Test Data Generation using Artificial Life

Harsh Bhasin
Delhi technological University

Shewani
M.Tech Scholar
A.I.T.M, Palwal

Deepika Goyal
Assistant Professor
A.I.T.M, Palwal

ABSTRACT
Test Data Generation is an intricate process which requires intensive manual labor and thus a lot of project time. There is an immediate need of finding out an effective technique for automating the process as manual Test Data Generation escalates the project cost. The paper proposes the use of Artificial Life in generating and minimizing the Test Cases. The work has been applied on some programs and the initial results are encouraging. The technique makes sure that all the modules are tested in accordance with their functional specifications by the Artificial Life Test Suite Generator (ALTSG). The initial results even points to an indication of the technique being better than its counterparts.

Keywords
Artificial Life, Test Data Generator, Program Analyzer, Path Selector.

1. INTRODUCTION
Software testing is a complex intricate process which not only requires extensive resources but also intelligence so as to comprehensively cover all the plausible cases. Testing is done via test cases which can be generated manually or via an automated system. Manual test data generation results in robust test case suit because of the ability of human mind to recognize the patterns which may lead to erroneous output. The human mind's ability arises out of the neurological decision making capability. Human body has millions of neurons and trillions of synapses. These synapses may each store a pattern and the interconnection between them leads to coherence between the patterns to generate outputs that are hard to obtain via an automated system. This ability of human beings is hard to be replicated by a program. The automated test case generator on the other hand, keeps these complexities aside; and dwell on the analyzer to cover all the possible paths in order to generate the data. The advent of the discipline saw a program being given to three different subsystems in order to generate the data. The various constituents of the system are analyzer, path selector and data generator. The analyzer comprehensively analyses the program by following one of the approaches discussed in the second section of the paper. The second component is the path selector. After analyzing the program the path which covers the program are to be identified. This is done for test case generation. The third part of the system consists of data generator which generates the test cases by putting constraints on the paths generated by the second component. In such systems the selection of the path become crucial and must be handled with both sound criteria and intelligence. The work tries to harness the virtues of artificial life system in order to generate test data. It is an attempt to imitate natural evolution and explore its application in testing.

The paper is segregated as follows. Section 2 of the paper contains the review of test data generation. The third section introduces the concept of artificial life; fourth section presents the proposed technique. Fifth section discusses the results and conclusions.

The review is carried out strictly in accordance with the guidelines proposed by kitchenham[9].

2. LITERATURE SURVEY
Testing plays a crucial role in the software development process. The quality and the reliability of a program depend on testing. Testing is not only for detecting bugs but it is also for checking the conformance to the functional and nonfunctional requirements of a program. Testing consumes the major part of the development of the software. Antonia [1] presented a complete overview of testing. Testing has to start at the requirement specification stage. It is a challenging activity that involves very high risks.

In order to reduce the cost of manual software testing and for increasing the reliability of testing, researchers have tried to automate the testing process [6]. For this, an automatic test data generator has been evolved which is a kind of a system that automatically generates the test data for a given program. Many kinds of automatic test data generator have been crafted so far. In the work, a program based automatic generators has been considered. The paper describes the basic concepts and working of the generator. For generating test data there are several methods: random, path-oriented, and goal oriented [2]. A test data generator consists of three parts: program analyzer, path selector, and Test Data Generator [2]. Several efforts have been made for automatic test data generation but none of them uses Artificial Life as its base. To solve the constraint satisfaction method rule based test data generation methods have also been proposed [2].

One of the works depicts the features of test data generators. The handling of Booleans, integers, reals, and arrays has been discussed in the work [2]. Some attempts for using pointers have been also made. The techniques like: constraint satisfaction, assertions, modules, path selection, pointers etc have been considered.

In another work by Neelam Gupta, a new program execution based approach, which uses an iterative relaxation technique, has been used to solve the problem that occurs during path oriented testing. The system is required to generate test data using a particular path [3]. In the work an input is chosen to generate the test data from the domain. The inputs are

iteratively selected for achieving the required output. In each iteration, the program statements which are appropriate are executed on each branch predicate thus identifying constraints. The constraints are solved to get the next input. This process is repeated for next input. The relaxation method used in obtaining the constraints provides feedback on the input to achieve the desired output. The method either finds a solution in one iteration or the path is declared infeasible. Here the relaxation technique is used in numerical analysis to improve the approximate solution of an equation.

The technique used in the above paper is an advanced application of the traditional relaxation method used in numerical analysis to obtain an accurate result of an equation by an iterative method. The results obtained from this method provide practical solution to generate test data automatically and it is more efficient. The work can also detect infeasible paths easily.

In the work by Ali [5] assertion based test data generation has been used for removing the faults [5]. This method uses assertions to increase the confidence in the software. The main idea of the above paper is to decompose each assertion into simple elements. The process of finding test data to execute each node is performed in parallel. Assertions are recognized as a strong tool for automatic run-time detection of software errors during testing and debugging. Assertion specifies a constraint that is applied to some computations. When the assertion estimates to be false during program execution, there exists a wrong state in the program. This paper proposes a concurrent assertion-based automated test data generation framework. The framework has been implemented on PC platform for Java programs. The purpose of this framework is to implement it on high performance parallel architecture with thousands of processors.

In the by Roger [4] the chaining approach for automated software test data generation has been used. In this test data are obtained based on the actual execution of the program under test. This approach used data dependency analysis to automatically identify statements that affect the execution of the selected statement. They form a sequence of statements that is to be executed prior to the execution of selected statements.

This method is an extension of the present execution oriented methods of test data generation which uses only the control flow graph in the search process. But chaining approach uses the data dependency technique which may enhance the test data generation. Chaining approach is used for node problem but can be used for branch testing and for data flow testing. For them different initial event sequences is generated.

Artificial life is an interdisciplinary study of life and life-like processes that uses a synthetic methodology [6]. Three wide branches of artificial life correspond to three different synthetic methods. 'Soft' Artificial life creates simulations or other purely digital constructions that exhibit life-like behavior, 'hard' artificial life produces hardware implementations of life-like systems, and 'wet' artificial life synthesizes living systems out of biochemical substances. The concept has evolved by Von Neumann [7] and others [6, 8].

As explained above test data generation is an important task. Many methods have been proposed for test data generation, but none of them based on the premise of artificial life. This approach is motivated by the observation that in many applications a significant number of faults are caused by interactions of a smaller number of parameters. The work proposes a new test generation algorithms based on artificial life techniques.

3. PROPOSED TECHNIQUE

In the technique henceforth called Test Data Generation using Artificial Life (TDGUAL). The Artificial Life methodology will be used to reduce the number of test cases. This technique uses modules as the basic unit. The technique is novel as it serves two purposes i.e. generation and minimization of the test cases.

In the literature review it was found that, some of the works depict time as a major factor compared to the coverage of test cases but in the proposed work coverage is given more importance. In the technique the stress is on the function coverage as testing is done to instill confidence in the software, therefore maximization of function coverage factor will lead to lesser time in the maintenance phase.

The technique intends to generate the test cases for all the modules of the program. The technique henceforth presents a new state machine called the Function Flow Graph (FFG). The FFG will cover all the modules and help in generating the test suit.

The automated test case generation is generally required for huge programs. Therefore, there will be many test suit cases hence requiring minimization. At this point Artificial Life (AL) technique comes to our rescue. The AL part will select the appropriate value of the variables in question.

The first phase of the proposed technique has been summarized as follows:

1) Set test case suite to a null set φ.
2) Start from the main module.
3) $\forall\ module\ m_i$ encountered perform the following tasks and put it onto the stack
    i) Identify $inputs\ a_i$, store the type and constraint of the input from the design view.
    ii) With the help of the above step design a test case and add it to the test case suite.
4) Once all the modules have been processed repeatedly pop modules from the stack and perform step a) and b).
5) $\forall\ Test\ cases\ T_i$, check the feasibility of the test case as per the design phase. If accepted put it in the final suite.

Figure 1 depicts the above algorithm**.**

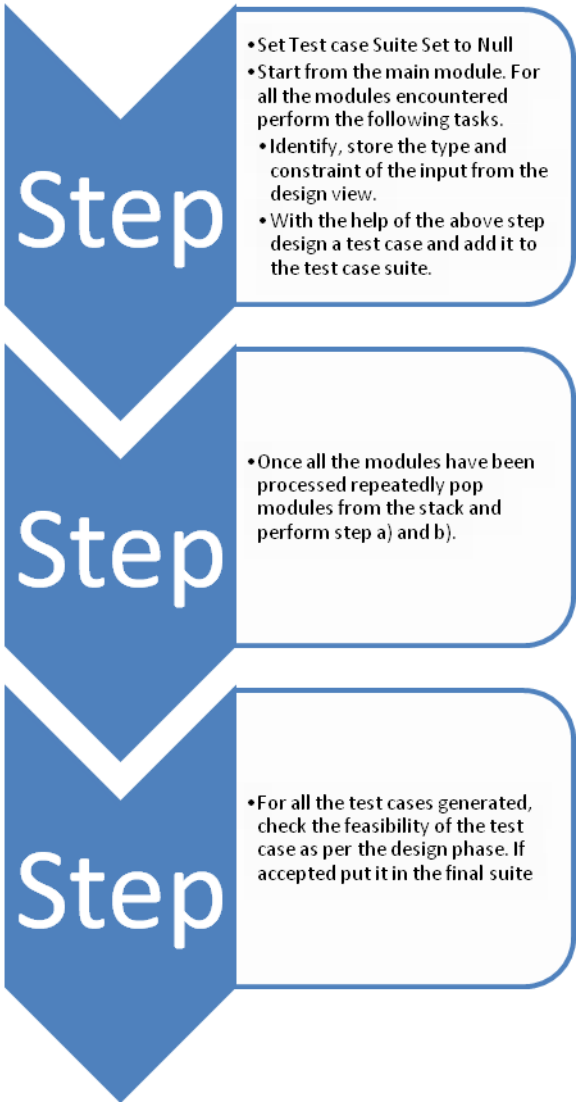

Fig 1: Phase 1: Test Data Generation using Artificial Life

A lexical analyzer program having 445 lines of code with 11 modules is taken as an example. From this Table 1 has been generated. Table 1 shows the modules and its attribute.

Table 1: Modules and its attribute

S.NO	MODULE NAME	PURPOSE	INPUT	RETURN TYPE	INPUT RANGE
1.	Lex_Ana(char *str)	Defines keywords and operators	Character	character	–128 to 127
2.	int IsMxIden(char ch)	For max identifier	Character	INTEGER	–128 to 127
3.	int IsIden(char ch)	For identifier	Character	INTEGER	–128 to 127
4.	int IsFloat(char ch)	For float	Decimal value	INTEGER	–128 to 127
5.	int IsOpr(char ch)	Defines operator	Character	INTEGER	–128 to 127
6.	int IsDel(char ch)	Defines delimiter	Character	INTEGER	–128 to 127
7.	int IsKey(char *str)	Defines keyword	Character	INTEGER	–128 to 127
8.	void Find_Lex()	Function for finding lex function	Character	VOID	–128 to 127
9.	void Rem_Dup( )	Function for duplicate keyword	Character	VOID	–128 to 127
10.	void Sep_Val()	Separate value	Character	VOID	–128 to 127
11.	void Display()	Function for display	Character	VOID	–128 to 127

Table 2. Design specifications of the modules

MODULE NO.	INPUT	OUTPUT
1.	Keywords are reserved and operators like +, =, <,>, &, *, % etc.	Character
2.	0 to 9 and a to z and A to Z	INTEGER
3.	0 to 9 and a to z and A to Z	INTEGER
4.	Decimal value	INTEGER

5.	+, =, <,>, &, *, % etc.	INTEGER
6.	Character	INTEGER
7.	Reserved	INTEGER
8.		VOID
9.		VOID
10.		VOID
11.		VOID

Algorithm for proposed technique is as follows:

1.$\forall\ module\ m_i$, where m_i being a module. Identify the calling sequence and start from the entry point for example the program taken for verification the FFG has main () as the entry point. Let Graph be G henceforth.

2. G is the graph formed in the above step.

For all e_i belongs to G such that e=(x, y) where x and y are the nodes in the graph. Identify the input values and output values of mi from table 2 which depicts the design specifications of the modules.

3. The input values and the legal values of output are known by the virtue of Table 2. We have to select from amongst these legal values to form a test case.

4.$\forall\ module\ m_i$, the test cases generated above are mapped to Langton's loop.

5. At each intersection with the centre of the loop while the process of creation of the new one, put the intersection point value in the Test Case.

6. In the same way repeat the process for all variables.

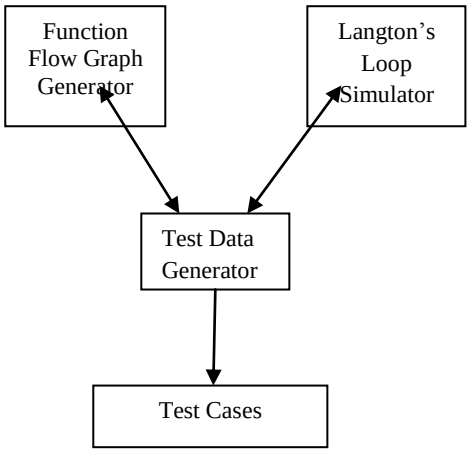

Fig 2 – Components of Test Data Generation Using Artificial Life

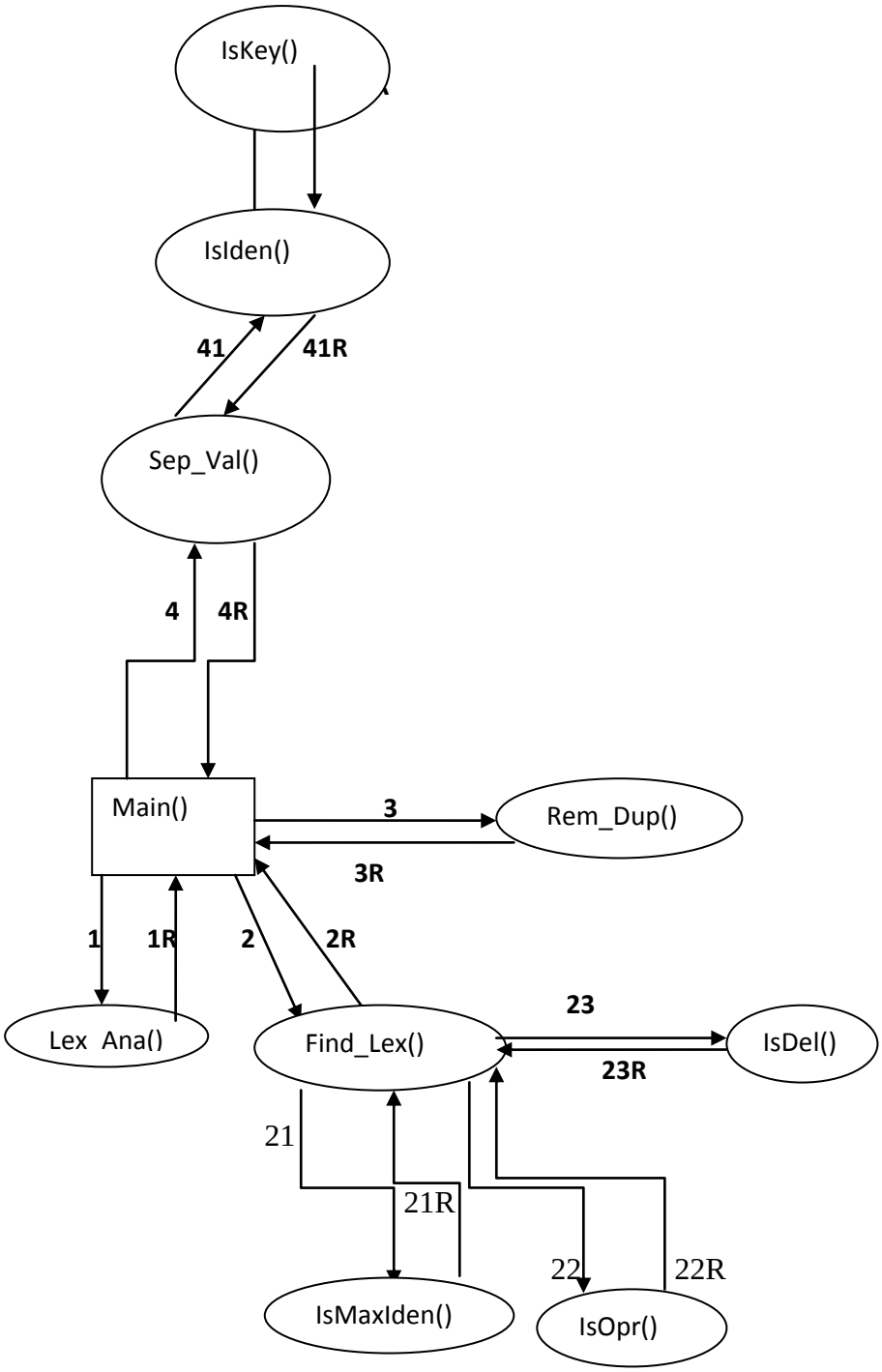

Fig 3 – Module State Diagram of the program taken for verification

Figure 2 shows the components of Test Data Generation using Artificial Life. In figure 3 the main ( ) function is calling Lex_Ana() and 1R shows that Lex_Ana() is returning some value. 2 shows that Main() is calling Find_Lex() and 2R means that Find_Lex() is returning some value. 3 shows that Main() is calling Rem_Dup() and 3R means that Rem_Dup() is returning some value. 4 shows that Main() is calling Sep_Val() and 4R means that Sep_Val() is returning some value. 21 means that Find_Lex() is calling IsmaxIden() 21R means that IsmaxIden() is returning some value. 22 means that Find_Lex() is calling IsOpr() 22R means that IsOpr() is returning some value. 23 means that Find_Lex() is calling IsDel() 23R means that IsDel() is returning some value. 41 shows that Sep_Val() is calling IsIden() and 41R means that isIden() is returning some value and 411 shows that IsIden() is calling IsKey() and 411R shows that IsKey() is returning some value.

Table 1 and Table 2 shows the design specifications of the above modules and Test Data which are generated are fed to the above Module State Machine.

4. CONCLUSIONS

Test Data Generation is immensely important and is an intricate process. The realization of its importance has helped generating a renewed interest in the field. The above work is novel in the sense that Artificial Life has never been used before to accomplish such task. It may also be stated that Test Data Generation has seldom been seen with the perspective of function calling. Moreover the technique works well for the program taken and is being applied on a large set of programs with varying lines of code. The initial results are encouraging. And it is sure to change the way we look at Artificial Life.

5. REFERENCES

[1] Bertolino, A., Marchetti, E.: A brief essay on software testing. In: Thayer, R.H., Christensen, M.J. (eds.) Software Engineering, 3rd edn. Development process, vol. 1, pp. 393–411. Wiley-IEEE Computer Society Press (2005)

[2] J. Edvardsson. A survey on automatic test data generation. In Proceedings of the Second Conference on Computer Science and Engineering in Linkoping, pages 21-28. ECSEL, October 1999.

[3] 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

[4] Roger Ferguston, B. Korel, The chaining approach for software test data generation, ACM Transactions on Software Engineering and Methodology (TOSEM) Volume 5.

[5] Ali M. Alakeel. 2010. A Framework for Concurrent Assertion – Based Automated Test Data Generation. Universityof Tabuk. Saudi Arabia.

[6] C. G. Langton (1984). "Self-reproduction in cellular automata". Physica D 10: 135–144.

[7] Von Neumann, John; Burks, Arthur W. (1966). "Theory of Self-Reproducing Automata." (Scanned book online). www.walenz.org. Archived from the original on 2008-01-05. Retrieved 2008-02-29.

[8] Wiener, N. 1948. Cybernetics, or Control and Communication in the Animal and the Machine. Wiley.

[9] B. A. Kitchenham et. Al. Systematic literature reviews in software engineering - A tertiary study, Information & Software Technology - INFSOF , vol. 52, no. 8, pp. 792-805, 2010.