Innovative Science and Technology Publications

International Journal of Future Innovative Science and Technology,
ISSN: 2454- 194X Volume-3, Issue-1, Jan - 2017

A GENETIC TOOL BASED RANDOMIZED UNIT TESTING FOR ENSURING QUALITY SOFTWARE

Baskaran.P

Assistant Professor,
Department of CSE,
SNS College of Engineering,
Coimbatore, TamilNadu, India.

E-Mail: baskarcse06@gmail.com

Jan – 2017

www.istpublications.com

Received: **Sep-2016** *Revised:* **Oct-2016** *Accepted:***Dec-2016** *Published:***Jan-2017**

A GENETIC TOOL BASED RANDOMIZED UNIT TESTING FOR ENSURING QUALITY SOFTWARE

Baskaran.P

Assistant Professor,
Department of CSE,
SNS College of Engineering,
Coimbatore, TamilNadu, India.

E-Mail: ***baskarcse06@gmail.com***

Abstract: Randomized unit testing deals with testing the individual software units by using the test cases randomly or testing randomly with the test cases. Though it is most effective of its kind, the thoroughness of the randomized unit testing varies based on the settings of parameters and relative frequencies which the methods are called. In this system two testing criteria's are described. Primarily Genetic Algorithms are used to find parameters for randomized unit testing. Designing Genetic Algorithm is quite a black art, where the reduced Genetic Algorithm (GA) achieves almost the same results as the full system but in only 10 percent of time. It also uses a Feature Subset Selection tool to assess the size and content of the representations within the GA. Here Nighthawk – two level genetic random test data generation system is used to generate test cases randomly for the Unit Testing. Secondarily a general unit testing is done based on the traditional way of testing. After the completion of unit testing, from both the test results quality of the software is ensured by using control charts. The deviations in the charts of the two testing methods ensure the efficient quality evolution of the system. The results reinforce the belief that the testing made by these methods ensures the better quality than the existing system.

Index Terms- *Genetic, Testing, Unit, test case, randomized, nighthawk, software quality*

.

1 Introduction

Software testing involves running a piece of software (the software under test, or SUT) on selected input data and checking the outputs for correctness. The goal of software testing is to force failures of the SUT and to be thorough. The more thoroughly the SUT have been testing an SUT without forcing failures, It will be the reliability of the SUT.

1.1 RANDOMIZED UNIT TESTING

Randomized unit testing is unit testing where there is some randomization in the selection of the target method call sequence and/or arguments to the method calls. Many researchers have performed randomized unit testing, sometimes combined with other tools such as model checkers. A key concept in randomized unit testing is that of value reuse. This term is used to refer how the testing engine reuses the receiver, arguments, or return values of past method calls when making new method calls. In previous researches, value reuse has mostly taken the form of making a sequence of method calls all on the same receiver object; latest research reported as, value reuse on arguments and return values as well. Randomized testing depends on the generation of so many inputs that it is infeasible to get a human to check all test outputs. An automated test oracle is needed. There are two main approaches to the oracle problem. The first is to use general purpose, "high- pass" oracles that pass many executions but check properties that should be true of most software.

For instance, Miller et al. [2006] judge a randomly generated GUI test case as failing only if the software crashes or hangs, despite the use of high-pass oracles, all of these researches found

randomized testing to be effective in forcing failures. The second approach to the oracle problem for randomized testing is to write oracles in order to check properties specific to the software. These oracles, like all formal unit specifications, are nontrivial to write; tools such as Daikon for automatically deriving likely invariants could help here. Since randomized unit testing does not use any intelligence to guide its search for test cases, there has always been justifiable concern about how thorough it can be, given various measures of thoroughness, such as coverage and fault-finding ability.

Michael et al. [2005] performed randomized testing on the well-known Triangle program; this program accepts three integers as arguments, interprets them as sides of a triangle, and reports whether the triangle is equilateral, isosceles, scalene, or not a triangle at all. They concluded that randomized testing could not achieve 50 percent condition/decision coverage of the code, even after 1,000 runs.

1.2 GENETIC ALGORITHM IN TESTING

Genetic algorithms were first described by Holland. Candidate solutions are represented as "chromosomes," with solutions represented as "genes" in the chromosomes. The possible chromosomes form a search space and are associated with a fitness function representing the value of solutions encoded in the chromosome. Search proceeds by evaluating the fitness of each of a population of chromosomes, and then performing point mutations and recombination on the successful chromosomes. GAs can defeat purely random search in finding solutions to complex problems. Goldberg argues that their power stems from being able to engage in "discovery and recombination of building blocks" for solutions in a solution space.

Both of these approaches evaluate the fitness of a chromosome by measuring how close the input is to covering some desired statement or condition direction. Finally, the class testing represents the sequence of method calls in a unit test as a chromosome; the approach features customized mutation operators, such as one that inserts method invocations.

1.3 PROBLEM STATEMENT

Since randomized unit testing does not use any intelligence to guide its search for test cases, there has always been justifiable concern about how thorough it can be, given various measures of thoroughness, such as coverage and fault-finding ability. Randomized testing can be enhanced via randomized breadth-first search of the search space of possible test cases, but pruning branches that lead to redundant or illegal values which would cause the system to waste time on unproductive test cases. In addition, most analysis-based approaches incur heavy memory and processing time costs. However the random testing is done, it does not shows the quality evolution of the software.

2. EXISTING SYSTEM

Randomized testing uses randomization for some aspects of test input data selection. Several studies have found that randomized testing of software units is effective at forcing failures in even well-tested units. However, there remains a question of the thoroughness of randomized testing. Using various code coverage measures to measure thoroughness, researchers have come to varying conclusions about the ability of randomized testing to be thorough. The thoroughness of randomized unit testing is dependent on when and how randomization is applied, e.g., the number of method calls to make, the relative frequency with which different methods are called, and the ranges from which numeric arguments are chosen.

The manner in which previously used arguments or previously returned values are used in new method calls, which we call the value reuse policy, is also a crucial factor. It is often difficult to work out the optimal values of the parameters and the optimal value reuse policy by hand. Later the testing is made the quality of testing cannot be calculated in any manner. Since the quality of testing are not projected out the evolution of software in the quality is still unknown.

PROBLEM STATEMENT

- Since randomized unit testing does not use any intelligence to guide its search for test cases, there has always been justifiable concern about how thorough it can be, given various measures of thoroughness, such as coverage and fault-finding ability.
- Randomized testing can be enhanced via randomized breadth-first search of the search space of possible test cases, but pruning branches that lead to redundant or illegal values which would cause the system to waste time on unproductive test cases.

- In addition, most analysis-based approaches incur heavy memory and processing time costs.
- However the random testing is done, it does not shows the quality evolution of the software.

3. PROPOSED SYSTEM

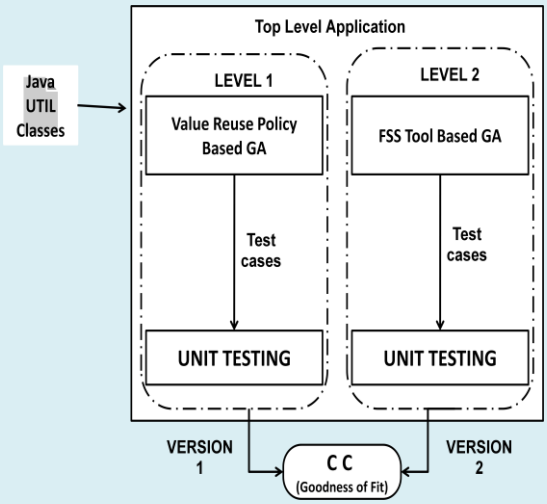


Fig. 1 System Architecture

The project involves a unit test data generator named Nighthawk. Nighthawk has two levels. The upper level is a genetic algorithm (GA) which uses fitness evaluation, selection, mutation, and recombination of chromosomes to find good values for the genes. The lower level is a randomized unit testing engine which tests a set of methods according to parameter values specified as genes in a chromosome, including parameters that encode a value reuse policy.

Goodness is evaluated on the basis of test coverage and number of method calls performed. Users can use Nighthawk to find good parameters, and then perform randomized unit testing based on those parameters. The randomized testing can quickly generate many new test cases that achieve high coverage and can continue to do so for as long as users wish to run it. After the test cases are generated from the generator, the testing is done for the chosen application on both the methods. Initially testing is done with the test case generated by the genetic algorithm based test case generator engine. Those results are noted down with the execution time of the corresponding test cases. Secondly the testing is done by general unit testing. Especially in JAVA, the jUNIT package be used for the testing. Then the results are compared for charting. By using control charts we can ensure the quality of the test made and the software evolution.

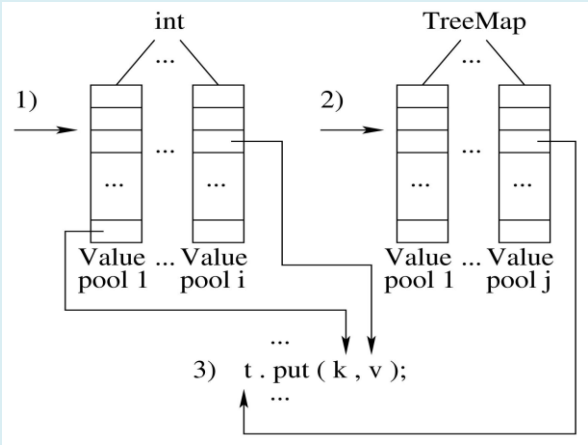
System Framework

The below figure shows the design of the NIGHTHAWK. The model has three stages to function.

Stage 1: Random values are seeded into the value pools for primitive types such as int, according to bounds in the pools.

Stage 2: Values are seeded into non primitive type classes that have initializer constructors by calling those constructors.

Fig. 2 Design of NIGHTAWK



Stage 3: The rest of the test case is constructed and run by repeatedly randomly choosing a method and receiver and parameter values. Each method call may result in a return value which is placed back into a value pool (not shown).

4 GENETIC ALGORITHM LEVEL

The system uses the space of possible chromosomes as a solution space to search, and apply the GA approach to search it for a good solution. It chose GAs over other meta heuristic approaches such as simulated annealing because it's a belief that recombining parts of successful chromosomes would result in chromosomes that are better than their parents. However, other meta heuristic approaches may have other advantages and should be explored in future work. The parameter to Nighthawk's GA is the set M of target methods. The GA performs the usual chromosome evaluation steps (fitness selection, mutation, and recombination). The GA derives an initial template chromosome appropriate to M, constructs an initial population of size p as clones of this chromosome, and mutates the population. It then loops for the desired number g of generations, of evaluating each chromosome's fitness, retaining the fittest chromosomes, discarding the rest, cloning the fit chromosomes, and mutating the genes of the clones

with probability m percent using point mutations and crossover (exchange of genes between chromosomes). The evaluation of the fitness of each chromosome c proceeds as follows: The random testing level of Nighthawk generates and runs a test case, using the parameters encoded in c. It then collects the number of lines covered by the test case. If it bases the fitness function only on coverage, then any chromosome would benefit from having a larger number of method calls and test cases since every new method call has the potential of covering more code. Nighthawk therefore calculates the fitness of the chromosome as:

(Number of coverage points covered) * (coverage factor) = (number of method calls performed overall).

The coverage factor sets to 1,000, meaning that the system is willing to make 1,000 more method calls (but not more) if that means covering one more coverage point. For the three variables mentioned above, Nighthawk uses default settings of p ¼ 20; g ¼ 50;m ¼ 20. These settings are different from those taken as standard in GA literature, and are motivated by a need to do as few chromosome evaluations as possible (the primary cost driver of the system). The population size p and the number of generations g are smaller than standard, resulting in fewer chromosome evaluations; to compensate for the lack of diversity in the population that would otherwise result, the mutation rate m is larger. The settings of other variables, such as the retention percentage, are consistent with the literature.

To enhance availability of the software, Nighthawk uses the popular open source coverage tool Cobertura to measure coverage. Cobertura can measure only line coverage (each coverage point corresponds to a source code line and is covered if any code on the line is executed). However, Nighthawk's algorithm is not specific to this measure.

Generating Data Sets:

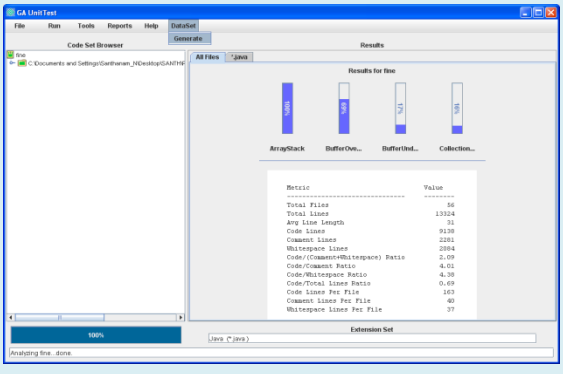


Fig.3 Description Tool for Data set alignment

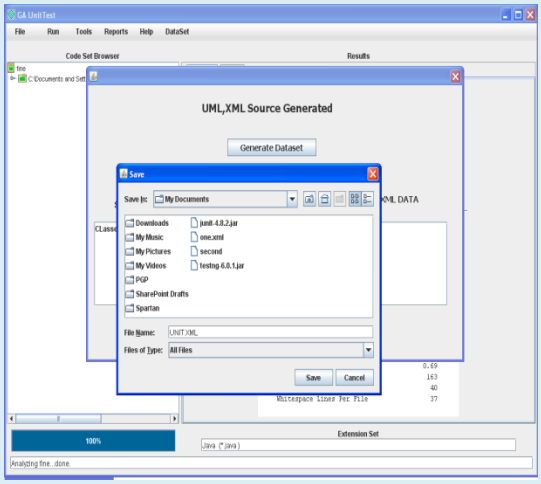


Fig. 4 XML Datasets

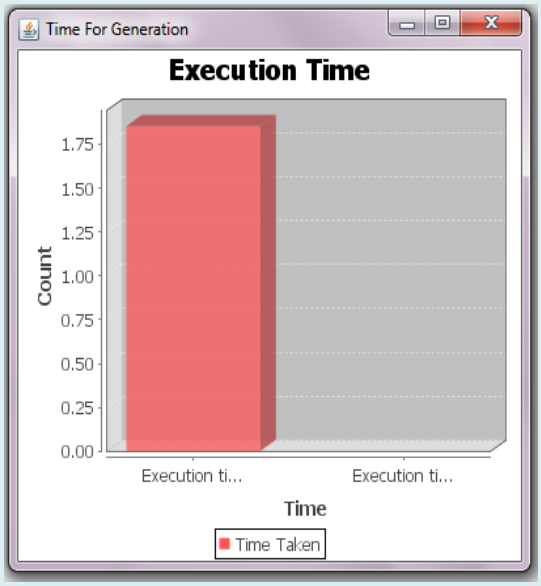


Fig.5 Coverage Measure

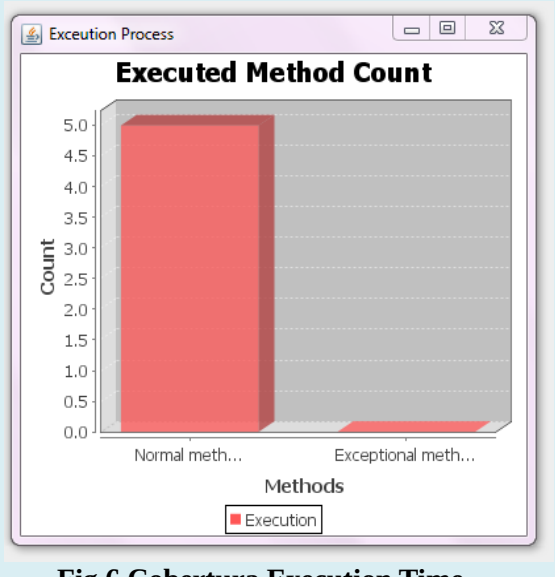


Fig.6 Cobertura Execution Time

5. CONCLUSION

Randomized unit testing is a superior technology that has been shown to be effective, but whose thoroughness depends on the settings of test algorithm parameters. In the system, Nighthawk, a system in which an upper level genetic algorithm automatically derives good parameter values for a lower level randomized unit test algorithm. It had been shown that Nighthawk is able to achieve high coverage of complex, real-world Java units, while retaining the most desirable feature of randomized testing: the ability to generate many new high-coverage test cases quickly. The control charts are used to evaluate the quality ensured after the test has been made. This system can be enhanced in the future by taking the huge databases for testing and the reduction of the complexity in handling the application data. Future work also includes the integration into Nighthawk of useful facilities from past systems, such as failure-preserving or coverage-preserving test case minimization. Integration of a feature subset selection learner into the GA level of the Nighthawk algorithm for dynamic optimization of the GA is also possible.

REFERENCES

[1] J.H. Andrews, S. Haldar, Y. Lei, and C.H.F. Li, "Tool Support for Randomized Unit Testing," Proc. First Int'l Workshop Randomized Testing, pp. 36-45, July 2006.

[2] J. Andrews and T. Menzies, "On the Value of Combining Feature Subset Selection with Genetic Algorithms: Faster Learning of Coverage Models," Proc. Fifth Int'l Conf. Predictor Models in Software Eng., http://menzies.us/pdf/09fssga.pdf, 2009.

[3] E.J. Weyuker, "On Testing Non-Testable Programs," The Computer J., vol. 25, no. 4, pp. 465-470, Nov. 1982.

[4] W.K. Leow, S.C. Khoo, and Y. Sun, "Automated Generation of Test Programs from Closed Specifications of Classes and Test Cases," Proc. 26th Int'l Conf. Software Eng., pp. 96-105, May 2004.

[5] J.H. Andrews and Y. Zhang, "General Test Result Checking with Log File Analysis," IEEE Trans. Software Eng., vol. 29, no. 7, pp. 634-648, July 2003.

[6] L.A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," IEEE Trans. Software Eng., vol. 2, no. 3, pp. 215-222, Sept. 1976.

[7] Periyasamy, Baskaran, and R. Ashok kumar. "A Combined Model with Test Prioritizing for Testing an Event Driven Software." American Journal of Software Engineering 3.1 (2015): 1-5.

[8] I. Kononenko, "Estimating Attributes: Analysis and Extensions of Relief," Proc. Seventh European Conf. Machine Learning. pp. 171-182, 1994.

[9] S. Berner, R. Weber, and R.K. Keller, "Enhancing Software Testing by Judicious use of Code Coverage Information," Proc. 29th Int'l Conf. Software Eng., pp. 612-620, May 2007.

[10] W.C. Hetzel, ed., Program Test Methods, Prentice-Hall, 1973.

[11] R. Hamlet, "Random Testing," Encyclopedia of Software Eng., Wiley,pp. 970-978, 1994.

[12] P. Tonella, "Evolutionary Testing of Classes," Proc. ACM/SIGSOFT Int'l Symp. Software Testing and Analysis, pp. 119-128, July 2004.

[13] K. Kira and L. Rendell, "A Practical Approach to Feature Selection," Proc. Ninth Int'l Conf. Machine Learning, pp. 249-256, 1992.

[14] C. Csallner and Y. Smaragdakis, "JCrasher: An Automatic Robustness Tester for Java," Software Practice and Experience, vol. 34, no. 11, pp. 1025-1050, 2004.

[15] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," IEEE Trans. Software Eng., vol. 27, no. 2, pp. 99-123, Feb. 2001.

[16] Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Artoo: Adaptive Random Testing for Object-Oriented Software," Proc. 30th ACM/IEEE Int'l Conf. Software Eng., pp. 71-80, May 2008.

[17] S. Vaucher et al., "Tracking Design Smells: Lessons from a Study of God Classes," Proc. 16th Working Conf. Reverse Eng. (WCRE 09), IEEE CS Press, 2009, pp. 145–154.

[18] S. Kan, Metrics and Models in Software Quality Engineering, Addison- Wesley, 2003.