**International Journal of Computational Intelligence Techniques**
ISSN: 0976-0466 & E-ISSN: 0976-0474, Volume 3, Issue 1, 2012, pp.-60-64.
Available online at http://www.bioinfo.in/contents.php?id=36

# GENETIC ALGORITHM BASED SOFTWARE TESTING SPECIFICALLY STRUCTURAL TESTING FOR SOFTWARE RELIABILITY ENHANCEMENT

**NIRPAL P.B. AND KALE K.V.**

Department of CS & IT, Dr. B.A. Marathwada University, Aurangabad- 431004, MS, India.
*Corresponding Author: Email- premal.nirpal@gmail.com

Received: January 21, 2012; Accepted: April 18, 2012

**Abstract-** In this paper we focus on software reliability with testing coverage, which will increase the software efficiency. Testing costs often account for up to 50% of the total expense of software development; hence any techniques leading to the automatic generation of test data will have great potential to considerably reduce costs. Hence, software quality managers have been looking for solutions to reduce testing costs and time. In our work, genetic algorithms that can automatically generate test cases to test selected path. This algorithm takes a selected path as a target and executes sequences of operators iteratively for test cases to evolve. We describe the implementation of our GA-based system and examine the effectiveness of this approach on a number of programs.
**Key words**- Software Testing, Test case generation, Path Coverage, Genetic Algorithms, Software Reliability.

**Citation:** Nirpal P.B. and Kale K.V. (2012) Genetic Algorithm Based Software Testing Specifically Structural Testing for Software Reliability Enhancement. International Journal of Computational Intelligence Techniques, ISSN: 0976-0466 & E-ISSN: 0976-0474, Volume 3, Issue 1, pp.-60-64.



## Introduction

The software testing is designed with the purpose of detecting defects for a given set of inputs and estimating the operational effectiveness and suitability of the program being developed. Software program could be viewed as a function that describes the relationship of an input to an output. The testing process is used to ensure that the program realizes the function and its problem could be represented by the testing problem. The essential components of a software program test are a description of the functional input range, the program in executable form, a description of the expected behavior, a way of observing program behavior and a method of determining whether the observed behavior conforms to the expected behavior. Generally, software-testing techniques are classified into two categories: static analysis and dynamic testing.

This paper includes six sections, background of various software testing techniques in section 2, a literature study of genetic algorithm based approaches to test data generator in section 3, introduction to genetic algorithms in section 4, experimental results and discussion in section 5, this paper concludes in section 6

## Software Testing Techniques

Software-testing techniques are classified into two categories: static analysis and dynamic testing. In static analysis, a code reviewer reads the source code of the program or software under test, statement by statement and visually follows the program logic flow by feeding an input. This type of testing is highly dependent on the reviewer's experience. Typical static analysis methods are code inspections, code walkthroughs and code reviews [1]. In contrast to static analysis, which uses the program requirements and design documents for visual review, dynamic testing techniques execute the program under test on test input data and observe its output. Usually, the term testing refers to just dynamic testing.

Dynamic testing can be classified into two categories: black-box and white-box. White-box testing is concerned with the degree to which test cases exercise or cover the logic flow of the program [1,2]. Therefore, this type of testing is also known as logic-coverage testing or structural testing, because it considers the structure of the program. Black-box testing, a.k.a. functional or specification-based testing, on the other hand, tests the function-

alities of software irrespective of its structure. Functional testing is interested only in verifying the output in response to given input data.

This paper focuses on structural testing. Competence of logic-coverage testing can be judged using different criteria: statement, decision, condition and path coverage [1,2].

**A literature study of Genetic algorithm based approaches to test data generator**

One of the major difficulties in software testing is the automatic generation of test data that satisfy a given adequacy criterion. To solve this difficult problem there were a lot of research works, which have been done in the last 19 years. Perhaps the most commonly encountered are random test-data generation, symbolic (or path-oriented) test-data generation, dynamic test-data generation and recently, test-data generation based on genetic algorithms (GAs).

Xanthakis et al. in [3] is presented the first work applying genetic algorithms to generate test data. In this work GAs are employed to generate test data for structures not covered by random search. A path is chosen by the user and the relevant branch predicates are extracted from the program. The GA is then used to find input data that satisfies all branch predicates at once, with the fitness function summing branch distance values.

Alan Schultz et al [4] in 1993 propose a machine learning techniques to evaluate autonomous vehicle software controller. A set of simulated fault scenarios is applied to controller and a genetic algorithm searches for significant combination of fault. This approach to find a minimum set of faults that produces degraded vehicle performance and maximum set of faults that can be tolerated without significant performance loss.

Pei et al. [5] in 1994 observed that most of the test data generators, which were developed in their era, were using symbolic evaluation. They observed that both static and gradient-descent-based dynamic testing was developed. However, they concluded that static testing was not practical, while the dynamic one was not effective. These drawbacks had inspired Pei et al. to develop a single-path-coverage test data generator that employs GA.

Hunt J. et al. [6] 1995 presents a genetic algorithm designed to search for significant input and output combinations to software control system. By "significant" is meant those which produce an output (or result) which is not in line with the original specification. It is intended that such a tool should be used to support the human tester by focusing their attention on areas of concern which they can investigate further.

Around the same time, Roper et al. [7] in 1995 developed a GA-based test data generator that has an aim to traverse all the branches within a target program. Their generator takes a program and instruments it automatically with probes to provide feedback on the branch coverage achieved.

Alander, J. et al [8] in 1996 studying possibilities to test software using genetic algorithm search. The idea is to produce test cases in order to find problematic situation like processing time extremes. The proposed test method comes under the heading of automated dynamic stress testing.

One year after, Jones et al. [9] in 1996 developed a similar GA-based test data generator to achieve branch coverage. Their major contributions are the use of a sequence of binary strings for individual representation, which is converted to a decimal number prior to the program execution and the use of unrolled control flow graph (CFG) to represent one, two, or more iterations for each loop, which makes the CFG acyclic. As each branch is executed, the test data generator automatically traverses the CFG to the next branch in a breadth-first manner.

Michael et al. [10] in 1997 implemented Korel's function minimization approach [21] in their GA-based test data generator. They have built a test data generator called GADGET (Genetic Algorithm Data Generation Tool), which has the ability to instrument a program automatically with no limitation in the programming language, but it has a restriction that it can only accept scalar inputs. GADGET has the condition-decision coverage as its adequacy criteria. GADGET uses simple GA as well as differential GA. The difference between differential GA and the simple GA is in the recombination process [21]. Michael et al.'s result shows that, in general, the simple GA outperforms the differential one. GADGET is considered to be the first test data generator to be tested against a large real-world program named b737, which is part of an autopilot system (real-world control software). Michael et al. reported that the performance of random test generation deteriorates for larger programs.

Tracey N. et al. [11] in 1998 develop a generalized test-case data generation framework based on optimization techniques. The framework can incorporate a number of testing criteria, for both functional and non-functional properties. Application of the optimization framework to testing specification failures and exception conditions is illustrated. The results of a number of small case studies are presented and show the efficiency and effectiveness of this dynamic optimization-base approach to generating test-data.

The work done by Pargas et al. [12] in 1999 is an improvement to Jones et al.'s work. The approach they presented also uses branch information to evaluate the fitness function, except it uses control dependency graph for the fitness evaluation, which they claimed it can give more precise fitness evaluation than Jones et al.'s and Michael et al.'s approaches. Pargas et al. parallelized GA to make it faster and also claimed that the approach can provide path coverage with minor modifications.

Lin and Yeh [13] in 2000 extended the work done by Jones et al. In their work, they used the path coverage criterion rather than branch coverage. They also extended the ordinary (weighted) hamming distance such that it can handle different ordering of the target paths that have the same branch nodes. The rationale here is that, in path testing, two different paths may contain the same branches but in different sequences, where the simple hamming distance is no longer suitable. They name the fitness function SIMILARITY, since it calculates the similar items with respect to their ordering within the two different paths, i.e. branches, between the current executed path and the target path. The greater SIMILARITY leads to the better fitness.

Bueno and Jino [14] in 2000 proposed an approach that utilizes control and data flow dynamic information. The proposed approach is meant to fulfill path coverage testing. In addition, it also tackles the identification of potentially infeasible program paths by monitoring the progress of the search for required test data. The approach considers a continual population's best fitness improvement as an indication that a feasible path is covered. On the other hand, attempts to generate test data for infeasible paths result,

invariably, in a persistent lack of progress.

Wegener et al. [15] in 2002 developed a fully automatic GA-based test data generator for structural software testing, specifically statement and branch coverage, of real-world embedded software systems. Their fitness function consists of two major building blocks: approximation level and normalized predicate local distance. The approximation level indicates the number of continuously matched branching nodes between the traversed branches by an individual and a target branches (they call it "partial aim"). The local distance is calculated for the individual by means of the branching conditions in the branching node in which the target node is missed. Unfortunately, the report does not describe the normalization scheme of the local distance value. Overall fitness value is the summation of the approximation level value and the local distance value. Although their tool works on only one partial aim after the other, it takes into consideration the execution of a test datum that usually leads to passing several partial aims. Thus, the test soon focuses on those partial aims that are difficult to reach. The stopping criteria used is full statement/branch coverage and number of generations; whichever is satisfied first.

Wegener et al.'s paper does not discuss as whether multiple targets can be covered in first attempt. However, the approach, or more precisely "the test control", evaluates all individuals generated with respect to all unachieved targets. Thus, other targets found by chance are identified and individuals with good fitness values for one or more targets are noted and stored for seeding the next subsequent testing of uncovered targets. Furthermore, they reported that full coverage of some programs is achieved, but not for all programs though. According to their research work, they are investigating whether infeasible statements/branches or the number of generations are some of the reasons for not being able to achieve full coverage in some programs.

Daz E., et al. [16] in 2003 in this paper, author explain how to created an efficient testing technique that combines Tabu Search with Korel's chaining approach. This technique automatically generates test data in order to obtain branch coverage in software testing.

Berndt D., et al. [17] in 2003 in this paper focuses on breeding software test cases using genetic algorithms as part of a software testing cycle. An evolving fitness function that relies on a fossil record of organisms results in interesting search behaviors, based on the concepts of novelty, proximity and severity. A case study that uses a simple, but widely studied program is used to illustrate the approach.

Tonella P., et al. [18] in 2004 in this paper, a genetic algorithm is exploited to automatically produce test cases for the unit testing of classes in a generic usage scenario. Test cases are described by chromosomes, which include information on which objects to create, which methods to invoke and which values to use as inputs. The proposed algorithm mutates them with the aim of maximizing a given coverage measure. The implementation of the algorithm and its application to classes from the Java standard library are described.

D. J. Berndt et al. [19] in 2005 in this paper explores strategies that combine automated test suite generation techniques with high volume or long sequence testing. Long sequence testing repeats test cases many times, simulating extended execution intervals. These testing techniques have been found useful for uncovering errors resulting from component coordination problems, as well as system resource consumption (e.g. memory leaks) or corruption. Coupling automated test suite generation with long sequence testing could make this approach more scalable and effective in the field.

James Miller et al. [20] in 2005 presents a new approach utilizing program dependence analysis techniques and genetic algorithms (GAs) to generate test data. A set of experiments using the new approach is reported to show its effectiveness and efficiency based upon established criterion.

Abdelhamid Bouchachia [21] in 2007 proposed hybrid algorithm is called Immune Genetic Algorithm (IGA). A full description of this algorithm is presented before investigating its application in the context of software test data generation using some benchmark programs. Moreover, the algorithm is compared with other evolutionary algorithms.

Yong Chen et al.'s [22] in 2009 proposed two fitness function based on branch distance (BDBFF) and another based on normalized extended Hamming distance (SIMILARITY) are both applied in GA-based path oriented test data generation. To compare performance of these two fitness functions, a triangle classification program was chosen as the example.

**Genetic Algorithms**

Genetic Algorithms begins with a set of initial individuals as the first generation, which are sampled at random from the problem domain. The algorithms are developed to perform a series of operations that transform the present generation into a new, fitter generation [22].

Each individual in each generation is evaluated with a fitness function. Based on the evaluation, the evolution of the individuals may approach the optimal solution.

The most common operations of genetic algorithms are designed to produce efficient solution for the target problem [15]. These primary operations include:

**Reproduction-** This operation assigns the reproduction probability to each individual based on the output of the fitness function. The individual with a higher ranking is given a greater probability for reproduction. As a result, the fitter individuals are allowed a better survival chance from one generation to the next.

**Crossover-** This operation is used to produce the descendants that make up the next generation. This operation involves the following crossbreeding procedures:

- Randomly select two individuals as a couple from the parent generation.
- Randomly select a position of the genes, corresponding to this couple, as the crossover point. Thus, each gene is divided into two parts.
- Exchange the first parts of both genes corresponding to the couple.
- Add the two resulted individuals to the next generation.

**Mutation-** This operation picks a gene at random and changing its state according to the mutation probability. The purpose of the mutation operation is to maintain the diversity in a generation to prevent premature convergence to a local optimal solution. The mutation probability is given intuitively since there is no definite way to determine the mutation probability [22].

Upon completion of crossover processing and mutation operations, there will be an original parent population and a new offspring population. A fitness function should be devised to determine which of these parents and offspring's can be survived into the next generation. After performing the fitness function, these parents and offspring's are filtered and a new generation is formed. These operations are iterated until the expected goal is achieved. Genetic algorithms guarantee high probability of improving the quality of the individuals over several generations according to the Schema Theorem.

**Experimental studies**

**Example of Triangle classification program**

Triangle classification program has been widely used in the research area of software testing [22, 24]. It aims to determine if three input edges can form a triangle and so what type of triangle can be formed by them. Fig. 1 gives source code of the program.

**An example program**

```
path1=;%instrument branch#1
Triangle='Not a Triangle';
        if((a+b>c)&&(b+c>a)&&(c+a>b))
path1=[path1 'a']; %instrument branch#2
        if((a~=b)&&(b~=c)&&(c~=a))
path1=[path1 'e']; %instrument branch#3
Triangle='Scalene';
        else
path1=[path1 'b']; %instrument branch#4
        if(((a==b)&&(b~=c))||((b==c)&&(c~=a))||((c==a)&&(a~=b)))
path1=[path1 'f']; %instrument branch#5
Triangle='Isosceles';
        else
path1=[path1 'c']; %instrument branch#6
Triangle='Equilateral';
        end
                end
        else
path1=[path1 'd']; %instrument branch#7
        end
                end
```

For test generation we have taken the triangle classification problem. Our experiment of the general path testing follows the four steps: Control flow graph construction, Target path selection, Test case generation and execution and Test result evaluation.

**Control flow graph construction**

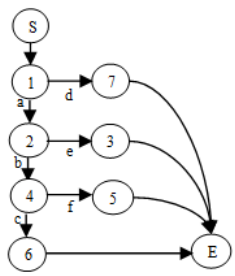


**Fig. 1-** Control flow graph of the triangle classification program

**Target path selection**

Path I: <d> //Not-a-triangle
Path 2: <ae> //Scalene
Path 3: <abf> //Isosceles
Path 4: <abc> //Equilateral
According to probability theory, the path <abc> is the most difficult path to be covered in path testing. Therefore, the path <abc> is selected as the target path.

**Test case generation and execution**

According to the genetic algorithms, an experimental tool for automatically generating test cases to test a specific path is developed

**Test result evaluation**

This step is to execute the selected path with the test cases found in step (3) and to determine whether the outputs are correct or not.

**Results**

In this experiment we have used GA for 1000 generations with n=15, initial population with 1000 test cases. The size of the chromosome is 3. Mutation rate is 0.01. Selection rate 0.5

*Table 1 Average number of test cases on the path of Fig. 1 of 1-10 generation*

| Generation | <abc> Equilateral | <d> Not a Triangle | <ae> Scalene | <abf> Isosceles | time | total test cases |
|---|---|---|---|---|---|---|
| 1 | 6 | 492 | 370 | 132 | 0.0753 | 1000 |
| 2 | 1 | 376 | 94 | 29 | 0.0416 | 500 |
| 3 | 0 | 347 | 104 | 49 | 0.0436 | 500 |
| 4 | 0 | 357 | 100 | 43 | 0.0438 | 500 |
| 5 | 0 | 369 | 97 | 34 | 0.0445 | 500 |
| 6 | 4 | 332 | 115 | 49 | 0.0442 | 500 |
| 7 | 2 | 354 | 92 | 52 | 0.0436 | 500 |
| 8 | 0 | 339 | 114 | 47 | 0.044 | 500 |
| 9 | 2 | 342 | 115 | 41 | 0.044 | 500 |
| 10 | 1 | 344 | 113 | 42 | 0.0432 | 500 |

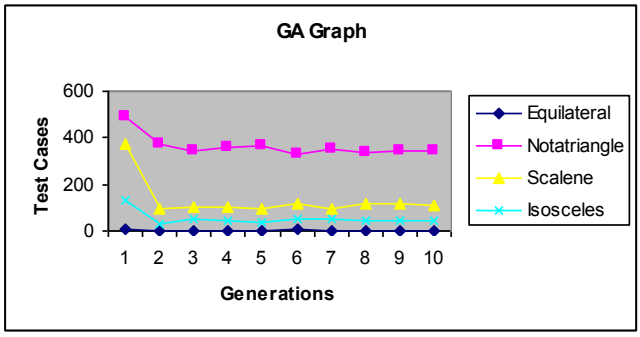


**Fig. 2-** Average number of test cases on the path of Fig. 1 of 1-10 generation

In this work the software tests data generation using genetic algorithm. Throughout this work triangle classification problem was used for experiment. During result process around 100 figures and almost 100 tables are used to show the outcome of this work. The output is evaluated by passing the proposed algorithm through 1000 generation to test the accuracy. The experimental work consists of various types of triangle likes Equilateral, Not a

Triangle, Scalene and Isosceles. The outcome is evaluated by verifying the number of various triangles generated in 1000 test cases.

We have used automation coverage for software reliability enhancement. This software testing metric gives the percentage of manual test cases automated.

$$\text{Automation Coverage} = \left[\frac{\text{Total No. of TC Automated}}{\text{Total No. of manual TC}} * 100\right]\% \quad \ldots (1)$$

Example: If there are 100 Manual test cases and one has automated 60 test cases then Automation Coverage = 60%.

*Table 2- Reliability achieved in percentage using automation coverage.*

| Sr. No | Generation | Number of automated test cases | Number of manual test cases | Percentage (%) |
|---|---|---|---|---|
| 1 | 1 | 1000 | 1000 | 100% |
| 2 | 2 | 500 | 1000 | 50% |
| 3 | 3 | 500 | 1000 | 50% |
| 4 | 4 | 500 | 1000 | 50% |
| 5 | 5 | 500 | 1000 | 50% |
| 6 | 6 | 500 | 1000 | 50% |
| 7 | 7 | 500 | 1000 | 50% |
| 8 | 8 | 500 | 1000 | 50% |
| 9 | 9 | 500 | 1000 | 50% |
| 10 | 10 | 500 | 1000 | 50% |

The final reliability in this work is calculated by equation number 1 and shown table it is 2 found that for test cases number 1 generation the reliability is 100% where as for remaining all 1000 generation it is 50%. The reliability trends of our research work are shown in Fig. 3.

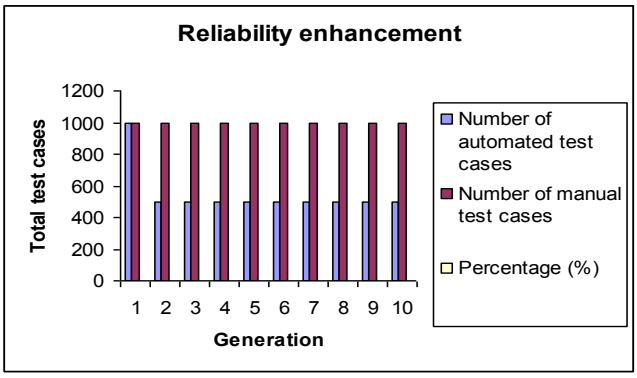


**Fig. 3-** Reliability achieved in percentage using automation coverage

## Conclusion

In this experimental work we have used triangle problem for test data generation. To carry out these experiments we have performed above presented results. The various experimental results which are carried out through triangle problem using genetic algorithm based test data generation. It is hoped that the limitations of this work would be considered as the beginning for the research in future. Effectiveness and Efficiency - the effectiveness and efficiency of the software can be improved.

## Acknowledgment

The authors wish to acknowledge UGC for the award of Research Fellowship under Fellowship in Sciences to Meritorious Students (RFSMS) scheme for carrying out this research.

## References

[1] Roger Pressman (1997) *"Software Engineering" A Practitioner's Approach 5th Edition, McGraw Hill*.

[2] Beizer B. (1990) *Software Testing Techniques 2nd Edition, International Thomson Computer Press*.

[3] Xanthakis S., Ellis C., Skourlas C., Le Gall A., Kastiskas S., Karapoulios K. (1992) *In 5th International Conference on Software Engineering and its Applications*, 625-636.

[4] Alan C. Schultz, John J. Grefenstette, aid Kenneth A. De Jong (1993) IEEE- *Test and Evaluation by Genetic Algorithms, Digital Object Identifier,* 8(5).

[5] Pei M., Goodman E.D., Gao Z. and Zhong K. (1994) *Automated Software Test Data Generation Using A Genetic Algorithm*.

[6] Hunt J. (1995) *Testing Control Software using a Genetic Algorithm", Working Paper, University of Wales, UK*.

[7] Roper M., Maclean I., Brooks A., Miller J. and Wood M. (1995) *Genetic Algorithms and the Automatic Generation of Test Data*.

[8] Alander J.T., Mantere T. and Turunen P. (1997) *Genetic Algorithm Based Software Testing*.

[9] Jones B., Sthamer H. and Eyres D. (1996) *Software Engineering Journal* 11(5), 299-306.

[10] Michael C.C., McGraw G.E., Schatz M.A. and Walton C.C. (1997) *Genetic Algorithms for Dynamic Test Data Generation", Technical report, Reliable Software Technologies, Sterling*.

[11] Tracey N.J., Clark J., Mander K. and McDermid J. (1998) 13*th IEEE Conference in Automated Software Engineering, Hawaii*.

[12] Pargas R.P., Harrold M.J. and Peck R.R. (1999) *Journal of Software Testing, Verification and Reliability*.

[13] Lin J.C. and Yeh P.L. (2000) *9th Asian Test Symposium*.

[14] Bueno P.M.S. and Jino M. (2000) *Fifteenth IEEE International Conference on Automated Software Engineering*, 209-218.

[15] Wegener J., Buhr K. and Pohlheim H. (2002) *Genetic and Evolutionary Computation Conference*.

[16] Daz E., Tuya J. and Blanco R. (2003) 18*th IEEE International Conference on Automated Software Engineering*, 310-313.

[17] Berndt D.J., Fisher J., Johnson L., Pinglikar J. and Watkins A. (2003) *Thirty-Sixth Hawai`i International Conference on System Sciences*.

[18] Tonella P. (2004) *ACM SIGSOFT international symposium on Software testing and analysis*, 119-128.

[19] Berndt D.J., Watkins A. (2005) 38*th Annual Hawaii International Conference on System Sciences, Track* 9.

[20] James Miller, Marek Reformat and Howard Zhang (2006) *Science Direct, Information and Software Technology,* 48, 586 -605.

[21] Abdelhamid Bouchachia (2007) *IEEE, Seventh International Conference on Hybrid Intelligent Systems*.

[22] Chen Yong and Zhong Yong (2008) *Fourth International Conference on Natural Computation*.