Evolutionary testing for crash reproduction

Soltani, Mozhan; Panichella, Annibale; Van Deursen, Arie

DOI
10.1145/2897010.2897015

Publication date
2016

Document Version
Accepted author manuscript

Published in
Proceedings - 9th International Workshop on Search-Based Software Testing, SBST 2016

Delft University of Technology
Software Engineering Research Group
Technical Report Series

Evolutionary Testing for Crash Reproduction

Mozhan Soltani, Annibale Panichella and Arie van Deursen

Report TUD-SERG-2016-013

TUD-SERG-2016-013

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Evolutionary Testing for Crash Reproduction

Mozhan Soltani
Delft University of Technology
The Netherlands
mozhan.soltani@gmail.com

Annibale Panichella
Delft University of Technology
The Netherlands
a.panichella@tudelft.nl

Arie van Deursen
Delft University of Technology
The Netherlands
Arie.vanDeursen@tudelft.nl

ABSTRACT

Manual crash reproduction is a labor-intensive and time-consuming task. Therefore, several solutions have been proposed in literature for automatic crash reproduction, including generating unit tests via symbolic execution and mutation analysis. However, various limitations adversely affect the capabilities of the existing solutions in covering a wider range of crashes because generating helpful tests that trigger specific execution paths is particularly challenging.

In this paper, we propose a new solution for automatic crash reproduction based on evolutionary unit test generation techniques. The proposed solution exploits crash data from collected stack traces to guide search-based algorithms toward the generation of unit test cases that can reproduce the original crashes. Results from our preliminary study on real crashes from Apache Commons libraries show that our solution can successfully reproduce crashes which are not reproducible by two other state-of-art techniques.

Keywords

Crash Reproduction, Genetic Algorithm, Search-Based Software Testing, Test Case Generation

1. INTRODUCTION

Debugging is the process of locating and fixing defects in software source code, which requires deep understanding about that code. Typically, the first step in debugging is to reproduce the software crash, which can be a non-trivial, labor-intensive and time-consuming task. Therefore, several automated techniques for crash reproduction have been proposed, including the use of core dumps to generate crash reproducible test cases [6, 9], record-replay approaches [1, 8, 10], post-failure approaches [2, 5], and approaches based on crash stack traces [3, 11].

However, the techniques mentioned above present some limitations which may adversely impact their capabilities in generating crash reproducible test cases. For example, core dumps are not always generated by software applications at the crash time, which may reduce the applicability of approaches which are merely based on using core dumps [6, 9]. Record-replay approaches apply dynamic mechanisms to monitor software executions, thus, leading to higher performance overhead [1, 8]. STAR [3] and MuCrash [11] are two novel approaches designed to deliver test cases that can reproduce target software crashes by relying on crash stack traces. STAR relies on backward symbolic execution to compute the crash triggering precondition [3]. However, inferring the initial condition of certain types of exceptions may be a complex task to accomplish by STAR. On the other hand, MuCrash applies mutation to update existing test cases to reproduce crashes [11]. While MuCrash can also reproduce certain crashes that STAR can reproduce, it fails to reproduce certain other crashes which are reproducible by STAR. As reported by Xuan et al. [11], the major reason for this failure is that reproducing those crashes requires frequent method calls which can not be recreated by directly applying mutation operators.

In this paper, we propose a novel approach for automatic crash reproduction through the usage of evolutionary search-based techniques, and crash stack traces. We implemented our solution as an extension of EvoSuite [4], and evaluated it on well-known crashes from the Apache Commons libraries. The main contributions of our paper can be summarized as follows:

- We provide a first formulation of stack-trace-based crash replication problem as a search-based problem;
- We define a novel fitness function to evaluate how close the generated test cases are to replicate the target crashes relying on stack traces only;
- We report the results of a preliminary study which shows the effectiveness of our solution compared to STAR and MuCrash.

The rest of the paper is structured as follows: Section 2 provides and overview on existing approaches on crash replication and provides background notions on search-based software testing. Section 3 presents our approach, while Section 4 describes our preliminary study. Finally, conclusions and future work are discussed in Section 5.

2. BACKGROUND AND RELATED WORK

In this section, we describe the two main related techniques for automatic crash reproduction, namely STAR [3] and MuCrash [11]. In addition, we provide an overview on search-based software testing and Genetic Algorithms.

SBST16, May 16-17, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2897010.2897015

Stack Trace based Crash Reproduction. STAR is an approach proposed by Chen and Kim [3] to identify crash triggering preconditions. It combines backward symbolic execution with a novel method sequence technique to create test cases that can produce test inputs to satisfy the identified crash triggering preconditions. The goal in STAR is to produce test cases that can crash at the same position and can generate stack traces as similar to target stack traces as possible. Chen and Kim also describe an empirical evaluation involving real crashes from three well-known open source projects. The results showed that STAR can successfully exploit 31 out of 52 crashes (59.6%) reported for the open source projects. Out of those exploitable crashes, 42.3% were successful reproduction of the reported crashes that revealed the crash triggering defect [3].

MuCrash is a more recent approach to automatic crash reproduction proposed by Xuan et al. [11]. It applies test case mutation to update existing test cases that can reproduce crashes, rather than generating new test cases which is the general strategy used in STAR [11]. Given a stack trace, MuCrash executes all the existing test cases on the program and selects test cases that cover the classes in the stack trace. Each selected test case produces a set of test case mutants, using a set of predefined mutation operators. The resulting test cases are executed on the program and the ones that can reproduce crashes are delivered to developers for debugging. MuCrash has been evaluated on 12 crashes reported for Apache Commons Collections library [11]. The result of the evaluation showed that MuCrash could successfully replicate 7 crashes out of 12.

We notice that none of the two approaches above provided an explicit formulation of the crash reproduction problem as a search-based problem, thus, they do not use any search-based algorithm to deal with crash reproduction. In this paper we conjecture that the usage of evolutionary test case generation technique can be effective in reproducing software crashes upon the definition of a specific fitness function, which is the key contribution of our paper.

Search-Based Software Testing (SBST) applies search-based optimization algorithms to seek solutions for various kinds of software testing problems. In the 1990s, there was a dramatic increase in work on metaheuristic search techniques, and since then, SBST has been applied in various testing problems [7], such as integration testing, functional testing, mutation testing, etc.

So far, the main metaheuristic search algorithms that have been applied in SBST include Hill Climbing, Simulated Annealing, and Genetic Algorithms [7].

Genetic Algorithms are closely related to the concept of survival of the fittest [7]. Solutions in the search space are referred to as "individuals" or "Chromosomes", which collectively form a "population". Since Genetic Algorithms maintain a population of solutions, multiple starting points are provided to the search with a corresponding larger sample of the search space (compared to local searches) [7]. The first population is randomly generated, and then iteratively recombined and mutated to evolve throughout subsequent iterations, called "generations". After a population is generated, best individuals are selected as parents for reproduction via crossover [7]. The selection is guided through using a fitness function, which is problem-specific. While fitter individuals are favored, a too strong bias towards them may result in their dominance in future generations [7]. Consecutively, the chance of premature convergence on a particular area of the search space may increase. This cyclic process of generating and selecting individuals goes on until either the Genetic Algorithm finds a solution or the allocated resources are consumed.

3. SEARCH-BASED CRASH REPLICATION

The key ingredient for a successful application of search-based techniques is the formulation of a proper fitness function to guide the search toward reaching the test goal. Then, such a function is optimized by search techniques, such as Genetic Algorithms, which use specific search operators to promote tests closer to cover the target goal and penalize tests with weak fitness values.

An optimal test case for crash reproduction has to crash at the same location as the original crash and produce stack traces as close to the original one as possible. Therefore, our fitness function has to exploit the information available in stack traces to measure the *closeness* of a test case to replicate the target crash. Usually a stack trace contains (i) the type of the exception thrown, and (ii) the list of methods being called at the time of the crash. For each called method, the stack trace also provides names and line numbers where the exception was generated. The first method in the trace is the root cause of the exception while the last one is the location where the exception was actually thrown. Therefore, the class containing the last method in the trace is the class to target for generating unit test, i.e., the class under test.

There are three main conditions that must hold to replicate a crash: (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. Therefore, our fitness function has to consider the three conditions above. Formally, let t be a given test to evaluate, we define the following fitness function:

$$f(t) = 3 \times d_s(t) + 2 \times d_{except}(t) + d_{trace}(t) \qquad (1)$$

where $d_s(t)$ denotes how far t is to execute the target statement, i.e., the location of the crash; $d_{except}(t) \in \{0, 1\}$ is a binary value indicating whether the target exception is thrown or not; and $d_{trace}(t)$ measures the distance between the generated stack trace (if any) and the expected trace.

For the line distance $d_s(t)$, we use the *approach level* and the *branch distance*, which are two well-known heuristics to guide the search for branch and statement coverage [7]. The *approach level* measures the distance (i.e., minimum number of control dependencies) between the path of the code executed by t and the target statement. The *branch distance* uses a set of well-established rules [7] to score how close t is to satisfy the branch condition for the branch on which the target statement is directly control dependent. For the trace distance $d_{trace}(t)$, we define a new distance function as reported below. Let $S^* = \{e_1^*, \ \ldots, e_n^*\}$ be the target stack trace to replicate, where $e_i^* = (C_1^*, m_1^*, l_1^*)$ is the i-th element in the trace composed by class name C_i^*, method name m_i^*, and line number l_i^*. Let $S = \{e_1, \ldots, e_k\}$ be the stack trace (if any) generated when executing the test t. We define the distance between the expected trace S^* and

Table 1: Real-world bugs used in our study

Bug ID	Version	Exception	Priority
ACC-4	2.0	NullPointerException	Major
ACC-28	2.0	NullPointerException	Major
ACC-35	2.1	UnsupportedOperationException	Major
ACC-48	3.1	IllegalArgumentException	Major
ACC-53	3.1	ArrayIndexOutOfBoundExceptions	Major
ACC-70	3.1	NullPointerException	Major
ACC-77	3.1	IllegalStateException	Major
ACC-104	3.1	ArrayIndexOutOfBoundsException	Major
ACC-331	3.2	NullPointerException	Minor
ACC-377	3.2	NullPointerException	Minor

Table 2: Detailed crash reproduction results

Bug ID	% Successful Replication	STAR [3]	MuCrash [11]
ACC-4	30/30	Yes	Yes
ACC-28	30/30	Yes	Yes
ACC-35	30/30	Yes	Yes
ACC-48	30/30	Yes	Yes
ACC-53	**28/30**	Yes	**No**
ACC-70	**30/30**	**No**	**No**
ACC-77	**30/30**	Yes	**No**
ACC-104	0/30	Yes	Yes
ACC-331	**10/30**	**No**	Yes
ACC-377	0/30	No	No

the actual trace S as follows:

$$D(S^*, S) = \sum_{i=1}^{\min\{k,n\}} \varphi\left(\text{diff}(e_i^*, e_i)\right) + \mid n - k \mid \quad (2)$$

where $\text{diff}(e_i^*, e_i)$ measures the distance between the two trace elements e_i^* and e_i in the traces S^* and S respectively; finally, $\varphi(x) \in [0, 1]$ is the widely used normalizing function $\varphi(x) = x/(x + 1)$ [7]. We say that two trace elements are equal if and only if they share the same trace components. Therefore, we define $\text{diff}(e_i^*, e_i)$ as follows:

$$\text{diff}(e_i^*, e_i) = \begin{cases} 3 & C_i^* \neq C_i \\ 2 & C_i^* = C_i \text{ and } m_i^* \neq m_i \\ \varphi\left(\mid l_i^* - l_i \mid\right) & \text{Otherwise} \end{cases} \quad (3)$$

Therefore, $\text{diff}(e_i^*, e_i)$ is equal to zero if and only if the two trace elements e_i^* and e_i share the same class name, method name and line number. Similarly, $D(S^*, S)$ in Equation 2 is zero if and only if the two traces S^* and S are equal, i.e., they share the same trace elements. Starting from the function in Equation 2, we define the trace distance $d_{trace}(t)$ as the normalized $D(S^*, S)$ function:

$$d_{trace}(t) = \varphi(D(S^*, S)) = D(S^*, S)/(D(S^*, S) + 1) \quad (4)$$

Consequently, our fitness function $f(t)$ assumes values within the interval $[0, 5]$, reaching a zero value if and only if the evaluated test t replicates the target crash.

4. PILOT STUDY

To evaluate the effectiveness of our solution for crash reproduction, we selected 10 bugs from the `Apache Commons Collections` library, a popular real world Java project with 25,000 lines of code. The selection of these bugs was not at random. These bugs have been used in the previous study on automatic crash reproduction when evaluating symbolic execution [3] and mutation analysis [11], which allows us to compare the results. The characteristics of the bugs, including type of exception and priority, are summarized in Table 1.

Prototype Tool We have implemented our fitness function in Evosuite [4], a popular unit test generation framework, widely used in research to generate unit tests targeting code coverage (e.g., statement coverage) or mutation score as testing criteria to maximize. Specifically, we defined a new coverage criterion (in addition to traditional coverage criteria already existing in Evosuite) consisting in maximizing the number of bugs (stack traces) to replicate. As search strategy, we used the traditional *one target* at a time approach, which consist of targeting one single bug (and the corresponding stack trace) at a time and running meta-heuristics, and genetic algorithms in particular, to optimize the fitness function. A value of zero for the fitness function means that the generated test case is able to replicate the targeted crash and, thus, it can be directly presented to developers for debugging purposes. The encoding schema is the same as used in Evosuite at the test case level. Thus, a chromosome is a randomly generated test case consisting of a variable sequence of method calls with random input. Random tests are then evolved as usual in genetic algorithms throughout selection, crossover and mutation operators. Pairs of existing tests (parents) are selected using the *tournament selection* according to their fitness function scores. New tests (offsprings) are then generated from their parents using a *single-point* crossover, which randomly exchange statements between the two parents. Finally, test cases are mutated by the *uniform mutation* that randomly adds, deletes, or changes statements with a given small probability. For all parameter values, we use the default setting in Evosuite since they provide good performance in traditional test case generation applications [4].

Experimental Procedure We applied our prototype tool to the selected crashes in order to generate test cases for reproducing them. In our pilot study, we set a maximum search budget of 2 minutes. Therefore, the search ended when either zero fitness was achieved or when the timeout was reached. Given the randomized nature of genetic algorithms, the search for each target bug/crash was repeated 30 times in order to verify whether crashes are constantly replicated or not. To assess whether the generated test cases are really helpful to fix the bugs -other than triggering the same stack trace- we performed a manual validation following the guidelines in [3, 11].

4.1 Results

Table 2 reports the number of times our prototype is able to replicate the target crashes (column 2). It also compares our crash results with two state-of-the-art methods, namely (i) STAR [3], and (ii) MuCrash [11]. The former uses symbolic execution while the latter is based on mutation analysis. As shown in Table 2, genetic algorithms allow us to reproduce 8 out of 10 crashes. Based on our manual check, all reproduced crashes are useful to fix the bug. For example, for bug ACC-70 our prototype generated within 10 seconds of search (on average) the test case depicted in Listing 1. According to our test, the crash is caused by a call to `previous()` when a `TreeListIterator` is instantiated with the first parameter (parent of the tree) set to `null`. Since inside the method `previous()` there is no check condition on such a parameter, a null pointer exception is generated. A simple fixing would consist of adding a check condition to

verify that the parent of the tree is not `null`.

```
public void test12() throws Throwable  {
TreeList treeList0 = new TreeList();
treeList0.add((Object) null);
TreeList.TreeListIterator l =
 new TreeList.TreeListIterator(treeList0, 732);
// Undeclared exception!
treeList_TreeListIterator0.previous();
}
```

Listing 1: Generated test for ACC-70.

For six bugs, our prototype constantly replicates the crash in all 30 independent runs. For ACC-53, there are only two out of 30 runs where a replication is not achieved. Finally, we find that the replication for ACC-331 is achieved only for some of the runs (33%). However, for such a class we notice that it requires specific method call sequence to be regenerated. Since our prototype does not invoke only methods and classes involved in the crash, it has minimal chance to call the right methods or to instantiate the correct objects. While this choice is useful to maintain diversity, it can have certain drawbacks. One natural extension would be to change the mutation operator in Evosuite in order to focus the search by using methods and objects of interest more frequently than others.

Comparing our results with those achieved by STAR [3] and MuCrash [11], we observe that there are bugs that can be reproduced by our technique and not by the alternative ones. In particular, for ACC-70 our prototype generates a test case (see Listing 1) which helps in replicating and fixing the bug. However, for such a bug both STAR and MuCrash are not able to generate useful tests. Crashes due to bugs ACC-53 and ACC-77 are replicable using our technique while they are not replicable using MuCrash [11]. Finally, STAR fails to reproduce ACC-331, which is instead covered by our prototype.

The results of our pilot study show the strength of evolutionary testing techniques, and evolutionary test case generation tools in particular, with respect to symbolic execution based on precondition analysis and mutation analysis. Theoretically speaking, evolutionary testing should imply a higher overhead of computing resources since tests have to be generated and executed. However, we notice that in our pilot study all crashes have been replicated in few (<10) seconds on average.

5. CONCLUSION

Manual crash reproduction is a labor-intensive and time-consuming task. Therefore, in this paper we propose a new search-based approach for generating unit test cases to replicate software crashes. Our solution uses a novel fitness function suitably, defined for crash reproduction and implemented as an extension of EvoSuite. By exploiting crash information from crash stack traces, the novel fitness function is used to guide test case generation algorithms toward the generation of tests directly consumable by developers to find the cause of the crash and fix the bugs.

This paper also reports the results of a preliminary study based on ten real crashes (and stack traces) related to bugs affecting the well-known Apache Commons libraries. The achieved results show that our solution is able to generate helpful tests for eight out of ten crashes. Moreover, our search-based solution is able to successfully replicate crashes not replicable using two state-of-the-art techniques for crash reproduction, namely STAR and MuCrash.

Considering the promising results achieved in this paper, the future work may have several possible directions. First, we plan to evaluate our search-based techniques on a wider sample of real crashes. We also plan to improve the fitness function and mutation operators in order to increase the likelihood of generating helpful test cases. Finally, a combination of genetic algorithms and symbolic execution is part of our future agenda.

6. REFERENCES

[1] S. Artzi, S. Kim, and M. D. Ernst. Recrash: Making software failures reproducible by preserving object states. In *ECOOP 2008–Object-Oriented Programming*, pages 542–565. Springer, 2008.

[2] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *PLDI*, pages 363–374. ACM, 2009.

[3] N. Chen and S. Kim. Star: Stack trace based automatic crash reproduction via symbolic execution. *IEEE Tr. on Sw. Eng.*, 41(2):198–220, 2015.

[4] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, Feb. 2013.

[5] W. Jin and A. Orso. Bugredux: reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering*, pages 474–484. IEEE Press, 2012.

[6] A. Leitner, A. Pretschner, S. Mori, B. Meyer, and M. Oriol. On the effectiveness of test extraction without overhead. In *International Conference on Software Testing Verification and Validation (ICST)*, pages 416–425. IEEE, 2009.

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

[8] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 284–295. IEEE Computer Society, 2005.

[9] J. Rossler, A. Zeller, G. Fraser, C. Zamfir, and G. Candea. Reconstructing core dumps. In *2013 IEEE Sixth Int. Conf. on Software Testing, Verification and Validation*, pages 114–123. IEEE, 2013.

[10] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for java. In *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering*, pages 114–123. ACM, 2005.

[11] J. Xuan, X. Xie, and M. Monperrus. Crash reproduction via test case mutation: Let existing test cases help. In *ESEC/FSE*, pages 910–913. ACM, 2015.

TUD-SERG-2016-013
ISSN 1872-5392