International Journal of Engineering and Advanced Technology (IJEAT)
ISSN: 2249-8958 (Online), Volume-2 Issue-3, February 2013

A Neural Network Approach for Randomized Unit Testing Based On Genetic Algorithm

R. Raju, P.Subhapriya

Abstract*—The goal of unit testing is to isolate each part of the program and show that the individual parts are correct. A unit test provides a strict, written contract that the piece of code must satisfy. As a result, it affords several benefits. Unit tests find problems early in the development cycle. In continuous unit testing environments, through the inherent practice of sustained maintenance, unit tests will continue to accurately reflect the intended use of the executable code in the face of any change. Depending upon established development practices and unit test coverage, up-to-the-second accuracy can be maintained. In this paper, a genetic algorithm to evolve a set of inputs. So the system called Nighthawk, which uses a genetic algorithm (GA) to find parameters for randomized unit testing that optimize test coverage. Therefore using a feature subset selection (FSS) tool to assess the size and content of the representations within the GA. The enhancement in this work is to introduce Neural network based unit testing , include some training sets for possible output and then apply the Genetic Algorithm. Therefore these results shows a better efficiency in the unit testing and reduce the test coverage.

***Keyword*— Unit testing, Genetic algorithm, Neural network, FSS.**

I. INTRODUCTION

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include, but are not limited to, the process of executing a program or application with the intent of finding software bugs (errors or other defects).Software testing can be stated as the process of validating and verifying that a software program/application/product:

1. Meets the requirements that guided its design and development;
2. Works as expected;
3. Can be implemented with the same characteristics.
4. Satisfies the needs of stakeholders

Manuscript published on 28 February 2013.
***** Correspondence Author (s)
R. Raju, Associate Professor , Department of Information Technology, Sri ManakulaVinayagar Engineering College, Puducherry, India.
P.Subhapriya, PG Student, Computer Science and Engineering, Sri ManakulaVinayagar Engineering College, Puducherry, India.

Software testing, depending on the testing method employed, can be implemented at any time in the development process. Traditionally most of the test effort occurs after the requirements have been defined and the coding process has been completed, but in the Agile approaches most of the test effort is on-going. As such, the methodology of the test is governed by the chosen software development methodology.

Different software development models will focus the test effort at different points in the development process. Newer development models, such as Agile, often employ test driven development and place an increased portion ofthe testing in the hands of the developer, before it reaches a formal team of testers. In a more traditional model, most of the test execution occurs after the requirements have been defined and the coding process has been completed. 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 we have tested an SUT without forcing failures, the more sure we are of the reliability of the SUT. 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.

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.

II. BACKGROUND

A. Feedback-Directed Random Test Generation

This work addresses random generation of unit tests for object-oriented programs. Such a test typically consists of a sequence of method calls that create and mutate objects, plus an assertion about the result of a final method call.

A test can be built up iteratively by randomly selecting a method or constructor to invoke, using previously computed values as inputs.[1] It is only sensible to build upon a legal sequence of method calls, each of whose intermediate objects is sensible and none of whose methods throw an exception indicating a problem.

The present a technique that improves random test generation by incorporating feedback obtained from executing test inputs as they are created. This technique builds inputs incrementally by randomly selecting a method call to apply and finding arguments from among previously constructed inputs. As soon as an input is built, it is executed and checked against a set of contracts and filters. The result of the execution determines whether the input is redundant, illegal, contract violating, or useful for generating more inputs. The technique outputs a test suite consisting of unit tests for the classes under test. Passing tests can be used to ensure that code contracts are preserved across program changes; failing tests (that violate one or more contract) point to potential errors that should be corrected.

Feedback-directed random testing scales to large systems, quickly finds errors in heavily tested, widely deployed applications, and achieve behavioral coverage on par with systematic techniques. The exchange of ideas between the random and systematic approaches could benefit both communities. The structural heuristics to [1]guide a model checker; the heuristics might also help a random test generator. Going the other way, this notion of exploration using a component set or state matching when the universe contains more than one object could be translated into the exhaustive testing domain. Combining random and systematic approaches can result in techniques that retain the best of each approach.

B. A Theory Of Predicate-Complete Test Coverage And Generation

The goal of *predicate-complete* testing (PCT) to be to cover all *reachable* observable states. The *n* predicates represent all the case-splits on the input that the programmer has identified.1 In the limit, each of the *m* statements may have different behavior in each of the 2*n* possible observable states, and so should be tested in each of these states.[2] It show that PCT coverage subsumes traditional coverage metrics such as statement, branch and multiple condition coverage and that PCT coverage is incomparable to path coverage. PCT groups paths ending at a statement *s* into equivalence classes based on the observable states the paths induce at *s*.

Consider a program with *m* statements and *n* predicates, where the predicates are derived from the conditional statements and assertions in a program, as well as from implicit run-time safety checks. An observable state is an evaluation of the *n* predicates under some state at a program statement. The goal of *predicate-complete* testing (PCT) is to cover every reachable observable state (at most *m* £ 2*n* of them) in a program. PCT coverage is a new form of coverage motivated by the observation that certain errors in a program only can be exposed by considering the complex dependences between the predicates in a program and the statements whose execution they control. PCT coverage subsumes many existing control-flow coverage criteria and is incomparable to path coverage. To support the generation of tests to achieve high PCT coverage, it shows how to define an upper bound *U* and lower bound *L* to the (unknown) set of reachable observable states *R*. These bounds are constructed automatically using Boolean (predicate) abstraction over modal transition systems and can be used to guide test generation via symbolic execution.

C. Nighthawk: A Two-Level Genetic-Random Unit Test Data Generator.

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, the ranges from which integer arguments are chosen, and the manner in which previously- used arguments or previously returned values are used in new method calls. [3]It is often difficult to work out the optimal values of these parameters by hand.

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 settings of certain parameters, such as the relative frequencies with which methods are called. This paper, describe a system that uses a genetic algorithm to find parameters for randomized unit testing that optimize test coverage. It compare the coverage results to previous work, and report on case studies and experiments on system options. 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.[4] it have described Nighthawk, a system in which an upper-level genetic algorithm automatically derives good parameter values for a lower-level randomized unit test algorithm. It have shown that Nighthawk is able to achieve high coverage of complex Java units. The code is available by writing to the first author. Future work 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

III. RELATED WORK

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 it have tested an SUT without forcing failures, the more sure of reliability of the SUT.[8] 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.

- In unit testing, the test coverage has been utilized through the optimal values of the parameters (inputs) handled for the testing.
- Each testing techniques provide a different perspective to improve the time complexity in unit testing.

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 call as 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. This PROJECT describes the Nighthawk unit test data generator. Nighthawk has two levels. 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. 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. 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. [6] The Nighthawk system described here significantly builds on this work by automatically determining good parameters. The lower, randomized testing, level of Nighthawk initializes and maintains one or more value pools for all relevant types, and draws and replaces values in the pools according to a policy specified in a chromosome. Chromosomes specify relative frequencies of methods, method parameter ranges, and other testing parameters. The upper, genetic algorithm, level searches for the parameter settings that increase the coverage seen in the lower level.

The information that Nighthawk uses about the SUT is only information about type names, method names, parameter types, method return types, and which classes are subclasses of others; this makes its general approach robust and adaptable to other languages. Designing a GA means making decisions about what features are worthy of modeling and mutating. For example, much of the effort on this project was a laborious trial-and-error process of trying different types of genes within a chromosome. To simplify that process, [5] it described experiments here with automatic feature subset selection (FSS) which lead us to propose that automatic feature subset selection should be a routine part of the design of any large GA system.

IV. PROPOSED SYSTEM

A. Input Process Module

Input Source code (a set "M "of Java methods). That code Divided into Modules. From that module codes Generate a GA chromosome "c", appropriate to "M", specified by chromosome c.[10]

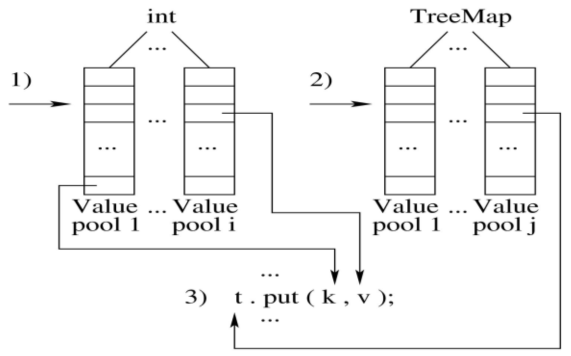

Fig. 1.Value pool initialization and use.

See Fig. 1 for a high-level view of how the value pools are initialized and used in the test case generation process. The algorithm chooses initial values for primitive type pools, before considering non primitive type pools. A constructor method is an initialize if it has no parameters, or if all of its parameters are of primitive types.

B. Chromosome Composition Module

A ***chromosome*** is composed of a finite number of genes. The return value of a method call as quasi-parameters of the method call then Take the space of possible chromosomes as a solution space to search, and apply the GA approach to search it for a good solution. Parameters and quasi-parameters have candidate types

a. receiver candidate type
b. parameter candidate type
c. return value candidate type

Receiver candidate type

A type is a receiver candidate type if it is a subtype of the type of the receiver. These are the types from whose value pools the receiver can be drawn.

Parameter candidate type

A type is a parameter candidate type if it is a subtype of the type of the parameter. These are the types from whose value pools the parameter can be drawn.

Return value candidate type

A type is a return value candidate type if it is a super type of the type of the return value. These are the types into whose value pools the return value can be placed.

C. Pooling Module

Judge a randomly generated GUI test case (*the word "choose" always used to mean specifically a random choice, which may partly depend on "c"*). Value pool initialization (All initializes are also reinitializes).

Define the set "CM" of callable methods to be the methods in "M"

1. Call description
2. Object description

Check correctness of return values and exceptions by providing a generator for ***"test wrapper"*** classes. The generated "**test wrapper"** classes can be instrumented with assertions. Test wrappers can be customized for precondition checking, result checking, or coverage enhancement.

Nighthawk's randomized testing algorithm is referred to as constructRunTestCase, and is described in Fig. 2 .It takes a set M of target methods and a chromosome c as inputs. It begins by initializing value pools, and then constructs andruns a test case, and returns the test case. It uses anauxiliary method called tryRunMethod, which takesa method as input, calls the method, and returns ,a cal ldescription. In the algorithm descriptions, the word"choose" is always used to mean specifically a random choice which may partly depend on c.tryRunMethod considers a method call to fail if and onlyif it throws an Assertion Error. It does not consider other exceptions to be failures since they might be correct responses to bad input parameters.[9] It facilitate checking correctness of return values and exceptions by providing a generator for "test wrapper" classes. The generated testwrapper classes can be instrumented with assertions.

Return values may represent new object instances neveryet created during the running of the test case. If these newinstances are given as arguments to method calls, they maycause the method to execute statements never yet executed.Thus, the return values are valuable and are returned to thevalue pools when they are created.Although it have targeted Nighthawk specifically atJava, note that its general principles apply to any object oriented or procedural language. For instance, for C, it would need only information about the types of parameters and return values of functions, and the types of fields instructs. struct types and pointer types could be treated as classes with special constructors, getters, and setters;functions could be treated as static methods of a singletargetclass.

Algorithm

Input: a set M of target methods; a chromosome **Output**: a test case.

Steps:

1) For each element of each value pool of each primitive type in $I_{M,}$ choose an initial value that is within the bounds for that value pool.
2) For each element of each value pool of each other type in I_{M} :
a) If t has no initializes, then set the element to null.
b) Otherwise, choose an initialize method i of t, and call **tryRunMethod**(i,e). If the call returns a non-null value ,place the result in the destination element.
3) Initialize test case k to the empty test case.
4) Repeat n times, where n is the number of method calls to perform:
a) Choose a target method m € $C_{M.}$
b) Run**tryRunMethod**(m,c).Add the returned call description to k.
c) If **tryRunMethod** returns a method call failure indication, return k with a failure indication.
5) Return k with a success indication.

Fig.2 Algorithm constructRunTestCase.

D. Knowledge Based Pooling Module

Judge a randomly generated GUI test case (*the word "choose" always used to mean specifically a random choice, which may partly depend on "c"*). Value pool initialization (All initializes are also reinitializes).

Define the set "CM" of callable methods to be the methods in "M"

1. Call description
2. Object description

Check correctness of return values and exceptions by providing a generator for ***"test wrapper"*** classes. The generated "**test wrapper"** classes can be instrumented with assertions. Training possible errors then Training possible outputs finally Training possible solutions. Customize the output. Test wrappers can be customized for precondition checking, result checking, or coverage enhancement.

E. Comparison Module

In the Comparison of pooling and Knowledge based first take Testing Result from Randomized Testing then take Testing Result from Enhanced Randomized Testing. It gives Comparison result.

V. SYSTEM ARCHITECTURE

The below figure shows the architectural diagram for this project. First, the general java code has been considered as an input to the system. By applying the genetic algorithm, the inputs have been divided into chromosomes. Since this is the existing system which makes the unit testing for a better efficiency. In this work, by applying Neural network approach, the inputs have been trained and it will be monitored in the test wrapper and customize the output. Finally, after applying this technique, the results being tested.

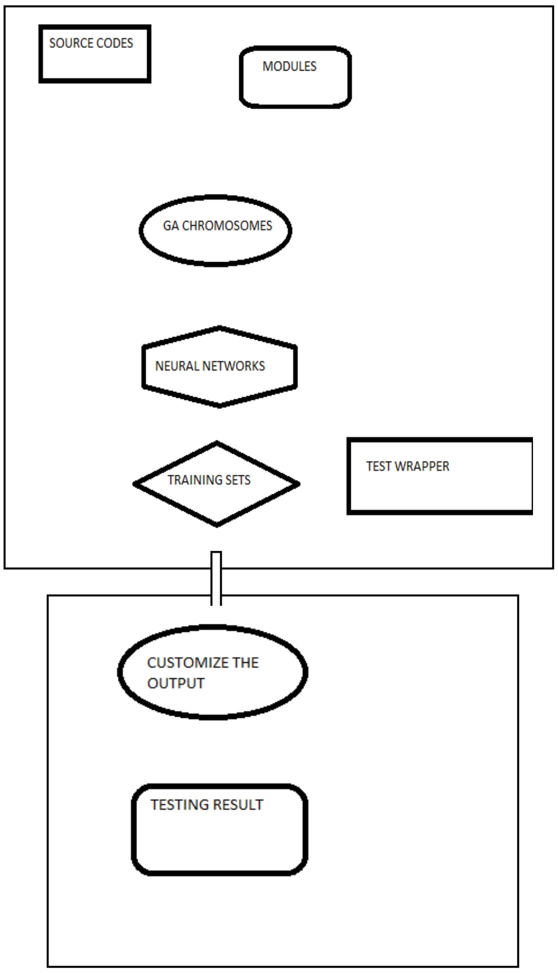

Fig. 3. System Architecture.

VI. NETWORKAPPROACH

A. Neural Networks

Traditionally, the term "neural network" has been used torefer to a network of biological neurons. The modern definitionof this term is an artificial construct whose behavior is basedon that of a network of artificial neurons.[7] These neurons areconnected together with weighted connections following a certainstructure. Each neuron has an activation function that describesthe relationship between the input and the output of theneuron . The data can be processed in parallel by differentneurons, and distributed on the weights of the connections between neurons. Different neural network models have been developed, including BP neural networks , RBF neural networks, self-organizing map (SOM) neural networks ,and adaptive resonance theory (ART) neural networks. A particularly important attribute of a neural network is that it clear from experience. Such learning is normally accomplished through an adaptive process using a learning algorithm. These algorithms can be divided into two categories: *supervised*, and *unsupervised* .

B. Supervised Vs. Unsupervised Learning

From a theoretical point of view, supervised and unsupervised learning differ only in the causal structure of the model. In supervised learning, the model defines the effect one set of observations, called inputs, has on another set of observations, called outputs. [13]In other words, the inputs are assumed to be at the beginning and outputs at the end of the causal chain. The models can include mediating variables bet en the inputs and outputs.

In unsupervised learning, all the observations are assumed to be caused by latent variables, that is, the observations are assumed to be at the end of the causal chain. In practice, models for supervised learning often leave the probability for inputs undefined. This model is not needed as long as the inputs are available, but if some of the input values are missing, it is not possible to infer anything about the outputs. If the inputs are also modelled, then missing inputs cause no problem since they can be considered latent variables as in unsupervised learning.

With unsupervised learning it is possible to learn larger and more complex models than with supervised learning.[11] This is because in supervised learning one is trying to find the connection between two sets of observations. The difficulty of the learning task increases exponentially in the number of steps between the two sets and that is why supervised learning cannot, in practice, learn models with deep hierarchies.

In unsupervised learning, the learning can proceed hierarchically from the observations into ever more abstract levels of representation. Each additional hierarchy needs to learn only one step and therefore the learning time increases (approximately) linearly in the number of levels in the model hierarchy.

C. Feed Forward

The feed forward neural network was the first and arguably simplest type of artificial neural network devised. In this network, the information moves in only one direction, forward, from the input nodes, through the hidden nodes (if any) and to the output nodes. [15] There are no cycles or loops in the network.

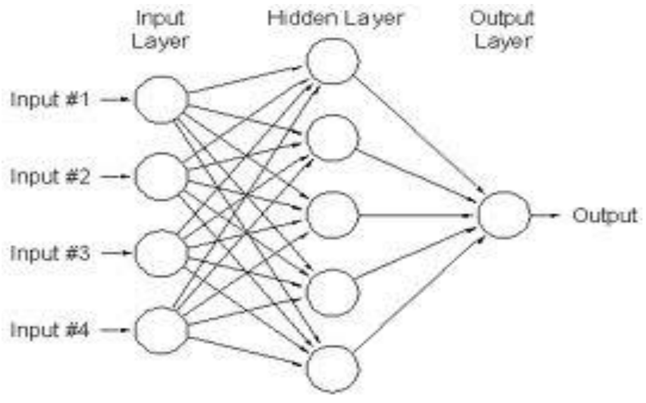

Fig.4 Feed forward neural network

first see how to compute the output of an already learned neural network which means that the entire network (topology, activation functions, weights, and biases) is known at this point. Then to construct a neural network to approximate any given function, but for now just assume that the neural network[10] is given and that it has been trained. It might be helpful to refer to the figure above while reading this section. Recall that the purpose of a feed-forward neural network is to approximate a function of multiple inputs and outputs. The inputs and outputs are continuous (real values). Each input to the neural network is fed as input into a different input neuron. The input neurons are connected to hidden neurons which perform transformations on the input. At the end of the graph are the output neurons. The output of these neurons is defined to be the output of the entire network. It is also possible to connect an input neuron directly to an output neuron [12](without any hidden neurons in between) but that is usually not a good idea because there are no hidden neurons to perform transformations on the data and then it would not be possible to approximate a function accurately.

Therefore the feed forward technique which consider as the best result for the unit testing operation and the result has been shown in the above graph.

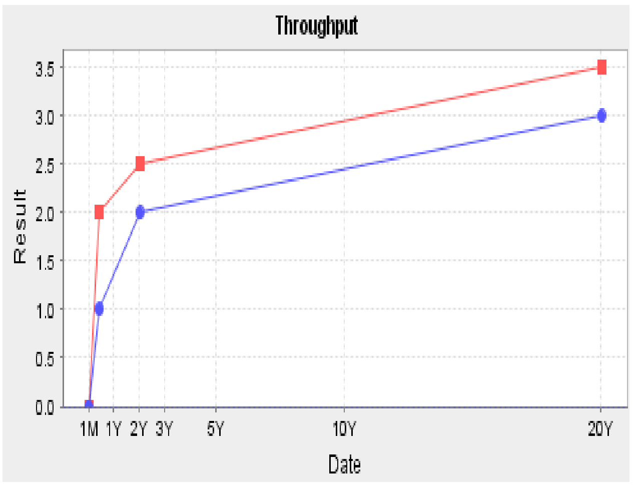

Fig 5: Graphical representation of neural network in unit testing.

VII. 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. In this paper, it have described Nighthawk, a system in which an upper level genetic algorithm automatically derives good parameter values for a lower level randomized unit test algorithm. [14]It have 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. And also shown that it able to optimize and simplify metaheuristic search tools. Metaheuristic tools(such as genetic algorithms and simulated annealers) 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 described here. In this way, FSS can be used to automatically find and remove superfluous parts of the search control. Since by applying neural approach the time complexity and minimization of the test cases can be improved better compared to the existing.

VIII. FUTURE WORK

Future work 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. It also wish to integrate a feature subset selection learner into the GA level of the Nighthawk algorithm for dynamic optimization of the GA. Further, it can see a promising line of research where the cost/benefits of a particular metaheuristic are tuned to the particulars of a specific problem. Here, it have shown that if surrender 10 th of the coverage, it 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 metaheuristics (with or without FSS) to particular problem domains.

REFERENCES

[1] Feedback-directed Random Test GenerationCarlos Pacheco1, Shuvendu K. Lahiri2, Michael D. Ernst1, and Thomas Ball21MIT CSAIL, 2Microsoft Research.

[2] A Theory of Predicate-Complete Test CoverageandGenerationThomasBalltball@microsoft.comMicrosoft Research April 23, 2004.

[3] Nighthawk: A Two-Level Genetic-RandomUnit Test Data Generator. James H. Andrews and Felix C. H. Li Department of Computer Science University of Western Ontario London, Ontario, CANADA N6A 5B7 andrews,cli9@csd.uwo.ca Tim Menzies Lane Department of Computer Science West Virginia University PO Box 6109, Morgantown, WV, USA 26506 tim@menzies.us

[4] Randomized Differential Testing as a Prelude to Formal Verification Alex Groce, Gerard Holzmann, and Rajeev JoshiLaboratory for Reliable Software _Jet Propulsion LaboratoryCalifornia Institute of TechnologyPasadena, CA 91109. USA.

[5] On the Value of Combining Feature Subset Selection with Genetic Algorithms: Faster Learning of Coverage Models James H. AndrewsUniversity of Western OntarioDepartment of Computer ScienceLondon, Ont., Canada, N6A 2B7 andrews@csd.uwo.ca. Tim Menzies Lane Department of CS & EE Morgantown, WV, USA tim@menzies.us.

[6] Randomized Unit Testing: Tool Support and Best Practices James H. Andrews, SusmitaHaldar, Yong Lei and Felix Chun Hang Li Report No. 663 Jan 2006.

[7] Using Neural Networks for Data Mining. Using Neural Networks for Data Mining Mark W.Craven School of Computer ScienceCarnegie Mellon University Pittsburgh PA mark craven_cs cmu edu Jude W Shavlik Computer Sciences Department University of Wisconsin_Madison Madison WI shavlik_cs .wisc .edu

[8] ".Software Testing Fundamentals—Concepts, Roles. and Terminology John E. Bentley, Wachovia Bank, Charlotte NC.

[9] "Genetic Algorithms for Randomized Unit Testing James H. Andrews, Member, TimMenzies, IEEE and Felix C.H.Li.IEEETRANSACTIONSONSOFTWAREENG,VOL.37,NO.1, JAN/FEB2011.

[10] SWAT: A Spiking Neural Network Training Algorithm for Classification Problems John J. Wade, Liam J. McDaid, Jose A. Santos, and Heather M. Sayers IEEE TRANSACTIONS ON NEURAL NETWORKS, VOL. 21, NO. 11, NOVEMBER 2010

[11] Effective Software Fault Localization Using an RBF Neural Network W. Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani Thuraisingham IEEE TRANSACTIONS ON RELIABILITY, VOL. 61, NO. 1, MARCH 2012

[12] P. Rowcliffe and J. Feng, "Training spiking neuronal networks with applications in engineering tasks," *IEEE Trans. Neural Netw.*, vol. 19, no. 9, pp. 1626–1640, Sep. 2008

[13] Semi-Supervised Learning via Regularized Boosting Working on Multiple Semi-Supervised Assumptions Ke Chen, Senior Member, IEEE, and Shihai Wang IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, VOL. 33, NO. 1, JANUARY 2011

[14] Evalution of genetic algorithm for conflict probe testing Computer science at rovanuniversity ,October 19,2011 IEEE Transaction.

[15] Toward the Training of Feed-Forward Neural Networks With the D-Optimum Input Sequence Marcin Witczak IEEE TRANSACTIONS ON NEURAL NETWORKS, VOL. 17, NO. 2, MARCH 2006