International Journal of Computer Theory and Engineering, Vol. 2, No. 2 April, 2010
1793-8201

Design Fault Tolerance System Using Genetic Algorithm Employing Mutation and Back-to-Back Testing

Rakesh Kumar and Kulvinder Singh

Abstract— **Software system should be reliable and available failing which huge losses may incur. To achieve these objectives a thorough testing is required. Adequacy of test cases is the key to the success. Despite the availability of a number of adequacy criteria, deterministic approaches to testing are not sufficient consequential to the need of automatic random and anti-random testing. Our research uses a novel method for the development of n-version of the software by creating the different mutation in software and test cases generation using the Genetic Algorithm. Its purpose is to eliminate software faults as possible by using lesser test cases in the testing phase. The test case generated by the use of Genetic Algorithm (GA) is compared with the results of totally random generated test cases. The method was applied to the specification of a sorting of array problem. The advantage of this analysis is that when we produce multiple versions, reliability of the software is likely to be better than if a single version is developed. The N-version software testing will helps to reduce the possibility of mistakes and inconsistencies in the process of software development and testing and the number of test cases required during the testing phase of the software system. In this paper a technique of generating the test cases and doing the testing automatically, employing genetic algorithm and Back-to-Back testing has been discussed.**

Index Terms— **Anti-random testing, Back-to-Back testing, Genetic algorithm, N-Version Programming, Random testing.**

I. Introduction

According to [1] testing is the process of executing a program with the intent of finding errors. It includes activities aimed at evaluating an attribute of a program and verifying that it meets its required results [2]. Effectively detecting the failures using the limited resource is a challenging task. A study conducted by National Institute of Standards and Technology (NIST) in 2002 reported that software bugs cost $59.5 billion to U.S. economy, a third of which could be avoided if a better testing would be done. Software faults will always exist in any software module of moderate size because the complexity of software is generally intractable and humans have limited capability to manage complexity. Identifying faults in software is easier said than done because software is not continuous, so testing boundary values as suggested in Boundary Value Analysis or selecting test cases using criteria such as path coverage are not ample to assure correctness and moreover exhaustive testing is infeasible. Thing are further complicated by the dynamic nature of programs. If a failure occurs during preliminary testing and the code is changed, then behavior of S/W on pre-error test cases that it passed before can no longer is guaranteed. So testing should be restarted.

An interesting analogy parallels the intricacy in software testing with the pesticide, referred to as the Pesticide Paradox [3]: *Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual*. But this alone will not guarantee to make the software better, because the Complexity Barrier [3] principle states: *Software complexity (and therefore that of bugs) grows to the limits of our ability to manage that complexity*. Eliminating the easy bugs results into another escalation of features and complexity, but this time there are subtler bugs to face, just to retain the reliability you had before.

Software is being used now in mission critical situations where failure is simply intolerable. From the point of view of a software development organization also, delivering products with defects results in loss of goodwill. Thus, the only alternative is to do it right the first time, before delivering the product to the customer [4]. In this paper, a technique of automatic test case generation using genetic algorithm (GA), back to back testing and mutation adequacy criteria has been purposed and the result were compared with random testing. Section 2 deals with importance of testing and test adequacy criteria including mutation adequacy. Section 3 gives a brief overview of back to back testing and section 4 cover the issues in automatic test cases generation using random and anti random testing. In Section 5, GA was discussed and section 6 covers the proposed technique followed by section 7 in which results were analyzed.

II. Importance Of Testing

Dependable system should be reliable, available, safe and secure. To achieve these objectives a number of techniques are being used such as fault avoidance, fault tolerance, fault removal, and fault evasion etc. Testing is an integral part for fault removal and is usually performed for the following purposes: (a) Quality assurance, (b) For Verification & Validation (V&V): Testing is used as a tool in the V&V

Manuscript received November 4, 2009.

Rakesh Kumar is with the Department of Computer Science and Applications (D.C.S.A), Kurukshetra University, Kurukshetra (K.U.K), India- 136 119(Phone: +91-98963-36145; e-mail: rsagwal@rediffmail.com)

Kulvinder Singh is with Department of Computer Science and Engineering, University Institute of Engineering & Technology (U.I.E.T), Kurukshetra University, Kurukshetra (K.U.K), India- 136 119 (Phone: +91-94162-24353, e-mail: kshanda@rediffmail.com).

process. Testers can make claims based on interpretations of the testing results whether the product works under certain situations or not. Testing for the purpose of validating the product works known as clean tests. The negative aspects are that it can only validate that the software functions for the specified test cases. A limited number of test cases cannot validate that the software functions for all situations. On the contrary, only one failed test is sufficient to show that the software does not work. Dirty tests refer to the tests aiming at breaking the software and software must have sufficient exception handling capabilities to survive a significant level of dirty tests. (c) For reliability estimation [5]: Software reliability has important relations with many aspects of software, including the structure, and the amount of testing it has been subjected to. Based on an operational profile (an estimate of the relative frequency of use of various inputs to the program [5]), testing can serve as a statistical sampling method to gain failure data for reliability estimation.

A. Adequacy of test cases

To locate the faults in the software, the test cases designed should be adequate and effective enough. A number of adequacy criteria have been proposed in the literature such as statement coverage, branch coverage, path coverage, loop coverage etc but studies reveal that no criterion is capable enough to identify all the bugs except exhaustive testing which is theoretically and practically not possible. Mutation testing has been established as a powerful approach to evaluate test cases and for comparing different testing strategies. Empirical studies show that the generated mutants provide a good indication of the fault detection ability of a test suite [6]. Mutation testing is an approach to verify the effectiveness of the test cases designed and has been proved successful with some limitations.

B. Mutation Adequacy

The mutation method is a fault-based testing strategy that measures the adequacy of testing by examining whether the test set used in testing can reveal certain types of faults. The core of a mutation-based testing is a set of operators that modifies the source code to inject a fault. The modified program is known as a mutant. A mutant is said to be killed relative to a test data set, if at least one test case generates different results between the mutant and the implementation. Else, the mutant is live. If no test case can kill a mutant, then it is either equivalent of the original implementation or a new test case needs to be generated to kill the live mutant, a method of enhancing a test data set. The adequacy of a test data set is measured by a *mutation score* (MS), which is the percentage of non-equivalent mutants killed by the test data. The *mutation score* for a set of test cases is:

$$\text{Mutation Score} = 100 \times \frac{D}{N - E}$$ *Where* D = Dead mutants, N = Number of mutants, and E = Number of equivalent mutants. A set of test is mutation *adequate* if its mutation score is 100%.

III. Back-To-Back Testing

In the systems where the reliability of the software is critical, the software module is implemented in a number of different versions by different teams, using common specification, a technique called N-version programming. Each version is executed in parallel. There outputs are compared using a voting system and inconsistent outputs are rejected. At least three versions of the module should be available. The assumption is that it is unlikely that different teams will make the same design or programming errors. [7, 8] describes this approach as fault avoidance. In Back-to-back testing, using lessons learned from N-version programming, [9] and [10] have suggested that that N version of the software be developed even when only a single version will be used. Test cases designed using other testing techniques is provided as test input to each version and their outputs can be compared by automatic tools. In case of the differences in the output, each of the versions is analyzed to identify the fault. This method depends on the basis that all the versions have been developed independently so if any version fails that will fail independently. In this paper total three different version of the same sorting programs were prepared independently from the same specification of the software and then subjected to thousand of test cases. In this research paper we use two types of test cases: one is totally random numbers, and second one is using the concept of the Genetic Algorithm. Both type of the test case are given to the N-version software after placing the mutant in any one version of the software.

IV. Automatic Test Case Generation

A. Random Testing

Random Testing (RT) randomly selects test cases/sequences of events from the input domain [1, 11]. The advantages of RT include its low cost, ability to generate numerous test cases automatically, generation of test cases in the absence of the software specification and source code and apart from these; it brings randomness into the testing process. Such randomness can best reflect the chaos of system operational environment; as a result, RT can detect certain failures unable to be revealed by deterministic approaches. All these advantages make RT irreplaceable in industry for revealing software failures [12, 13, 14, 15, 16, 17, 18, 19, 20, 21]. This approach may produce a large number of event sequences that are not legal & hence not executable, wasting valuable resources. Moreover, the test designer has no control over choice of event sequences; they may not have acceptable test coverage. Random testing selects arbitrarily test data from the input domain & then these test data are applied to the program under test. The automatic production of random test data, drawn from uniform distribution, should be the default method by which other systems should be judged, [22]. The random generation of tests identifies members of the sub domains arbitrarily, with a homogeneous probability which is related to the cardinality of the sub domains. Under these circumstances, the chances of testing a function, whose sub domain has a low cardinality with regard

to the domain as a whole, is much reduced. A random number generator generates the test data with no use of feedback from previous tests. The tests are passed to the procedure under test, in the hope that all branches will be traversed [23].

B. Adaptive Random Testing

Adaptive Random Testing (ART) is an enhancement of Random Testing (RT). It has been introduced to improve the fault detection effectiveness of RT for the situations where failure-causing inputs are clustered together [24, 25]. Such situations do occur frequently in real life programs as reported in [26, 27, 28]. When failure-causing inputs are concentrated in regions (Known as the failure regions [26]), intuitively speaking, keeping test cases apart shall enhance the effectiveness of RT. Therefore, ART does not just randomly generate but also evenly spreads test cases or it generates fewer duplicate test cases. Studies [29, 30, 31, 32,34] shows that ART can be very effective in detecting failures when there exist continuous failure regions inside the input domain as compared to RT. Since ART is as simple as RT and preserves certain degree of randomness, ART could be an effective replacement of RT.

V. Genetic Algorithm (GA)

GA is a search technique used to find exact or approximate solutions to optimization and search problems. GAs represents a class of adaptive search techniques & procedures based on the processes of natural genetics & Darwin's principal of the survival of the fittest. There is a randomized exchange of structured information among a population of artificial chromosomes. When GAs are used to solve optimizations problems, good results are obtained surprisingly quickly. A problem is defined as maximization of a function of the kind $f(x_1, x_2, \ldots x_m)$ where $(x_1, x_2, \ldots, x_m)$ are variables which have to be adjusted towards a global optimum. Three basic operators responsible for GA are (a) selection, (b) crossover & (c) mutation. Crossover performs recombination of different solutions to ensure that the genetic information of a child life is made up of the genes from each parent. GAs may be differentiated from more conventional techniques as (a) in GA a representation for the sample population must be derived, (b) GAs manipulates directly the encoded representation of variables, rather than manipulation of the variables themselves, (c) GAs use stochastic rather then deterministic operators, (d) GAs search blindly by sampling & ignoring all information except the outcome of the sample, (e)GAs search from a population of points rather than from a single point, thus reducing the probability of being stuck at a local optimum, which make them suitable for parallel processing. In the context of S/W testing, the basic idea is to search the domain for input variables which satisfy the goal of testing. With the above defined, GA is defined as follows:

Procedure GA(φ, θ, n, r, m)
// φ is the fitness function for ranking individuals
// θ is the fitness threshold, which is used to determine when to halt
// n is the population size in each generation (e.g., 100)
// r is the fraction of the population generated by crossover (e.g., 0.6)
// m is the mutation rate (e.g., 0.001)
P:= generate n individuals at random// initial generation is generated randomly
while max (φ (h_i)) < θ **do**
//define the next generation S (also of size n)
Reproduction step: Probabilistically select (1-r)n individuals of P and add them to S, where the probability of selecting individual h_i is
Prob(h_i)= φ (h_i) / $\sum$(φ (h_j))
Crossover step: Probabilistically select r*n/2 pairs of individuals from P according to Prob(h_i)
for each pair (h_1, h_2), produce two offspring by applying the crossover operator and add these offspring to S
Mutate step: Choose m% of S and randomly invert one bit in each
P := S
end_while
Find b such that φ (b) = max (φ (h_i))
Return (b)
end_proc

VI. Purposed Method Of Testing

Using GA is one proposed way to test application [33]. This method generates test cases based on the theory that good test coverage can be attained by simulating a novice user who would follow a more random path while an expert user of a system will follow a predictable path through an application ignoring many possible system states that would never be achieved. Therefore, it is more desirable to create test suites that simulate novice usage because they will test more. The obscurity lies in generating test suites that simulate 'novice' system usage. Novice paths are not the random paths. First, a novice user will learn over time and generally will not make the same mistakes repeatedly and secondly, a novice user is following a plan and probably has some domain or system knowledge.

The algorithm of proposed automatic test case generation approach using GA, mutation testing and back to back testing is as under:

1) Write the module *V*// i.e. the module to be tested
2) Generate N versions of *V* i.e. $V_1 \ldots V_n$.// for back to back testing
3) P:=Generate Test Cases using structural and functional testing techniques //Generate a test set of n test cases
4) FAIL:=FALSE // initialize the variable
5) While (Not (terminating condition)) do {
   Generate the mutant of *V*
   While (~FAIL)
   {

Generate the next generation S (of size n) from P using Genetic Algorithm.
Perform back to back testing
If failure {
FAIL=TRUE
Then add the test case killing the mutant to the population P
} } }
End.

In order to make the experiment realistic, an attempt was made to choose an application that would normally be a candidate for the inclusion of fault tolerance. The problem that was selected for programming is a simple and realistic data structure sorting system. The N version program read some data that represents as test cases an array (integer or float). The outputs from the N-version software are compared to check the correctness of the system. To check the efficiency of N-version system and the completeness of the test set, tests are performed by introducing the mutant in the software. This program was originally written in MATLAB, and the program has been subjected to several thousand test cases. The figure 1 shows the block diagram for proposed approach of test case generation.

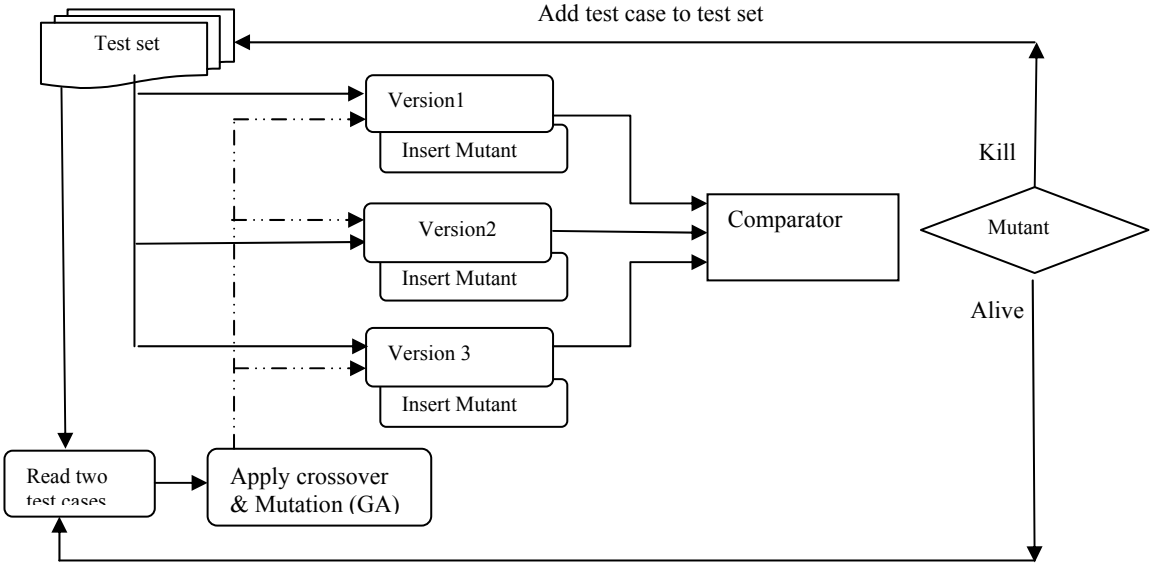


Figure1 shows the block diagram for N-Version program development.

Assumptions made are as under:

A. Encoding

Direct value encoding can be used in problems where some more complicated values such as real numbers are used. In value encoding, each chromosome is a sequence of some values. Values can be no matter which connected to the problem, such as (real) numbers, chars or any objects.

B. Selection

From a population of individuals, we wish to give the fitter individuals a better chance to survive to the next generation. We not use the simple criterion "keep the best individuals." It turns out the nature that it does not kill all the unfit genes. They usually become recessive for a long period. Then they may mutate to something useful. Therefore, there exists tradeoff for better individuals and diversity. The individuals are selected according to Rank selection criteria. Rank selection ranks the population after that every chromosome receives fitness value determined by this ranking. The worst individual will have the fitness 1, the second worst 2 etc. and the best individual will have fitness N (number of chromosomes in population).

VII. Results

Fault propagation spreads the faulty result in a problem to the output and causes a failure of the program. It can be revealed by an execution of the program. A fault may be any occurrence of program in any particular version that causes that version to fail when that software is executed on some test case. The numbers of faults found in the individual versions is shown in Table 1. All of these faults have been found and corrected. Many of the faults were unique to individual versions but several occurred in more than version.

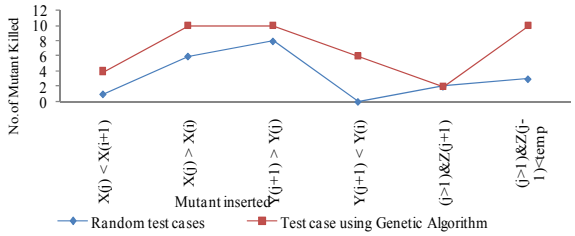
Table 1 shows the numbers of mutant kill per no. of test case applied.

Sr. No	Original expression	After Mutant expression	No. of mutant killed per No. of test cases applied							
			Random test case Generation				**test case Generation by application of GA**			
			10	50	100	1000	10	50	100	1000
1	X(j) < X(i)	X(j) < X(i+1)	1	2	2	23	4	12	18	187
2	X(j) < X(i)	X(j) > X(i)	6	33	60	581	10	48	98	978
3	Y(j+1) < Y(j)	Y(j+1) > Y(j)	8	36	56	623	10	47	93	982
4	Y(j+1) < Y(j)	Y(j+1) < Y(i)	0	4	37	45	6	11	16	164
5	(j>1)&Z(j-1)	(j>1)&Z(j+1)	2	3	9	54	2	6	11	31
6	(j>1)&Z(j-1)>temp	(j>1)&Z(j-1)<temp	3	30	51	582	10	48	99	921

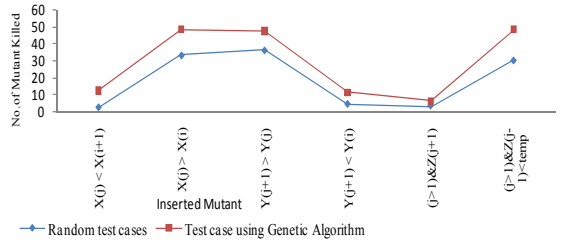
Table2 shows the improvement (%age) of killing mutants in random vs. GA.

Sr. No	N-Version Programs selected	Original expression	After Mutant expression	Random test case mutant Kill(%age)	Proposed method mutant Kill(%age)	Improvement (%age)
1	Selection Sort	X(j) < X(i)	X(j) < X(i+1)	4.5	25.25	20.75%
2		X(j) < X(i)	X(j) > X(i)	60.5	98	38.5%
3	Bubble Sort	Y(j+1) < Y(j)	Y(j+1) > Y(j)	67.5	96.25	28.75%
4		Y(j+1) < Y(j)	Y(j+1) < Y(i)	12.37	28.6	16.23%
5	Insertion Sort	(j>1)&Z(j-1)	(j>1)&Z(j+1)	10.1	11.5	1.4%
6		(j>1)&Z(j-1)>temp	(j>1)&Z(j-1)<temp	49.75	71.75	22%

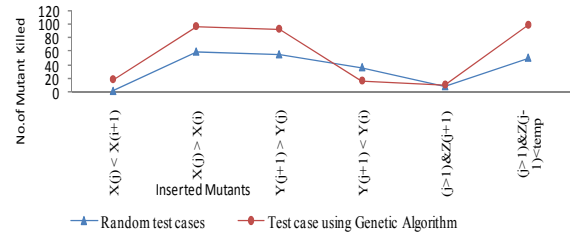
The graphs depict the comparison approach of random vs. Genetic Algorithm.



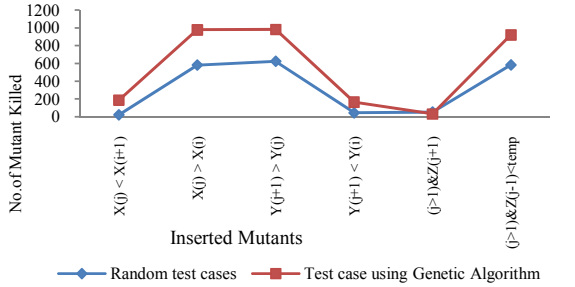
Graph 1. Compare the numbers of mutant kill per 10 test cases in random test vs.GA test cases.



Graph2.Compare the numbers of mutant kill per 50 test cases in random test vs.GA test cases



Graph 3. Compare the numbers of mutant kill per 100 test cases in random test vs.GA test cases.



Graph4. Compare the numbers of mutant kill per 1000 test cases in random test vs.GA test cases.

VIII. Conclusion

Genetic Algorithms are easy to apply to a wide range of optimization problems, like the traveling salesperson problem, inductive concept learning, scheduling, and layout problems. Software testing is also an optimization problem with the objective that the efforts consumed should be minimized and the number of faults detected should be maximized. Software testing is considered most effort consuming activity in the software development. Although a number of testing techniques and adequacy criteria have been suggested in the literature but it has been observed that no technique/criteria is sufficient enough to ensure the delivery of fault free software consequential to the need of automatic test case generation to minimize the cost of testing. As discussed the techniques like random and anti-random testing techniques have shown the good results. The proposed technique using GA and employing back-to-back testing and mutation adequacy criteria has shown the average 21% and maximum 39% improvement over the random test case generation. Although the cost incurred in producing N versions of the same module will be large but by using the technique judiciously in those modules only where a high level of reliability is required, the benefits accrued override the cost incurred.

References

[1] Myers G. J., The Art of Software Testing. Wiley, New York, 2nd edition, 1979.
[2] Hetzel, William C., The Complete Guide to Software Testing, 2nd ed. Publication Wellesley, 1988. ISBN: 0894352423.
[3] Beizern Boris, *Software Testing Techniques,* Second Edition, 1990.
[4] Srinivasan Desikan, Gopalaswamy Ramesh, Software Testing-Principles and Practices. Pearson Education, Fifth Impression, 2007.
[5] Lyu R. Michael, Handbook of Software Reliability Engineering. McGraw-Hill publishing, 1995, ISBN 0-07-039400-8
[6] Andrews J., Briand L., Labiche Y., Is mutation an appropriate tool for testing experiments?, Proc. Of the 27th International Conference on Software Engineering. ACM Press, New York, NY, USA, 2005, pp. 402-411.
[7] Avizienis, A. (1985) The N-version approach to fault-tolerant software. IEEE transaction on Software Engineering, SE-11 (12), 1491-501.
[8] Avizienis, A. (1995) A methodology of N-Version programming. In Software Fault Tolerance (lyu, M. R., ed.) Chichester: John Wiley & Sons, 23-46.
[9] Brilliant, S.S., J. C. Knight, "The consistent Comparison Problem in N-Version Software", ACM Software Engineering Notes, Vol. 12, no. 1, January 1987, pp. 29-34.
[10] Knight, J., and Ammann P., "Testing Software using Multiple Versions", Software Productivity Consortium, Report No. 89029N, Reston VA, June 1989.
[11] Hamlet. R., Random testing. In J. Marciniak, editor, Encyclopedia of Software Engineering. John Wiley & Sons, second edition, 2002.
[12] Cobb R. and Mills H. D., Engineering software under statistical quality control. IEEE Software, 7(6):45–54, 1990.
[13] Dab'oczi T., Koll'ar I., Simon G., and Megyeri T., Automatic testing of graphical user interfaces. In Proceedings of the 20th IEEE Instrumentation and Measurement Technology Conference 2003 (IMTC '03), pages 441–445, Vail, CO, USA, 2003.
[14] Forrester J. E. and Miller. B. P., An empirical study of the robustness of Windows NT applications using random testing. In Proceedings of the 4th USENIX Windows Systems Symposium, pages 59–68, Seattle, 2000.
[15] Miller B. P., Fredriksen L., and So. B., An empirical study of the reliability of UNIX utilities. Communications of the ACM, 33(12):32–44, 1990.

[16] Miller B. P., Koski D., Lee C. P., Maganty V., Murthy R., Natarajan A., and Steidl J.. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical Report CS-TR-1995-1268, University of Wisconsin, 1995.
[17] Miller. E., Website testing. http:// www.soft.com/ eValid/ Technology/ White. Papers/ website.testing.html, Software Research, Inc., 2005.
[18] Nyman. N., In defense of monkey testing: Random testing can find bugs, even in well engineered software. http: // www.softtest.org /sigs/material /nnyman2.htm, Microsoft Corporation.
[19] Sen K., Marinov D., and Agha G., Cute: a concolic unit testing engine for c. In ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, pages 263–272, New York, NY, USA, 2005.ACM Press.
[20] Slutz. D., Massive stochastic testing of SQL, In Proceedings of the 24th International Conference on Very Large Databases (VLDB 98), pages 618–622, 1998.
[21] Yoshikawa T., Shimura K., and Ozawa T., Random program generator for Java JIT compiler test system. In Proceedings of the 3rd International Conference on Quality Software (QSIC 2003), pages 20–24. IEEE Computer Society Press, 2003.
[22] Ince, D.C.,"The automatic generation of test data", The Computer Journal, Vol. 30, No. 1, pp. 63-69, 1987.
[23] Watt D. A., Wichman B. A., Sayward F. G., & Findlay W., "ADA language and methodology", 1987.
[24] Chen T. Y., Leung H., and Mak I. K., Adaptive random testing. In Proceedings of the 9th Asian Computing Science Conference, volume 3321 of Lecture Notes in Computer Science, pages 320–329, 2004.
[25] Mak. I. K., On the effectiveness of random testing. Master's thesis, Department of Computer Science, University of Melbourne, 1997.
[26] Ammann P. E. and Knight J. C.. Data diversity: an approach to software fault tolerance. IEEE Transactions on Computers, 37(4):418–425, 1988.
[27] Bishop P. G.. The variation of software survival times for different operational input profiles. In Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS-23), pages 98–107. IEEE Computer Society Press, 1993.
[28] Finelli. G. B., Nasa software failure characterization experiments. Reliability Engineering and System Safety, 32(1–2):155–169, 1991.
[29] Chen T. Y., Eddy G., Merkel R. G., and. Wong P. K. Adaptive random testing through dynamic partitioning. 4th International Conference on Quality Software (QSIC 04), pages 79–86, Braunschweig, Germany, 2004. IEEE Computer Society Press.
[30] Chen T. Y. and Huang D. H.. Adaptive random testing by localization. In Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04), pages 292–298. IEEE Computer Society, 2004.
[31] Chen T. Y., Kuo F. C., Merkel R. G., and Ng S. P., Mirror adaptive random testing. Information and Software Technology, 46(15):1001–1010, 2004.
[32] Chen T. Y., Kuo F. C., and Zhou Z. Q., On the relationships between the distribution of failure-causing inputs and effectiveness of adaptive random testing. In Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE 2005)), pages 306–311, Taipei, Taiwan, 2005.
[33] Kasik, D.J. and George, H. G., Toward automatic generation of novice user test scripts. Proceedings of the Conference on Human Factors in Computing Systems: Common Ground, pages 244-251, New York, 13-18 Apr. 1996, ACM Press.
[34] Chan, K. P. Chen T. Y., and Towey D., Restricted random testing: Adaptive random testing by exclusion. Accepted to appear in International Journal of Software Engineering and Knowledge Engineering, 2006.