");

The test case that replicates the crash is shown below:

```
public void testTheCrash() {
        HashMap<Integer, Integer> hashMap0 = new HashMap<Integer, Integer>();
        TransformedMap transformedMap0 = new TransformedMap(hashMap0, (Transformer) null, (Transformer) null);
        transformedMap0.putAll(hashMap0);
    }
```

Figure 5: Testing the crash

In this test an empty HashMap item is initialised and then it is transformed by using the TransformedMap constructor. After this, the transformedMap object is passed as an argument to the putAll function.

To fix this crash from occuring we add the following code on top of the function transformMap in class TransformedMap. In case an empty Map argument is given, whose size is 0, we return that same empty map in order to interrupt the error from propagating and reaching the exception.

```
if (map.size()==0) {
                return map;
        }
```

Figure 6: Fixing the crash

Using Evocrash in the assignment

In the second part of the assignment where we had to use EvoCrash, when testing the NPE code and having no tests to check where the problem is, we provided as input for EvoCrash in the command line the Stack Trace log file and the -Dtarget_frame accordingly. We were able to see that the automated test generated was the one calling the append function with a null LoggingEvent as argument and of course that replicates the crash, which is that the append function cannot handle null events.

Example of using EvoCrash

As an example of how EvoCrash could have helped us in our programming tasks, we could take a simple example of the inheritance problem. If there is a problem with the parent class, in all the offsprings of this class it would almost be impossible to spot the error.

In Evocrash we tested the following simple case of inheritance where FreeEmployee is the base class and it's constructor is not made to handle null objects. So we are dealing with a NullPointerException.

Figure 7: Fixing the crash

In Evocrash when determining the -Dtarget_frame=5 and by putting in the Stack Trace the line where the potential crash is in the class Employer, we were given an automated test crash where the constructor was called with a null argument.

Evocrash: Future work and Improvements

The future work may take several directions, including [4]:

1. enhancing the fitness function implemented in EvoCrash
2. extending the comparison between EvoCrash and the other techniques, which considerably would depend on the availability of the tools
3. evaluating EvoCrash for industrial projects

Usefulness and Establishment of Evocrash

Manual crash replication is a labor-intensive task. Developers faced with this task need to reproduce failures reported in issue tracking systems, which all too often contain insufficient data to determine the root cause of a failure.

Hence, EvoCrash [4] help us to save cost and time, and in this stage can be a great help for the software development process. Evocrash as an evolutionary search based approach, synthesizes a crash reproducible test case by using a guided genetic algorithm (GGA) for crash reproduction in combination with the development of a smart fitness function that improves the calculation of stack trace distance and finally it outperforms previous testing approaches in automated crash reproduction.

Moreover, Evocrash proved to be very effective in relation with Crash Coverage, since the test generated by EvoCrash results in the generation of the same type of exception at the same crash line as reported in the crash stack trace. In terms of Test Case Usefulness, Evocrash found to be also very effective, since the generated test case by EvoCrash managed to reveal the actual bug that causes the original crash.

For all the aforementioned reasons, Evocrash would be a very useful addition in our daily programming routines.

References

[1] Wikipedia, *Search Based Software Engineering*, https://en.wikipedia.org/wiki/Search-based_software_engineering

[2] McMinn, Phil. *Search based software test data generation: A survey.* Software Testing Verification and Reliability 14.2 (2004): 105-156.

[3] Fraser, Gordon, and Andrea Arcuri. *Evosuite: automatic test suite generation for object-oriented software.* Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ACM, 2011.

[4] Soltani, Mozhan, Annibale Panichella, and Arie van Deursen. *A Guided Genetic Algorithm for Automated Crash Reproduction.* Proceedings of the 39th International Conference on Software Engineering (ICSE 2017). ACM, 2017.