Software Testing Cost Reduction with Genetic Algorithms and Neural Networks

A. Watkins
College of Business
University of South Florida St Petersburg
140 7th Avenue South
St Petersburg, Florida 33701

Sergio Davalos
Milgard School of Business
University of Washington Tacoma
1900 Commerce St
Tacoma, WA 98402

***Abstract*—Highly complex and interconnected systems may suffer from intermittent or transient software failures. These are particularly difficult to diagnose without large quantities of test cases. This research focuses on a hybrid method for generating test cases. A genetic algorithm is first used to automatically generating large numbers of test cases to form a comprehensive test suite. These test suites are then used to train a neural network for regression testing and test suite augmentation. The results indicate that the genetic algorithm can produce a balanced test suite that, when combined with a neural network, can reduce the costs of software testing by reducing system run-time and human interaction.**

***Keywords*— Software test data generation, genetic algorithms, neural networks, software testing component**

I. INTRODUCTION (*HEADING 1*)

In August 2003, a software flaw in an energy management system caused an alarm to fail in a regional control center. This set off a chain reaction that left many parts of the northeastern U.S. shrouded in darkness and cost the City of New York alone close to $1 billion in losses [1]. *Ex post* analysis of this widespread blackout indicates that an alarm system failure led to a delay in bringing up a back-up system, which subsequently failed when the large number of unprocessed system events created during the down-time overflowed the process input buffer [2]. Although this particular system failure drew worldwide attention due to its effects on business, government, and everyday lives, it was not an isolated incident. Outages due to software failures have had an impact on every industry and inconvenienced millions of users worldwide.

Complex systems such as electric grids, telecommunication exchanges and military support systems are systems composed of systems created from component parts that may or may not have been designed to work together. When tightly linked or coupled, reciprocal actions among the various elements of the overarching system can lead to failures that are both unpredictable and catastrophic [3]. As our systems grow in scale and rely increasingly on new technology, risk reduction via more comprehensive and thorough testing is imperative at every stage of the development lifecycle, including after every system update. Unfortunately this isn't always the case, as the Royal Bank of Canada that a programming error in a routine update caused a delay in the processing of deposits, withdrawals and payments [4,5].

There are many techniques for system testing, one of which involves generating suites of test data and noting anomalies as data is run through the system. Sometimes, however, after system changes it may be impossible to generate test data for both legacy and new systems due to platform changes or because of the high cost of running parallel systems for testing. The research presented here addresses this issue by focusing on techniques for generating test suites that can be used to model a system therefore reducing costs in parallel system runs. This paper is an extension to earlier work completed at a unit testing level that used genetic algorithms (GA) to generate test suites rich in failure causing test cases [6] and to localize failures to facilitate system-level debugging [7]. The focus of this current work is to generate test suites to train neural networks (NN) to act as surrogates for the original (unchanged) system during regression testing. In essence, the function of the NN is to augment the test suite with test cases that highlight any adverse consequences resulting from the system changes.

This paper is structured as follows: first, is a review of the relevant literature on software testing, with a focus on performance and regression testing issues, and places the work in context. Next, the system under test is described and the GA is used to generate test suites for this system. Finally, these test suites are used to train a NN to act as an oracle during systems testing and again for regression testing.

II. SOFTWARE TESTING AT A SYSTEM LEVEL

Testing at a systems level is less concerned the inputs to the system and is instead focused on workloads and physical resources [8] which is in other terms, the behavior or performance of the system given some state [9]. As such, it is black-box or behavioral testing [10]. Although there is little agreement as to just which system behaviors to test, a review of the relevant literature provides a number of strategies (see

[7, 11, 12, 13]). These strategies include timing issues, network and tier loads, elapsed running time, as well as cyclical patterns such as month and day of the week. Test cases are generated that consist of input parameters along with specific system environmental attributes. The resulting set of test cases forms the test suite.

Even after initial testing and delivery of the system to the end user, the responsibility of the developer is not complete. No system exists without upgrades and each modification of the system requires the retesting of the system to confirm that any changes have not adversely affected the system. This type of testing, regression testing, raises a number of issues that are distinct from those associated with initial system testing, such as strategies for generating new test cases that examine the modifications, selecting a subset of existing test cases to execute the modified program, and creating a new suite for use on future versions of the software [14].

To ascertain the effect of changes to the system, newly generated test cases are typically executed on both the old and new systems. In some cases, however, the original system may be inaccessible or it may be too costly to run both the new and old versions [15]. Nevertheless, given access to the original test suite, it is possible to model the behavior of the old system using a NN. For example, Anderson et al. [20] successfully used a NN to automatically classify fault severity by training the NN to recognize whether a test case would produce a failure; if so, only then would the test case be executed on the target system. The work of Aggarwal et al. [17] took this a step further using a NN as a test oracle. However the relatively low accuracy rate reported of 15.9% suggests further work is needed in this area. It appears that a key issue may be how the test suite is generated, the authors of [20] used gradient analysis, while [17] generated test suites randomly.

Clarke et al [18] argues that metaheuristics such as simulated annealing, GAs and tabu search should be applied to software engineering problems such as test data generation. (An extensive survey of these testing techniques can be found in [19] while [20] provides details of fitness function construction and [6] provides background on the use of GAs for system level testing.) Ahmed et al. [17] used a GA approach focused on a multiple paths generator. Srivastava et al. [22] following in this vein focused on the most critical paths. This enables better performance but can result in local optima.Premal B. Nirpal et al [23] also used this method with good results. Khamis et al. [24] used a reduced set approach. However, the reduced set can result in out of sample errors. McCart et al. [25] examined three techniques for improving the performance of GA test generation. In this research, we take a different approach,combining neural network with GA generated test suites. Comparisons of GA generated suites to those generated randomly have shown in many cases that the GA can produce a better suite for testing purposes – the question this research seeks to answer is whether GA generated test suites are better for training NNs on complex systems.

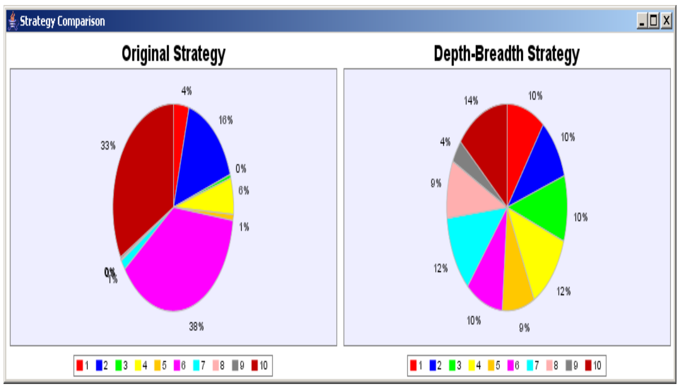
III. THE EXPERIMENTAL SYSTEM - LOBNET

The failure patterns of complex systems are difficult to diagnose because the system can fail in unexpected ways due to the interaction of the components. The complex system used in this research is a lab-built, multi-tiered, distributed system for targeting ballistic weapons called LOBNET. LOBNETwas built to mirror the distributed systems that characterize evolving network-centric warfare components. The system has 40 input parameters and a test case for the system includes the input parameters plus thirteen environmental attributes that focus on prolonged periods of execution, excessive load factors, and repetitive failures over time [for a full list of the environmental attributes see 6]. In order to investigate intermittent failures in the system, ten exceptions of varying complexity were injected using the system environmental attributes and input parameters. Each of these throws an exception code (coded 1-10, with 0 indicating a successful execution) and Table I gives the percentage of the search space covered by each exception. Exceptions are not necessarily errors in code that need to be fixed but rather represent system states that may require attention. The solution in some situations may require changes to the code; for example, a load-balancing algorithm may need to be changed, the hardware might need to be upgraded, or the system specification and/or expectations might need to be revised [11].

TABLE I. EXCEPTION NUMBER AND PERCENT OF SEARCH SPACE COVERED BY EXCEPTION.

Exception Number	% of Search Space
1	1.19%
2	2.38%
3	0.22%
4	1.63%
5	0.05%
6	2.20%
7	2.00%
8	0.06%
9	0.08%
10	0.15%

A GA is used to generate a test suite for testing LOBNET that contains test cases which explore the exception-producing regions of the code. In addition, the suite should be evenly balanced over all exceptions – that is, have a comparable number of test cases representing each exception. The GA uses a failure pursuit (FP) fitness function to guide its search. In the FP approach, the notion of goodness of fit is relative rather than absolute because it evolves with the population, enabling subsequent generations to take advantage of any insights gleaned in previous ones [6]. To accomplish this, a fossil record (essentially a chronological database) was established to capture all previously generated test cases and information about the type of failure generated, if any, when each test case was run. In addition, the search strategy of the GA was devised in such a way that it uses a breadth first strategy to give a higher reward to novel test cases when there are too few different error causing cases and a depth first strategy to build up failure cases when there are too few of a specific error. The result is a much better balanced test suite as shown in Figure 1, the chart on the left shows an example



distribution using a FP strategy without the breadth-depth strategy while on the right incorporates FP with a breadth-depth search. Suites generated by the FP-GA, as well as an equivalent number generated randomly, were used by the NN to create models of LobNet.

Fig. 1.A comparison of the fitness function that emphasized a depth-first approach (left) to a revised strategy (right) that emphasized depth and breadth. The depth-breadth strategy generates more test cases for all the exceptions instead of in-depth coverage of just a few.

IV. NEURAL NETWORK AS A SYSTEM MODEL

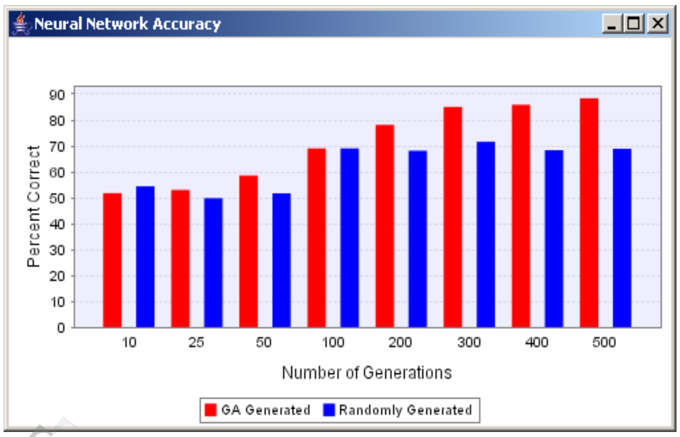
The literature identifies two different applications for NNs within the software testing lifecycle: as an oracle for the system and as a surrogate for the original system in regression testing. In the following subsections, a NN is trained to determine whether it is a viable strategy for system level testing in both these approaches.

A. Using a NN as an Oracle

Aggarwal et al. [17] use the NN as a test oracle that gives the correct output for a test case. This output is compared to the actual system output. Its purpose is to replace the human who must verify each test case to determine if the actual output matches the expected output. Their research investigated only a small unit testing problem, but highlighted an interesting dilemma: in order for the neural network to be used as an oracle, enough test cases must be generated and executed in the target system to train the network. LobNet is a much larger system and it was necessary to determine how large the test suite must be for accurately predicting the outcome of a new test case.

We use test suites of the following sizes: 600 (10 generations), 1500 (25 generations), 3,000 (50 generations), 6,000 (100 generations), 12,000 (200 generations), 18,000 (300 generations), 24,000 (400 generations), and 30,000 (500 generations) to train the NN. After training, the NNs were tested on a separately generated suite and the percentage of correctly classified test cases was recorded. At 18,000 test cases, the accuracy at classifying exception and non-exception test cases was 85%, and a high of 88% was reached at 30,000 generations. In comparison, randomly test generated suite only reached a high of 71% for 18,000 test cases.

The random suites training accuracies were lower because they contained higher percentages of non-exception test cases. This makes this method less effective at classifying exception cases. For example, for test suites with 30,000 cases, the GA-generated suite was able to classify correctly 80% of the



exception causing cases, while the randomly generated suite only classified 25% of these cases correctly. Therefore, larger GA generated test suites are better suited for training the NN than the randomly generated ones. Fig. 2 presents The accuracy percentages indicate how many test cases the model correctly classified – for example at 500 generations the GA generated test suite is 88% accurate while the randomly generated is 69% accurate.

Fig. 2.A comparison of GA-generated and randomly generated test suites when modeled with a neural network. The accuracy percentages indicate how many test cases the model correctly classified – for example at 500 generations the GA generated test suite is 88% accurate while the randomly generated is 69% accurate.

B. Regression Testing

Regression testing confirms that system modifications have not aversely affected the system operations. However, running two systems side by side for comparison purposes is not always feasible whether because time is short, prohibitive costs, or the original system is no longer available. In the following experiments, it is assumed that the original system is unavailable for side-by-side testing, but that the test suite from the original system test is accessible. A model for regression testing using GAs and NNs is shown in Fig. 3.

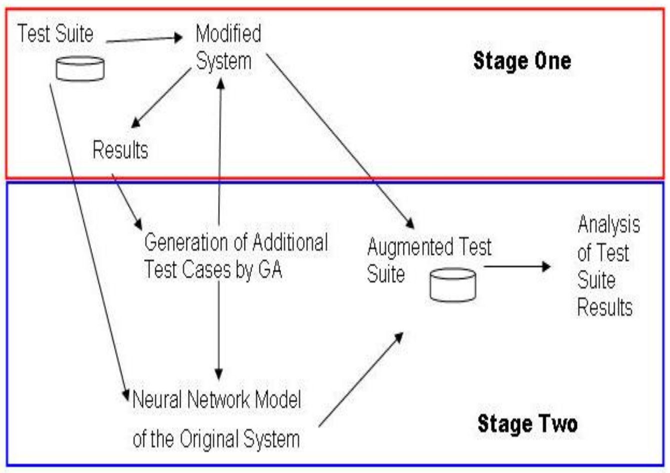

Test Suite
Modified System
Stage One
Results
Generation of Additional Test Cases by GA
Augmented Test Suite
Analysis of Test Suite Results
Neural Network Model of the Original System
Stage Two

Fig. 3. The regression testing model is broken into two stages; the second stage uses the NN as a surrogate model of the original system during the test suite augmentation process. The augmented test suite is analyzed for modification changes and discrepancies reported.

The model has two linked stages; the first involves the execution of the existing test suite on the modified system. The goal is to determine whether the system modifications caused any change to the case classification. For example, determining whether a case that previously caused an exception is now a successful. The test suite augmentation process in stage two begins upon completion of stage one,. The goal is to build a body of evidence of these changes for assessment by the system designers and developers.

A NN model of the original system is first built using the existing test suite, and, then, as each test case is generated, it is executed on the modified system and the NN model. The aim is to generate more test cases that have the following characteristics: first, those that are successful in the model but cause exceptions in the revised system; and second, those that caused an exception but are now successful. This means that the fitness function of the GA must put a greater emphasis on these characteristics. Therefore, the fitness evaluation includes a severity scale where new exceptions are rewarded more than existing ones. Using this scale, the augmented test suite will consist of more test cases that highlight system changes.

To test the revised fitness function, the original LOBNET system was 'fixed', that is one of the exceptions thrown by the system was removed, but when the exception was 'fixed' a new exception was created. The existing test suite is executed on the modified system and the exceptions found are recorded with their exception type, this revised suite is then used to create a NN model of the original systems. Using the NN as a surrogate for the original system, additional test cases are generated by the GA (although it is possible to seed the first population of the GA with test cases that raised queries). This new test suite augments the original but the bulk of its test cases are in the specific modified regions of the system but still engages in some exploration.

Using this technique it is now possible to pinpoint the causes of the exceptions, fix them, and retest the system. Fig. 4 and Fig. 5 show the test case distribution before and after augmentation, Fig. 4 shows a number of test cases generated for the error region and Fig. 5 demonstrates the greater amount of additional test data that can be generated for a specific exception region when combining the oracle and the new system.

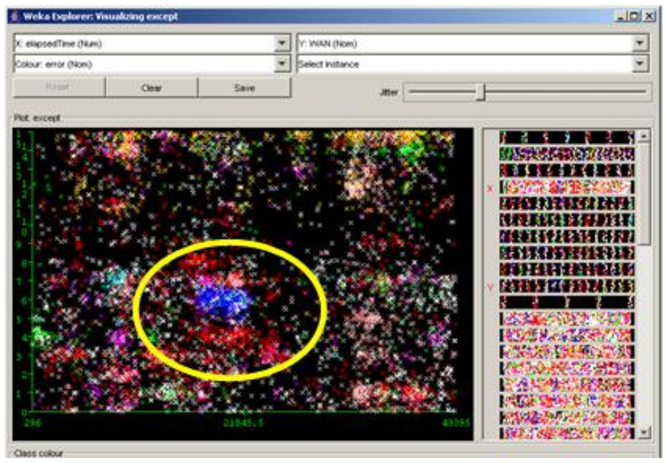


Fig. 4. The distribution of test cases prior to regression test suite generation. While there are many cases within the error region need further augmentation.

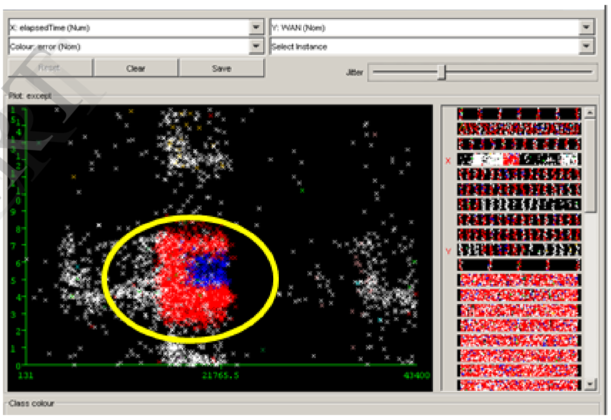


Fig. 5. The distribution of test cases after augmentation. Many more test cases are now within the new error region – indicated by the circle.

V. CONCLUSION

This paper demonstrates that the GA can be used to generate failure-rich test suites for system level testing. Using these test suites, NNs are able to build a model of the system that can be used as either a system oracle or for regression testing.Other generation techniques, for example simulated annealing and tabu search, could be used as well and future research could compare these results.

Computationally intelligent techniques can assist software developers in the testing of complex, distributed systems. Exception causing test cases are not necessarily system failures, but unanticipated system limitations, uncovered only during the testing process. We investigated the use of a neural network in the software testing process. The GA and the NN model performed well during regression testing and were successful in augmenting the test suite with test cases that focused on system modifications. The NN is able to model the system under test, but an important consideration before use in other environments will be the risk tolerance level of the system.

REFERENCES

[1] D. Barrett, "Feds deny New York's request for more blackout aid, capping relief at $5 million," The Associated Press State and Local Wire. November 7, 2003.

[2] U.S.- Canada Power Systems Outage Task Force Final Report., 4/1/2004.

[3] C. Perrow, Normal Accidents. Princeton University Press, New Jersey, 1999.

[4] Computerworld, 6/7/2004,

[5] J. Langton, "Royal Bank's Pain has Other Canadians Wary," The American Banker, 6/14/2004

[6] A. Watkins, E. Hufnagel, D. Berndt, and L. Johnson, "Using Genetic Algorithms and Decision Tree Induction to Classify Software Failures," International Journal of Software Engineering and Knowledge Engineering, 16(2), 2006, 269-291.

[7] A. Watkins, D. Berndt, J. Fisher, L. Johnson, J. Pinglikar, and K. Aebisher, "Breeding Software Test Cases for Complex Systems," Proc. 37th Hawaii International Conf. on Systems Sciences, 2004, pp. 90303c.

[8] G. Denaro, A. Polini, and W. Emmerich, "Early Performance Testing of Distributed Software Applications," Proc. of WOSP, 2004, p 94 – 102.

[9] J. Musa, and A.F. Ackerman, "Quantifying software validations: when to stop testing?" IEEE Software, 6(3), 1989, 19-27.

[10] B. Beizer, Black-box testing: techniques for functional testing of software and systems. John Wiley & Sons, New York, 1995.

[11] E. Weyuker, "Testing component-based software: A cautionary tale," IEEE Software, 15(5), 1998, 54-59.

[12] E. Weyuker, and F. Vokolos, "Experience with performance testing of software systems: Issues, an approach, and case study," IEEE Transactions on Software Engineering, 26(12), 2000, 1147 – 1156.

[13] J. Wegener, and M. Grochtmann, "Verifying timing constraints of real-time systems by means of evolutionary testing," Real-time Systems, 15, 1998, 275-298.

[14] P. Frankl, G. Rothermel, K.Sayre, and F.Vokolos, "An Empirical Comparison of Two Safe Regression Test Selection Techniques," Proc. 2003 International Symposium of Empirical Software Engineering, 2003.

[15] L. White, V. Narayanswamy, T. Friedman, M. Kirschenbaum, and P.Piwowarski, M. Oha, "Test Manager: A Regression Testing Tool,"*Proc Conf. on Software Maintenance*, 1993, 338-347.

[16] C. Anderson, A. Mayrhauser, and R. Mraz, "On the Use of Neural Networks to Guide Software Testing Activities,"*ProcInternational Test Conference*, 1995.

[17] K. Aggarwal, Y. Singh, A. Kaur, and O. Sangwan, "A Neural Net Based Approach to Test Oracle,"*ACM Software Engineering Notes*, *29*(4), 2004, 1-6.

[18] J. Clarke, J.J.Solado, M. Harman, R. Hierons, B. Jones, M.Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd, "Reformulating software engineering as a search problem,"*IEE Proceedings – Software*, *150*(3) 2003; 161-175.

[19] P. McMinn, "Search-based software test data generation: a survey,"*Software Testing, Verification and Reliability*, *14*, 2004, 105-106.

[20] A. Watkins, and E.M. Hufnagel, "Evolutionary Test Data Generation: A Comparison of Fitness Functions, *Software: Practice and. Experience,36*(1), 2006, 95-116.

[21] Moataz A. Ahmed, and Irman Hermadi, GA-based multiple paths test data generator, Computers & Operations Research, 35 (2008) 3107 – 3124

[22] Praveen Ranjan Srivastava and Tai-hoon Kim,"Application of Genetic Algorithm in Software Testing,"International Journal of Software Engineering and Its ApplicationsVol. 3, No.4, October 2009 87-95

[23] Premal B. Nirpal, and K. V. Kale, "Using Genetic Algorithm for Automated Efficient Software Test Case Generation for Path Testing," Int. J. Advanced Networking and Applications 911 Volume: 02, Issue: 06, Pages: 911-915 (2011)

[24] Abdelaziz M. Khamis, Moheb R. Girgis. and Ahmed S. Ghiduk,"Automatic Software Test Data Generation For Spanning Sets Coverage Using Genetic Algorithms," Computing and Informatics, Vol. 26, 2007, 383–401

[25] McCart, James; Berndt, Donald; and Watkins, Alison, "Using Genetic Algorithms for Software Testing: Performance ImprovementTechniques," AMCIS 2007 Proceedings. Paper 222, 2007.