Volume 3, Issue 8, August 2013 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

Software Testing Using Soft computing Technique

[1] P. Anil babu, [2] K.Koteswara Rao.
Department of Computer Science and Engineering
GMR Institute of Technology, Rajam, AP, India

***Abstract:** Testing software is essential to ensure quality in IT systems. One of the testing methods is randomized testing which is effective for unit testing. The effectiveness of randomized testing is influenced by certain parameters. One such parameter is frequency of method calls. In this paper we implement a genetic algorithm for finding parameters required by randomized testing. This will help in maximum test coverage. The problem with GA is that the representations in GA are bulky whose size and content has to be reduced. We build a tool which will reduce the representations of GA substantially. This will reduce a great deal of latency while achieving the results same as full representations of GA. We built a prototype application for randomized test case generation. The empirical results are encouraging and the tool can be used in real time applications.*

***Keywords:** Software testing, genetic algorithms, feature selection, randomized testing*

1. Introduction

Software testing is a process of testing the software for correctness. It will help the development team to know the flaws in the system. When certain functional requirement is not met, it can be considered to be a flow. Testing helps in finding and fixing such flaws in software systems. The more faults are discovered the good for the quality of software. Randomization is the characteristics exhibited by randomized testing. Many prior works in the literature such as [1], [2] revealed that the randomized testing is effective for discovering faults in software units. However, its thoroughness has been questioned. For this reason research was carried out on the measures of thoroughness pertaining to randomized testing. Such code coverage measures are useful in finding thoroughness of randomized testing [3]. The Thoroughness of this testing mechanism is based on the way the randomization is applied. For instance it is based on the arguments range, value reuse policy, method calls, returned values and so on. Finding optimal parameters is often difficult task. For unit test data generation, Nighthawk can be used. This tool works in two levels. In the first Level it tests methods based on the parameter values which are given in the form of genes that are part of chromosome. The other level of tool is GA which evaluates fitness function with the help of mutation, selection and combination of chromosomes. There are many tools for GA. One such tool is Nighthawk. Many gene types that are generated by tools like Nighthawk can be pruned. Thus it can achieve performance that reflects in the speed of the test case generation application.

Unit testing is a process of testing a subset of an application. The subset might be a single method or a collection of methods or even a class or module. Such methods to be tested in unit testing are known as target methods. Selecting the methods to be tested randomly is what done by randomized testing. Randomized unit testing has been around in the research circles for many years [3], [4], [5], [6]. Vale reuse is the important point in randomized unit testing. It does mean that the test engine reuses arguments, return values and method calls. Earlier value reuse was limited to only method calls that reoccur. In this paper we implemented a tool for randomized testing which makes use of GA for parameter selection and other mechanism for pruning the parameter size and content thus making it much faster than general GA representations. The parameter finding involves method names, arguments, return types classes and sub classes and so on. The remainder of this paper is structured into the following sections. Section 2 reviews literature. Section 3 descries the proposed approach for randomized test case generation and information about prototype implementation. Section 4 presents experimental results while section 5 concludes the paper.

2. Prior Work

Randomized testing has been around for the fast 30 years and it is effective as it can generate plenty of test cases in less time [7], [8]. This will ensure that it can generate test cases that can uncover flaws in the software under test. To achieve this test oracle [9] is required. For this problem can be solved in two ways. The first type are known as “high-pass” oracles while the second class is meant for properties specific to software under test [10], [11]. No intelligence is associated with randomized unit testing. In [5] triangle problem is used for randomized testing. They made a conclusion that could not achieve 50% of coverage in code in spite of 1000 runs in case of triangle problem. Visser et al. [3] performed comparison between model-checking approaches and also randomized unit testing and concluded that randomized testing is better than the model checking approaches. Many researchers observed that the thoroughness of randomized testing is based on the implementation. Randomized sequences of method calls are used by Doong and

Frankl et al. [4]. By changing parameters they could find the faults in software effectively. Java Vector class was tested by Antoy and Hamlet [12] against a formal specification for finding faults. Another fact found by Andrews and Zhang [13] through randomized testing is that the more parameters and ranges can discover more faults in software. Randomized breadth-first search can be used to enhance randomized testing [1]. In addition to this pruning branches which are not required also can help improve the performance of randomized testing.

Out of all the approaches the approach followed by Pacheco et al. is very much similar to this paper. The main difference is that in this paper long test sequences are used to achieve thoroughness while in that paper uses shorter sequences. They focused on contracts for units while we gave importance to code coverage as it is the best measure for thoroughness. Test data generation is important in randomized test case generation. Generating test data through symbolic execution has been around since 1976 [14], [15]. A thorough set of test cases can be generated by these approaches. There is one tool named TESTGEN [16] transforms each condition in the program to one of the forms such as e<=0 or e<0 thus the condition is reduced to either true or false. Iterative relaxation of constraints is explored in [17] on given input data. In [18] goal directed reasoning is sued to generate test sequences. Model checkers are also used recently in Java using a tool such as Pathfinder [19]. These tools sometimes use lossy randomized search in order to find paths in the CUTE and DART systems [20], [21]. Range of different condition is limited by some approaches such as TESTGEN. However, it can't be used to find paths where pointers are involved.

Genetic algorithms are also used in randomized testing. The GAs is explored in [22], [23], [24], [25] and [26]. The candidate solutions in case GAs are represented as chromosomes. Chromosome concept is used by Tonella [27] for class testing. It also uses mutation operators that are customized for inserting new invocations. A genetic random test generation system was proposed in [28] and named it as Nighthawk which will do its job in two steps. In the first step it generates parameters using GA and in the second step it reduces the parameter size and content to make it much faster.

2.1 Motivation behind the work

Testing is an essential aspect of software development. Testing helps in finding and fixing bugs. Automatic testing of software applications is important besides manual testing. Traditionally testing has been given important in which various kinds of testing are available. They include black box testing, white box testing and so on. However, automatic test case generation that has maximum structural coverage is a challenging problem. The existing solutions use many techniques like Random Walk, Hill Climbing, and Genetic Algorithms and so on. However, the GA will produce parameters that can be used to generate test case for maximal structural coverage. Nevertheless, it is essential to have the parameters pruned in order improve performance further. This is the motivation behind this project which will prune the parameters generated by GA. The parameter pruning is done using the logic which is similar to the functionality of FSS tool. This will help to improve the performance of algorithm without causing the coverage problem.

2.2 problem statement

Testing has been around since the inception of software development. However, initially testing was carried out in trial and error basis. Later on it was given much importance in software development life cycle. However, generating automated test cases has many challenges and an open problem to be addressed. Many researchers suggested solutions for the problem. However, in this paper we focus on the GA (Genetic Algorithm) usage for the purpose of generating test cases. The genetic algorithm is best used to find parameters based on which test cases are generated. However, there is possibility of pruning the parameters generated by GA so as to improve the process of test case generation. For this reason it is essential to use a tool such as FSS or its equivalent solution in order to generate test cases efficiently without compromising the coverage software under test. In this paper the problem of parameter pruning is made using a custom built logic while the GA is used to find out parameters required for generating test cases.

3. Proposed Approach to Randomized

The methodology used to generate test cases is presented here. An application is built that takes software under test as input. First of all a Genetic Algorithm is used to extract parameters that can be used to generate test cases in such a way that the generated test cases can cover the SUT fully. However, there is scope for pruning parameters so as to improve the performance of test case generation without the scarifying the structural coverage of the SUT. To reiterate this, the steps followed include:

1. Find out parameters using GA for test case generation.
2. Prune the parameters using FSS or equivalent tool.
3. Generate test cases for given SUT that have full coverage.

3.1 Test Case Generation

We use genetic algorithm for randomized testing. Andrews et al. [29] developed a tool known as Nighthawk. The algorithms presented by them are used in our Genetic algorithm as well. They used FSS (Feature Subset Selection) tool for pruning parameters. However, we have implemented out own logic for pruning part. The pruning is performed for reducing size and content of representations of GA. This will reduce test cases and still achieve maximum coverage. The schematic overview of the proposed architecture is as shown in fig. 1.

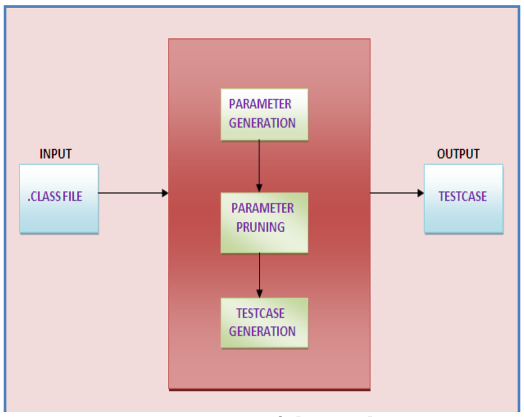
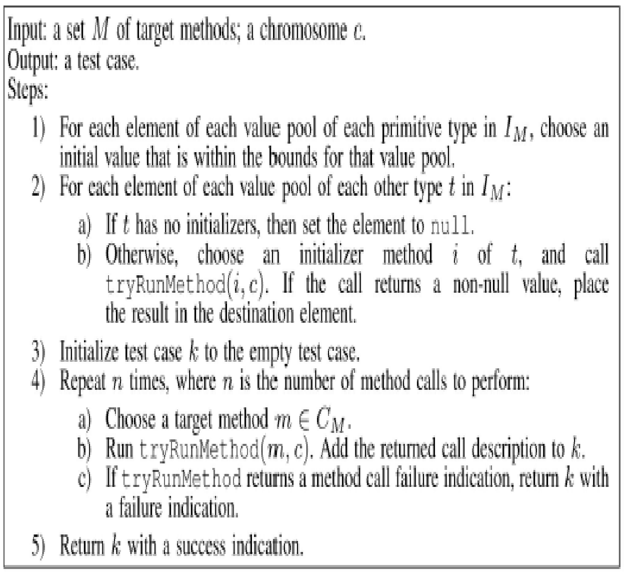


Fig. 1 – Overview of the Architecture

As can be seen in fig. 1, the proposed architecture overview is shown. It takes .class file as input and performs operations such as parameter generation, parameter pruning and test case generation. The parameters identified by the GA are pruned by the custom logic built by us. After pruning with reduced parameters, the application generates test case which will have optimal coverage of the given class.

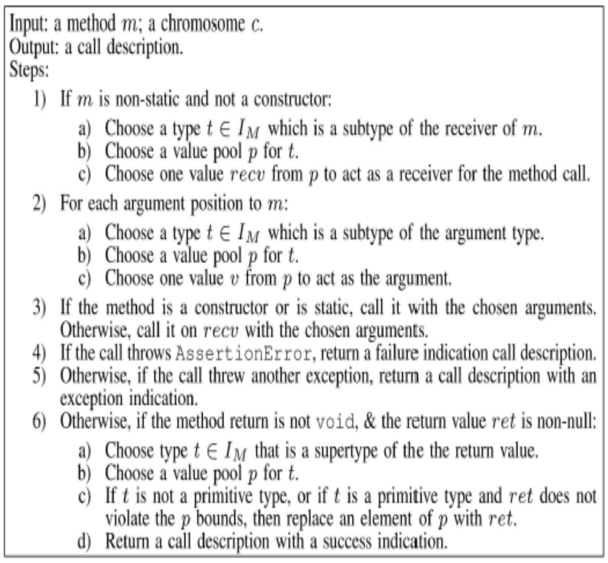
3.2 Algorithms Used



```
Input: a set $M$ of target methods; a chromosome $c$.
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 $t$ in $I_M$:
       a) If $t$ has no initializers, then set the element to null.
       b) Otherwise, choose an initializer method $i$ of $t$, and call
          tryRunMethod($i, c$). 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 $m \in C_M$.
       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 for Randomized Test Case Generation (excerpt from [13])

As can be seen in fig. 2 the randomized test case generation algorithm takes a set of target methods and a chromosome as input and generates test case. In the process of generating test case it invokes algorithm presented in fig. 4 for obtaining a method call description.



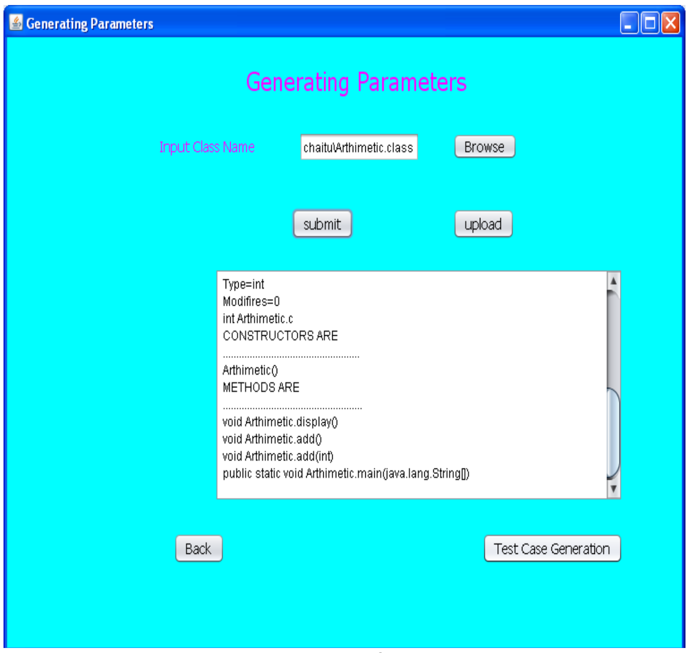
```
Input: a method $m$; a chromosome $c$.
Output: a call description.
Steps:
  1) If $m$ is non-static and not a constructor:
       a) Choose a type $t \in I_M$ which is a subtype of the receiver of $m$.
       b) Choose a value pool $p$ for $t$.
       c) Choose one value $recv$ from $p$ to act as a receiver for the method call.
  2) For each argument position to $m$:
       a) Choose a type $t \in I_M$ which is a subtype of the argument type.
       b) Choose a value pool $p$ for $t$.
       c) Choose one value $v$ from $p$ to act as the argument.
  3) If the method is a constructor or is static, call it with the chosen arguments.
     Otherwise, call it on $recv$ with the chosen arguments.
  4) If the call throws AssertionError, return a failure indication call description.
  5) Otherwise, if the call threw another exception, return a call description with an
     exception indication.
  6) Otherwise, if the method return is not void, & the return value $ret$ is non-null:
       a) Choose type $t \in I_M$ that is a supertype of the the return value.
       b) Choose a value pool $p$ for $t$.
       c) If $t$ is not a primitive type, or if $t$ is a primitive type and $ret$ does not
          violate the $p$ bounds, then replace an element of $p$ with $ret$.
       d) Return a call description with a success indication.
```

Fig. 3 – Algorithm to get Call Description (excerpt from [13])

As can be seen in fig. 3, the algorithm is used to obtain a method call description. It takes a method and chromosome as input and returns call description with success or exception indication.

3.3 Prototype Implementation

A prototype application is built to demonstrate the proof of concept. The application is meant for testing Java applications. It takes a Java class as input and generates randomized test cases. The main application UI is as shown in fig. 5.



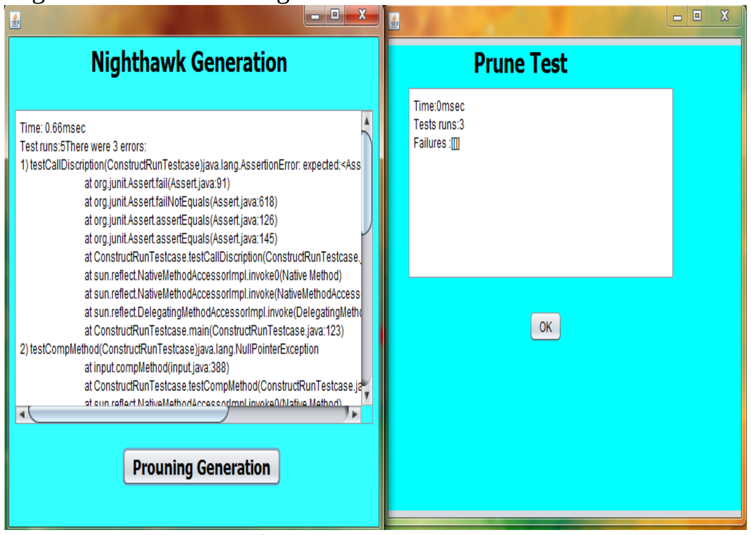
```
Type=int
Modifires=0
int Arthimetic.c
CONSTRUCTORS ARE
..................................................
Arthimetic()
METHODS ARE
..................................................
void Arthimetic.display()
void Arthimetic.add()
void Arthimetic.add(int)
public static void Arthimetic.main(java.lang.String[])
```

Fig. 4 – The main UI of the application.

As can be seen in fig. 4, the application allows the user to choose a .class file for which test cases are to be generated. First of all it generates GA representations. Then the GA representations are pruned in order to reduce the size and content for performance improvement. Instead of using FSS tool, we have built custom logic in order to prune the parameters. After pruning the application facilitates random test case generation.

4. Experimental Results

The environment used for the implementing the proposed prototype for randomized test case generation is a PC with 4GB RAM, Core 2 Dual processor running Windows 7 OS. For application development Java programming language is used. Net Beans IDE is used for rapid application development. The difference between nighthawk test case generation and test case generation using FSS is as shown in fig. 6



```
Time: 0.66msec
Test runs:5There were 3 errors:
1) testCallDiscription(ConstructRunTestcase)java.lang.AssertionError: expected:<Ass
        at org.junit.Assert.fail(Assert.java:91)
        at org.junit.Assert.failNotEquals(Assert.java:618)
        at org.junit.Assert.assertEquals(Assert.java:126)
        at org.junit.Assert.assertEquals(Assert.java:145)
        at ConstructRunTestcase.testCallDiscription(ConstructRunTestcase.
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccess
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMeth
        at ConstructRunTestcase.main(ConstructRunTestcase.java:123)
2) testCompMethod(ConstructRunTestcase)java.lang.NullPointerException
        at input.compMethod(input.java:388)
        at ConstructRunTestcase.testCompMethod(ConstructRunTestcase.ja
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
```

```
Time:0msec
Tests runs:3
Failures :[]
```

Fig. 6 – Experimental results for randomized unit testing using nighthawk & FSS.

5. Conclusions and Future Work:

Randomized unit testing is one of the testing methods which is proved to be effective. However, its perfectness depends on the parameters of the testing algorithm. In this paper we have designed a mechanism that is meant for finding best parameter values. This mechanism works in two levels. In the first phase GA finds parameters to be focused on while the second phase is meant for pruning the size and content of the GA representations are reduced in order to improve the performance of the randomized testing mechanism. We also implemented a prototype application that demonstrates the proof of concept. The empirical results revealed that the proposed randomized testing mechanism is effective and proves the hypothesis that the effectiveness or thoroughness of randomized testing depends on the settings of certain parameters. In future we explore the combination of our approach and other testing techniques for maximum test coverage effectively.

References:

[1] C. Pacheco, S.K. Lahiri, M.D. Ernst, and T. Ball, "Feedback-Directed Random Test Generation," Proc. 29th Int'l Conf. SoftwareEng., pp. 75-84, May 2007.

[2] A. Groce, 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.

[3] W. Visser, C.S. P_as_areanu, and R. Pela´nek, "Test Input Generation for Java Containers Using State Matching," Proc. Int'l Symp.Software Testing and Analysis, pp. 37-48, July 2006.

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

[5] S. Antoy and R.G. Hamlet, "Automatically Checking an Implementationagainst its Formal Specification," IEEE Trans. SoftwareEng., vol. 26, no. 1, pp. 55-69, Jan. 2000.

[6] K. Claessen and J. Hughes, "QuickCheck: A Lightweight Tool forRandom Testing of Haskell Programs," Proc. Fifth ACM SIGPLANInt'l Conf. Functional Programming, pp. 268-279, Sept. 2000.

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

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

[9] E.J. Weyuker, "On Testing Non-Testable Programs," The ComputerJ., vol. 25, no. 4, pp. 465-470, Nov. 1982.ANDREWS ET AL.: GENETIC ALGORITHMS FOR RANDOMIZED UNIT TESTING 93

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

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

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

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

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

[15] J.C. King, "Symbolic Execution and Program Testing," Comm.ACM, vol. 19, no. 7, pp. 385-394, 1976.

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

[17] N. Gupta, A.P. Mathur, and M.L. Soffa, "Automated Test DataGeneration Using an Iterative Relaxation Method," Proc. Sixth Int'lSymp. Foundations of Software Eng., pp. 224-232, Nov. 1998.

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

[19] W. Visser, C.S. P_as_areanu, and S. Khurshid, "Test Input Generationwith Java PathFinder," Proc. ACM/SIGSOFT Int'l Symp.Software Testing and Analysis, pp. 97-107, July 2004.

[20] P. Godefroid, N. Klarlund, and K. Sen, "DART: DirectedAutomated Random Testing," Proc. ACM SIGPLAN Conf. ProgrammingLanguage Design and Implementation, pp. 213-223, June2005.

[21] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit TestingEngine for C," Proc. 13th ACM SIGSOFT Int'l Symp. Foundations ofSoftware Eng., pp. 263-272, Sept. 2005.

[22] J.H. Holland, Adaptation in Natural and Artificial Systems. Univ. ofMichigan Press, 1975.

[23] D.E. Goldberg, Genetic Algorithm in Search, Optimization, andMachine Learning. Addison-Wesley, 1989.

[24] L. Rela, "Evolutionary Computing in Search-Based SoftwareEngineering," master's thesis, Lappeenranta Univ. of Technology,2004.

[25] R.P. Pargas, M.J. Harrold, and R.R. Peck, "Test-Data GenerationUsing Genetic Algorithms," J. Software Testing, Verification andReliability, vol. 9, pp. 263-282, Dec. 1999.

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

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

[28] J. Andrews, F. Li, and T. Menzies, "Nighthawk: A Two-LevelGenetic-Random Unit Test Data Generator," Proc. 22nd IEEE/ACMInt'l Conf. Automated Software Eng., http://menzies.us/pdf/07asenighthawk.pdf, 2007.

[29] James H. Andrews, Member, IEEE, Tim Menzies, Member, IEEE, and Felix C.H. Li, "Genetic Algorithms for Randomized Unit Testing", IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 37, NO. 1, JANUARY/FEBRUARY 2011.