Interaction Fault Detection using Combinatorial Interaction Testing and Random Testing

Gowtham.N,
PG Scholar,
Department of Information Technology,
Bannari Amman Institute of Technology,

Sengottuvelan.P, Ph.D.
Associate Professor,
Department of Information Technology,
Bannari Amman Institute of Technology,

ABSTRACT
Software product lines are the common trend in software development which helps in reducing the development cost. Mostly the interaction faults are very difficult to identify during the process of debugging. By the use of combinatorial testing a set of features can be identified and all small combinations can be verified to a certain level only. By introducing random testing can improve the accuracy and ratio of t-wise fault detection. Through random testing can acquire a higher level of improvements over the combinatorial testing which will be under the budgetary limit of the product. Random testing can provide minimum guarantees on the probability of fault detection at any interaction level using the set of theories. For example, random testing becomes even more effective as the number of features increases and converges toward equal effectiveness with combinatorial testing. Given that combinatorial testing entails significant computational overhead in the presence of hundreds or thousands of features, the results suggest that there are realistic scenarios in which random testing may outperform combinatorial testing in large systems. Furthermore, in common situations where test budgets are constrained and unlike combinatorial testing, random testing can still provide minimum guarantees on the probability of fault detection at any interaction level. However, when constraints are present among features, then random testing can fare arbitrarily worse than combinatorial testing.

Index Terms:
Combinatorial testing, random testing, t-wise fault

1. INTRODUCTION
Software testing is a process of analyzing the effectiveness of software. Testing has two objectives in order to determine if the software performs as expected, which are to identify the differences between existing and expected conditions, which this differences are sometimes called 'bugs', but it now can this denotes a defect in the performance of either hardware or software. Secondly is to enable the expected performance of software to be determined before it is used for business purposes and this type of testing is to determine whether the right mix of software, hardware and personnel can satisfy the business needs. Software testing process is important in software development activities. Therefore, this industrial is focused on software testing process that currently practices.

National Institute of software testing last estimated the annual cost of software defects as approximately $59 billion. They also suggest that approximately $22 billion can be saved through more effective testing. Testers need to be more thorough in testing, yet they need to perform testing within a prescribed budget. Systematic approaches of combination testing have been suggested to complement current testing methods in order to improve rates of fault detection. Category partitioning is a base of systematic approaches as finite values (options) for parameters are identified for testing. Each of the finite parameter-values may be tested at least once, in specified combination together, or in exhaustive combination. The simplest and least-thorough combination testing approach is to test all values at least once. The most thorough is to exhaustively test all parameter-value combinations. However, exhaustive testing of all possible combinations is too expensive for most systems. Testers may place constraints to limit tests from category partitioning; however, this can be an unsatisfactory solution when constraints are arbitrarily selected to limit the number of tests. Combination strategies may be a better solution to limiting tests as they systematically test combinations of parameter-values across a system. Combination testing has been applied with ad-hoc methods, stochastic models, and combinatorial designs. Ad-hoc methods include tests developed by hand in which testers attempt to create representative tests to catch problems that they anticipate. Anti-random testing attempts to provide tests that minimize overlap using Cartesian products or Hamming distance. An example of a stochastic model includes Markov chains to simulate usage. Combinatorial designs have been applied to test *t*-way interactions of parameter values.

As software testing process is important in software development activities, and SBS system is also important for core banking application, an intensive software testing process is required to be developed and provided into this organization. This section clearly discussed and summarized every comparison study that being conducted on several testing characteristic for SBS system. Finally, a proposed Customized Software Testing Process (CSTP) is clearly defined based on these comparison studies

Unsupervised learning methods such as clustering techniques are a natural choice for analyzing software quality in the absence of fault proneness labels. Clustering algorithms can group the software modules according to the values of their software metrics. The underlying software-engineering assumption is that fault-prone software modules will have similar software measurements and so will likely form clusters. Similarly, not-fault-prone modules will likely group together. When the clustering analysis is complete, a software engineering expert inspects each cluster and labels it fault prone or not fault prone.

Combination testing has been applied with ad-hoc methods, stochastic models, and combinatorial designs. Ad-hoc methods include tests developed by hand in which testers attempt to create representative tests to catch problems that they anticipate. Anti-random testing attempts to provide tests

that minimize overlap using Cartesian products or Hamming distance usage modeling but do not systematically cover a system.) The success of combination strategies hide upon correct identification of parameters and their suitable values for testing. Indeed, if parameters are missing, or category partitioning does not select suitable values for parameters, then any combination strategy may fail. Tester identifies this threat early on since work is an extension of combination testing work. The *Quad tree algorithm* [3] method depicts, about various clustering algorithms that prevail to partition a dataset by some means of similarity. In this project, a Quad Tree based *Expectation Maximization (EM)* algorithm has been applied for predicting faults in the classification of datasets. K-Means is a simple and popular approach that is widely used to cluster/classify data. However, K-Means does not always guarantee best clustering due to varied reasons. The proposed EM algorithm is known to be an appropriate optimization for finding compact clusters. EM guarantees elegant convergence. EM algorithm assigns an object to a cluster according to a weight representing the probability of membership. EM then iteratively rescores the objects and updates the estimates. The error-rate for K-Means algorithm and EM algorithm are computed, denoting the number of correctly and incorrectly classified samples by each algorithm. Result consists of charts showing on a comparative basis the effectiveness of EM algorithm with quad tree for fault prediction over the existing Quad Tree based K-Means (QDK) model.

2. SOFTWARE FAULT DETECTION

Fault detection models are used to improve software quality and to assist software inspection by locating possible faults Model performance is influenced by a modeling technique and metrics. The performance difference between modeling techniques appears to be moderate and the choice of a modeling technique seems to have lesser impact on classification accuracy of a model than the choice of a metrics set. To this end, decided to investigate the metrics used in software fault prediction and to leave the modeling techniques aside. In software fault prediction many software metrics have been proposed. The most frequently used ones are those of MOOD metrics suite) (QMOOD metrics suite), (CK metrics suite), Many of them have been validated only in a small number of studies. Contradictory results across studies have often been reported. Even within a single study, different results have been obtained when different environments or methods have been used. Nevertheless, finding the appropriate set of metrics for a fault prediction model is still important, because of significant differences in metrics performance. This, however, can be a difficult task to accomplish when there is a large choice of metrics with no clear distinct ion regarding their usability a preliminary mapping study on software metrics. The study was broad and included theoretical and empirical studies which were classified in the following categories: development, evaluation, analysis, framework, tool programs, and use literature survey.

3. PROPOSED SYSTEM

In order to improve the probability of finding faults, CIT aims at generating test suites with high coverage of feature interactions. Sample N test cases at random, where the value of each of the features in the ith column is randomly chosen with uniform probability from 1; . . . ; vi. Notice that this procedure could be repeated q times and then select the best test suite, where the best test suite would be the one with highest number of covered t-wise interactions. Based on the computational time tester can afford (i.e., the testing budget), tester could run random testing with q as high as necessary to obtain a full t-wise coverage. In this project, however, have to only consider the case q = 1. Larger values of q would, of course, lead to higher fault detection. In a comparison with CIT, tester could choose a value for q that corresponds to the CIT overhead required to generate covering arrays. Lower bounds related to some probabilities that describe the dynamics of random testing when applied to find interaction faults. Assume P to be the probability that random testing triggers at least one failure. A probability is always bounded in (0; 1). Random testing might or might not trigger a failure when run once. Depending on the problem instances, the probability P could significantly vary. To prove six theorems regarding the effectiveness of random tests when applied to unconstrained combinatorial interaction testing (CIT) problems. These theorems provide general results that cannot be obtained with empirical studies, though the latter are necessary to refine understanding. First, proved that for any t-wise CIT problem (independently of its properties), random testing would always have at least a 63 percent probability of triggering at least one failure related to t-wise interaction faults (if any is present) when compared against any CIT tool using the same number of test cases. Second, perhaps more importantly, this probability increases with the number of features present in the software under test and converges to 1 (for infinite number of features), that is, to equal effectiveness with CIT techniques. Given that current CIT tools suffer severe scalability problems in the presence of large numbers of features, thus leading to significant execution overheads to generate covering arrays, this additional time could be easily used by random testing to run more test cases if an automated oracle is available. The results suggest that in such situations random testing would be effective at detecting interaction faults. To overcome these types of problems previously used the CIT method and reduced a certain level by the use of combination features and removed the bugs present in them which caused the faults. In this project by using random testing method can reduce the impact of interaction faults on the software systems as well as analyzing their improvements in their performance consequently.

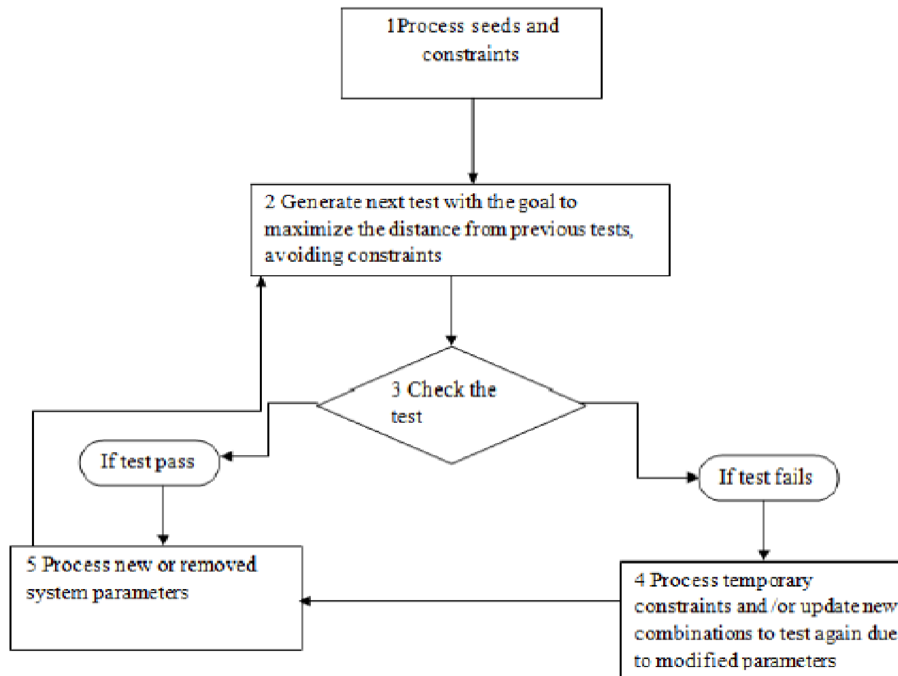

Figure 1 System Architecture

3.1 Feature Model Extraction

Feature models are frequently used to present the commonalities and variability within variant rich systems such as Software Product Lines (SPLs). Feature Models are initially created during the requirements analysis where each feature 'represents a system property that is relevant to some stakeholder' Utilizing a hierarchical structure, different notations and cross tree dependencies feature models are able to determine which feature combinations or products are permitted and which are forbidden. In this module Feature models provide a hierarchical and structured representation of the SPL requirements. Thus, it seems to be promising to benefit from such a representation of SPL requirements for testing purposes. Algorithms ensuring a certain degree of requirements coverage within product derivation can take advantage of the feature model.

3.2 Random Testing

Random testing module uses test configurations that correspond to sets of features of test cases. A feature is an attribute of generated test inputs that the generator can directly control, in a computationally efficient way. For example, an API-based test generator might define features corresponding to inclusion of API functions (e.g., push and pop); a program generator might define features corresponding to the use of language constructs (e.g., arrays and pointers); and a media-player tester might define features over the properties of media files, e.g., whether or not the tester will generate files with corrupt headers. A feature determines a configuration of test generation, not the System Under Test (SUT). In particular, this module are configuring which aspects of the SUT will be tested (and not tested) only by controlling the test cases output. Features can be thought of simply as constraints on test cases, in particular those the test case generator lets us control.

3.3 Lower Bounds

In this module the lower bounds which related to some probabilities that describe the dynamics of random testing when applied to find interaction faults. Assume P to be the probability that random testing triggers at least one failure. A probability is always bounded in (0; 1). Random testing might or might not trigger a failure when run once. Depending on the problem instances, the probability P could significantly vary. A lower bound to the probability that random testing triggers at least one failure related to t-wise faults for a given test suite size N. This theorem is then used to prove. A high lower bound that is independent of N to the effectiveness of random testing compared to CIT. Assume P to be the probability that random testing triggers at least one failure. A probability is always bounded in (0; 1). Random testing might or might not trigger a failure when run once. Depending on the problem instances, the probability P could significantly vary. For example, on very faulty software could have P = 1, whereas P could be much lower in cases where only a single feature combination triggers failures. Because before running Feature Oriented Software Development community (FOSD) [2] uses the features of a feature model as fundamental artifacts within the development process of variant rich systems such as SPLs. There, the features are linked to various different artifacts such as code fragments, behavioral models, requirements, specifications, documentations and tests.

Any large empirical study it would not be possible to know P in advance, then it would be of practical interest to know a lower bound b (i.e., P= b) that is valid for any problem instance.

3.4 Detection of Multiple Faults

This module is used to find the detection of multiple faults in products. So far the system has only analyzed random testing from the point of view of triggering at least one failure. Software testing can only trigger failures and not directly reveal faults. If one has a test suite in which more than one test case fails, then some or all of these failures might or might not be related to the same fault. To distinguish among faults, a software tester has to debug and fix the code, and then rerun the test suite to see if any test case is still failing. Real-world systems typically contain several faults and it would hence be important to study how well random testing fares in revealing a set of combinatorial interaction faults

4. RESULTS AND DISCUSSION

In many realistic testing environments, testers may not have time to run entire test suites that cover all possible parameter-value combinations, or even all lower strength *t*-way parameter-value combinations. They may also be in an environment in which partial test suites are run and then testing needs to adapt. Also measure the overlap in the number of 2-way, 3-way, and 4-way combinations covered in the respective tests. Hence break the studies into two experiments with two goals in mind. First, combinatorial testing has been criticized as an ineffective testing method that offers little benefit over random tests. However, this criticism is supported only in small study and contradicts other existing literature that reports on the success of combinatorial testing. In addition, previous work only reports the number of faults found with test suites of specific sizes that cover all *t*-way interactions; they do not report how fast the test suites localize faults, nor what happens if a tester cannot run an entire test suite. Therefore, the work here on distance based testing closely examines the overlap of combinations.

The simulation results indicate that random testing is effective only when one runs far too many tests, and hence a comparison of random and structured schemes can be misleading when one fixes a large number of tests in advance to be run. For instance, these simulations indicate that distance-based tests can be more effective in locating faults sooner and in fewer tests than random tests, especially when faults are more complex (i.e: more parameter-values interact to cause faults) and faults do not cluster around only few parameter-values.

Table 1 Comparison on performance

Number of tests	Existing t=2	Proposed t=2
10	39%	93%
50	64%	100%
100	76%	100%
250	90%	100%
500	98%	100%
750	99%	100%
1500	100%	100%
No. tests for t-way coverage	122	15

In the experiments, the overlap of combinations covered in the distance-based tests and random tests is quite high in the earliest tests, but this finding does not scale; more than 10 times as many random tests are needed to cover all 2-way combinations in the experiments. In the TCAS experiment, distance-based tests work well in finding 2-way interaction faults, but mixed results are observed for 3-way, 4-way, 5-way testing. Indeed, a closer look at the tests and data observe that the 3-way interaction faults involve many of the same parameter-values. When testing actual software here, random testing is quite competitive in the earliest tests but is not competitive after approximately 45 to 80 tests are run. Hamming distance appears to be more effective than uncovered combinations distance than have seen in the previous simulations. While the distance-based testing works well in this example to identify 2-way interaction faults, initial experiments with t = {3} do not exhibit any clear pattern. Have to find that uncovered combinations and Hamming distance metrics sometimes appear to be more successful at finding the first fault, however, maximizing distance with either of these approaches is not particularly effective. Attribute this to the characteristics of the faults - the numerous faults injected into the TCAS system cluster around similar parameter-values. In cases when faults cluster, these notions of distance may not be adequate. Currently studying alternate notions of distance that do not penalize next tests based on their proximity to a fault. One can expect such an alternate notion of distance to serve well in locating Clustered faults.

5. CONCLUSION

Distance-based testing is a systematic testing technique that may be used to augment current testing practices. The methodology is an abstraction of static combination strategies that have been proposed in the past. Instead of generating test suites that are run as a whole, an adaptive one-test-at-a-time process is more flexible. Tests are adaptively generated as systems can undergo modifications. System components may be added, removed, modified or temporarily unavailable and tests will adapt. The effectiveness of the strategy is examined for an actual system and in simulation by measuring the rate of fault detection of dispensed tests.

Distance-based testing can be instantiated using a number of different combination strategies, considered recent controversy on combination strategies when conducting experiments. For instance, a specific example of distance based testing, implemented with “uncovered combinations” has been reported with mixed reviews. The majority of empirical studies report that it is a useful approach, while other work suggests that it offers little benefit over random testing. Software testing process identifies what the test activities to carry out and when, which what is the best time to accomplish those test activities. Hence, even though testing may differs between organizations, there is still a testing process that need to be performed has the basic phases such as test planning, test design, test execution and test evaluation. Further exploration of distance metrics are needed for systems in which interaction faults cluster around a smaller number of parameter-values. By using the ACO algorithm and SWARM intelligence method can make the fault detection easier. The dataset has to be taken as noisy for revealing them from the original faults that are associated with the system code. Most important attribute can be found for fault prediction and this work can be extended to further programming languages.

6. REFERENCES

[1] NIST, “The economic impacts of inadequate infrastructure for software testing,” March 2003.

[2] M. Grindal, J. Offutt, and S. Andler, “Combination testing strategies: a survey,” *Software Testing, Verification, and Reliability*, vol. 15, no. 3, pp. 167–199, March 2005.

[3] T. J. Ostrand and M. J. Balcer, “The category-partition method for specifying and generating functional tests,” *Communications of the ACM*, vol. 31, no. 6, pp. 676–686,June 1988.

[4] Y. K. Malaiya, “Antirandom testing: getting the most out of black-box testing,” in *Proceeding of the International Symposium on Software Reliability Engineering*, October 1995, pp. 86–95.

[5] J. A. Whittaker and M. G. Thomason, “A markov chain model for statistical software testing,” *IEEE Transactions on Software Engineering*, vol. 20, no. 10, pp. 812–824, 1994.

[6] R. C. Bryce and C. J. Colbourn, “Prioritized interaction testing for pair-wise coverage with seeding and avoids,” *Information and Software Technology Journal (IST, Elsevier)*, vol. 48, no. 10, pp. 960–970, October 2006.

[7] K. Burr and W. Young, “Combinatorial test techniques: Table based automation, test generation, and code coverage,” in *Proceedings of the International Conference on Software Testing Analysis and Review*, October 1998, pp. 503–513.

[8] S. R. Dalal, A. Karunanithi, J. Leaton, G. Patton, and B. M. Horowitz, “Model-based testing in practice,” in *Proceedings of the International Conference on Software Engineering*, May 1999, pp. 285–294.

[9] S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino, “Applying design of experiments to software testing,” in *Proceedings of the International Conference on Software Engineering*, October 1997, pp. 205–215.32

[10] D. Kuhn and M. Reilly, “An investigation of the applicability of design of experiments to software testing,” in *Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop*, October 2002, pp. 91–95.

[11] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, “Software fault interactions and implications for software testing,” *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 418–421, October 2004.

[12] R. Mandl, “Orthogonal latin squares an application of experiment design to compiler testing,” *Communications of the ACM*, vol. 28, no. 10, pp. 1054–1058, October 1985.

[13] C. Yilmaz, M. B. Cohen, and A. Porter, “Covering arrays for efficient fault characterization in complex configuration spaces,” *IEEE Transactions on Software Engineering*, vol. 31, no. 1, pp. 20–34, January 2006.

[14] R. C. Bryce, C. J. Colbourn, and M. B. Cohen, “A framework of greedy methods for constructing interaction tests,” in *Proceedings of the 27th International Conference on Software Engineering*, May 2005, pp. 146–155.

[15] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, “The combinatorial design approach to automatic test generation,” *IEEE Software*, vol. 13, no. 5, pp. 82–88, October 1996.

[16] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge, “Constructing test suites for interaction testing,” in *Proceedings of the International Conference on Software Engineering*, May 2003, pp. 28–48.

[17] C. J. Colbourn, “Combinatorial aspects of covering arrays,” *Le Matematiche (Catania)*, vol. 58, pp. 121–167, 2004.

[18] R. C. Bryce and C. J. Colbourn, “A density-based greedy algorithm for higher strength covering arrays,” *Software Testing, Verification, and Reliability*, vol. 19, no. 1, pp. 37– 53, 2009.33

[19] K. Tai and L.Yu, “A test generation strategy for pair-wise testing,” *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 109–111, January 2002.

[20] Y. Tung and W. Aldiwan, “Automating test case generation for the new generation mission software system,” in *IEEE Aerospace Conference*, March 2000, pp. 431–37.