A Survey on Test Generation Techniques used in Evosuite Automatic Unit Test Generation Tool

V. Ravi Kanth
M. Tech. Student: dept. of CSE
JNTUCEA, Anantapur

***Abstract*— Generating test cases automatically rather than doing it manually reduces the effort and time of testers. Evosuite, a result of research work by Arcuri et al, is such a tool which generates test cases with assertions for java programs and achieves high code coverage with the generated test cases. Evosuite had undergone so many changes continuously throughout its development for achieving better results; meanwhile it implemented several techniques for test case generation. This paper aims to present a comprehensive study of those techniques which are involved in the development of Evosuite.**

***Keywords*— Testcase; automated test generation; environment**

I. INTRODUCTION

In software testing test case generation is the most labor-intensive work for testing people as it requires more effort to write and execute them than other tasks. However, it is a very important activity hence it evaluates the software system to find whether it meets its intended requirements or not. Unit testing, the initial phase in entire software testing is performed on the software system units which cannot be decomposed further. Unit testing will ensure that all the individual units of software system are performing their functionality or not. Unit testing will be performed by writing test cases and executing them. These test cases are nothing but the calls to the specifications of the unit under test. A test suite is the combination of these sorts of test cases where each test case will have a different goal. To perform this whole process manually requires lot of effort from testers. This had given motivation for developing automatic test case generation techniques.

Evosuite is a tool which generates unit test suites automatically for code written in java. It applies a novel hybrid approach [1] for generating test cases for java code while unit testing performed. Since Evosuite is fully automated, there is no need for any tester to do manual testing; instead tester just needs to select the class to be tested and then the test cases are generated automatically with a mouse-click.

Evosuite uses genetic algorithm for generating tests and then applies some post processing techniques for adding assertions to exhibit the behavior of the tested units. Arcuri et al has made continuous research on this research prototype for achieving high performance levels in automatic test generation. As part of the development, Arcuri et al tried several variations of techniques with various development criterions like mutation analysis, coverage criteria and triggering undeclared exceptions …etc. And there are many techniques are proposed with Evosuite for addressing particular problem.

Evosuite was evaluated against a large group of software systems to prove its practical value. This large base contains 100 open source projects, 10 most popular open source projects according to Source Forge website, 7 industrial projects and 11 projects which are generated automatically [2]. The reason behind selecting such large code base is to evaluate Evosuite and is to ensure that there are no threats to its external validity. And this evaluation also tries to prove the fact that Evosuite will not only work well for the selected projects but also its performance can scale up to complexity of the real systems. These large set of projects used for empirical set up are referred as SF110 corpus and contains 23,886 java classes which are big enough for setting a large empirical assessment. In the process of improving the performance, reaching up to the expected qualities and getting rid of some drawback Arcuri et al have used number of advanced techniques to enhance the ability of Evosuite tool. For each enhancement the tool is evaluated against SF110 to prove that tool with extensions was making progress towards efficient performance or not.

Evosuite participated in unit testing competitions held at SBST 2013[3], SBST 2015 [4] and SBST 2016 [5], where the tool was applied for getting results in tool competition at the International Workshop on Search-Based Software Testing and achieved ranks first, second and first with respective overall scores of 156.95, 190.6 and 1126.7 in the respective years. Thus Evosuite proved to be the best unit test generation tools for java code at higher software testing environmental competitions. Hence these performances emphasis on the significance of the techniques used for enhancing Evosuite tool and notifies that these methods are worth to software community. In the following section this paper presents a detailed explanation about methods and techniques used in developing Evosuite unit test generation tool. Thus the following techniques are considered in this paper which includes:

1. Mutation Analysis
2. Combining search based and constraint based testing
3. Generation of parameterized unit tests
4. Handling environmental dependencies
5. Whole test suite generation

It is worth noting that there are many other techniques that are used in development of Evosuite but are not covered in this paper. For keeping the paper in reasonable size, we limited our selection of topics to above important techniques. Hence, after brief representation of these techniques in the next section, this paper will review the techniques.

II. TECHNIQUES USED IN DEVELOPING EVOSUITE

Evosuite is an automatic unit test generation tool for java code, which uses an evolutionary approach to generate the test suites. It works at byte code level hence there is no need of source code to be available for testing and it is fully automated i.e. whenever Evosuite used along with its eclipse and Intellij plug-ins, the user just needs to select Unit Under Test and the tests are generated with just a mouse click. It can be also used on command line by using following syntax.

java –jar evosuite.jar <target> [options]

Here target can be either name of a class to be tested or it can be a jar file. There are many options one can use here but the most important option is *projectCP*, which sets the class path for test generation.

A. Mutation analysis

Mutation analysis [6] is a process of seeding mutants which are known as artificial defects driven into programs for detecting purpose. Evosuite uses an automated approach to evolve test suites which detect these mutants. As part of mutation analysis Evosuite present an approach called as u-test that uses mutations rather than structural properties for coverage criterion. This u-test approach allows tester for getting guidance for *what to test for* and also *where to test.* This allows Evosuite to generate test oracles effectively that gives rise to automation of test generation. By generating oracles along with test cases µtest also makes it simple for checking if the assertions which are generated are valid. In case the generated assertion is not valid then it is obvious that the bug has been found. µtest uses following approaches for effective mutation analysis.

- Uses a genetic algorithm which effectively detects mutants by breed method call sequences.
- Generating minimum number of mutants when compared to earlier executions of test cases and their undetected mutants.
- Considering changes to the state of the program as mutants that causes maximal impact on the program behavior and thus reducing assessment effort.
- Optimizing test cases by removing all irrelevant objectives and hence reducing the long sequences of test cases which leads to shorter test cases that are easier to understand.

The mutation analysis approach followed by Evosuite provides improvement when compared to structural coverage is that it not only shows where to test but also assists in selecting what should be verified for. Evosuite applied this mutation analysis that results test suites which are significantly better than manually written ones when it comes to finding defects.

B. Combining search based and constraint based testing

Modern automated test generators are based on meta-heuristic search techniques or constraint based solvers [7]. These approaches have their advantages and also disadvantages respectively. For example search based testing finds inputs by applying relevant algorithms which generates suitable tests and this approach scales good for any code with any criterion only when heuristic approach provides needed guidance. Whereas constraint based solvers which are independent of heuristic methods will use dynamic symbolic execution to enhance the efficiency of constrains to be solved. There is a scope for getting results whenever these two methods are combined that contains great potential for achieving better results rather than applying individually. Evosuite intrinsically combines both approaches although it appears as if it is only search based approach from top level.

The procedure in which the combination of these two approaches will work can be described as; a search based approach generates a candidate solutions population. A special mutant operator will be added for avoiding the scenario where the search might just get stuck. This operator negates the path conditions which represents the execution of a candidate solution. And then constraint solver come into play where it produces input which was mutated and guaranteed to get diverted from original execution path. And thus raising the efficiency of search based testing which implicitly improves the constraint solver approach. Eventually the combination results two advantages first search based approach uses DSE to improve exploration and to overcome problematic areas in the search landscape and the second is DSE uses search based approach as a search wrapper to control the conditions where constraint solver fails.

C. Generation of parameterized unit tests

Often automatic test generation tools focus on providing tests that covers the programs behavior without providing any oracles as it is considered the duty of tester or user to find what the test does and how to decide the correctness of the resulted behavior. This is a difficult task for testers because it needs to verify the output of the test and understanding what a test does. To overcome this issue Fraser et al proposed a technique for generating parameterized unit tests [8] in which symbolic pre and post conditions are presented for characterizing the test input and the test result. This approach uses test generation and mutation to systematically generalizing pre and post conditions. This technique presents a novel approach that explores both pre and post conditions which are embedded in tests in the form of parameterized unit tests.

This approach will be implemented by separating test code from test input which provides benefit of dropping large amount of generated code and was replaced by symbolic parameters, consequently reducing the size of the test cases. And the next one was to filter the irrelevant behavior from the important one by variation: which seeds the defects by mutations that changes some post conditions. This approach suggests oracles that are effective for finding defects by identifying relevant pre conditions and also filter out overlying post conditions. This approach also converts a concrete method sequence to a more expressive test which requires fewer computation steps and achieves high coverage when compared to normal concrete tests.

D. Handling environmental dependencies by mocking approach

When generating test cases for object oriented software like java, the automatic test generation tools mainly faces two kinds of problems: one is generated test cases may not cover

the code which is under test hence its execution depends on its environment like File system, where the execution depends on the contents of a file. Second, even if the code to be tested covered by tests, there is no guarantee that these tests will be success when they executed later on the same code resulting unstable tests. In object-oriented software systems the Class Under Test often embedded in an interwoven environment which surrounded by number of dependencies. These environmental dependencies may involve operating system, file system, data bases and networking interactions.

To overcome these problems caused by environmental dependencies [9] Evosuite uses an approach based on mocking. Here mocking referred as replacing the original classes with the mocked versions. In java, a class interacts with its environment by using library classes such as java.file for interacting with file systems, java.sql for interacting with data bases, java.io for interacting with input/output streams and java.net for networking interactions. Hence Evosuite tries to bring the environment under control instead of user classes by mocking library classes. As Evosuite works at byte code level it uses byte code instrumentation to mock these interactions based Java Agent technology. Java agent is responsible for instrumenting and altering the byte code of the classes loaded. It is done by calling methods on the class InstrumentingAgent. For example the methods like *InstrumentingAgent.intialize()* will set up the agent, *activate()* and *deactivate()* will be used for start and stop byte code instrumentation respectively.

Whenever the test case executes the CUT, the original interaction was redirected towards mock environment which was controlled by Evosuite. For example when the CUT interacts with file system, Evosuite will instrument the CUT to use virtual file system rather than original one. This mocking approach was implemented by overriding standard library classes to perform as intended by Evosuite to get control over the environment. This technique improves the coverage of the Evosuite when it was evaluated against SF110 corpus.

E. Whole test suite generation

A common application of search based test generation tools is generating test cases for achieving various coverage criterions like branch, line and mutants...etc. Instead of generating individual test cases for achieving these coverage goals one at a time, whole test suite generation optimizes entire test suites towards satisfying all goals at the same time. Empirical evidence suggests that this whole test suite generation [10] obtains better results than individual test cases. But there are some questions to be answered like a) whether the results generalize beyond branch coverage b) whether the whole test suite generation only optimized by targeting coverage goals not already covered.

The fitness function guides to cover all goals according to the whole test suite generation which follows the process as instead of searching for a single test for each coverage goal in sequence, the search will proceed to a set of coverage goals at the same time. To achieve this Arcuri et al proposed a technique where the use an archive can lead to better results, but it may have some side effects, as the use of an archive would require special search operators. Designing these operators will require further research. Moreover the incorporation of the archive as part of the whole test suite generation gives raise to the question of whether it can still be regarded as evolution of test suites.

III. CONCLUSIONS

This paper presents a survey of some of the prominent techniques used by Evosuite automatic test generation tool as part of its development. Each technique presented in this paper carries its significance for the enhancement of the tool and not only had they contributed to the development of Evosuite but also to the environment of automatic test generation which is the reason why those are selected for study in this paper. Every time Evosuite implemented a technique it was evaluated against SF110 corpus to prove that there are no threats to its external validity. Although the techniques presented in this paper achieves higher coverage with test case generation there are still some issues with seeding strategies which can further improve the Evosuite tools performance and those are left for future work.

REFERENCES

[1] G. Fraser and A. Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In ACM Symposium on the Foundations of Software Engineering (FSE), pages 416–419, 2011.

[2] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 24, no. 2, p. 8, 2014.

[3] , "Evosuite at the SBST 2013 tool competition," in International Workshop on Search-Based Software Testing (SBST), 2013, pp. 406–409.

[4] U. Rueda, T. E. Vos, and I. Prasetya, "Unit testing tool competition - round three," in International Workshop on Search-Based Software Testing (SBST), 2015.

[5] Gordon Fraser, A.Arcuri, "Evosuite at the SBST 2016 Tool Competetion" in International Workshop on Search-Based Software Testing (SBST), 2016

[6] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In ISSTA'10: Proceedings of the ACM International Symposium on Software Testing and Analysis, pages 147–158. ACM, 2010.

[7] A. Arcuri and X. Yao. Search based software testing of object-oriented containers. Information Sciences, 178(15):3075–3095, 2008.

[8] N.Tillmann and W. Schulte, "Parameterized unit tests," in ACM Symposium on the Foundations of Software Engineering (FSE), ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 253–262.

[9] A.Arcuri, G.Fraser, and J.P.Galeotti, "Automated unit test generation for classes with environmental dependencies", *in IEEE/ACM int. Conference on Automated Software Engineering(ASE). ACM,2014, pp.79-90.*

[10] G. Fraser and A. Arcuri. "Evolutionary generation of whole test suites." In International Conference On Quality Software (QSIC), 2011.