Search Based Software Engineering: How Crash Replication Helps in Finding Bugs

Kwadjo Anobaah Nyante, Liza Everon, Tugce Arican

Electrical Engineering, Mathematics and Computer Science (EEMCS/EWI) Faculty: University of Twente, The Netherlands

March 19, 2017

Abstract

In recent times, Search Based Software Engineering(SBSE) has become a powerful tool in the arsenal of many software developers and testers. This branch of software engineering enables its users to convert a software engineering problem into a computational search problem that can be tackled with meta-heuristic algorithms, thus, finding a "good" solution to the computationally expensive problem in a "reasonable" amount of time. Today SBSE is applied to many aspects of the Software Development Life Cycle such as: Requirements Engineering, debugging and maintenance, Optimization and testing, etc. In this paper, we explore how SBSE can be applied to software testing, specifically crash replication. Crash replication simplifies crash debugging by automatically reproducing real-world software failures. The resulting crash stack traces are then analyzed (post-failure analysis) by the developers to determine the root causes of the problem. Recent advances in this area still find real-world crash replication a tough nut to crack due to factors such as: environment dependencies, path explosion and time complexity. This paper explores how EvoSuite*(a novel test generation tool)* and EvoCrash*(a tool based on EvoSuite which uses a Guided Genetic Algorithm and a novel fitness function)* can be employed to simplify the process of finding bugs in typical software applications, highlighting the core strengths and possible improvements that can be made to these tools to enhance their efficiency.

Keywords: Computational Search, Meta-heuristic, Crash Replication, Guided Genetic Algorithm, Fitness Function

1 Introduction

Many activities in Software Engineering can be stated as computational problems. In a computational problem, we are given an input that, without loss of generality, we assume to be encoded over the alphabet **{0,1}**, and we want to return as output a solution satisfying some property: a computational problem is then described by the property that the output has to satisfy given the input. The following are some types of computational problems: decision problems, search problems, optimization problems, and counting problems. These problems can be solved using three types of algorithms: Exact Algorithms, Heuristic Algorithms or Meta-heuristic Algorithms.
Exact algorithms use techniques of operations research such as linear programming or dynamic programming. Although exact algorithms are guaranteed to find an optimal solution in a finite amount of time, this finite amount of time may increase exponentially with respect to the dimensions of the problem, especially for NP-hard problems.
One solution to NP-hard problems is to use Heuristic Algorithms. Although there is no guarantee that a heuristic algorithm will find an optimal solution, there is a good chance that it will find a near optimal solution in a "reasonable" amount of time.
But alas, heuristic algorithms are problem specific and thus provide problem-dependent solutions. Meta-heuristic algorithms are alternatives which are high-level problem independent algorithmic frameworks that provide guidelines or strategies to develop heuristic optimization algorithms.
Thus, meta-heuristic algorithms provide a more general purpose approach to solving optimization problems.

SBSE is a sub-field of Software Engineering that seeks to reformulate Software Engineering problems as "search problems". This is not to be confused with textual or hyper-textual searching but rather computational searching. In SBSE, an optimal / near optimal solution is sought from a defined search space consisting of possible solutions. The search is guided by a fitness function which basically distinguishes between better and worse solutions. The fitness function, therefore, acts as a metric for determining the quality of discovered solutions. Meta-heuristic algorithms are mostly used for the computational search because the search space is typically too large to be explored exhaustively.

In the light of the above, it is crystal clear that the importance of SBSE cannot be underestimated in solving Software Engineering problems. This is especially true in the field of test data generation. Test data generation, an important part of software testing, is the process of creating a set of data for testing the adequacy of new or revised software applications.The use of dynamic memory allocation in most of the code written in industry is the most severe problem that the Test Data Generators face as the usage of the software then becomes highly unpredictable. Due to this it becomes harder to anticipate the paths that the program could take making it nearly impossible for the Test Data Generators to generate exhaustive Test Data. SBSE with the help of meta-heuristic algorithms make it easy to generate test data. Sometimes additional code in the target programming language is generated with the test solution to make it directly executable. This is called test code generation.

Lastly, by easily generating test cases, SBSE reduces the manual effort by developers in crash reproduction by automatically generating new test cases to trigger crashes.

2 Automatic Test Generation with EvoSuite

"What is the smallest set of test cases that cover all branches of my program?". This is a typical question software testers / developers want answered. The answer to this question is not always straight forward especially when test cases are automatically generated. In this section, we discuss the core concepts regarding how EvoSuite automatically generates test cases to answer these type of questions.

There are various techniques used to search for test cases in a given search space. A naive way is to do random searching, thus, iteratively picking a random test solution in the search space until a good solution is found. However, it should be noted that since this kind of search is not guided (no fitness function used), it usually fails to find globally optimal solutions. Only a small local area of the search space is covered (low coverage). Other methods such as Hill Climbing, Simulated annealing and Evolutionary algorithms are better alternatives. These methods employ a fitness function to guide the search thus increasing its coverage. No matter what test generation / search method is chosen, its efficiency depends heavily on how fast it finds globally optimal solutions and the area of the search space covered (coverage). An ideal method should cover near 100% of the searh area(high coverage) and also find an optimal solution within a "reasonably" short amount of time.

EvoSuite is a search-based tool that automatically generates unit tests for Java classes, targeting branch coverage and other coverage criteria (E.g mutation). Instead of generating one test at a time for each test goal, EvoSuite generates a suite of tests attempting to satisfy all goals simultaneously. There is evidence that the overall coverage achieved with this approach is superior to that of targeting individual coverage goals[5]. It works at the Java bytecode level, i.e., it does not require source code. It is fully automated and requires no manually written test drivers or parameterized unit tests. For example, when EvoSuite is used from its Eclipse and IntelliJ plugins, a user just needs to select a class, and tests are generated with a mouse-click.

The Test Data Generator follows the following steps: Program Control Flow Graph Construction, Path Selection and Test Generation. First the test constructs a Program Control Flow Graph. A Control Flow Graph of a program is a representation, using graph notation, of all paths that might be traversed through a program during its execution. Next a path is selected based on the coverage criterion chosen. By default the criterion for path selection is branch coverage. Generally a branch is considered infeasible if there are no program inputs that execute that branch. Lastly, tests are generated from searching the search space guided by a fitness function. An optimal test is one which covers all feasible branches/methods and is minimal in the total number of statements. These tests are assigned a relatively higher fitness value because there are no other test suites that exist which cover all feasible paths and have a lower total number of statements.

EvoSuite uses an evolutionary approach to derive test suites in the test generation stage: A genetic algorithm evolves candidate individuals (chromosomes) using operators inspired by natural evolution (e.g., selection, crossover and mutation), such that iteratively better solutions with respect to the optimization target (e.g., branch coverage) are produced. For details on this test generation approach we refer to [5].

To find defects in software, one needs both test cases that execute the software systematically and oracles to assess the correctness of the observed behavior when running the test cases. For this every test case EvoSuite generates comes with assertions for classes written in Java code. This is done using a novel hybrid approach that generates and optimizes whole test suites towards satisfying the chosen coverage criterion, for each test suite, EvoSuite suggests possible oracles by adding small and effective sets of assertions (assertions are selected by mutation analysis) that concisely summarize the current behavior. 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. As the generated unit tests are meant for human consumption [8], EvoSuite applies various post-processing steps to improve readability (e.g., minimizing). For more details on the tool and its abilities we refer to [9,10].

3 Evocrash

EvoCrash is an automated crash reproduction tool which uses EvoSuite for creating JUnit tests. EvoCrash aims to generate a stack trace as close to the original stack trace as possible. In order to achieve this criteria, it uses crash stack traces to create test cases. EvoCrash suggests a novel fitness function and Guided Genetic Algorithm for test generation.

3.1 Novel Fitness Function

The fitness criterion is calculated according to three main components: the line where the exception is originally thrown, the line where the exception is actually thrown, and the similarity between original and generated stack traces. The fitness function value of a test **t** can be calculated as follows.

$$f(t) = 3 * d_{\mathrm{s}}(t) + 2 * d_{\mathrm{except}}(t) + d_{\mathrm{trace}}(t) \tag{1}$$

$d_s(t)$ denotes how far t and targeted frame are from each other, $d_{except}(t)$ is a binary value that indicates whether the exception is thrown or not, and $d_{trace}(t)$ measures the distance between generated and targeted stack traces.

While calculating ds(t), heuristic approaches such as approach level and branch distance are used. Distance between generated and targeted stack traces can be basically (novel distance) calculated according to following formula.

$$D(S^*, S) = \sum_{i=1}^{n} min\left\{diff({e_{\mathrm{i}}}^*, e_{\mathrm{j}}) : e_{\mathrm{j}} \,\epsilon\, S\right\} \tag{2}$$

S* denotes the target stack trace to replicate, while S represents generated one. e_i and e_j denote ith and jth element in stack traces of S* and S respectively.

This novel distance measures the difference between element ${e_i}^*$ in S* and its closest element e_j in S. Then $d_{trace}(t)$ can be calculated as the normalized D(S*,S) function as follows.

$$d_{\mathrm{trace}}(t) = \varphi(D(S*, S)) = D(S*, S)/(D(S*, S) + 1) \tag{3}$$

3.2 Guided Genetic Algorithm

EvoCrash uses a novel genetic algorithm called the Guided Genetic Algorithm (GGA). GGA is quite similar to the regular genetic algorithm used in Evosuite except with a few improvements on initial population generation, crossover operator and mutation operator. Furthermore, GGA gives higher priority to the methods involved in target failure; hence, increasing the overall probability of triggering the target crash. The algorithm starts by generating an initial population of random tests. Then, guided crossover and mutation operators are applied to the population. At the end of the algorithm a fitted test is obtained.

3.3 Initial Population Generation

EvoCrash uses an improved Initial Population Generation Algorithm. It randomly selects statements and gives precedence to the methods involved in the stack trace. This approach increases the overall probability to trigger a target crash. In each iteration, the algorithm creates a test case **t** with a random size of statements. These statements are randomly inserted from either target methods, $\mathbf{M_{crash}}$, or member classes **C**(Class Under Test). In the first iteration, crash methods are inserted with a low probability, **1 / size** where size denotes the number of methods in $\mathbf{M_{crash}}$. This probability will automatically increase when no target method is inserted during the loop. Inserting methods also requires additional elements such as primitive variables, objects, or class constructors. A method named **INSERT_METHOD_CALL** instantiates these elements.

3.4 Guided Crossover Operator

EvoCrash applies a single-point crossover operator in order to expand the searching space. However, single-point crossover may move all target method calls to one offspring, while other offspring loses its target method calls. A novel approach called Guided Crossover checks the offspring after crossover operator is applied. If one of the offspring(e.g. O_1) loses its target method calls, the algorithm reverts the changes and defines the offspring(O_1) as an identical copy of its parent(P_1). Moreover, moving target methods from one offspring to another may create non well-formed tests because of the required elements such as class constructors, objects, etc. A procedure called *correction* is responsible of inserting all required objects, primitive types, etc.

3.5 Guided Mutation Operator

Mutation is applied for inserting, changing, or deleting statements from tests. Mutation is applied with a low probability, **1/n**. Inserting statements means adding a new method at a random point in the test **t**. This procedure also includes the initialization of primitive variables, declaring objects, etc. During the changing operation, mutation has to handle two conditions: changing a statement declared with primitive variable and changing a method or construction call. In the first condition, the operator replaces the primitive variable with a random value. If the statement is a method or constructor call, then the operator replaces the call with another method or constructor call having the same return type with same parameter types. The Deleting operation randomly removes statements from test **t**. Removing a statement also requires removing related objects, variables, etc. During mutation, the test may lose its target method calls. In this case, the algorithm tries to insert one or more target method calls. If this fails, the mutation process is terminated.

3.6 Post Processing

Statements are randomly inserted during this stage. Although the final test can be directly used by developer, it may contain some statements which play no vital role in crash reproduction. In order to make tests more concise and understandable, GGA applies a post processing step based on test optimization routines defined in EvoSuite, namely *test minimization* and *value minimization*. Test minimization is based on the Greedy algorithm (basically removes all statement that do not affect the final fitness value). Value minimization shortens the identified numbers and simplifies the randomly generated strings.

3.7 EvoSuite vs EvoCrash

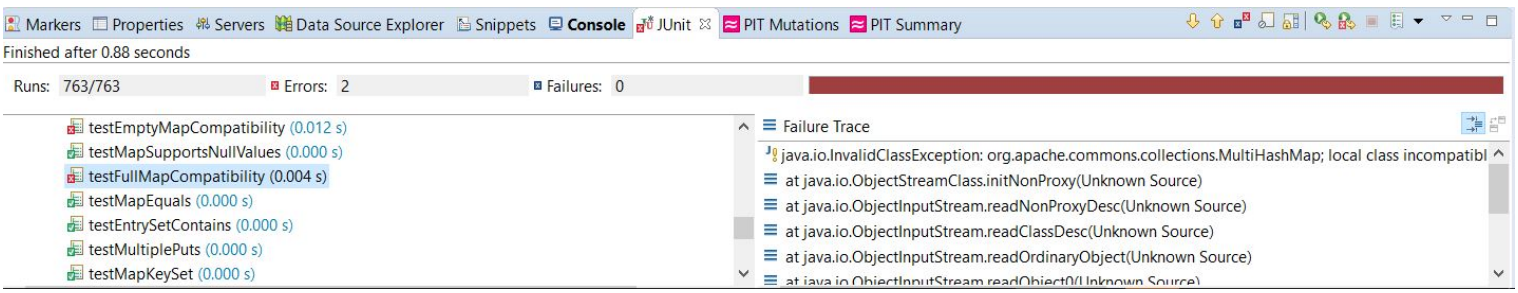
At first glance, EvoSuite and EvoCrash may seem very similar, however, on a closer look, the major differences can be easily noticed.

Item	EvoCrash	EvoSuite
Initial Population	gives higher importance to the methods involves in crash stack frame while generating initial population	gets a set of random tests suites to generate initial population
Fitness Function	compares stack frames element wise	makes comparison prefix wise.
Evolutionary Search	uses Guided Genetic Algorithm: modifications are in new crossover and mutation operators. Modifications check offspring whether they lose their target method calls.	uses genetic algorithm: it also uses crossover and mutation operators; yet, it does not check whether the offspring have a target method call.
Test Suite Size	Test suites are smaller in size because of the target method criteria.	It does not have that criteria, thus test suites are larger.

Table 1: This table shows the differences between EvoCrash and EvoSuite

4 Debugging With EvoCrash

To test the effectiveness of EvoCrash, we tested the tool on **apache-commons-collection-2.0** (a collection of open-source reusable Java components) which contains various programming elements such as maps, lists, etc. The library already comes with various unit tests, therefore, we begun by running these test using **JUnit 4.0** in **Eclipse Neon.2 Release (4.6.2) build 20161208-0600.** JUnit run a total of **763** tests in **0.88 seconds**. Out of these tests, 2 errors were found which are the *testEmptyMapCompatibility()* and *testFullMapCompatbity()* tests.



Next, we run EvoCrash in the command line pointing to the the file containing the apache commons library. The following code was used. For details on how to use EvoCrash in the command line see [14].

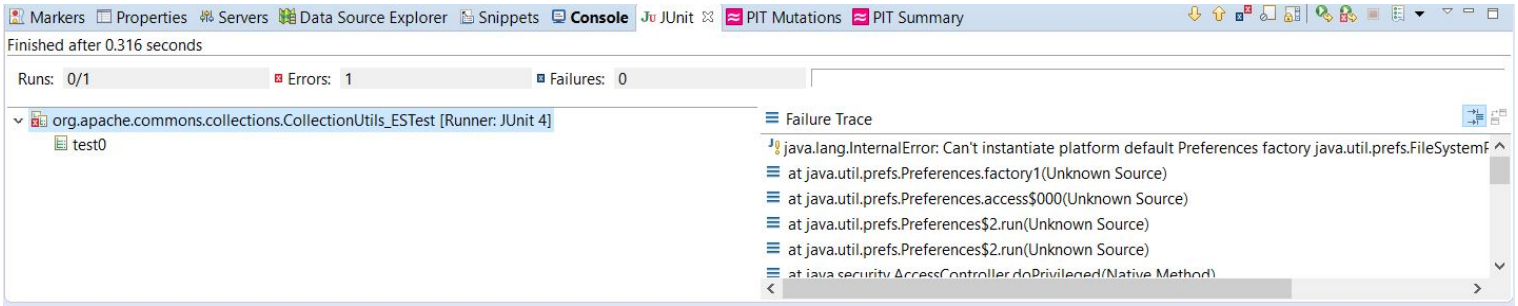
```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\HARPAZO>java -jar evocrash-master-1.0.0-jar-with-all-dependencies.jar -projectCP targets\ACC-bins\ACC-2.0\commo
ns-collections-2.0.jar -class "org.apache.commons.collections.CollectionUtils" -Dtest_dir=GGA_test -DEXP=stacks\ACC\ACC-
4\ACC-4.log -generateTests -Dcriterion=CRASH -Dtarget_frame=1
```

One test was successfully generated which was written to the current directory in a folder named *GGA_test*.

```
INFO  evo_logger - * Going to analyze the coverage criteria
INFO  evo_logger - * Coverage analysis for criterion CRASH
INFO  evo_logger - * Passed the target level 1, so stopping the parser!
INFO  evo_logger - * EvoCrash: The target call is public!
INFO  evo_logger - * Coverage of criterion CRASH: 100%
INFO  evo_logger - * Total number of goals: 1
INFO  evo_logger - * Number of covered goals: 1
INFO  evo_logger - * Generated 1 tests with total length 7
INFO  evo_logger - * Resulting test suite's coverage: 100%
INFO  evo_logger - * Generating assertions
INFO  evo_logger - * Resulting test suite's mutation score: 0%
INFO  evo_logger - * Compiling and checking tests
```

On exploring the directory, we noticed two java files were created: *CollectionUtils_ESTest.java* and *CollectionUtils_ESTest_scaffoldin* *CollectionUtils_ESTest.java* contains the test, which is test0().On running the test case created (*test0()*) with JUnit, we discovered that an actual error had been produced which was not catered for in the already existing tests. There was also an extensive stack trace to help in debugging the error. The results are shown below.



The results show the importance of using automated testing tools, since writing manual tests cannot possibly cover all possible crash scenarios especially for such a large program.

5 How EvoCrash Helped

EvoCrash aims to create test cases as close to the original crash as possible. Generated test cases point to the original crash point i.e original class, method, and statement. Thus, in the debugging exercises performed, we noticed that first of all, there was no need to write a manual test. Also debugging took less time since the test was generated in a relatively short time with an extensive and comprehensive crash information on the stack trace. It is also important to know that apart from tracing bugs, EvoCrash could also prove useful in Quality Assurance.

After taking a look at the track record of EvoCrash (ability to replicate 82% of 50 real-world crashes) and getting a better understanding about how it actually works, our team came to a consensus, that the tool could prove useful in everyday programming situation and we decided to adopt it for testing and developing high quality software applications.

6 Conclusion and Possible Improvements

In spite of the several advantages provided by EvoCrash, we believe that there is always room for improvement. Paramount on the list of possible improvements is in the simulation of the programming environment. We noticed that after the test

code is generated, it sometimes require human intervention to execute the program. Thus, there is still some dependency on external files. Lastly, EvoCrash faces issues when generating tests for programs with multiple nested if statements. We believe that this means the program encounters an uncharacteristically large number of paths (Path Explosion). Therefore, we suggest that branch coverage alone may not be the best option in this scenario. EvoCrash should be extended to be able to use multiple coverage criteria simultaneously without overly increasing test generation time.

References

[1]Jifeng Xuan, Xiaoyuan Xie and Martin Monperrus "Crash reproduction via test case mutation: let existing test cases help"

[2[5]G. Fraser and A. Arcuri, "Whole Test Suite Generation", IEEE Transactions on Software Engineering, vol. 39, no. 2, pp. 276-291, 2013.

[3]Mark Harman1, Phil McMinn2, Jereson Teixeira de Souza3, and Shin Yoo1 "Search Based Software Engineering: Techniques, Taxonomy, Tutorial"

[4]Stanford University CS254: Computational Complexity Handout 2 Luca Trevisan March 31, 2010

[5] G. Fraser and A. Arcuri, EvoSuite at the SBST 2016 Tool Competition, in 9th International Workshop on Search-Based Software Testing (SBST16) at ICSE16, 2016, pp. 33-36.

[6] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, Does automated white-box test generation really help software testers? in Proceedings of the 2013 International Symposium on Software Testing and Analysis. ACM, 2013, pp. 291301

[7] G. Fraser and A. Zeller, Mutation-driven generation of unit tests and oracles, IEEE Transactions on Software Engineering (TSE), vol. 28, no. 2, pp. 278292, 2012.

[8]G. Fraser and A. Arcuri, "Whole Test Suite Generation", IEEE Transactions on Software Engineering, vol. 39, no. 2, pp. 276-291, 2013.

[9] G. Fraser and A. Arcuri, EvoSuite: Automatic test suite generation for object-oriented software. in ACM Symposium on the Foundations of Software Engineering (FSE), 2011, pp. 416419.

[10] , EvoSuite: On the challenges of test case generation in the real world (tool paper), in IEEE Int. Conference on Software Testing, Verication and Validation (ICST), 2013.

[11] Mozhan Soltani, Annibale Panichella and Arie van Deursen, "Automated Crash Reproduction"

[12] 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.

[13]Sai Charan Taj Chitirala, Comparing the effectiveness of automated test generation tools "EVOSUITE" AND "Tplaus", July 2015.

[14]https://www.evosuite.org "EVOSUITE OFFICIAL WEBSITE".