Paul POCATILU, PhD
Email: ppaul@ase.ro
Economics Informatics and Cybernetics Department
The Bucharest Academy of Economic Studies

A FRAMEWORK FOR TEST DATA GENERATORS ANALYSIS

Abstract. *Test data generation is an important process for software testing and code analysis. The efficiency of a test data generator is crucial for very complex projects in terms of speed, effort and duration. There are different types of input data that a generator could provide and different ways to implement these automated generators for test data. In this paper is proposed a system for evaluation of test data generators. The proposed system is used to compare different implementations of test data generators like random or based on genetic algorithms.*

Key words: *software testing, test data generator, path coverage, genetic algorithm, control flow, indicator.*

JEL Classification: C49, C61, L86

1. INTRODUCTION

Every software system requires a rigorous testing process in order to achieve o high level of confidence. Software testing is a process that aims to find errors in software. This process involves several methods, techniques and strategies. The software that is tested could be approached as a black box or a white box. Software testing process is described from theoretical and practical point of view in numerous papers like Meyers (2004), Pressman (2009), Sommerville (2011) and Ivan and Pocatilu (1999).

The software testing process can be view using a systemic approach as depicted in Figure 1.

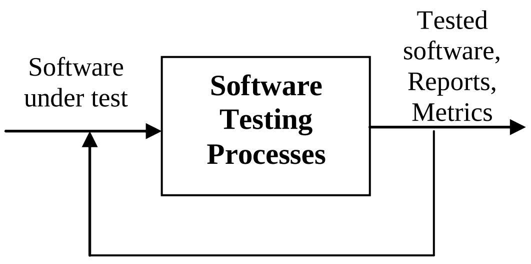

Figure 1. Software testing process as a system

The main input for the software testing process is the software under test (SUT) that is analyzed at a source code level (white box) or at a functional level (black box). One approach in finding errors is to generate data sets that will be used to compare the actual results with expected results. Data set generation processes are time consuming, even these processes are automated.

The main outputs of the software testing process are the reports and the metrics collected. The reports include the test results. The metrics are very important for the continuous improvement of quality (Gao et al., 2011).

The use of genetic algorithms for test data generation is an actual topic, examples of test data generators (TDG) based on genetic algorithms are the researches of Malhotra and Garg, M. (2011), Shimin and Zhangang (2011) Alshraideh et al. (2013) and others.

The paper is structured as follows.

Section 2 presents test data generation process. The concepts of path, Control Flow Graph (CFG), Control Flow Diagram (CFD) and code coverage are presented.

Section 3 analyzes random test generators implementation. Random test data generators are exemplified.

Section 4 deals with test data generators based on genetic algorithms. It also presents an analysis of fitness function efficiency.

Section 5 presents the proposed indicators used in test data generators analysis. It also presents an aggregated indicator that represent a unified way for comparison of test data generators.

Section 6 presents the results of applying the proposed aggregated indicator for two test data generators used for several SUTs. The results are analyzed.

The paper ends with conclusions and proposed work.

2. TEST DATA GENERATION

Test data generators (TDG) implemented to work based on program structure use the associated representation to generate test data in order to achieve the desired degree of coverage. The coverage can be seen at a statement, branch, path, module (function) or class level.

Each program can be represented as a Control Flow Graph (CFG). This consists of nodes associated to instruction, sentences, functions or blocks, depending on the required level of detail, and the links between nodes. CFG is represented as a graph or as a tree. Another program representation is by using Control Flow Diagrams (CFD).

For example, let’s consider a C# function that sorts three numbers in ascending order:

```
void sort3(ref int a, ref int b, ref int c)
A    {
       int t;
B      if (a > b)
       {
         t = a;
C        a = b;
         b = t;
        }
D       if (a > c)
        {
          t = a;
E         a = c;
          c = t;
         }
F       if (b > c)
        {
          t = b;
G         b = c;
          c = t;
         }
H    }
```

A simplified CFG associated to *sort3()* function is given in Figure 2. Within this graph examples of paths are ABDFH, ABCDFGH or ABDFGH.

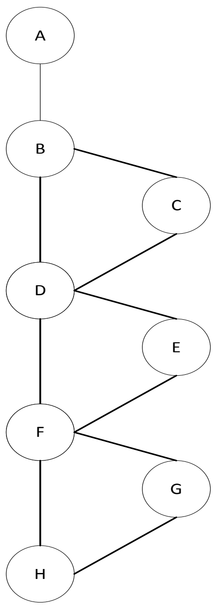

Figure 2. A simplified CFG of *sort3()* function

The associated tree structure of the *sort3()* function, based on the CFG, is given in Figure 3. As it can be seen, the program tree is a transformation of CFG by multiplication of nodes. In this representation the leaves nodes represents the exit points of the function (Pocatilu et al, 2001).

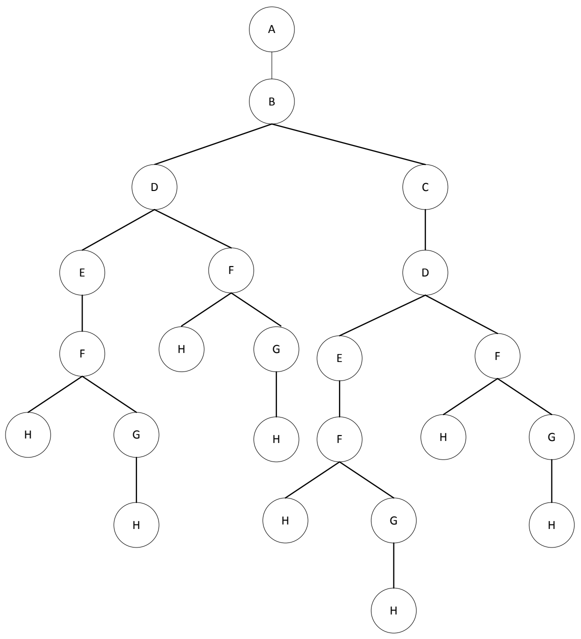

Figure 3. Associated tree of *sort3()* function

In order to test the *sort3()* function and to cover all feasible paths, test data will be generated. It is not necessary to test all paths within the program. A simple measure is the minimum independent path given by the cyclomatic number (McCabe, 1976). For the *sort3()* function the cyclomatic number (V) is 4, resulting from 10 edges (*E*) and 8 nodes (*N*) and one connected component (*P*):

$$V(G) = E - N + 2P \qquad (1)$$

Table 1 presents data sets that covers all feasible paths of the *sort3()* function. For example, the path ABCDEFH is not feasible, and cannot be covered by any set of data.

Table 1. Generated test data for *sort3()* function

Set	a	b	c	Path
1	44	44	-70	ABDEFH
2	-48	-6	-101	ABDEFGH
3	-99	-2	105	ABDFH
4	-118	38	-74	ABDFGH
	-	-	-	ABCDEFH

5	88	-34	-92	ABCDEFGH
6	-124	-125	-5	ABCDFH
7	89	-38	65	ABCDFGH

For this example, the four independent paths that will cover all statements and branches within the program are 1, 2, 3 and 6.

An automated test data generator (TDG) includes several modules, depending on the implementation. Even the implementation is different, there are several common modules like the SUT analyzer. This module:

- analyzes the source code;
- generates the CFG;
- calculates required source code metrics (lines of code, cyclomatic complexity etc.).
- instruments the source file by inserting calls to trace function in order to identify the path covered by running the SUT.

This module helps in code instrumentation and provide inputs for specific test data generators.

3. RANDOM TEST DATA GENERATION

The common way used to generate test data is by providing random values for parameters.

A TDG based on random data generation includes the specific module that implements the following actions:

- generates input parameters randomly;
- calls the injected SUT with generated parameters;
- records path coverage for the SUT;
- check if the covered path is new to be added to the list;
- a new set of input parameters is generated.

The process ends when the required level of coverage is achieved or the number of runs reaches a given level.

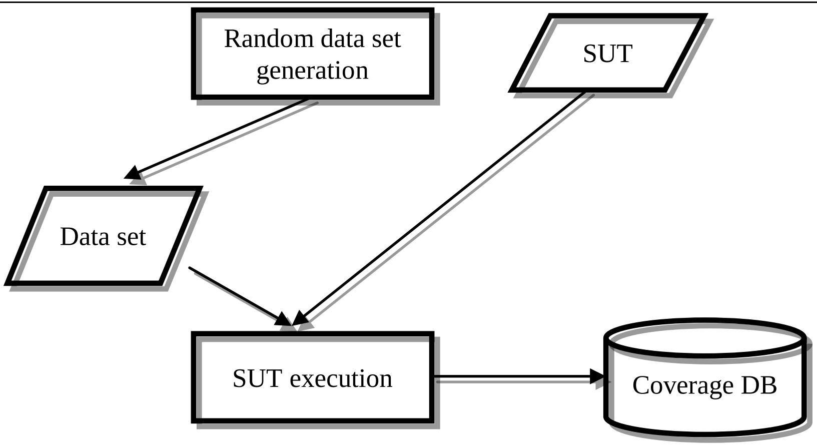

Figure 4. Random test data generator

Figure 4 depicts the components of a random test data generator. The SUT is executed using randomly generated test data.

A random test data generator can provide:

- simple data types numbers;
- homogeneous data types as array and matrices;
- complex structures as records and classes;

The generated data sets can be in memory or stored in files, depending on the requirements.

Usually, the input parameters are generated randomly, without any analysis of the previously generated data sets.

In order to improve test data generators based on random values, the input parameters could be included in classes and the previously generated values to be analyzed in order reduce the number of execution of injected SUTs and to increase the performances of the test data generator.

4. TEST DATA GENERATORS BASED ON GENETIC ALGORITHMS

Genetic algorithms represents optimization techniques used to determine an optimal solution by searching in a large solution space. Solution are chosen based on natural selection as detailed in Haupt and Haupt (2004), Maries and Dezsi (2011) and Visoiu (2011).

Initially, candidate solutions are generated and grouped within a population. Solutions or individuals are encoded as chromosomes using a finite alphabet (0 and 1, discrete or continuous values or strings). For example, a chromosome could represent four 8-bits values encoded into a 32-bit allele or could be encoded in an array with four elements.

Each step of the genetic algorithm (generation) produces a new population of possible solutions. The new population is based on the individuals from previous population.

A specific function (fitness) is used to evaluate each individual. Natural selection is used to select chromosomes with a higher fitness value. The chromosomes with good fitness have best chances to be kept from one generation to the next one.

First, chromosomes with higher fitness value are selected to produce new chromosomes. This process is called selection. Next, parts of solutions are used to generate new solutions by using genetic operators:

- Crossover – alleles from two chromosomes (parents) are swapped in order to produce new chromosomes. The result will be chromosomes that have parts from both parents (offspring).
- Mutation – alleles are changed in order to produce new chromosomes (for example, a bit is toggled).

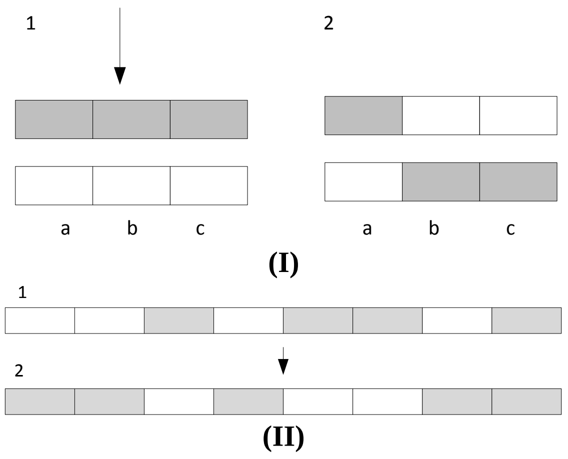

Figure 5. Using crossover (I) and mutation (II)

Crossover is done selecting two parents and choosing a random number from 1 to last encoded parameter-1. After that, offspring are created by exchanging the parts starting with selected point, as depicted in Figure 5 (I).

For all encoded parameters a random number is generated. If this is above the selected mutation probability, then it will take place. At a bit level, for each bit of the selected encoded parameter, the values are toggled, from 0 to 1 and from 1 to 0, as depicted in Figure 5 (II).

Generally, a genetic algorithm consists of the following steps:

1. Initialize the population
2. Evaluate the fitness for each chromosome
3. While (stop condition is not true)

a. Select chromosomes for the new generation based on their fitness;
b. Transform the chromosomes in the population by using crossover and mutation;
c. Evaluate the fitness for each chromosome.

The genetic algorithm stopping condition depends on the problem, and it could include criteria like:

- when a given number of generations is reached;
- when an certain solution achieves a specified level of fitness;
- when there are no variation of individuals from one generation to the next one.

After the candidates for the new population are selected, new solutions are generated using genetic operators until the stopping condition is not met.

Due to their characteristics, genetic algorithms can be successfully used in test data generators. A specific termination condition is to achieve the required degree of coverage (statements, branches, paths, modules or blocks). The percent of coverage should be chosen wisely and the way is chosen depends on the SUT.

The TDG system based on genetic algorithm includes the specific module that implements the following actions:

- Initially it generates input parameters randomly;
- Calls the injected SUT with generated parameters;
- Records path coverage for the SUT;
- GA is applied based on the results and it generates a new set of input parameters
- The process ends when the required level of coverage is achieved or the number of runs reaches a given level.

Every defined fitness function for a genetic algorithm used for test data generators has to be analyzed using several criteria in order to prove its efficiency. As example of analysis, Pocatilu and Mihai (2001) presented a fitness function named Inverse Similarity of Coverage that is based on path length (*len*) and the degree of similarity (*sym*) of the paths covered using generated test data. Let's consider that *p* is the associated probability of a node from CFG to give an error. Hence, the probability that an individual will detect an error increases with its length. For a chromosome *a*, the error detection probability, p_{err}, is given by:

$$p_{err} = 1 - k \times p^{len(a)} \qquad (2)$$

where *len(a)* is the length of the path covered by chromosome *a* and *k* is a correction factor that depends on the complexity of the SUT, with $k \in [0, 1]$.

If we consider three chromosomes *a*, *b* and *c*, with $len(a) \leq len(b) \leq len(c)$, and the reference chromosome is *b*, then

$$p_{err}^{c} = 1 - (1-p)^{len(c)\ - len(shared_c)} \qquad (3)$$

where *len(c)* is the length of path given by chromosome *c* and *len(shared$_c$)* is the length of the segment shared by chromosome *c* and the reference chromosome *b*, and

$$p_{err}^{a} = 1 - (1-p)^{len(a) - len(shared_a)} \qquad (4)$$

where *len(shared$_a$)* is the length of the path shared by *b* with *a*. It is known that *len(shared)$_c$>len(shared$_a$)* and since on average the length of both paths is identical $\overline{len}(c) = \overline{len(a)} = \overline{len}$, where $\overline{len}$ is the average tree traversal path length, results that on average $\overline{p_{err}^{a}} > \overline{p_{err}^{c}}$.

5. TDG PERFORMANCE INDICATORS

In order to measure the effort required to generate test data several indicators are defined.

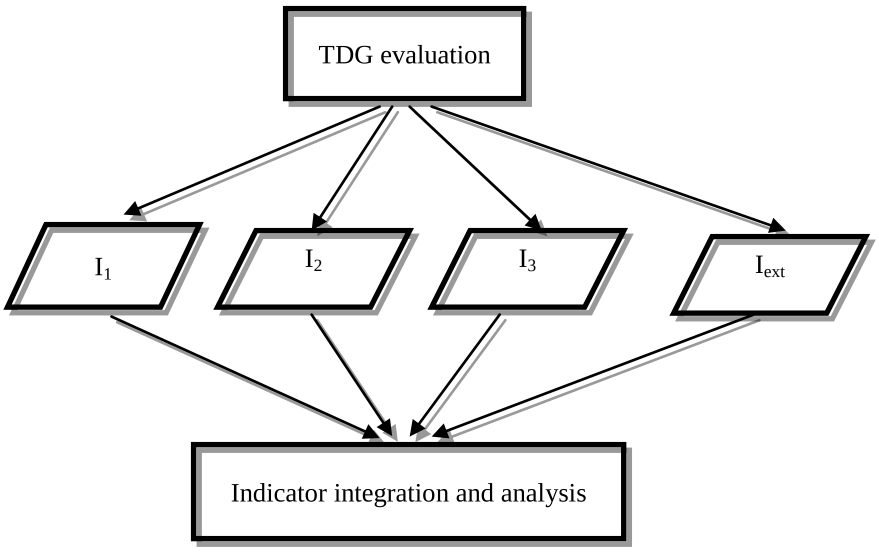

Figure 6. Proposed framework for TDG analysis

Figure 6 depicts the general framework used to analyze test data generator. I_1, I_2, …, I_{ext} are indicators used to define common and specific performance metrics for test data generators. The indicators are normalized and aggregated so that:

, with (5)

where w_i are weights defined by the user depending on the requirements.

All indicators need to be normalized in order to obtain values in the same domain. The normalized indicators ( ) could be determined by calculating the minimum and maximum recorded values, and:

- the minimum value will be 0 or 1;
- the maximum value will be 1 or 0;
- values between minimum and maximum will have proportional values between 0 and 1.

Another normalization technique involves the calculation of ratio between the indicator and the sum of all values for that indicator.

The indicator has to be validated in order to determine if it characterized or not by the properties: sensitivity, non-compensatory, non-catastrophic (Ivan and Pavel, 2011).

Let's consider *n* the number of times the test data generator runs in order to provide *n* sets of complete test data. These runs are executed in order to measure the performance of the test data generator. The number of executions depends on the user settings.

Total number of runs (*NR*) represents the number of execution of SUT required to achieve the required coverage for target paths. *NR* is calculated using the number of runs (*nr*) for each generation *i*:

(6)

The number of runs increases as the SUT complexity increases. Also, this is influenced by the test data generator efficiency.

A specific indicator that measure the *efficiency* (*ef*) of a test data generator is calculated using the number of required data sets (DS) and the total number of generated data sets (TDS):

(7)

Efficiency indicator has value 1 when the all data sets are generated sequentially without any duplicates and tend to zero when the number of tries is very high.

The average indicator for AEF takes into account each runs and is calculated as:

(8)

Average duration (*AD*) represents the average duration for a run in order to obtain all data sets required to cover all the required paths. The average is calculated starting from the duration of each individual run (*d*):

(9)

Usually, the duration of a run is expressed in milliseconds (ms). It is important to use the same configuration in order to obtain comparable values for this indicator.

C represents the test data generator *complexity*. This can be measured in several ways like Cyclomatic complexity (CC), Halstead complexity, function points (*FP*), or line of codes (*KLOC*).

For example, a proposed aggregated indicator for measuring the performance of a test data generators could be defined as:

(10)

or

(11)

The aggregated indicator is calculated for one SUT, running the TDG multiple times. Also, the indicator could be aggregated in order to include several SUTs used as basis for performance assessment.

The aggregated indicator has values between 0 and 1, the best TDG being the closest to 1.

The weights are calculated based on the required test data generator performances. They differ from user to user and are application specific.

The framework can be extended using other user-defined indicators.

6. RESULTS AND DISCUSSIONS

In order to apply the indicators, two test data generators were used and four programs for testing. The following configuration was used for these tests: Windows 8 64 bits operating system, Intel Core i7 processor and 4 GB of RAM. The software was developed using in C# using Visual Studio.

The software under tests are presented in Table 2. These are simple programs in order to focus on the current research. After the framework is set up, the research should focus on more complex SUTs.

Table 2. Software under test

SUT	Description	Number of feasible paths
QuadEq	Determines the solutions for quadratic equation	4
Sort3	Sort three numbers	7
TriClass	Triangle classification	5
GCD	Calculate the greatest common divisor (GCD) using Euclid's algorithm	8

The first test data generator generates data randomly. In consists of a single class having cyclomatic complexity 19. The complexity is reduced due algorithm simplicity. A better version of the data generation algorithm will lead to an increased complexity and better results.

The second test data generator is based on genetic algorithms (GA). It consists of three classes with cumulated cyclomatic complexity of 76. The fitness function is based on Inverse Similarity of Coverage.

The recorded indicator values are presented in Table 3. The SUT column represents the program for which data was generated, and the following columns are associated to three indicators presented in previous section.

Table 3. Performance indicators results

SUT	NR		AD (ms)		C (Cyclomatic)	
	Random	**GA**	**Random**	**GA**	**Random**	**GA**
QuadEq	74131.88	46277.44	0.1	0.6	19	76
Sort3	555.53	586.08	0.008	0.02		
TriClass	151929.47	37145.6	0.02	0.05		
GCD	50.4	72.16	0.00008	0.002		

As it can be seen from Table 3, for two SUTs (QuadEq and TriClass) the number of runs is higher for random TDG (two and four times more) than the GA test data generator.

In all cases the average duration for random test data generation is less than the average duration for GA-based test data generation. The differences are very high for Sort3 and GCD.

The complexity of GA solution for TDG is far higher than the complexity of random TDG.

The normalized values for the recorded values of the indicators are presented in Table 4.

Table 4. Performance indicators normalized

SUT	NR		AD		C (Cyclomatic)	
	Random	GA	Random	GA	Random	GA
QuadEq	0.38	0.62	0.86	0.14	0.86	0.14
Sort3	0.51	0.49	0.71	0.29		
TriClass	0.20	0.80	0.71	0.29		
GCD	0.59	0.41	0.96	0.04		

Using the normalized values the aggregated indicator is calculated as, for QuadEq, for random TDG:

(12)

and for GA TDG:

(13)

For example, if $w_1 = 1$, and $w_2 = w_3 = 0$, then results that GA based TDG is superior to random TDG. That means, that for this case, the user is interested only in the number of runs and the other values are not important. This is equivalent to a classification of TDGs by the number or runs (the smaller the better).

If all weights are equals ($w_1 = w_2 = w_3 = 0.33$), it will result that random test data generator is better than the GA-based test data generator.

After the analysis of these value, the next step is to optimize the TDG in order to reduce the complexity and to identify the bottlenecks. This involves removal of unnecessary data sets based on recorded data, or fitness function optimization for GAs.

7. CONCLUSION AND FUTURE WORK

The proposed method presents a framework that helps to compare test data generators implemented using different algorithms. The current research shows that test data generators can be easily compared and analyzed if:

- all will run against the same programs under test;
- they will be instrumented in order to collect required data for indicators;
- the same testing platform will be used for analysis;

Next steps includes:

- development of new indicators;
- aggregated indicator validation using more test data generators;
- implementation of a system in order to automate the analysis process;

This represents an important step in achieving a high degree of quality for software applications.

REFERENCES

[1] **Meyers, G. J. (2004)*, The Art of Software Testing*, *Second Edition*, *John Wiley & Sons*, New Jersey;

[2] **Pressman S. (2009)**, ***Software Engineering: A Practitioner's Approach*. *7th ed.***, *McGraw-Hill*, New York;

[3] **Sommerville I. (2011)**, ***Software Engineering. 9th ed.***, *Addison-Wesley*, Boston;

[4] **Ivan, I., Pocatilu, P. (1999)**, ***Testarea software orientat obiect***, *Inforec Publishing House*, Bucharest;

[5] **McCabe, J. T. (1976)**, ***A Complexity Measure***, *IEEE Transaction on Software Engineering*, Vol. SE-2, No. 4, December, pp. 308-320;

[6] **Haupt, R. L., Haupt, S. E. (2004)**, ***Practical Genetic Algorithms***, Second Edition, John Wiley & Sons, Inc., Hoboken, New Jersey;

[7] **Maries I., Dezsi D. (2011)**, ***A Genetic Algorithm for Community Formation in Multi-Agent Systems***, *Economic Computation and Economic Cybernetics Studies and Research*, vol. 45, no. 3, 199-212;

[8] **Visoiu, A. (2011), *Deriving Trading Rules Using Gene Expression Programming***, *Informatica Economica*, vol. 15, no. 1, pp. 22-30;

[9] **Pocatilu, P., Mihai, T., Cazan, D. (2001)**, ***Test Data Generation Using Genetic Algorithms***, *Proceeding of the Workshop on Evolutionary Algorithms*, Bucharest ;

[10] **Pocatilu, P., Mihai, T. (2001)**, ***An Evolutionary Method for Test Data Generation***, *Proc. of the Fifth Symposium on Economic Informatics*, 10-13 May, Bucharest, pp. 761-764;

[11] **Malhotra, R., Garg, M. (2011)**, ***An Adequacy Based Test Data Generation Technique Using Genetic Algorithms***, *Journal of Information Processing Systems*, Vol.7, No.2, June, pp. 363-384;

[12] **Shimin, L., Zhangang, L. (2011), *Genetic Algorithm and its Application in the path-oriented test data automatic generation***, *Procedia Engineering*, Vol. 15, pp. 1186-1190;

[13] **Gao K, Khoshgoftaar T.M., Wang H, Seliya N. (2011)**, ***Choosing Software Metrics for Defect Prediction: An Investigation on Feature Selection Techniques***. *Software - Practice and Experience*, vol. 41, pp. 579–606;

[14] **Ivan, I., Pavel S.L. (2011), *Quality Metrics System for Very Large Collections***, *Economy Informatics*, vol. 11, no. 1, pp. 163-177;

[15] **Alshraideh, M. A., Mahafzah, B. A., Salman, H. S. E., Salah, I. (2013), *Using Genetic Algorithm as Test Data Generator for Stored PL/SQL Program Units***, *Journal of Software Engineering and Applications*, vol. 6, pp. 65-73.