ijecs open access www.ijecs.in
International Journal Of Engineering And Computer Science ISSN: 2319-7242
Volume 5 Issues 6 June 2016, Page No. 16846-16854

Test Path Generation Using Cellular Automata

Vandana[1], Neha Sewal[2], Shailja Gupta[3]

[1]M.Tech Scholar, NGF College of Engineering and Technology,
Department of Computer Science and Engineering, Palwal
saivandana4@gmail.com
[2]Assistant Professor, NGF College of Engineering and Technology,
Department of Computer Science and Engineering, Palwal
neha.sawal@gmail.com
[3]Assistant Professor, Manav Rachna University,
Department of Computer Science and Engineering, Faridabad
shailjaguptaymca@gmail.com

Abstract: ***The growth of software engineering can justifiably be attributed to the advancement in Software Testing. The quality of the test cases to be used in Software Testing determines the quality of software testing. This is the reason why test cases are primarily crafted manually. However, generating test cases manually is an intense, complex and time consuming task. There is, therefore, an immediate need for an automated test data generator which accomplishes the task with the same effectiveness as manual crafting of test cases. The work presented intends to automate the process of Test Path Generation with a goal of attaining maximum coverage. The work presents a technique using Cellular Automata (CA) for generating test paths. The work opens the window of Cellular Automata to Software Testing. The approach has been verified on programs selected in accordance with their Lines of Code and utility. The results obtained have been verified.***

Keywords: Test Path Generation, Cellular Automata, Software Testing, Path Coverage**.**

1. Introduction

Manual test data generation is tedious and it accounts for a considerable percentage of time and cost. So, there is a need to make Automatic Test Data Generation (ATDG) more efficient, both in terms of time and coverage. This will enhance the reliability of testing and the quality of the product being tested. As per the literature, the best test case suite is the one that maximizes the coverage of code and at the same time minimizes the Oracle cost. Oracle cost is the combination of the cost of executing the entire test suite and the cost of checking the system behavior as the whole [1]. The coverage is maximum when there is at least one test case in the suite for each independent path of the Program under Test (PUT). To minimize the Oracle cost, the number of test cases to be executed must be reduced.

The work being carried out takes into account the coverage criteria and minimization of the oracle cost. These bi-objective optimization criteria can be achieved by taking the best possible test case, from amongst the set of test cases generated, for each individual path. In order to do so, just one test case would be considered for a path, so the need of minimizing the oracle cost is eliminated. The task of generating paths is accomplished using the concept of Cellular Automata (CA). The procedure also considers programs having many modules and thus paves the way for its applicability to component based programming as well. This is necessary as most of the existing test data generators fail to consider the interaction between modules.

The rest of the paper is organized as follows. Section 2 presents the literature Review. Section 3 presents the proposed architecture. Section 4 demonstrates the result, followed by the conclusion and future scope in section 5.

2. Literature Review

Today, the main challenge in front of software engineering is crafting of automated test cases. An extensive review has been conducted to study the existing techniques to craft the test cases. The work done so far is classified into various categories depending on the technique applied by various researchers, which are as follows.

- Techniques based on Genetic Algorithms
- Pairwise Testing approach for Test Data generation
- Random Technique
- Other Techniques
- Hybrid Approach
- Multi-Objective Approach

Various Techniques have been proposed to automate the process of Test Data Generation in which many techniques are based on Evolutionary Algorithms. A systematic survey was conducted by McMinn in 2004, on Search based software Test Data Generation [2]. Various papers related to these techniques have been studied and thoroughly analyzed. Most of the researchers used Genetic Algorithms (GAs) to solve the problem of ATDG. Various tools were generated using GAs such as GADGET [3], TDGen [4], TGen [5], KMGA [6], et. al.

Pairwise testing is a significant approach for software testing as it provides efficient error detection at a very low cost. It shows a good balance by linking the magnitude and effectiveness of combinations. It requires, combination of any two parameter values that has to be covered by at least one test case [7, 8]. From the point of pairwise there are some pre-defined rules to calculate the test cases directly from the mathematical functions, which are known as algebraic strategy [9]. On the other side, computational approaches are based on

the calculation of coverage of generated pairs, followed by an iterative or random search technique to create test cases.

Random testing selects arbitrarily test data from the input domain and then these test data are applied to the program under test. The automatic production of random test data, drawn from a uniform distribution, should be the default method by which other systems should be judged [10]. Statistical testing is a test case design technique in which the tests are derived according to the expected usage distribution profile. [11], [12] and [13] suggested that the distribution of selected input data should have the same probability distribution of inputs which will occur in actual use (operational profile or distribution which occurs during the real use of the software) in order to estimate the operational reliability.

Besides GAs, other techniques have also been proposed such as Tabu Search [14], Differential Evolution Technique [15], Scatter Search approach [16], Iterative Relaxation method [17], Combinatorial Interaction testing technique [18], Constraint based technique [19], Generalized Extremal Optimization [20], etc.

Some of the researchers use the combination of two techniques to solve the problem of automatic test data generation. Liu et al. proposed a relation-based test method that combines the specification-based and the implementation-based testing approaches [21]. K. Dahal and A. Hossain focused on generating software test data using UML based software specifications and genetic algorithm [22]. X. Shen, Q. Wang, P. Wang, Bo Zhou proposed the hybrid scheme of genetic algorithm and tabu search that came to known as GATS algorithm [23]. Singla et al. presents an automatic test data generation technique that uses a new Algorithm called CGPSA (Combined Genetic-Particle Swarm Algorithm) that is based on a combination of Genetic Algorithms (GAs) and Particle Swarm Optimization (PSO) [24]. Perumal et al. introduced a novel approach to automated test data generation for software programs using a combination of heuristics involving Cuckoo and Tabu Search [25].

Evolutionary Algorithms are also used in multi-objective test data generation. In these algorithms minimizing oracle cost and maximizing the coverage are the two objectives. Lakhotia et al. reformulated the problem of ATDG into MOTDGP (Multi-Objective Test Data Generation Problem) for the first time [26] and then modified by Harman et al. in 2010 [27]. Various Multi-Objective algorithms have been proposed such as NSGA-II, MOCell, SPEA2, PEAS, etc. K. Deb et al. proposed an algorithm named NSGA-II based on GAs [28]. The MOCell (Multi-Objective Cellular Genetic Algorithm) technique was proposed by Nebro et al. It is an algorithm which is a combination of Cellular approach and Genetic Algorithms [29]. It takes the concept of neighborhood from Cellular Automata and amalgamates it with GAs. SPEA2 (Strength Pareto Evolutionary Algorithm) is an algorithm proposed by Zitler et al. [30]. Knowles and Corne proposed a metaheuristic approach PAES that found the diverse solutions in the Pareto Optimal set [31]. The algorithm does not make use of crossover operator, but instead finds the solution by just modifying the current solution.

3. Proposed Architecture

Cellular Test Path Generator (CTPG) generates test cases for a Program Under Test (PUT) by using its Control Flow Graph (CFG). The CFG is used to generate paths. From the paths generated redundant paths are removed. This minimizes the test cases and hence the cost. It may be noted that a test case is to be defined for each path. The main aim of the path generation algorithm is to maximize the path coverage.

The use of exact algorithms for generating independent path is a costly process. Therefore, a randomized technique is used for generating the paths. It may be noted that this technique does not guarantees complete path coverage. However, the plausibility of maximizing the coverage is very high. The proposed technique amalgamates Genetic Algorithms (GAs) and Cellular Automata (CA) in order to generate paths.

The proposed architecture consists of three phases.

- Control Flow Graph Generation Phase
- Rule Selection Phase
- Paths and Test Case Generation phase

The first two phases may execute in parallel. After the completion of first two phases, third phase is executed, that provides the desired results.

In the system, user provides two inputs, first one is the Program for which test data is to be generated, and the second one is the value of ‘N’, number of test cases. The value of ‘N’ is decided in such a way that, it is larger than the number of distinct paths required. This is because, the technique used to generate the test cases is random in nature. Due to randomness, paths may be repeated.

The architecture for the proposed system is shown in Fig. 1.

The *Control Flow Graph Generation Phase* takes program as input, and generates corresponding CFG as the output. *Rule Selection Phase* generates the patterns of CA, and gives an array of $22 \times 50 \times 10$ as its output. The outputs obtained from *Control Flow Graph Generation Phase* and *Rule Selection Phase*, along with the value of ‘N’, acts as the input for *Paths and Test Case Generation Phase.* This Phase produces independent paths. Test cases are crafted for each independent path which acts as the output for this Phase as well as the whole system.

3.1 Control Flow Graph Generation Phase

This phase takes program as input and generates its corresponding CFG. This phase constitute of two components:-

- CFG Generator
- CFG Minimizer

3.1.1 CFG Generator Component

CFG Generator component takes program as input and gives its corresponding CFG as its output.

3.1.2 CFG Minimizer Component

CFG Minimizer component of *Control Flow Graph Generation Phase* takes the above generated CFG as its input, and generates the corresponding minimized CFG. This component performs two functions. First one is to minimize the CFG and the second one is to rename the nodes of CFG. The minimized CFG may always have less number of nodes. This can be done by grouping the sequential statements into one node.

3.2 Rule Selection Phase

This module selects the rules of CA, which helps us to generate test paths. Following are the three components of this module:-

- Pattern Generator Component
- Pattern Minimizer Component
- Rule Selector Component

3.2.1 Pattern Generator Component

This module generates the patterns for all the rules. There are total 256 rules of CA. Corresponding to each rule, a pattern of 100×100 is generated. Each pattern is stored in a 100×100 matrix. Thus, a 3D matrix of $256 \times 100 \times 100$ is obtained.

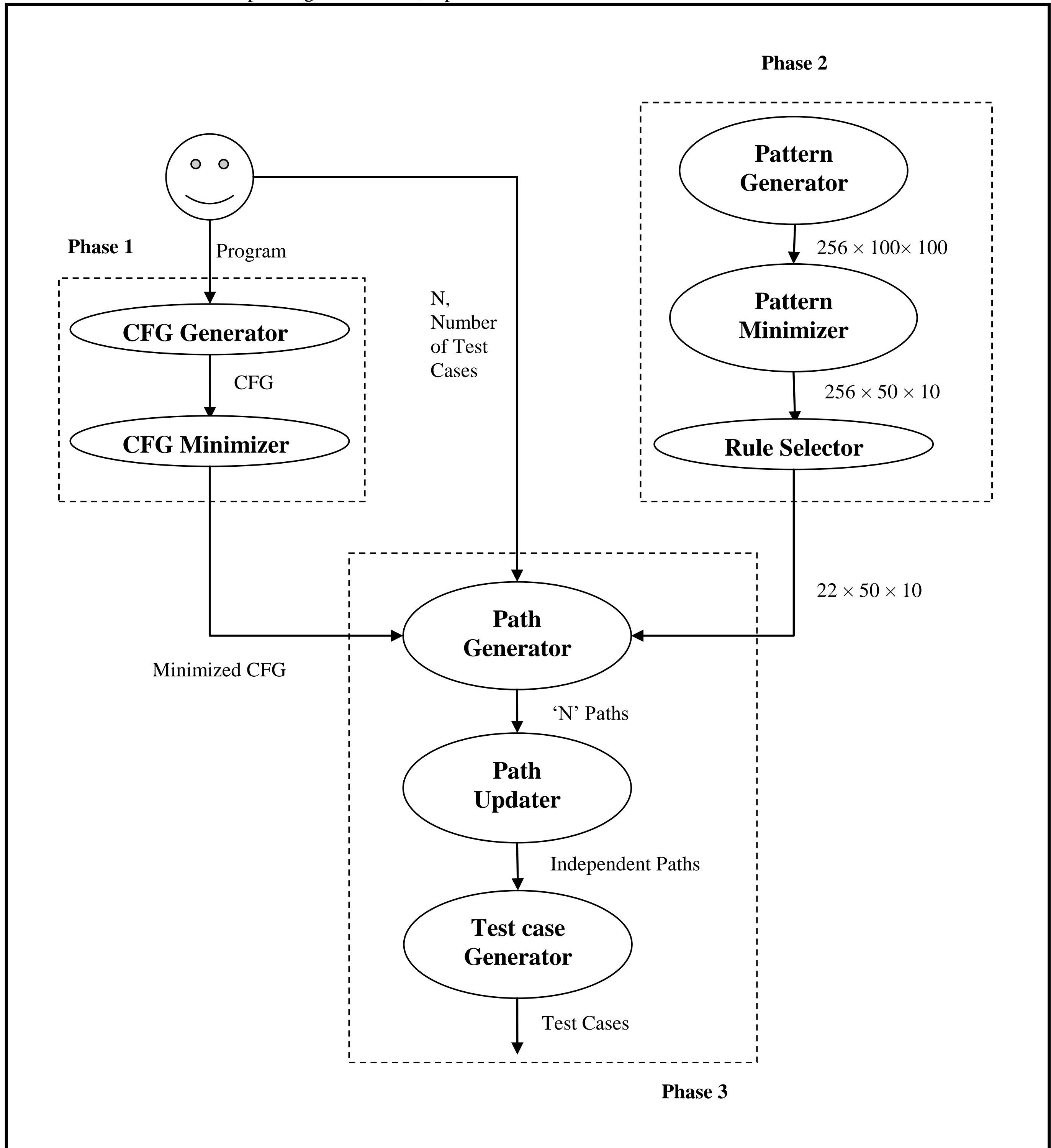

Figure 1: Architecture of Cellular Test Data Generator

3.2.2 Pattern Minimizer Component

This component takes a matrix of 100×100 as input, and gives a matrix of 50×10 as its output. This step is repeated for all the 256 rules. So, a 3D matrix of $256 \times 50 \times 10$ is obtained. The matrix is minimized in the following way.

Each row of the matrix is divided into 10 equal parts having 10 bits each. These parts are converted into a single bit depending on the majority of 0's or 1's in each part. If number of 1's is greater than or equal to number of zeroes, then, that part is replaced by 1; else it is replaced by 0. This step would

result in a matrix of order 100 × 10. This step is followed by removal of the first 50 rows from the resultant matrixes, thus resulting in the matrices of order 50 × 10.

3.2.3 Rule Selector Component

This component analyzes the 256 matrices of order 50 × 10, on the basis of randomness. To check the randomness of the pattern, coefficient of autocorrelation metric is used. The value is calculated as follows:

$$r_k = \frac{\sum_{i=1}^{N-k} (Y_i - \bar{Y})(Y_{i+k} - \bar{Y})}{\sum_{i=1}^{N} (Y_i - \bar{Y})^2}$$

Where, value of k is assumed as 1,
r = value of autocorrelation for the particular region.
N = 50, as there are 50 rows in the region which represents the state of cells at 50 discrete time steps.
Y_i = Value of ith row, which is calculated by multiplying the individual values in the respective row by 2^{-5} to 2^4.
$\bar{Y}$ = mean of all the Y_i values obtained.

$$\bar{Y} = \frac{\sum_{j=1}^{N} Y_j}{N}$$

Where, j varies from 1 to 50.

The value of autocorrelation lies between 1 and -1. Regions having value near to 0, are considered to be more randomly distributed than the other regions. After a detailed analysis, regions having coefficient of autocorrelation between -0.2 and 0.2 are considered. Rest of the regions are discarded. 22 such matrices corresponding to 22 rules are selected. The above matrices are stored in a 3D matrix [ ][ ][ ]. The first index of the 3D matrix depicts the index of the rule, and next two indices depict the rows and column of corresponding pattern.

3.3 Paths and Test Case Generation Phase

This phase takes CFG of the program under test, an array of 22 × 50 × 10, and number of paths to be generated as input, and produces independent paths, and for each independent path a test case is crafted which are the output for this phase. The architecture for this phase is shown in Fig. 2.

The *Path and Test case Generation Phase* consists of six components:-

- Branch Count Component
- Gen Pop Component
- Bit Extractor Component
- Next Node Evaluator Component
- Path Updater Component
- Test Case Generator Component

3.3.1 Branch Count Component

This component takes CFG and Current Node of the path as input and generates a number ‘m’ and ‘n’ as output. ‘m’ depicts the number of bits to be extracted. For the first time, current node is the starting node of the CFG. This component performs three functions.

First function is to calculate the number of branches of the current node using the CFG. Let ‘n’ be the number of branches of the node.

Second function is to calculate a number ‘m’ which satisfies the following condition.

$$2^m \geq n$$

Third function is to send a signal to *Gen Pop Component*, to randomly select ‘M’ chromosomes from the population of 50 chromosomes and send the chromosomes to the *Bit Extractor component.*

3.3.2 Gen Pop Component

This component generates an initial population of size 50 × 15. The population is generated only once for each path. This function also randomly selects ‘m’ chromosomes, when receive signal from *Branch Count Component* and send them to the *Bit Extractor Component.*

3.3.3 Bit Extractor Component

This This component takes ‘m’, ‘n’, ‘m’ chromosomes, and a 3D array of order 22 × 50 × 10, say pattern[ ][ ][ ], as input and gives branch number as the output.

The component makes use of 3 methods.

- ExtractRuleBits
- ExtractRowBits
- ExtractColumnBits

The three functions work as follows.

The chromosome of the GAs has 15 cells each, in which the first 5 bits depict the rule number, the next 6 bits depict the row of that particular rule and next 4 bits depict the column number.

ExtractRuleBits method extracts the first 5 bits from the chromosome. The 5 extracted bits gives a binary number which is converted into a decimal number. The modulus of that number with 22gives the rule number.

ExtractRowBits method extracts the next 6 bits from the chromosome. Decimal number obtained by these 6 bits when operated with modulus operator with 50, gives the row number of that particular rule.

ExtractColumnBits method extracts the last 4 bits from the chromosome. These 4 bits when converted into a decimal number and operated on modulus operator with 10, gives the column number.

The bit is extracted from the pattern[ ][ ][ ] array, where rule number, row number, and column number act as the indices.

The above process has been examplified in Fig. 3.

The above process is repeated ‘m’ times and ‘m’ bits are obtained. These ‘m’ bits are converted into a decimal number which is then moduled with ‘n’ to obtain the branch number.

3.3.4 Next Node Evaluator Component

This component takes branch number, current node and CFG as the input and calculates the next node of the path as the output. The process has been exemplified in Fig. 4.

The node generated act as the input for *Branch count Component*.

The above 4 components are executed again and again, until the end of CFG is reached, i.e. a complete path is generated from starting node of CFG to the end node of CFG.

The whole procedure is repeated ‘N’ times, where ‘N’ is the number of paths to be generated. At the end of whole procedure, a 2D array is generated which consist of the paths. These ‘N’ paths act as the input for *Path Updater Component*.

DOI: 10.18535/ijecs/v5i6.21

3.3.5 Path Updater Component

This Component takes the above generated paths as input, and gives the independent paths as output. There may be a case in which a single path is generated more than once, but, a path is to be considered only once. Since, there is no point of considering the path twice, as it will only lead to more number of test cases, with no increase in quality of testing. Therefore, redundant paths are removed.

3.3.6 Test Case Generator Component

This component takes the set of independent paths as input, and generates the test cases for each path which act as the output of this component as well as the output of the whole document.

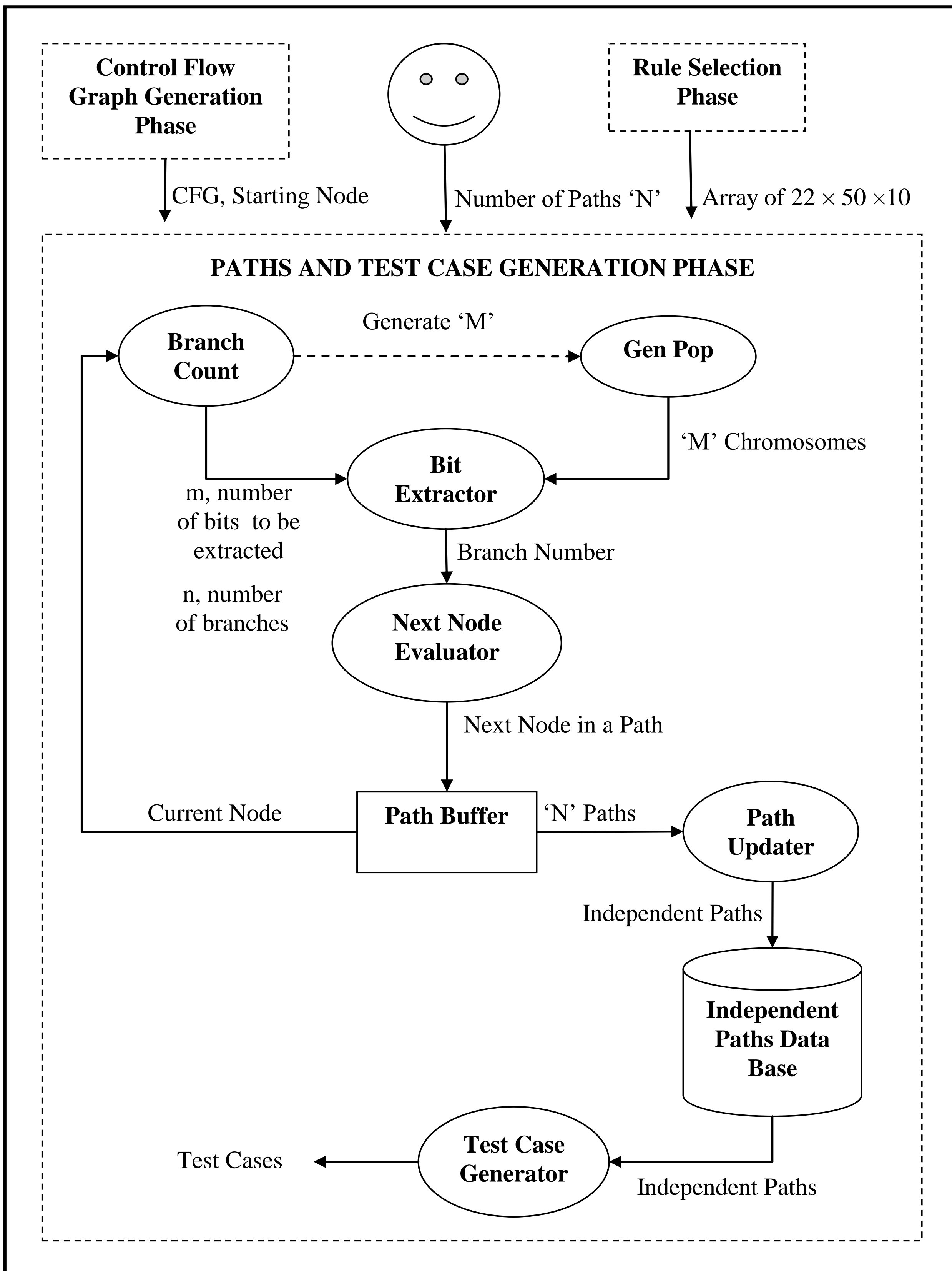

Figure 2: Architecture of Path Generation Module

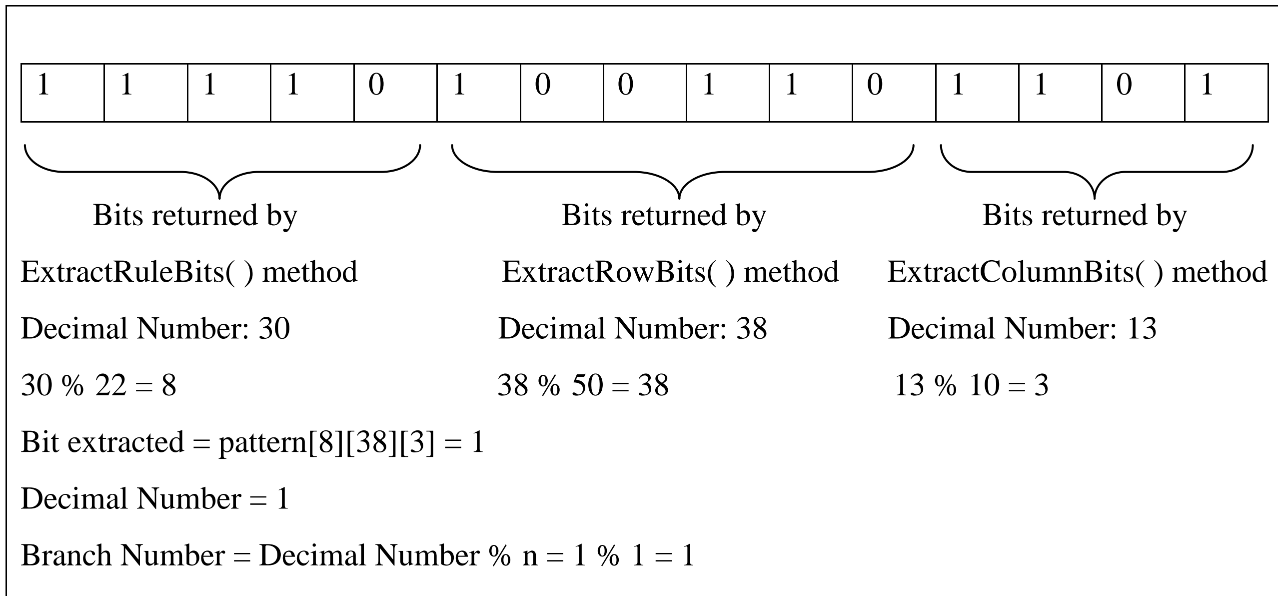

Figure 3: Examplification of Bit Extractor Component

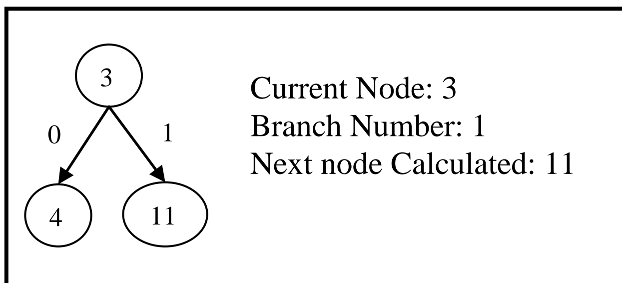

Figure 4: Examplification of Next Node Evaluator Component

4. Results

The approach mentioned has been implemented. The implemented system takes CFG of the program as the input. The CFG acts as the input for the Paths and Test Case Generation phase, and all of its components are implemented. The Rule Selection Phase has also been implemented which results in the selection of 22 rules out of the 256 rules of CA. Independent paths act as the output for the system.

To verify the technique proposed, various programs have been taken and executed. The programs were taken in accordance with their lines of code, applicability and the constructs used in them. Programs are taken such that all the constructs for which the approach is proposed are taken, and experiments are conducted on them. The details of the programs taken are given in Table 1. The results obtained are summarized in Table 2.

Table 1. Programs Description

Sr. No.	Program Name	Program Code	Constructs	Nesting Level
1	Even Number or Not	ENN	If else	1
2	Income Tax Calculator	ITC	Else-If Ladder	1
3	Quadratic Equation Solver	QES	Nested if else	2
4	Vowel or Not	VN	Switch Case	1
5	Bubble Sort	BS	Loops	2
6	Matrix Multiplication	MM	Loops	3
7	Menu Driven Program	MDP	Break, Continue, Goto	2

Table 2. Percentage of Path Coverage for the selected programs

Sr.No.	Program Code	Path Coverage
1	ENN	97.4
2	ITC	98.56
3	QES	96.32
4	VN	95
5	BS	91.06
6	MM	85.87
7	MDP	82.23

The average path coverage is calculated for our approach and is found to be 92.35%

The proposed technique is also compared with six other existing techniques also. The techniques used for comparison are as follows.

- Random Testing Approach (RT)
- Standard Genetic Algorithm (SGA)
- Generalized Extremal Optimization (GEO)
- Variation of Generalized Extremal Optimization (GEO_{var})
- Test Case Generation Using Scatter Search Approach (TCSS)
- Evolutionary Testing Approach (ET)

Other techniques taken for comparison produced their experimental results on different set of programs, whose code is not available with us. So, for the comparison purpose, average path coverage for the whole set of programs is calculated for each technique, and then compared with our approach. The average path coverage for our approach and other existing approaches has been listed in Table 3.

Table 3: Comparison with other existing techniques

CTPG	RT	SGA	GEO	GEO_{var}	TCSS	ET
92.35	85	93.80	93.41	93.64	88.30	99.8

5. Conclusion and Future Scope

The proposed work uses CA to generate test cases in a novel way, which is statistically sound and uses autocorrelation as its base, thus making sure that requisite patterns are selected from CA. Most of the procedural constructs like nested condition statements, switch cases, loops has been handled. Test case generation follows path coverage hence a time tested technique instills confidence in the proposed architecture. 7 programs were selected for testing and the results show that the test cases are as good as manual test cases.

In the work an assumption was made regarding the randomization of patterns. It was assumed that after 50 generations of a pattern, the randomization will be more than earlier generations. The fact was proved using the Pearson's Coefficient of Autocorrelation. In future, this process can be automated to find out the point from where randomization starts.

In the automation of path generation procedure, most of the constructs have already been handled. But, Function calling has not been handled yet. A modification in above technique can be made to handle functions in a software.

The process of path generation has been automated in C#. The process of test case generation can also be automated with the help of software specifications.

References

[1] Ferrer, J., Chicano, F. and Alba, E. 2012. Evolutionary algorithms for the multi-objective test data generation problem. Softw: Pract. Exper. 42, 1331–1362.

[2] McMinn, P. 2004. Search-based software test data generation: a survey. Research Articles, Software Testing, Verification & Reliability. 14, 2, 105-156.

[3] Michael, C.C., McGraw, G.E., Schatz, M.A. 2001. Generating software test data by evolution. IEEE Transactions on Software Engineering. 27, 12, 1085–1110.

[4] Miller, J., Reformat, M., Zhang, H., 2006. Automatic test data generation using genetic algorithm and program dependence graphs. Information and Software Technology 48 (7), 586–605.

[5] Pargas, R.P., Harrold, M.J., Peck, R.R., 1999. Test data generation using genetic algorithms. Journal of Software Testing, Verification and Reliability 9 (4), 263–282.

[6] Roya Alavi and Shahriar Lofti. IPCSIT vol. 14 (2011) IACSIT Press, Singapore The New Approach for Software Testing Using a Genetic Algorithm Based on Clustering Initial Test Instances 2011 International Conference on Computer and Software Modelling.

[7] Jangbok Kim, Kyunghee Choi, Daniel M. Hoffman, Gihyun Jung, "White Box Pairwise Test Case Generation", in proceedings of the IEEE Seventh International Conference on Quality Software, Oregon, USA, 2007.

[8] Zainal Hisham Che Soh, Syahrul Afzal Che Abdullah, Kamal Zuhari Zamli, "A Parallelization Strategies of Test Suites Generation for t-way Combinatorial Interaction Testing", in

proceedings of the IEEE International conference on Information Technology, International Symposium , Kuala Lumpur, Malaysia, 2008.

[9] M. I. Younis, K. Z. Zamli, N. A. Mat Isa, “Algebraic Strategy to Generate Pairwise Test Set for Prime Number Parameters and Variables”, in proceedings of the IEEE international conference on computer and information technology, Kuala Lumpur, Malaysia, 2008.

[10] Ince, D. C.: 'The automatic generation of test data', The Computer Journal, Vol. 30, No. 1, pp. 63-69, 1987

[11] Taylor R.: 'An example of large scale random testing', Proc. 7th annual Pacific North West Software Quality Conference, Portland, OR, pp. 339-48, 1989

[12] Ould, M. A.: 'Testing - a challenge to method and tool developers', Software Engineering Journal, pp. 59-64, March 1991

[13] Duran, J. W. and Ntafos S., 'A report on random testing', Proceedings 5th Int. Conf. on Software Engineering held in San Diego C.A., pp. 179-83, March 1981

[14] Eugenia Díaz , Javier Tuya , Raquel Blanco , José Javier Dolado, A tabu search algorithm for structural software testing, Computers and Operations Research, v.35 n.10, p.3052-3072, October, 2008 [doi>10.1016/j.cor.2007.01.009]

[15] Ricardo Landa Becerra, Ramón Sagarna, Xin Yao: An evaluation of Differential Evolution in software test data generation. IEEE Congress on Evolutionary Computation 2009: 2850-2857.

[16] Raquel Blanco , Javier Tuya , Belarmino Adenso-Díaz, Automated test data generation using a scatter search approach, Information and Software Technology, v.51 n.4, p.708-720, April, 2009

[17] Neelam Gupta , Aditya P. Mathur , Mary Lou Soffa, Automated test data generation using an iterative relaxation method, Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering, p.231-244, November 01-05, 1998, Lake Buena Vista, Florida, United States [doi>10.1145/288195.288321]

[18] Ferrer, J., Kruse, P., Chicano, F., Alba, E. 2012. Evolutionary algorithm for prioritized pairwise test data generation. GECCO 2012. 1213-1220.

[19] Jifeng Chen, Li Zhu, Junyi Shen, Zhihai Wang, Xinjun Wang, An Approach on Automatic Test Data Generation with Predicate Constraint Solving Technique, International Journal of Information Technology, Vol.12, No.3, 2006, pp. 132-141.

[20] Bruno Teixeira de Abreu, Eliane Martins, Fabiano Luis de Sousa: Generalized extremal optimization: an attractive alternative for test data generation. GECCO 2007: 1138

[21] Shaoying Liu , Yuting Chen, A relation-based method combining functional and structural testing for test case generation, Journal of Systems and Software, v.81 n.2, p.234-248, February, 2008 [doi>10.1016/j.jss.2007.05.036]

[22] K. Dahal, A. Hossain, “Test Data Generation from UML State Machine Diagrams using GAs”, International Conference on Software Engineering Advances, 0-7695-2937-2/07 © 2007 IEEE. pp:834-840.

[23] X. Shen, Q. Wang, P. Wang, Bo Zhou, “Automatic Generation of Test Case based on GATS Algorithm”, 2007AA04Z148, supported by Nation 863 Project.

[24] Sanjay Singla, H. M. Rai, Priti Singla, Automatic Test data Generation Approach using Combination of GA and PSO with Dominance Concepts, International Journal of Electronics Engineering, Vol. 3, No. 1, 2011, pp. 95-98.

[25] Krish Perumal, Jagan Mohan Ungati, Gaurav Kumar, Nitish Jain, Raj Gaurav, Praveen Ranjan Srivastava, Test data generation: a hybrid approach using cuckoo and tabu search, Proceedings of the Second international conference on Swarm, Evolutionary, and Memetic Computing (SEMCCO’11), PP. 46-54.

[26] Harman, M., Lakhotia, K., McMinn, P. 2007. A multi-objective approach to search-based test data generation. GECCO ’07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, ACM: New York, NY, USA. 1098–1105.

[27] Harman, M., Kim, S. G., Lakhotia, K., McMinn, P., Yoo, S. 2010. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. Proceedings of the 3rd International Workshop on Search-Based Software Testing (SBST) in conjunction with ICST 2010, IEEE: Paris, France. 182–191.

[28] Deb, K., Pratap, A., Agarwal, S., Meyarivan, T. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation. 6, 2, 182–191

[29] Nebro, A. J., Durillo, J. J., Luna, F., Dorronsoro, B., Alba, E. 2009. MOCell: A cellular genetic algorithm for multiobjective optimization. Int. J. Intell. Syst. 24, 7, 726–746.

[30] Zitzler, E., Laumanns, M., Thiele, L., 2001. SPEA2: Improving the strength pareto evolutionary algorithm. Technical Report 103, Gloriastrasse 35, CH-8092 Zurich, Switzerland.

[31] Angeline, P. J., Michalewicz, Z., Schoenauer, M., Yao, X., Zalzala A. 1999. The Pareto Archived Evolution Strategy: A New Baseline Algorithm for Pareto Multiobjective Optimisation, IEEE Press: Mayflower Hotel, Washington D.C., USA. Vol.1.