Delft University of Technology

Software Testing and Reverse Engineering CS4110

REPORT

Testing Assignment

Group 14

Author:
Jake Beinart (4646223)
Pim van den Bogaerdt (4215516)
Ade Setyawan Sajim (4608232)

Email:
J.S.Beinart@student.tudelft.nl
P.vandenBogaerdt@student.tudelft.nl
A.Sajim@student.tudelft.nl

20 March 2017

1 Introduction

In a software development process, debugging is the process of finding and resolving errors that prevent the proper working of a system. During debugging, one usually starts by looking at the crash stack trace and trying to reproduce the crash. One way to reproduce the crash is by debugging manually. Unfortunately, manual crash reproduction is exhausting and time-consuming [1].

Search-based software engineering attempts to take a normal software engineering problem, such as test code generation or debugging, and solve it using computational search methods. This involves dividing the problem into a set of possible solutions in a certain space. When applied to debugging, the space can e.g. be the set of all parameters and statements that can be used to create or recreate a specific crash that the user needs to fix.

EvoCrash does exactly this: by accepting the stack trace of the crash as input, and using a Guided Genetic Algorithm (GGA) to combine generated tests, it attempts to reproduce the stack trace of the error that has occurred [2]. This obviously presents an invaluable asset to developers who just can't seem to replicate a crash in their code.

1.1 EvoCrash Way to Automatically Generate Tests

EvoCrash is a tool that attempts to reproduce a crash, given a stack trace of it. Tests in EvoCrash are first generated with a bias towards methods that are included in the crash stack frames. An initial population of tests is generated in such a way that each of the firstborn tests include at least one of these higher priority methods. Otherwise, the tests are filled with public member classes of available classes.

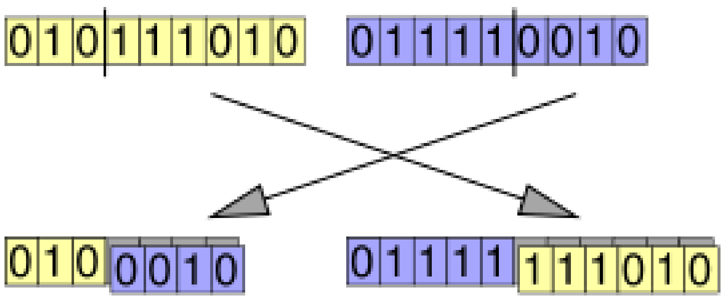

Figure 1: Crossover example [3]. The bits should be regarded as statements.

From this point, tests can be subject to crossover or mutation. During the crossover process, two parent tests have a section of their code swapped and recombined to create offspring classes (as seen in Figure 1). This can, however, cause an offspring test to lose its priority method, so EvoCrash checks for this and replaces any offspring that lose their priority methods entirely to be replaced by a copy of their parents instead. The guided mutation process augments the tests by either adding, removing, or changing statements in the test. If a target method is lost in this step, the mutation process loops until a target method is re-inserted; otherwise, it is terminated.

1.2 EvoCrash Methodology

This section is based on [2].

In this section, we will consider EvoCrash in more detail. EvoCrash generates an initial test set, manipulates it using methods inspired by genetics, and finally selects a best test case.

EvoCrash bases the search for a crash test on the Java stack trace given by the user. The stack trace contains fundamental information for how to reproduce the exception: it includes the exception name as well as the code site where the exception is thrown. Moreover, it contains call sites that consecutively lead to the exception throw site. These call sites are relevant because they might (but not necessarily) give an indication where the root cause is.

The main part of EvoCrash is a *Guided Genetic Algorithm*. This algorithm attempts to get closer and closer to a test case that reproduces the crash. In order to do so, an initial set of tests is created, and then two phenomenons from genetics are applied iteratively: *crossover* and *mutation*. These create new test cases derived from previous ones. Like in biology, the “fittest” test survives. Here, “fittest” refers to how close the test is to reproducing the crash.

More specifically, the tests in the initial test set are created by inserting call statements both to methods in the class that throws the exception and to methods in classes that appear in the stack trace. The probability of picking a call of either type depends on what calls were previously inserted. When this insertion process is done, the test is made compilable by e.g. constructing objects and parameters for the calls.

The crossover technique refers to combining two tests by swapping the first k statements, resulting in two new tests. It is noted, however, that crossover can possibly move the relevant call statements to one test case, rendering the other test case less useful. If this happens, the crossover is undone. If crossover does not give this issue, then the new tests are made compilable similar to before.

The mutation technique refers to inserting, changing and deleting statements from a test. To avoid losing calls that are relevant for triggering the crash, this process is guided as well. For example, when updating a method call, the new arguments may be fetched from arguments used in earlier calls. Presumably, such arguments are more likely to trigger the crash than using random values. (In fact, this idea seems essential for triggering the bug in our binary search tree, which we explain below.)

Lastly, the fittest test is selected, which is then minimized by removing unneeded statements and cleaning up (i.e., making shorter) random numbers used in the test. To determine which test is the fittest, EvoCrash uses a fitness function which depends on how far a test is to executing the crashing statement, and if so, the similarity between the obtained stack trace and the provided one. The statement distance is computed by considering, for example, to what extent the test satisfies the predicates needed to reach the crashing statement. The stack trace similarity is computed by comparing *each pair* of stack trace elements, rather than elements on the same line in the trace, so as to avoid penalizing two stack traces where an identical element occurs on different lines. The stack trace distance is also normalized to the range $[0, 1]$.

2 Results

In this assignment, every group member was asked to perform some debugging and testing experiments. Further details on the results of the assignment is provided below.

2.1 How EvoCrash Helped in The Debugging Assignments

2.1.1 Debugging Assignment

During the experiments, it was found that using EvoCrash to complete the assignment was not entirely necessary. We proceeded to debug some without using an EvoCrash test case.

As a concrete example, one of us did the `IAE.GroupB` case, which is an exception thrown in Apache Commons when an empty map is passed to a certain method. No crash test case was given, yet it was not difficult to find the culprit. The stack trace contains the exception message concerning a capacity being zero. It was feasible to consider the stack trace, and conclude how a zero capacity can end up at the throw site. The reason the missing test was not a problem is that the exception does not heavily depend on state of objects, and so the stack trace contains sufficient information to reproduce the crash. In contrast, the crash we describe in the next subsection does depend on state.

2.1.2 Self-Study: Binary Search Tree

Besides doing the given tasks, we also attempted to apply EvoCrash to a toy class we wrote ourselves. Specifically, we created a simple binary search tree implementation with methods `find(int)`, `insert(int)` and `delete(int)`. We tried to make this bug-free but we did add a custom throw statement when some non-trivial condition holds.

More concretely, when attempting to delete a key in a binary search tree where the key's node has two children, the key has to be replaced with the largest key in the node's left subtree to ensure the tree is still ordered. In case this largest key's node is on a depth ≥ 3 from the root, we throw a null pointer exception. This does not make any semantic sense; we did so to make the code crash when a non-trivial condition holds: for the crash to occur, several nodes have to be added and then a specific type of node has to be deleted.

After some help from Ms Soltani, we managed to get EvoCrash to reproduce the crash. The test indeed inserts keys and then deletes a key in such a way that we crash due to a node on depth ≥ 3. It is worth noting that the key used for deleting is indeed inserted through a previous

```
java.lang.NullPointerException
    at tree.Tree.delete(Tree.java:110)
    at tree.Main.main(Main.java:21)
```

Figure 2: Stack trace for binary search tree crash. Note that it does not include the state of the tree.

```
Tree tree0 = new Tree();
tree0.insert((-2217));
tree0.insert(443);
tree0.insert(2229);
tree0.insert(3);
tree0.insert(371);
// Undeclared exception!
try {
  tree0.delete(443);
  fail("Expecting exception: NullPointerException");

} catch(NullPointerException e) {
  //
  // no message in exception (getMessage() returned null)
  //
  verifyException("tree.Tree", e);
}
```

Figure 3: Test case generated by EvoCrash for the binary search tree case.

statement (see Figure 3), relating to the guided mutation technique that we mentioned in the section on EvoCrash Methodology.

As we said, our code is a contrived example where we know how to trigger the crash. Still, we believe EvoCrash shows its potential because the stack trace itself is not very useful. Indeed, the crash depends on the state (structure of the nodes) of the tree, which is not captured by the stack trace (see Figure 2). A test case that shows how to build the tree so as to trigger the crash seems helpful in a real-world case. Also, EvoCrash did not take too long to generate the test case (less than one minute). On the other hand, the test case is not perfectly clean: the keys inserted are pretty random/big (namely $-2217, 443, \ldots$, which could be normalized to a permutation of $1, \ldots, 5$) even though EvoCrash is said to apply so-called “value minimization” [2, p.6]. Also, the numbers are *sometimes* wrapped in double parentheses, as in `tree.insert((-2217))`, and we don’t see the point of this.

2.2 EvoCrash’s Role in Helping Programming Endeavours

Failure can happen to any software. Unfortunately, some failures are hard to analyze and reproduce. When using EvoCrash, it could help the developer in creating a test case for a failure automatically. To manually make the correct test case that reproduces a failure can be challenging since one needs to analyze the source code, utilize the debug tools inside the

IDE, and may also see the documentation, which can be a time greedy activity. This time cost increases rapidly with the growing of the project scale. Having a test case that could regenerate duplicate the failure will ease the developer in debugging the problem and solving it.

By using EvoCrash, one obtains a free regression test to prevent the crash from occurring in the future. This feature is an excellent benefit since it helps the developer to ensure the system still performs correctly after it was altered or interfaced with other software.

An excellent example of how EvoCrash could be useful is by applying this piece of software to help contributors in a large open source project, e.g. Netty [4], to solve the reported issues inside its repository. This project has more than $300,000$ lines of code [5] which makes an analysis possibly complicated, especially for non-experts and new contributors. By using EvoCrash to create test cases for them, the time consumed for error checking and debugging of a project can be reduced. Thus, the development speed can be maintained or even better, accelerated.

Even though such an example exists, a further look at how EvoCrash works in a complex project is needed since we think EvoCrash still has some limitations. These limitations can be a challenging for EvoCrash and might make EvoCrash does not perform as it supposed to be, especially in a complicated and large project. We elaborate on this in the next subsection.

2.3 Improvements for EvoCrash

After using EvoCrash, some suggestions were raised, especially on how to improve EvoCrash. Below are several notes about how EvoCrash could be improved.

- **Ease for New Users** - Having an UI and more extensive documentation would make EvoCrash much more navigable for new users.
- **Data Visibility** - Providing data on objects that are created will allow the user to see changes made to objects that aren't currently visible.
- **Integration with IDEs** - Using EvoCrash as a plugin in Eclipse would streamline the process of debugging and centralize the programming process in the IDE.
- **Quality of Life** - Having an option to limit the output to only the found test case would increase readability. Additionally, cleaning up the test cases when - for example - only one object is created, name the object `t` rather than `t0`. Cleaning up the double parentheses (mentioned earlier) would also increase readability.
- **General Ideas** - Value minimization could be improved since our test with the binary search tree yielded large numbers. There was also confusion around the test case because it seemed to us that it must be run with a special test runner. Instead, we copied the test body to a separate file to check whether it works.

2.4 Discussion on Future Use

Our group had diverse opinions on the use of EvoCrash and agreed that using EvoCrash as an early attempt to reproduce a failure or bug is a good idea: it operates quickly and using a tool to solve a problem in a little time automatically is never a bad idea.

We did, however, also agree that the use of EvoCrash is very limited to the Java world. We are not sure whether it can be used in complex environments such as web containers, or to other object oriented languages that it wasn't designed to handle. If you're someone who comes from a university where Java isn't the main language taught (as is the case for one of our group members) or aren't in a position to be developing in Java, EvoCrash would be of little help without significant redesign.

Based on the previous paragraph, we suggest that further development of EvoCrash is performed continuously. Several suggestions mentioned above can be a start on how EvoCrash could be improved. Another recommendation that we propose is by making EvoCrash an open source project. By having a lot of programmers and a dedicated community, the development of EvoCrash could be accelerated.

3 Conclusion

During this assignment, we have come to understand more about automatic debugging and testing, and also how using EvoCrash could help in this regard. By having EvoCrash generate the test cases, testing and debugging can be done easier and faster. Unfortunately, in some cases, manual analysis can be a better solution than using EvoCrash. EvoCrash also has several limitations which can obstruct the process of testing and debugging.

References

[1] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, T. Zimmermann. "What makes a good bug report?". Proc. 16th ACM SIGSOFT Int. Symp. Found. Softw. Eng., pp. 308-318. 2008.

[2] M. Soltani, A. Panichella, A. van Deursen. *A Guided Genetic Algorithm for Automated Crash Reproduction*. TU Delft. 2016.

[3] CreationWiki. (2016, July 21). Genetic algorithm. Retrieved March 16, 2017, from http://creationwiki.org/Genetic_algorithm.

[4] The Netty Project. (n.d). Netty: Home. Retrieved March 15, 2017, from http://netty.io.

[5] The Netty Project. (n.d). netty/netty: Netty project - an event-driven asynchronous network application framework. Retrieved March 15, 2017, from https://github.com/netty/netty.