An Empirical Study of Randomized Unit Tests using Nighthawk System

N.Rajasekaran
Assistant Professor, Department of Computer Application
Kongu Arts and Science College
Bharathiar University
Tamil Nadu, India

K.K.Sureshkumar
Assistant Professor,
Department of MCA
Kongu Arts and Science College
Bharathiar University
Tamil Nadu, India

N.M.Elango, PhD.
Professor and Head,
Department of MCA
RMK Engineering College
Anna University of Technology
Chennai, India

ABSTRACT

Randomized testing has been shown to be an effective method for testing software units. However, the thoroughness of randomized unit testing varies widely according to the input which is provided by the user. Such as the relative frequencies with which methods are called to be improved the thoroughness of randomized unit testing. In this paper the system, describes genetic algorithm based parameter finding for randomized unit testing that optimizes test coverage. Here the unit test data will be generated by nighthawk system. The system can be viewed as two levels, lower level and upper level. Randomized unit testing engine is a lower level, which tests a set of methods according to parameter values specified as genes in a chromosome, including parameters that encode a value reuse policy. The upper level is a genetic algorithm (GA) which uses fitness evaluation, selection, mutation and recombination of chromosomes in order to find out good values for the genes. Integrity is evaluated on the basis of test coverage and number of method calls performed. To find good parameters users can use Nighthawk and then perform with randomized unit testing based on those parameters. Many new test cases can quickly generate by randomized testing that achieve high coverage, and can continue to do so for as long as users wish to run it. In this research the test coverage results of Nighthawk are compared with manual unit testing results [6]. The Nighthawk system produced maximum test coverage results in less timing based on the genetic algorithm comparing with manual unit testing results.

General Terms

Software Testing, Unit testing, Genetic algorithms

Keywords

Randomized Unit Testing, Feature Subset Selection, Nighthawk. Software under Test (SUT).

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 goals of software testing are to force failures of the SUT, and to be thorough. The more thoroughly tested an SUT without forcing failures, the surer produces the reliability of the SUT. Randomized testing is the practice of using randomization for some aspects of test input data selection [15]. Various research and studies have found that randomized testing for software products is very effective and efficient at forcing failures in even well-tested units. However, the randomized testing is forcing failures in even well tested software units but the question will be raised it’s thorough enough or not. 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 highly dependent on parameters that control when and how randomization is applied. These parameters include 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 returned value or previously used arguments are again used in new method calls, which referred to as the value reuse policy, is also a crucial factor. It will be very difficult to find out the optimum solutions of the parameter and the optimal value reuse policy by hand.

1.1 Randomized Unit Testing

Random testing is a form of functional testing that is useful when the time needed to write & run directed tests is too long (or the complexity of the problem makes it impossible to test every combination). Release criteria may include a statement about the amount of random testing that is required. For example, we have a requirement that there should be no random failures for 2 weeks prior to release (that is 2 weeks of continuous random testing on 50 workstations).One of the big issues of random testing is to know when a test fails. As with all testing, an oracle is needed. You could rely in assertions in the code as your sole oracle (i.e. you throw random inputs at the code, possibly from multiple threads, and if no GPF happens in 2 weeks then you assume it's OK). In other situations, common with hardware development, you have two different implementations of the same specification (one is 'the golden model’; the other is 'the implementation'. If they both agree to a defined accuracy then the test passes. When doing random testing you must, of course, ensure that your tests are sufficiently random, and that they cover the spec. repeating the same test for 2 weeks doesn't tell you anything. It is often claimed, correctly, that random testing is less efficient than directed testing. But you must consider the time needed to write random test generator vs. the time to write a set of directed tests (or generators). Once you have a random test generator, you computer(s) can work 24 hours a day generating new tests.

The rest of the paper is organized as follows. Section 2 of this paper reveals previous work of genetic algorithms of testing. Section 3 focuses on the objectives of the study. Section 4 discusses about the proposed system of nighthawk system and randomized testing level. In Section 5, methodology and algorithm to construct run test cases are explained. The results and findings comparison of the manual unit testing and nighthawk testing are dealt in Section 6. The work is concluded and possible future enhancements are discussed in Section 6.

2. LITERATURE REVIEW

2.1 Genetic Algorithms for Testing

Genetic Algorithms (GAs) are adaptive heuristic search algorithm premised on the evolutionary ideas of natural selection and genetic. The basic concept of GAs is designed to simulate processes in natural system necessary for evolution, specifically those that follow the principles first laid down by Charles Darwin of survival of the fittest. As such they represent an intelligent exploitation of a random search within a defined search space to solve a problem. This algorithm is purely used for random search in finding solutions and search too many complex problems. First pioneered by John Holland in the 60s, Genetic Algorithms has been widely studied, experimented and applied in many fields in engineering worlds. Not only does GAs provide alternative methods to solving problem, it consistently outperforms other traditional methods in most of the problems link. Many of the real world problems involved finding optimal parameters, which might prove difficult for traditional methods but ideal for GAs. However, because of its outstanding performance in optimisation, GAs has been wrongly regarded as a function optimiser. In fact, there are many ways to view genetic algorithms. The review of Rela's describes 122 applications of metaheuristic search in software engineering, 44% of the software applications related to testing. Approaches to GA test suite generation can be black-box (requirements-based) or white-box (code-based); here this paper focus on white-box approaches, since this approach is coverage-based and therefore white box testing represent a set of testing data as a chromosome. In this each genes encode one input value to the software [21, 22]. Michael.C.C [23] represents the similarity of test data and comparing various strategies for augmenting the GA search [9]. Both of the above mentioned two approaches evaluate the fitness of chromosome the input is to covering some desired statement or condition direction. Guo et al [16] generate unique input-output (UIO) sequences for protocol testing using a genetic algorithm; the sequence of genes represents a sequence of inputs to a protocol agent, and the fitness function computes a measure related to the coverage of the possible states and transitions of the agent. The GA can of course be re-run to generate more test cases, but there is a good performance penalty since each run of the genetic algorithm generates only one new test case. In contrast, in our approach, each run of the GA results in a parameter setting for randomized testing which one can be applied and effective many times to generate many distinct high-coverage test cases? All analysis-based approaches share the disadvantage of requiring a robust parser and source code analyzer that can be updated to reflect changes in the source language. These complex tools are not often provided by language providers. Our approach does not require source code or byte code analysis, instead depending only on the robust Java reflection mechanism and commonly-available coverage tools. For instance, our source code was initially written with Java old versions (1.6) in other old or versions in mind, but worked seamlessly on the Java 1.7 versions of the java. util classes, despite the fact that the source code of many of units had been heavily modified to introduce templates. However, model-checking approaches have other strengths, such as the ability to analyze multithreaded code, further supporting the conclusion that the white box and model-checking approaches are complementary [25].

3. OBJECTIVES OF THE STUDY

The main objective of this research paper is the Nighthawk systems unit test data generator. It's used to generate the high test coverage in short period. Nighthawk has two levels. The lower level is a randomized unit testing which test set of methods according to the parameter specified as input genes in a particular chromosome. This includes different parameters that encode value reuse policy. The upper level in this algorithm is fitness evaluation, selection and mutation and recombination of chromosomes to find good values for the genes [19]. Goodness is evaluated on the basis of test coverage and number of method calls performed [4]. Using the Nighthawk system the user can find very good argument and 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. In this paper, the optimization techniques for genetic algorithms tools like nighthawk also discussed. Using FSS techniques the randomization can prune many of Nighthawk's mutate (gene types) without compromising coverage. The pruned Nighthawk tool achieves nearly the same coverage as full Nighthawk (90%) and does 10 times faster. So this research should recommends that meta-heuristic search based software engineering tools should also routinely perform subset selection.

4. PROPOSED SYSTEM

4.1 Nighthawk System

The Nighthawk system described in this paper significantly builds on this work by automatically determining various methods and its parameters used in the given classes, we developed Nighthawk, a genetic random test data generation system, using this system further carried out experiments and comparing it with manual unit testing and finding the optimal setting of program switches [13,17]. Unlike the methods discussed in the above Nighthawk's genetic algorithm does not result in a single test input. We take a class it used in inventory application. This class contains many methods with different arguments. Using Nighthawk ***"ClassParser"*** method we give class name as input, the output will be generated automatically that is test case. It contains method name and each its argument list. We can get Output as a test case in random manner. Using this we can test an application.

The results of our research encouraged to expand the scope of the GA to include method parameter ranges, value reuse policy and other randomized testing parameters. The result was very effective when using Nighthawk implementation of test data. In this research, first outline the lower randomized-testing level of Nighthawk, and then describe the chromosome that controls its operation. After that, depict the genetic-algorithm level and the end user interface [18]. Finally, it describes the use of automatically generated test wrappers for precondition checking, result evaluation and coverage enhancement.

4.2 Randomized Testing Level

Here randomized testing present a simplified description of the algorithm that the lower, randomized-testing, level of Nighthawk uses to construct and run a test case. The algorithm takes two parameters: a set *C* of Java classes and a *GA* chromosome *C* appropriate to *C*. the chromosome controls aspects of the algorithm's behavior, such as the number of method calls to be made. In this paper *C* is the set of "*Class Name*". And *m* the type of method corresponding to *M* is the following sets of types: All types of receivers, parameters and return values of methods in *M*. All primitive types that are the types of parameters to constructors of other types of interest [10]. Each type is associated with an array of value pools, and each value pool for contains an array of values of type. Each value pool for a range primitive type (a primitive type other than Boolean and void) has bounds on the values that can appear in it. The number of value pools, number of values in each value pool, and the range primitive type bounds are specified by the chromosome. The GA algorithm first chooses initial values for primitive type pools, and then moves on to non primitive type pools. Here define a constructor method to be an initialize if it has no parameters, or if all its parameters are of primitive types. Define a constructor to be a re initialize if it has no parameters, or if all its parameters are of types and define the set of callable methods to be the methods in plus the reinitializes of the types. A call description is an object representing one method call that has been constructed and run [7]. It consists of the method name, an indication of whether the method call succeeded, failed or threw an exception, and one object description for each of the receiver, the parameters and the result (if any). A test case is a sequence of call descriptions, together with an indication of whether the test case succeeded or failed [2]. Nighthawk's randomized testing algorithm is referred to as constructRunTestCase. It takes a set of target methods and a chromosome as inputs. It begins by initializing value pools, and then constructs and runs a test case, and returns the test case.

Input: a class name Output: generated test case.

Steps:

1) Choose an application for testing.
2) Each application having many modules and many classes.
3) Each Class having many target method
4) Choose any class from a module as genes.
5) Using ***ClassParser()***, ***getClass, getDeclaredMethods***, method in ***apache.java*** as *chromosomes*.
6) Using the above *chromosomes can generarate* test case
 Contains method names and its argument list.

5. METHODOLOGY

An auxiliary method called ***DynamicDataDemo*** and ***ClassParser*** which takes a class as input, this method calls the all methods in that and returns a call description. In algorithm descriptions, the word **"Random Data"** is always used to mean specifically a random choice which may partly depend on the Chromosome. ***m.getName*** considers a method call to fail if and only if it throws an Assertion Error. It does not consider other exceptions to be failures, since they might be correct responses to bad input parameters [20]. A separate mechanism is used for detecting precondition violations and checking correctness of return values and exceptions. These concern the treatment of nulls, the treatment of String, and the treatment of Object. The receiver of a method call cannot be null, and no parameter can be null unless ***m.getArgument*** Types chooses it to be. If ***m.getArgumentTypes*** fails to find a non-null value when it is looking for one, it reports failure of the attempt to call the method; ***ClassParser*** tolerates a certain number of these attempt failures before terminating the test case generation process. Being initialized with ***"strings"***. Some default strings are supplied by the system, and the user can supply more. Formal parameters of type ***java.lang.Object*** stand for some arbitrary object, but it is usually sufficient to use a small number of specific types as actual parameters; Nighthawk uses only ***int*** and ***string*** by default. A notable exception to this rule is the parameter to the equals () method, which can be treated specially by test wrapper objects. Java.lang.String is treated as if it is a primitive type, the values in the value pools chromosomes.

DynamicDataDemo:
Input:Class C Chromosome ch as argument;
Output: a test case.
Steps:

```
1) If C is not a static and constructor class
2) For each method in a class C
a) Select ClassParser ("class name comes here"); method in
Apche configuration file.
b) Choose a class name as argument chromosome.
c) Class C getting testing method name and its return type.
3) If the method is constructor or static call it with the chosen
arguments. Otherwise call it as receiver.
4) If the method call threw an error, return failure indication.
5) If the method calls other expression, return a call
description using Java Exception handling.
6) Otherwise if the method return type is not void and not
null, the type t is not primitive and returns a call description
with success indication.
```

Table 1: A Nighthawk gene type and its methods

Gene type	Occurrence	Type	Description
Nightawk	One for whole chromosome	int	N method call to be made
class.getName()	One for each and method	Int	Getting method name
ClassParser()	One for whole chromosome	All accepted data types	Initial method for putting class name for execution
clazz.getMethods	One for each position	Int, float	Get method name
MethodTroubleReturns()	Common for all chromosome	Int of float	Return any error in the corresponding methods
ClassNotFound Exception ex()	One for whole chromosome	char	Return error message when the class is not fount
getInventory()	User defined chromosome	Int, float	It contains many sub method
getPayroll()	User defined chromosome	Int, float	It shows employee payroll
Cmdb()	User defined chromosome	Int, float	Configuration management
Billing()	User defined chromosome	Int, float	Billing system for customers

6. FINDINGS AND RESULTS

In nighthawk system, the time taken to test the inventory, payroll, billing and CMDB application is (MM:SS) 15:05, 10:23, 11:13 and 18:00 respectively and the utilization of CPU is 60%, 55%, 50% and 65% respectively. In manual unit testing, the time taken to test the inventory, payroll, billing and CMDB application is (MM:SS) 25:55, 22:1, 17:05

and 30:17 respectively and the utilization of CPU is respectively 110%, 105%, 90% and 115%. Comparing with manual testing, the nighthawk system saves 10 min 50 Sec for inventory, 12 Min 22 Sec for payroll, 06 Min 08 Sec for Billing and 12 Min 17 Sec for CMDB applications. And the CPU utilization is also reduced approximately 50% in all applications.

In this research it is found that in nighthawk system, the CPU utilization and the turnaround time is less than manual unit testing. The result of the proposed system Nighthawk was compared with the result of manual unit testing. The comparison shows, that the Nighthawk system (with random test data) tests the given application more quickly and efficiently comparing with the manual unit testing. The result comparison is given in the Table 2. The results are represented in the charts as shown in the Fig 1 and Fig.2.

Table 2: Results Comparison of Manual Unit Testing and Nighthawk Testing

S.No.	Application Name	SLOC*	Manual Unit Testing		Nighthawk Testing	
			Turn Around Time	CPU Utilization	Turn Around Time	CPU Utilization
1	Inventory	1123	00:25:55	0 - 110	00:15:05	0 - 60
2	Payroll	956	00:22:17	0 - 105	00:10:23	0 - 55
3	Billing	750	00:17:05	0 - 90	00:11:13	0 - 50
4	CMDB*	1310	00:30:17	0 - 115	00:18:00	0 - 65
SLOC – Source Line of Code **CMDB** – Configuration Management Database						

Fig 1: Nighthawk Testing

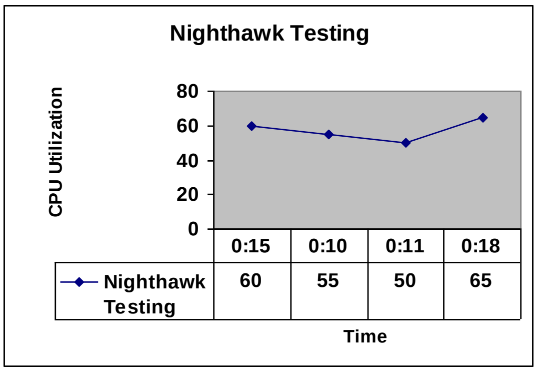
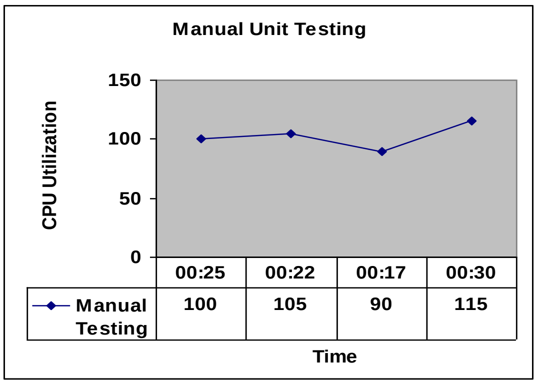


Fig 2: Manual Unit Testing



6. CONCLUSION AND FUTURE ENHANCEMENT

6.1 Conclusion

Randomized unit testing is a promising technology that has been shown to be effective, but whose thoroughness depends on the settings of test algorithm parameters and test cases. In this paper, Nighthawk were described, a system in which use genetic algorithm automatically derives methods and parameters in a module or a class in any kind of testing software applications. The comparison shows 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. In this research the test coverage results of Nighthawk were compared with manual unit testing results. The Nighthawk system produced maximum test coverage results in less timing based on the genetic algorithm and Feature Subset Selection (FSS) techniques comparing with manual unit testing results. And this research shows that we were able to optimize and simplify metaheuristic search tools. Metaheuristic tools (such as genetic algorithms) typically mutate some aspect of a candidate solution and evaluate the results. If the effect of mutating each aspect is recorded, then each aspect can be considered a feature and is amenable to the FSS processing. In this way, FSS can be used to automatically find and remove superfluous parts of the search control.

6.2 Future Enhancement

Future enhancement includes the integration into Nighthawk of useful facilities from past systems, such as failure-preserving or coverage-preserving test case minimization, and further experiments on the effect of program options on coverage and efficiency. Also wish to integrate a feature subset selection learner into the GA level of the Nighthawk algorithm for dynamic optimization of the GA. Further, can see a promising line of research where the cost/benefits of a particular metaheuristic are tuned to the particulars of a specific problem. This research shows that if we surrender one 10th of the coverage, we can run Nighthawk 10 times faster. While this is an acceptable trade-off in many domains, it may unsuitable for safety critical applications. More work is required to understand how to best match metaheuristic (with or without FSS) to particular problem domains.

7. REFERENCES

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

[3]. Andrews.J, F. Li, and T. Menzies, "Nighthawk: A Two-Level Genetic-Random Unit Test Data Generator," *Proc. 22nd IEEE/ACM Int'l Conf. Automated Software Eng.,*

[4]. Andrews.J 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.,*

[5]. Anatoly's and R.G. Hamlet, "Automatically Checking an Implementation against its Formal Specification," *IEEE Trans. Software Eng.,* vol. 26, no. 1, pp. 55-69, Jan. 2000.

[6] Ball.T, "A Theory of Predicate-Complete Test Coverage and Generation," *Proc. Third Int'l Symp. Formal Methods for Components and Objects,* pp. 1-22, Nov. 2004.

[7]. Ciupa.I, 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.

[8]. Claessen.K and J. Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs," *Proc. Fifth ACM SIGPLAN Int'l Conf. Functional Programming,* pp. 268-279, Sept. 2000.

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

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

[11].Doong.R.K and P.G.Frankl,"The ASTOOT Approach to Testing Object-Oriented Programs," *ACM Trans. Software Eng. and Methodology,* vol. 3, no. 2, pp. 101-130, Apr. 1994.

[12].Ernst M.D., 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.

[13].Godefroid.P, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 213-223, June 2005.

[14].Goldberg.D.E, Genetic Algorithm in Search, Optimization, and Machine Learning. Addison-Wesley, 1989.

[15].Groce .A, G.J. Holzmann, and R. Joshi, "Randomized Differential Testing as a Prelude to Formal Verification," *Proc. 29th Int'l Conf. Software Eng.,* pp. 621-631, May 2007.

[16].Guo.Q, R.M. Hierons, M. Harman, and K. Derderian, "Computing Unique Input/Output Sequences Using Genetic Algorithms," *Proc. Third Int'l Workshop Formal Approaches to Testing of Software,* pp. 164-177, 2004.

[17].Gupta.N, A.P. Matcher, and M.L. Soffa, "Automated Test Data Generation Using an Iterative Relaxation Method," *Proc. Sixth Int'l Symp. Foundations of Software Eng.,* pp. 224-232, Nov. 1998.

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

[19].Holland J.H, Adaptation in Natural and Artificial Systems. University of Michigan Press, 1975.

[20].King J.C, "Symbolic Execution and Program Testing," Comm. ACM, vol. 19, no. 7, pp. 385-394, 1976. Kira.K and L. Rendell, "A Practical Approach to Feature Selection," *Proc. Ninth Int'l Conf. Machine Learning,* pp. 249-256, 1992.

[21].Korel.B, "Automated Software Test Generation," *IEEE Trans.Software Eng.,* vol. 16, no. 8, pp. 870-879, Aug. 1990.

[22].Leow W.K, 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.

[23].Michael C.C, G. McGraw, and M.A. Schatz, "Generating Software Test Data by Evolution," *IEEE Trans. Software Eng.,* vol. 27, no. 12, pp. 1085-1110, Dec. 2001.

[24].Miller B.P, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Comm. ACM,* vol. 33, no. 12, pp. 3244, Dec. 1990.

[25].Owen.D and T. Menzies, "Lurch: A Lightweight Alternative to Model Checking," *Proc. 15th Int'l Conf. Software Eng. and Knowledge Eng.,* pp. 158-165, July 2003.