International Journal of Innovation and Scientific Research
ISSN 2351-8014 Vol. 2 No. 2 Jun. 2014, pp. 296-300
© 2014 Innovative Space of Scientific Research Journals
http://www.ijisr.issr-journals.org/

# Test Suite Generation using Genetic Algorithm and Evolutionary Techniques with Dynamically Evolving Test Cases

***K.Ramesh and P.Manivannan***

Assistant Professor, Department of Information Technology,
V.R.S. College of Engineering & Technology,
Arasur, Villupuram Dt, Tamilnadu, India

Copyright © 2014 ISSR Journals. This is an open access article distributed under the ***Creative Commons Attribution License***, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

**ABSTRACT:** Test oracles are mostly written manually by the user once test data are generated. This is because of the fact that each bug requires a different input and different data. This is a very difficult and time consuming task as the tester must produce quick and meaningful test cases for testing. However, the major problem of this approach is that they can cover only one goal at a time. They are dependent on one another and sometimes are not replicable. This paper presents a new approach by which test oracles are generated automatically by the usage of evolutionary algorithm. This method has successfully allowed bug identification in thousands of classes and it is quick to use.

**KEYWORDS:** Search-Based Software Engineering, Length, Branch Coverage, Genetic Algorithm, Evolutionary Algorithm.

## 1 INTRODUCTION

The goal of automatic test data generation in unit testing is to generate test data that can satisfy a given test coverage criterion. A common criterion for unit testing is branch coverage, i.e. the test set should execute every branch in the program unit under test [11]. Branch coverage is also a common criterion for assessing research in automatic test data generation and is the criterion adopted for the empirical investigation reported. A software test consists of an input that executes the program and a definition of the expected outcome. Many techniques to automatically produce inputs have been proposed over the years, and today are able to produce test suites with high code coverage [09]. Yet, the problem of the expected outcome persists and has become known as the oracle problem. This means that if we produce test inputs, then a human tester needs to specify the oracle in terms of the expected outcome.

## 2 LITERATURE SURVEY

Metaheuristic search techniques have been extensively used to automate the process of generating test cases, and thus providing solutions for a more cost-effective testing process [12]. This approach to test automation, often coined "Search-based Software Testing" (SBST), has been used for a wide variety of test case generation purposes. Since SBST techniques are heuristic by nature, they must be empirically investigated in terms of how costly and effective they are at reaching their test objectives and whether they scale up to realistic development artifacts. This paper [14] presents the results of a systematic, comprehensive review that aims at characterizing how empirical studies have been designed to investigate SBST cost-effectiveness and what empirical evidence is available in the literature regarding SBST cost-effectiveness and scalability. This paper [2] presents a novel approach to automatic software test data generation, where the test data is intended to cover program branches which depend on string predicates such as string equality, string ordering and regular expression matching. A search-based approach is assumed and some potential search operators and corresponding evaluation functions are assembled [11]. Their performance is assessed empirically by using them to generate test data for a number of test programs. A novel approach of using search operators based on programming language string operators and parameterized by string literals from the program under test is introduced. These operators are also assessed empirically in generating test data for the test programs and are shown to provide a significant increase in performance [2].

The natural mate-selection behavior of preferring individuals which are somewhat (but not too much) different has been proved to increase the resistance to infection of the resulting offspring, and thus fitness. Inspired by these results we have investigated the improvement obtained from diversity induced by differences between individuals sent and received and the resident population in an island model, by comparing different migration policies, including our proposed multiculturaity methods, which choose the individuals that are going to be sent to other nodes based on the principle of multiculturalists; the individual sent should be different enough to the target population, which will be represented through a proxy string (computed in several possible ways) in the emitting population.

The use of search algorithms for test data generation has seen many successful results. For structural criteria such as branch coverage, heuristics have been designed to help the search [5]. The most common heuristic is the use of approach level (usually represented with an integer) to reward test cases whose executions get close (in the control flow graph) to the target branch. To solve the constraints of the predicates in the control flow graph, the branch distance is commonly employed. These two measures are linearly combined [4]. Because the approach level is more important, the branch distance is normalised, often in the range. In this paper, we analyze different types of normalizing functions [13]. We found out that the one that is usually employed in the literature has several flaws. We hence propose a different normalizing function that is very simple and that does not suffer of these limitations. In the presence of an internal state, often a sequence of function calls is required to test software.

In fact, to cover a particular branch of the code, a sequence of previous function calls might be required to put the internal state in the appropriate configuration. Internal states are not only present in object-oriented software, but also in procedural software (e.g., static variables in C programs). In the literature, there are many techniques to test this type of software [7]. However, to the best of our knowledge, the properties related to the choice of the length of these sequences have received only a little attention in the literature. In this paper [5], we analyze the role that the length plays in software testing, in particular branch coverage. We show that, on "difficult" software testing benchmarks, longer test sequences make their testing trivial.

## 3 TYPES OF TESTS

The purpose of testing is to discover errors. Testing is the process of trying to discover every conceivable fault or weakness in a work product. It provides a way to check the functionality of components, sub-assemblies, assemblies and/or a finished product. Software system meets its requirements and user expectations and does not fail in an unacceptable manner. There are various types of test. Each test type addresses a specific testing requirement.

### 3.1 SYSTEM TEST:

System testing ensures that the entire integrated software system meets requirements. It tests a configuration to ensure known and predictable results.

### 3.2 UNIT TESTING:

Unit testing involves the design of test cases that validate that the internal program logic is functioning properly, and that program inputs produce valid outputs. All decision branches and internal code flow should be validated. It is the testing of individual software units of the application .it is done after the completion of an individual unit before integration.

### 3.3 INTEGRATION TESTING:

Integration tests are designed to test integrated software components to determine if they actually run as one program. Testing is event driven and is more concerned with the basic outcome of screens or fields.

### 3.4 FUNCTIONAL TEST:

Functional tests provide systematic demonstrations that functions tested are available as specified by the business and technical requirements, system documentation, and user manuals. Organization and preparation of functional tests is focused on requirements, key functions, or special test cases.

### 3.5 White Box Testing:

White Box Testing is a testing in which in which the software tester has knowledge of the inner workings, structure and language of the software, or at least its purpose.

### 3.6 Black Box Testing:

Black Box Testing is testing the software without any knowledge of the inner workings, structure or language of the module being tested.

### 3.7 Acceptance Testing:

User Acceptance Testing is a critical phase of any project and requires significant participation by the end user. It also ensures that the system meets the functional requirements.

## 4 Test Suite Generation

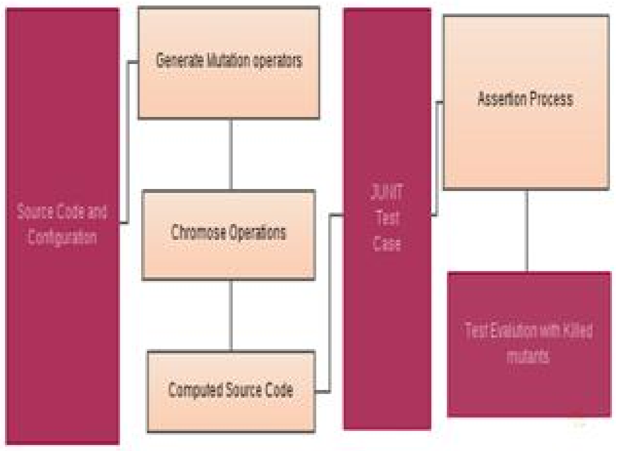
Source Code and Configuration
Generate Mutation operators
Chromose Operations
Computed Source Code
JUNIT Test Case
Assertion Process
Test Evalution with Killed mutants

*Figure 1- Architecture Diagram for Test Suite Generation*

### 4.1 Mutation Test Generation:

It uses mutation testing to produce a reduced set of assertions that maximizes the number of seeded defects in a class that are revealed by the test cases. These assertions highlight the relevant aspects of the current behavior in order to support developers in identifying defects, and the assertions capture the current behavior to protect against regression faults.

### 4.2 Genetic Algorithm:

Genetic Algorithms (GAs)[14] qualify as metaheuristic search technique and attempt to imitate the mechanisms of natural adaptation in computer systems. Population of chromosomes is evolved using genetics-inspired operations, where each chromosome represents a possible problem solution. In each iteration of the evolution, a new generation is created and initialized with the best individuals of the last generation (elitism). Then, the new generation is filled up with individuals produced by rank selection, crossover, and mutation.

### 4.3 Branch Coverage:

In branch coverage as test criterion, although the approach can be generalized to any test criterion. A program contains control structures such as if or while statements guarded by logical predicates; branch coverage requires that each of these predicates evaluates to true and to false [14]. A branch is infeasible if there exists no program input that evaluates the predicate such that this particular branch is executed.

## 4.4 RANDOM TEST CASES:

Random test case are used to evaluate the mutation testing ,Sampling a test case at random means that each possible test case in the search space has a nonzero probability of being sampled, and these probabilities are independent. For example, given a maximum length L, if each test case was sampled with uniform probability, then sampling a short sequence would be extremely unlikely [9].

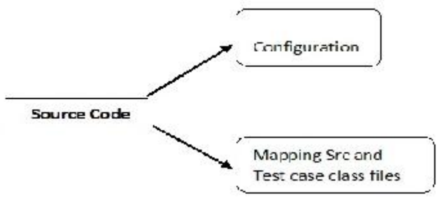

*Figure 2- Mapping Source Code (Level 0)*

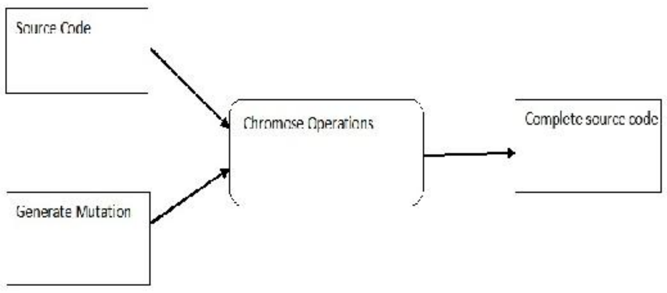

*Figure 3- Chromosome Operations (Level 1)*

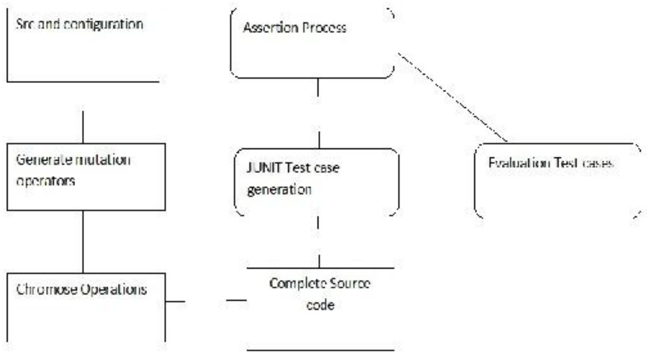

*Figure 4- JUNIT Test Case Generation (Level 2)*

## 5 CONCLUSION

Coverage criteria are a standard technique to automate test generation. In this paper, we have shown that optimizing whole test suites toward a coverage criterion is superior to the traditional approach of targeting one coverage goal at a time. In our experiments, this results in significantly better overall coverage with smaller test suites. While we have focused on branch coverage in this paper, the findings also carry over to other test criteria. Consequently, the ability to avoid being misled by infeasible test goals can help in overcoming similar problems in other criteria, for example, the equivalent mutant problem in mutation testing. Even though the results achieved with EVOSUITE already demonstrate that whole test suite generation is superior to single target test generation, there is ample opportunity to further improve our EVOSUITE prototype.

For example, there is potential in combining search-based test generation with dynamic symbolic execution (e.g., [12]), and search optimizations such as testability transformation or local search should further improve the achieved coverage. Furthermore, there are general enhancements in the literature of search algorithms that we could integrate and evaluate in EVOSUITE, as, for example, island models (e.g., see the recent [3]) and adaptive parameter control. In our empirical study, we targeted object-oriented software. However, the EVOSUITE approach could be easily applied to procedural software as well, although further research is needed to assess the potential benefits in such a context. The approach presented in this paper aims at producing small test suites with high coverage such that the developer can add test oracles in terms of assertions. Although keeping the test suites small is helpful in this respect, the oracle problem is still very difficult. In this respect, we are investigating ways to support the developer by automatically producing effective [12] assertions and, to ease understanding; we try to make the produced test cases more readable [16].

## REFERENCES

[1] S. Ali, L. Briand, H. Hemmati, and R. Panesar-Walawege, “A Systematic Review of the Application and Empirical Investigation of Search-Based Test-Case Generation,” IEEE Trans. Software Eng., vol. 36, no. 6, pp. 742-762, Nov./Dec. 2010.

[2] M. Alshraideh and L. Bottaci, “Search-Based Software Test Data Generation for String Data Using Program-Specific Search Operators: Research Articles,” Software Testing, Verification, and Reliability, vol. 16, no. 3, pp. 175-203, 2006.

[3] L. Araujo and J. Merelo, “Diversity through Multiculturality: Assessing Migrant Choice Policies in an Island Model,” IEEE Trans. Evolutionary Computation, vol. 15, no. 4, pp. 1-14, Aug. 2011.

[4] A. Arcuri, “It Really Does Matter How You Normalize the Branch Distance in Search-Based Software Testing,” Software Testing, Verification and Reliability, http://dx.doi.org/10.1002/stvr.457, 2011.

[5] A. Arcuri, “A Theoretical and Empirical Analysis of the Role of Test Sequence Length in Software Testing for Structural Coverage,” IEEE Trans. Software Eng., vol. 38, no. 3, pp. 497-519, May/ June 2011.

[6] A. Arcuri and L. Briand, “Adaptive Random Testing: An Illusion of Effectiveness?” Proc. ACM Int’l Symp. Software Testing and Analysis, 2011.

[7] A. Arcuri and L. Briand, “A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering,” Proc. 33rd Int’l Conf. Software Eng., pp. 1-10. 2011,

[8] A. Arcuri and G. Fraser, “On Parameter Tuning in Search Based Software Engineering,” Proc. Third Int’l Conf. Search Based Software Eng., pp. 33-47, 2011.

[9] A. Arcuri, M.Z. Iqbal, and L. Briand, “Black-Box System Testing of Real-Time Embedded Systems Using Random and Search-Based Testing,” Proc. 22nd IFIP Int’l Conf. Testing Software and Systems, pp. 95-110, 2010,

[10] A. Arcuri, M.Z. Iqbal, and L. Briand, “Random Testing: Theoretical Results and Practical Implications,” IEEE Trans. Software Eng., vol. 38, no. 2, http://doi.ieeecomputersoc..org/10.1109/TSE.2011.121, pp. 258-277, Mar./Apr. 2011.

[11] A. Arcuri and X. Yao, “Search Based Software Testing of Object-Oriented Containers,” Information Sciences, vol. 178, no. 15, pp. 3075-3095, 2008.

[12] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos, “Symbolic Search-Based Testing,” Proc. IEEE/ACM 26th Int’l Conf. Automated Software Eng., 2011.

[13] L. Baresi, P.L. Lanzi, and M. Miraz, “Testful: An Evolutionary Test Approach for Java,” Proc. IEEE Int’l Conf. Software Testing, Verification and Validation, pp. 185-194, 2010.

[14] B. Baudry, F. Fleurey, J.-M. Je´ze´quel, and Y. Le Traon, “Automatic Test Cases Optimization: A Bacteriologic Algorithm,” IEEE Software, vol. 22, no. 2, pp. 76-82, Mar./Apr. 2005.

[15] C. Csallner and Y. Smaragdakis, “JCrasher: An Automatic Robustness Tester for Java,” Software Practice and Experience, vol. 34, pp. 1025-1050, 2004.

[16] W. Feller, An Introduction to Probability Theory and Its Applications, vol. 1, third ed. Wiley, 1968.