ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 1, January 2014

Automated Evolutionary Test Data Generation through Poisson Distribution

[1]S.Gopikrishnan, [2]S.Bhuvenswari, [3]N.Vaitheeka

[1] Assistent Professor, Department of Computer Science and Engineering, Karpagam College of Engineering, Coimbatore, India

[2,3] Assistent Professor, Department of Information Technology, Karpagam College of Engineering, Coimbatore, India

ABSTRACT: Testing is the most important analytic quality assurance measure for software. The systematic design of test cases is crucial for test quality. Structure - oriented test methods, which define test cases on the basis of the internal program structures, are widely used test data generation in program testing is the process of identifying a set of data which satisfies given testing criterion. Most of the existing test data generators use symbolic evaluation to derive test data.
Evolutionary testing is a promising approach for the automation of structural test case design which searches test data that fulfill given structural test criteria by means of evolutionary computation. In this paper we present our evolutionary test environment, which performs fully automatic test data generation for most structural test methods based on actual execution of statistical testing through poisson distribution. We shall report on the results gained from the testing of real-world software modules. For most modules we reached full coverage for the structural test criteria.

KEYWORDS: Test Data Generation, Poisson distribution, Evolutionary Test, Structural Test case Design.

I.INTRODUCTION

A great number of today's products is based on the deployment of embedded systems. In industrial applications embedded systems are predominantly used for controlling and monitoring technical processes. There are examples in nearly all industrial areas, for example in aerospace technology, railway and motor vehicle technology, process and automation technology, communication technology, process and power engineering, as well as in defense electronics. Nearly 90% of all electronic components produced today are used in embedded systems.

In order to achieve high quality in the development of embedded systems, central importance is attributed to analytical quality assurance. In practice, the most important analytical quality assurance measure is dynamic testing. Thorough testing of the systems developed is essential for product quality. The aim of the test is to detect errors in the system under test, and, if no errors are found during comprehensive testing, to convey confidence in the correct functioning of the system. This is the only procedure which allows the testing of dynamical system behavior in a real application environment.

The most significant weakness of the test is that the postulated functioning of the tested system can, in principle, only be verified for those input situations selected as test data. Testing can only show the existence and not the nonexistence of errors. Therefore, the correctness proof can only be produced by a complete test. In practice, a complete test, with the exception of a few trivial cases, is not executable because of the enormous amount of possible input situations. Thus, the test is a sampling procedure. Accordingly, a task which is essential to testing is the selection of an appropriate sample containing the most error-sensitive test data.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 1, January 2014

Among the different test activities (test case design, test execution, monitoring, test evaluation, test planning, test organization, and test documentation – see Fig. 1) test case design is of essential importance.

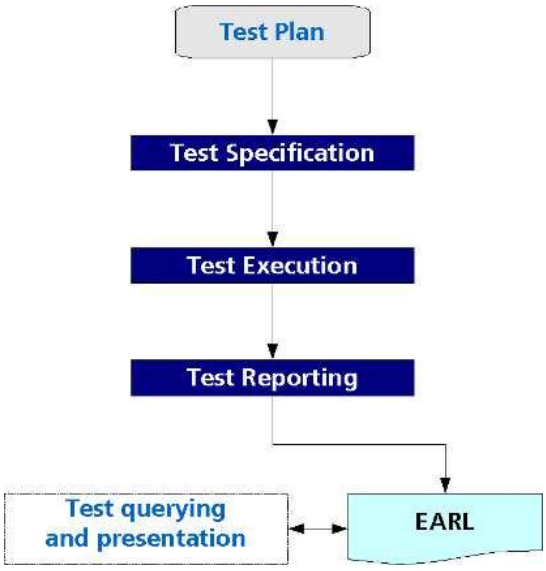

Figure 1: structure of test case generation

II. AUTOMATIC TEST-DATA GENERATION

Automated test-data generators can be divided into
Three classes – random, static and dynamic. Random test-data generation is easy to automate, but problematic [7–9]. First, it produces a statistically insignificant sample of the
possible paths through the software under test (SUT). Second, it may be expensive to generate the expected output-data for the large amount of input data produced. Finally, given that exceptions occur only rarely, the input domain which causes an exception is likely to be small. Random test-data

generators may never hit on this small area of the input domain. Static approaches to test-data generation generally use symbolic execution. Many test-data generation approaches presented in the literature use symbolic execution to obtain structural test-data [10–14]. Symbolic execution works by traversing a control flow graph of the SUT and building up symbolic representations of the internal variables in terms of the input variables, for the desired path. Branches within the code introduce constraints on the variables. Solutions to these constraints represent the desired test-data. A number of problems exist with this approach. Using symbolic execution it is difficult to analyses recursion, array indices which depend on input data and some loop structures. Also, the problem of solving arbitrary constraints is known to be non decidable.
Dynamic test-data generation involves execution of the SUT and a directed search for test-data that meets the desired criterion. The dynamic approach was first suggested in 1976 by Miller [15]. The work of Korel et al. built on this using locally directed search techniques [8,16–18]. This is further expanded by Gallagher et al. [19]. Local search techniques only work effectively for linear continuous functions. Consequently, these techniques are likely to become stuck at a local optimum and fail to locate the required global optimum [20]. The use of global optimization techniques for dynamic test data generation has been investigated more recently in an attempt to overcome this limitation [9,20–22].

III. THE TEST-DATA GENERATION PROBLEM

A control flow-graph is a directed graph which represents the control structure of a program. It can

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 1, January 2014

be described as follows [19]: G = (N,E, s, e), where N is a set of nodes, E is a set of edges of the form (ni, nj) represents a possible transfer of control from the basic block ni to the basic block nj . For branch instructions the edges are associated with a branch predicate. This describes the conditions which must hold for the branch to be taken.

A program is driven down a path in the control flow graph by the values of its input variables and the global state. These can be described as the vector x = αx1, x2, . . . xnα. Each variable will have an associated domain, Di , which can be determined from the variable's type. The total input space can be defined as the cross-product of each of these domains, D = D1 × D2 × · · · × Dn.

3.1. Raising exceptions

Firstly, consider the case of a user-defined exception. User-defined exceptions must be explicitly raised in Ada by executing the raise statement. Assuming the raise statement is contained in basic block ni ∈ N (more accurately a raise statement will always be the last statement in a basic-block as it causes an unconditional branch), the problem of testing the raising of a user-

Predefined exceptions are raised when the language rules are violated at run-time and in response to hardware errors. Test-data alone cannot test the raising of exceptions in response to hardware errors. For these hardware errors integration with a fault injection technique [20] is required. The focus, in this paper, is the generation of test-data which violates run-time language rules. In Ada there are a number of predefined exceptions [5]:

Constraint Error – data going out of range. Program Error – control-structure violation. Storage Error – running out of storage space. Tasking Error – general communications failure between tasks.

Our major concern is the development of software for safety critical systems. This type of development is often carried out using a 'safe' subset of a language. This allows the application of static analysis and potentially proofs to show adequate system safety. Our major concern with testing has been these kinds of systems, hence we have focused on the SPARK-Ada language [21,22]. SPARK-Ada allows testing for Constraint Error exceptions to be the focus of the work to date. The SPARK-Ada tool-set [21] mitigates against other pre-defined exceptions through

language restrictions or static analysis. Tasking Error cannot occur as Ada tasking is not part of SPARK Ada. Storage Error is also unlikely to occur as dynamic memory allocation is not being used and therefore storage requirements can be calculated statically. The situations where Program Error exceptions can occur are detected by the SPARK Examiner static analysis tool.

3.2. Automated test data generation (ATDG)

Most of the work on Software Testing has concerned the problem of generating inputs that provide a test suite that meets a test adequacy criterion. The schematic representation is presented in Fig.3. Often this problem of generating test inputs is called 'Automated Test Data Generation (ATDG)' though, strictly speaking, without an oracle, only the input is generated.

Fig.2 illustrates the generic form of the most common approach in the literature, in which test inputs are generated according to a test adequacy criteria [6].

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 1, January 2014

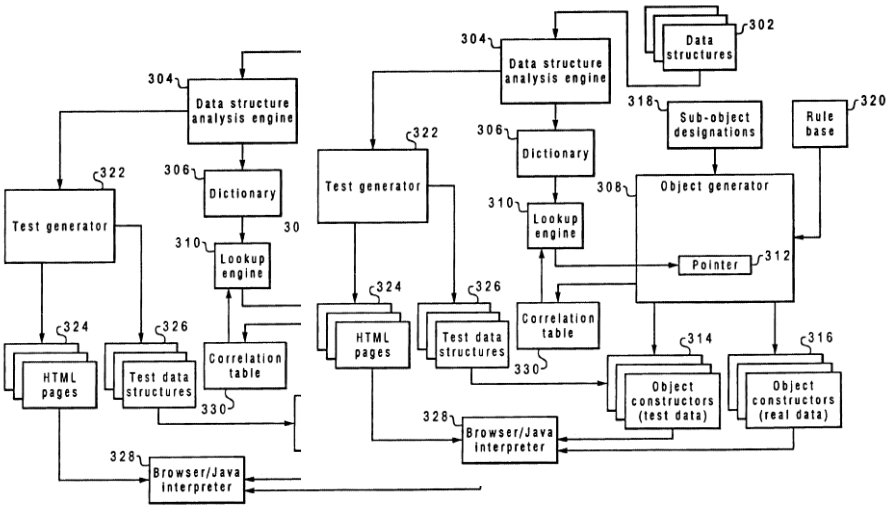

Figure 2: Evolutionary Test Environment Automated Test Data Generation

IV. EVOLUTIONARY TESTING

Evolutionary Testing (ET) [18] is a search-based software testing approach based on the theory of evolution. It formulates the task to generate relevant test data (relevant in terms of the testing objective at hand, such as maximizing structural coverage) as one or several search problems. Each search problem consists of the definition of the search space based on the input domain of the target program (e.g., its relevant parameters), and a fitness function that ET constructs. In the case of structural testing, such a search problem aims at finding a test data leading to the coverage of a particular branch. Each search problem is tried to be solved using an evolutionary algorithm: a pool of candidate test data, the so-called individuals, is iteratively manipulated by applying fitness evaluation, selection, mutation, and crossover in order to eventually obtain a relevant test data. Better fitness values are assigned to individuals that are better able to solve the search problem at hand, e.g., coming closer to covering the target branch during execution. [11].

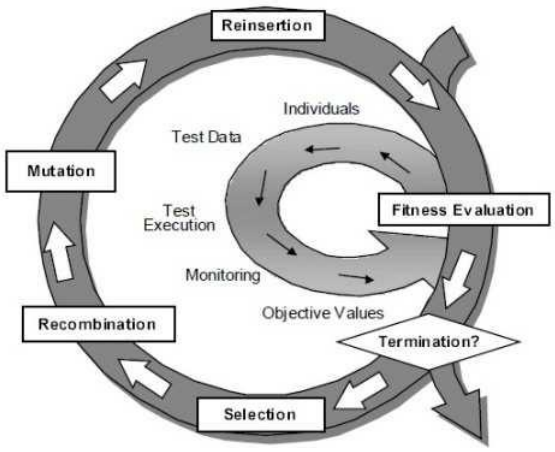

Figure 3: Structure of Evolutionary Testing

4.1. Evutionary test environment

In order to automate test case design for different structural testing methods with evolutionary tests we have developed a tool environment which consists of six components:

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 1, January 2014

• Parser for the analysis of test objects,
• Graphical user interface for the specification of the input domain of the test objects,
• instrumented which captures program structures executed by the generated test data,
• test driver generator which generates a test bed running the test object with the generated test data,
• test control which includes the identification and administration of the partial aims for the test and which guarantees an efficient test by defining a processing order and storage of initial values for the partial aims,
• toolbox of evolutionary algorithms to generate the test data.

4.2. parser

The parser identifies the functions in the source files which form the possible test objects. It determines all necessary structural information on the test objects. Control- flow and data-flow analyses are carried out for every test object. These analyses determine the interface, the control- flow graph, the contained branching conditions with their atomic predicates, as well as semantic information on the used data structures, e.g. the organization of user-defined data types.

GUI for interface specification To ensure efficient test data generation and to avoid the generation of inadmissible test data from the beginning, the tester may have to define the test object interface determined by the parser more precisely. For this, the developed tool environment provides a graphical user interface that displays the test objects and their interfaces as they have been determined by the parser. The tester can limit the value ranges for the input parameters and enter logical dependencies between different input parameters. These will then be considered during test data generation. It is also possible to enter initial values for single or for all input parameters. As a result, test data of a previous test run or data of an already existing functional test, as well as specific value combinations for single input parameters, can be used as a starting point for test data generation (seeding).

4.3. Test driver generator

The test driver generator generates a test bed that calls the test object with the generated individuals and returns the monitoring results provided by the execution of the instrumented test object to the test control. When the test object is called by the test driver, the individuals are mapped onto the interface of the test object. It is important that user specifications for the test object interface are taken into account. Individuals that do not represent a valid input are extracted and assigned a low fitness value.

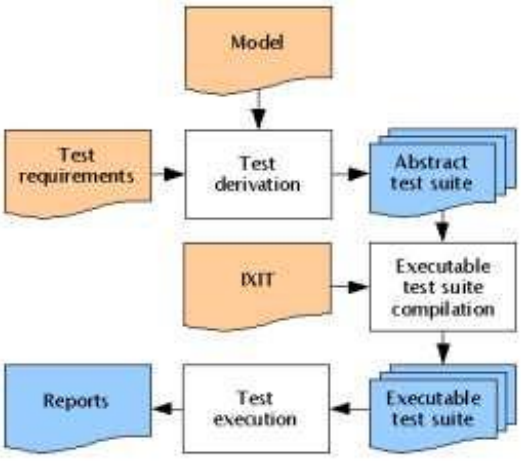

Figure 4: Test Case Generator for Evolutionary Testing

4.4. General scheme for Test case design

Now let us consider a given coverage criterion C. As a preliminary remark, note that the set of elements EC(D) must be finite, otherwise the quality of test would be zero. This implies, in particular, that the coverage

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 1, January 2014

criterion "all paths" is irrelevant as soon as there is a cycle in the description, like in our example (figure 4). Thus, this criterion has to be bounded by additional conditions, for example "all paths of length α n", "all paths of length between given n1 and n2", or "all paths which take at most m times each cycle in the graph". For the sake of simplicity, we consider in the following that paths are generated within P>n, the set of paths of length _ n that go from vs to ve. We consider two cases, according to the nature of the elements of EC(D). If EC(D) denotes a set of paths in the graph, we immediately state that the quality of test is optimal if the paths of EC(D) are generated uniformly, i.e. any path has the same probability 1=jEC(D)j to be generated. Indeed, if the probability of one or several paths was greater than 1=jEC(D)j, then there would exist at least one path with probability less than 1=jEC(D)j, therefore the quality of test would be lower.

Now, we consider the case where the elements of EC(D) are not paths, but are constitutive elements of the graph as, for example, vertices, edges, or cycles. Clearly, uniform generation of paths does not ensure optimal quality of test in this case. Ideally, the distribution on paths should ensure that the minimal probability to reach any element of EC(D) is maximal. Unfortunately, computing this distribution would require the resolution as as many equations as paths. This is generally impracticable. Thus we propose to generate a path in two steps:
1. pick at random one element e of EC(D), according to a suitable probability distribution.

2. generate uniformly at random one path of length _ n that goes through e. (This ensures a balanced coverage of the set of paths which cross e.)

4.5. Poisson Distribution

Often we are interested in the number of events which occur in a specific period of time or in a specific area of volume:
Number of alpha particles emitted from a radioactive source during a given period of time Number of telephone calls coming into an exchange during one unit of time Number of diseased trees per acre of a certain woodland Number of death claims received per day by an insurance company Characteristics Let X be the number of times a certain event occurs during a given unit of time (or in a given area, etc).
The probability that the event occurs in a given unit of time is the same for all the units. The number of events that occur in one unit of time is independent of the number of events in other units. The mean (or expected) rate is λ. Then X is a Poisson random variable with parameter λ and frequency function

p(x) =λ x/x!*e¡ ^λ; x = 0; 1; 2; : : :

4.6.Poisson distribution for an Evauationary test.

The problem consists in choosing the suitable probability distribution over EC(D) in order to maximize the quality of test. Given EC(D) = fe1; e2; : : : ; emg, with m >
0, the probability pi for the element ei (for any i in (1 α m)) to be reached by a path is

1 $$p_i = \pi_i + \sum_{j \in (1..m)-\{i\}} \pi_j \frac{\alpha_{i,j}}{\alpha_j}, \longrightarrow$$

where

- α_i is the number of paths of $\mathcal{P}_{\leq n}$ which takes element e_i;
- $\alpha_{i,j}$ is the number of paths which take both elements e_i and e_j; (note that $\alpha_{i,i} = \alpha_i$ and $\alpha_{i,j} = \alpha_{j,i}$);
- π_i is the probability of choosing element e_i during step 1 of the above process.

Indeed, the probability of choosing element ei in step 1 is _i; and the probability of reaching ei by drawing a random path which goes through another element ej is αi;j αj. The above equation simplifies to

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 1, January 2014

$$p_i = \sum_{j=1}^{m} \pi_j \frac{\alpha_{i,j}}{\alpha_j} \qquad 2$$

Let us illustrate this with our example. Given the coverage criterion "all the edges" and given n = 10, Table 1 presents the coefficients αi;j , where i and j denote letters from 'a' to 'k'. For example, the value '9' in row 'f' and column 'c' means that λ c;f = 9, i.e. there are exactly 9 paths of length lower or equal to 10 from vs to ve which cross both edges c and f in the graph of Figure 4.

The corresponding linear program is shown in Table 5. Each line, but the last one, is an in equation which corresponds to a row in Table 4. The first term of the in equation is p min, the value to be maximized.

	a	b	c	d	e	f	g	h	i	j	k
a	9	0	9	0	5	7	5	5	6	6	3
b	0	5	3	5	1	2	1	4	3	3	2
c	9	3	12	3	6	9	6	8	9	8	4
d	0	5	3	5	1	2	1	4	3	3	2
e	5	1	6	1	6	3	6	3	5	5	1
f	7	2	9	2	3	9	3	7	7	5	4
g	5	1	6	1	6	3	6	3	5	5	1
h	5	4	8	4	3	7	3	9	7	7	2
i	6	3	9	3	5	7	5	7	9	6	3
j	6	3	8	3	5	5	5	7	6	9	0
k	3	2	4	2	1	4	1	2	3	0	5

TABLE 1: Table of α i,j

The second term is one of the pi's, computed according to Formula 2. For example, the first line means that p min must be lower or equal to pa, the probability of reaching edge 'a' with a random path. By maximizing p min, one maximizes the lowest pi, so that the quality of test is optimal. The last line ensures that the probabilities λi that we are searching for sum to 1.

V. EVALUATION

This section presents the results of an evaluation of the optimization based approach to generating test data for exception conditions. The evaluation has been performed in two parts. Firstly, a collection of small Ada 95 programs have been used to provide a preliminary assessment of the ability of the system to generate test-data to raise particular exceptions. Test-data to achieve exception condition coverage has also been targeted. Secondly, integration of the test-data generation and proof of exception freeness is evaluated using the code for a civil aircraft engine controller.

5.1 Simple examples

A number of Ada 95 programs have been used to evaluate the effectiveness of this test-data generation approach. The routines are between 10 and 200 lines of code. Square simply squares the input parameter. However, the data-types are defined such that an overflow exception will be raised with large input values. IntSqrt is an integer square-root routine that uses a binary search algorithm. Find performs either a linear or binary search to locate a value in an array. A user-defined exception and handler is invoked if a binary search is requested on an unsorted array.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 1, January 2014

SUT Name	INP	NE	EF	C	T
Square	20 002	1	1	N/A	0.5s
Int Sqrt	10 000	3	3	N/A	1.1s
Find	$1e+44$	2	2	100%	2.0s
Remainder	$1e+8$	2	2	N/A	2.4s
Tomorrow	286 440	5	5	100%	4.8s
Convert	$1.7e+10$	7	7	N/A	17.2s
BigInt Div	$1e+50$	4	3	100%	36.2s
Total		24	23		

Table 2: Evaluation results (C - branch coverage of exception handlers, T - test-data generation time).

Remainder calculates the remainder and quotient given two input parameters. Tomorrow calculates tomorrow's day, date, month and year taking into account leap year calculations. User-defined error handling is used to validate the input date. Convert performs conversions between binary, octal, decimal, and hexadecimal and Roman numeral strings. BigInt Div performs an integer division using arbitrary length integer abstract data types. Error seeding was used on a number of the programs to introduce errors that allowed exceptions to be generated. automatically generated to test exceptions. However, these results are only on relatively trivial programs. The authors' focus is the provision of automatic testing solutions for high-integrity safety-critical systems. A more detailed evaluation of the test-data generation technique is presented in the next section for such a system.

VI. CONCLUSIONS

Many of the approaches for automated software test-data generation presented in the literature are inflexible or have limited capacity. Optimization techniques in contrast offer a flexible and efficient approach to solving complex problems. To allow the optimization based framework to generate test data for a specific testing criterion it is necessary only to devise a suitable fitness function. this paper we presented our evolutionary test environment, which performs fully automatic test data generation for most structural test methods based on actual execution of statistical testing through poisson distribution. We shall report on the results gained from the testing of real-world software modules.

VII. FURTHER WORK

The results presented above show that it is possible to use Poisson distribution to generate test-data for the testing of exceptions. However, more research is required in order to assess the limitations of the approach. The expression structure in the safety-critical code was very simple. This allowed the automatic simplifier to discharge the vast majority of the verification conditions itself. The test-data generation could then be targeted towards only the remaining verification conditions. The application of the test-data generation to a system.

Where exceptions can be raised from many expressions (and indeed sub expressions) may be very time consuming and no longer practical. A new search would be required for every possible point where an exception may be raised. However, safety-critical systems by their very nature tend to have simple control-flowed expression structuring as was the case with the engine controller code.

REFERENCES

[1] A.Denise, M.-C. Gaudel and S.-D. Gouraud, "A Generic Method for Statistical Testing" In 5th IEEE International Conference on Software Engineering, pages 179–183, San Diego, march 2001.
[2] Barnes J. High Integrity Ada: The SPARK Approach; Addison- Wesley, 1997.
[3] Clark J, Tracey N. Solving constraints in LAW. LAW/D5.1.1(E), European Commission - DG III Industry, 1997. Legacy Assessment Workbench Feasibility Assessment.
[4] Deo N. Graph Theory with Applications to Engineering and Computer Science; Prentice-Hall, 1974.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 1, January 2014

[5] Gallagher MJ, Narashimhan VL. ADTEST: A test data generation suite for ada software systems. IEEE Transactions on Software Engineering 1997; 23(8):473–484.
[6] Jones B, Sthamer H, Eyres D. Automatic structural testing using genetic algorithms. Software Engineering Journal 1996; 11(5):299–306.
[7] Jones BF, Sthamer HH, Eyres DE. Generating test-data for Ada procedures using gentic algorithms. Genetic Algorithms in Engineering Systems: Innovations and Applications; IEEE, September 1995; 65–70.
[8] Joachim Wegener, Kerstin Buhr, Hartmut Pohlheim, "automated test data generation for structural testing of embedded software system by evolutionary testing" IEC 65A Software for Computers in the Application of Industrial Safety-Related Systems (Sec 122).
[9] Kirkpatrick Jr S, Gelatt CD, Vecchi MP. Optimization by simulated annealing. Science 1983; 220(4598):671–680.
[10] Mark Harman1 Fayezin Islam2 Tao Xie3 Stefan Wappler4 "Automated Test Data Generation for Aspect-Oriented Programs" AOSD'09, March 2–6, 2009, Charlottesville, Virginia, USA. Copyright 2009 ACM 978-1-60558-442-3/09/03 ...$5.00.
[11] N. Tracey, J. Clark, K. Mander and J. McDermid, "Automated test- data generation for exception conditions" SOFTWARE—PRACTICE AND EXPERIENCE Softw. Pract. Exper. 2000; 30:61–79
[12] Ntafos SC. A comparison of some structural testing strategies. IEEE Transactions on Software Engineering 1988; 14(6):868–874.
[13] P. Maragathavalli, "SEARCH-BASED SOFTWARE TEST DATA GENERATION USING EVOLUTIONARY COMPUTATION", International Journal of Computer Science & Information Technology (IJCSIT), Vol 3, No 1, Feb 2011
[14] Praxis Critical Systems. Spark-Ada Documentation 2.0, 1995.
[15] Tracey N, Clark J, Mander K. Automated program flaw finding using simulated annealing. International Symposium on Software Testing and Analysis; ACM/SIGSOFT, 1998; 73–81.
[16] Tracey N, Clark J, Mander K. The way forward for unifying dynamic test case generation: The optimisation-based approach. International Workshop on Dependable Computing and Its Applications; IFIP,1998; 169–180.
[17] Tracey N, Clark J, Mander K, McDermid J. An automated framework for structural test-data generation. Proceedings of the International Conference on Automated Software Engineering; IEEE, October 1998.
[18] Tracey N, Clark J, McDermid J, Mander K. Integrating safety analysis with automatic test-data generation for software safety verification. Proceedings of the 17th International Conference on System Safety; IEEE, August 1999.
[19] Voas JM, McGraw G. Software Fault Injection: Inoculating Programs Against Errors; Wiley, 1998.
[20] Watkins AL. The automatic generation of test data using genetic algorithms. Proceedings of the 4th Software Quality Conference,1995; 2:300–309.
[21] Xanthakis S, Ellis C, Skourlas C, Le Gall A, Katsikas S, Karapoulios K. Application des algorithmes genetiques au test des logiciels. Proceedings of 5th International Conference on Software Engineering, 1992; 625–638.
[22] Zhu H, Hall PAV, May JHR. Software unit test coverage and adequacy. ACM Computing Surveys 1997; 29(4):366–427.