A Genetic Algorithm-based System for Automatic Control of Test Data Generation

Paul POCATILU, Ion IVAN

Bucharest University of Economic Studies,
6, Piata Romana, Bucharest, Romania,
ppaul@ase.ro, ionivan@ase.ro

Abstract: Software testing is an important process that helps to develop high quality software. This process is more time consuming when applied to control systems. This involves the use of several testing strategies, techniques and methodologies. At the module level one of test technique is to assure as much as possible code coverage. This is accomplished using several methods, one of them being automatic test data generation. Test data generation can be done manually, randomly or using combinatorial optimizing techniques. Another technique involves the use of genetic algorithm (GA). The paper presents a system that involves an automatic control of test data generation. It also provides implementation details of a test data generator (TDG) based on GA that uses a specific fitness function called Inverse Similarity of Coverage (ISC). The test data generator is a module of the proposed system. The results show that the proposed solution, GA-TDG, has far better results in many relevant situations than random test generators regarding the number of software under test (SUT) runs.

Keywords: software testing process, control system, genetic algorithm, test data generation, fitness function, code coverage, random data generation, control decision.

1. Introduction

One of the most important goals for software providers is to deliver high quality software to customers, especially if the software is embedded in a real-time, a control system or a safety critical system. Software quality involves many processes like software testing [1], [3], [5] and software metrics collection and analysis, according to [14].

Software testing processes target the study of software stability as a systemic approach. The management of these processes is influenced by various factors, and it is very important to identify the processes that can be optimized and have a better control over them.

The software embedded in control systems is implemented using algorithms with different levels of complexity. Regardless of their complexity, the software has to be tested applying several strategies.

Every control system requires a rigorous testing. The software embedded in a control system has to be tested thoroughly before going in production. One important step in testing is test data generation.

Test data generators (TDG) work based on the internal structure of the software under test. TDGs require source code access or code execution results supplied by instrumentation and execute the software in a simulator or on the device. Data set generation processes are time consuming, even these processes are automated. A possible solution that optimizes these processes involves the use of genetic algorithms (GA) in test data generators. By using GA the required level of coverage for code execution or states activation can be achieved having a better control using the input parameters.

Usually, TDGs use the associated control flow graph (CFG), control flow diagram (CFD) or tree of the program in order to produce test data. Software metrics are strongly related to analysis of test data generators in order to improve them and to find better solutions.

Test data generators are built so large variations of input data sets will influence in a similar way the program behavioral characteristics (duration, number of segment covered etc.). The stability and predictability of TDG are very important for testing system control. That means that we could approximate the duration of test data generation process based on the inputs (programs, CFG, CFD, constraints etc.).

The objective of this paper is to highlight the benefits of our proposed solution that can be implemented for any control system.

The paper is structured as follows.

The section Software Testing Process and Test Data Generation presents the main software testing strategies, focusing on structural testing. Also, in this section test data generators are analyzed.

The GA-TDG Solution section deals with the description of the dedicated module within the system based on genetic algorithm. The section analyzes the fitness function and details the practical implementation of our genetic algorithm. Here are presented the resulted solution GA-TDG and the solution used for random test generation, RND-TDG.

These sections include references to previous works in approached areas.

In the Experimental Results section, we present data collected after several runs of the GA-TDG compared with the results collected from the RND-TDG.

Discussions section analyzes the results of our algorithm.

The paper ends with conclusions and future work for this current research.

2. Software Testing Process and Test Data Generation

The objective of software testing is to find errors. Software testing involves planning, tests design, tests running, measurement and data collection. There are two main software strategies [1], [5], [6]: functional testing and structural testing.

Functional testing or black box testing does not require knowledge about the internal structure of program but it requires the existence of program specifications. Based on the specifications, it can be determined if the output of a program is correct for the given input data. Functional testing includes boundary value analysis and equivalence class methodologies.

Structural testing or white-box testing is based on internal structure of the program and is done at code level. Structural testing is focused on program coverage at required levels: statements, branches, paths, blocks, data flows or functions. Structural testing is adequate for unit testing due to code complexity.

The main input of the system is the software under test. The output is tested software, with identified errors fixed and associated reports and metrics. The feedback loop is essential in order to have a system under control.

Every program has associated a Control Flow Graph (CFG) structure. This is represented as a graph but could be transformed in a tree structure. The transformation implies a multiplication of nodes within the CFG.

The nodes from CFG correspond to statements, or sequence of statements or functions. The correspondence is given by the required degree of detail.

The following nodes can be identified in a tree or graph structure associated to a program:

- a root (entry) node corresponding to the first statement of the program;
- intermediate nodes corresponding to statements within the program;
- leave nodes (exits) corresponding to statements at the end of a path within the program.

A path within a program is characterized by an entry node, intermediary nodes and an exit node. The number of paths within a program depends on the number of decision blocks (if-then-else, switch) and loops (while, do-while, for). The number of paths within a function can be very large because every loop or decision block increases the number of paths. Even a function with several lines of code can have an infinite number of paths when it contains loops, so an exhaustive testing of all paths within a function is usually impossible [7]. The minimal number of independent paths that can be used for path coverage is given by the cyclomatic complexity coverage. The McCabe's cyclomatic number of a function with associated graph G is given by [13]:

$$V(G) = e - n + 2p \quad (1)$$

where:

e – number of edges from the graph G;

n – number of nodes from the graph G;

p – number of connected components.

V(G) represents the number of linearly independent paths through a function or program. These paths can be combined to generate all possible paths within the function or program. A smaller number of paths than the cyclomatic complexity number will lead to uncovered paths within the program or function under test. The cyclomatic complexity is independent of the programming language and it measures the number of decision points in a software unit [13].

Test data is generated manually or using dedicated software. It is difficult to generate large sets of data manually and test data generators (TDG) are used instead. TDG are programs that based on specific criteria, generate test data for software under test (SUT). TDG can generate data from simple values to complex ones (arrays, linked lists, trees, objects, files etc.).

TDGs produce random or conducted data based on program specifications or based on program structure. TDGs based on program structure use the associated program's CFG in order to generate test data.

3. GA-TDG Solution

An automatic control of test data generation leads to improved software testing processes. Our proposed system based on genetic algorithm includes two modules: the source code analyzer and instrumentation (SUT analyzer) and the test data generator based on genetic algorithm (GA-TDG).

Genetic algorithms are optimization techniques that involve searches of a large solution space for an optimal solution to the given problem based on natural selection [11], [12], and [19].

Genetic algorithms (GA) are used in various domains, including software testing, especially on automatic test data generation.

The use of genetic algorithms for test data generation has been the subject of many research papers, like [2], [4], [9], [10], [16] and [18]. The authors propose different fitness functions and compare the results of theirs algorithms with other evolutionary algorithms or with random test generators.

The input of the genetic algorithm is the instrumented function that will return the covered path with current solution and a list of paths that need to be covered. The output is the set of generated test data.

Figure 1 depicts the architecture of our proposed system that helps to control and to manage test data generation processes.

The type and the constraints of the input parameters of the SUT have to be identified correctly in order to encode them into a chromosome.

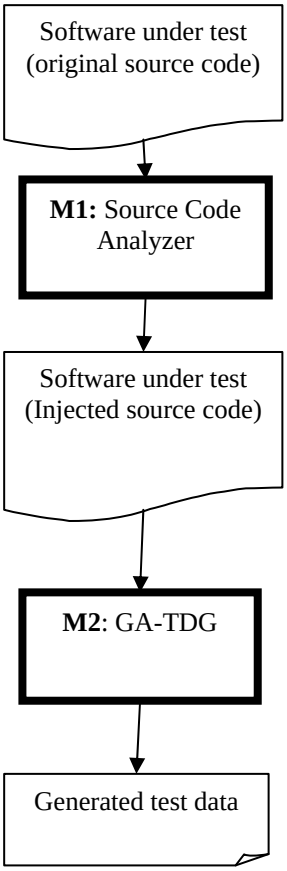

Figure 1. The technology of a GA-based system for automatic control of test data generation

In order to choose a representative population size, the McCabe's cyclomatic number could be a good start. Based on the cyclomatic number, that gives the minimum independent path within a CFG, the population size can be selected as a multiple of this metric.

The solution of our algorithm is a set of chromosomes that represents the input parameters for the SUT.

The steps of the proposed genetic algorithm for test data generation are:

S1. Initiate chromosome population with random values;

S2. Create an empty solution set (*S*);

S3. While (termination condition is not true)

a) Run SUT for every chromosome (test data) and collect coverage metrics;

b) Add chromosomes not already included to the solution set *S*; if the solution is complete (all required path have been covered) then stop;

c) Evaluate the current population by calculating the fitness for each individual;

d) Select the best chromosomes for the next population based on their fitness;

e) Generate the new population by applying crossover and mutation.

Each individual (potential solution) covers a path of the program CFG. The solution set *S* is generated by encouraging chromosomes to cover paths that have the maximum length and minimum similarity characteristics. By using this fitness functions, the chromosome selection process will provide individuals that assure maximum degree of coverage. The fitness of an individual is proportional with the length of the traversal path generated by the individual.

For each generation, fitness function evaluates every solution focusing on covered path. Based on their fitness function, solutions are selected and genetic operators are used to generate new solutions. For example, in [2] is defined the fitness function Last Block Traversal Probability with Bonus (LBTPB). The fitness bonus helps the genetic algorithm to search for unexplored paths of the SUT. The LBTPB fitness function is compared with another fitness function Inverse Path Probability (IPP). In [9] the fitness function is defined based on the number of predicates (decisions, loops) on the evaluated path. In [20] the fitness function is based on test data available for a path. As the number of data increases, the fitness value will decrease.

Our fitness function *f(a)*, initially presented in [8], named *Inverse Similarity of Coverage (ISC)*, applied to chromosome under evaluation *a* is given by:

$$f(a) = \frac{len(a)}{sym(a)},\ \mathrm{f} : N^* \rightarrow \mathrm{R}^* \qquad (2)$$

In (2), *len(a)* is the length of the path given by the chromosome *a*. The number of nodes in the path gives the length of the path.

The similarity function *sym(a)* is given by:

$$sym(a) = \max_{i \in P}(share(a,i)) \qquad (3)$$

where *a* is the chromosome under evaluation, *i* is a chromosome within the population *P* and *share(a, i)* is the function that returns the shared path of two chromosomes. As it can be seen, chromosome similarity is defined as the longest shared path among chromosomes from the current population, which is the number of common tree nodes.

The idea behind the *sym(a)* function is that the closer to the tree root a chromosome branches from all previously discovered chromosomes the higher the probability to increase coverage and discover an error [8].

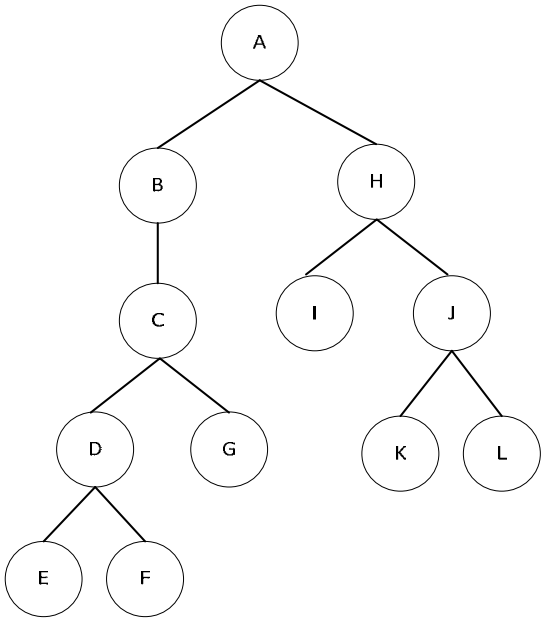

Figure 2. Example of tree structure

As example, we will consider the tree structure depicted in Figure 2 that is based on a part of a CFG associated to a SUT. Let's suppose that at a given iteration within the population there are four chromosomes that return the paths: ABCDE, ABCG, AHI, and AHJK. Based on the calculated fitness, presented in Table 1, the chromosomes 4 and 1 have a higher chance to be chosen for the next generation in order to find the best solution.

To summarize, the GA searches for individuals that can increase the solution coverage by including nodes closer to the tree root. This will increase the chances to discover unexplored paths.

Table 1. Example results

Chromosome	Path	len	sym	fitness
1	ABCDE	5	3	1.66
2	ABCG	4	3	1.33
3	AHI	3	2	1.5
4	AHJK	4	2	2

Our TDG using GA was implemented for .Net platform using C# programming language as GAT_G class. The Chromosome class was defined. Population is implemented as a set of chromosome in a class named ChromosomeSet.

For simplicity all input parameters have the same type, sbyte. They are encoded as sbyte arrays. Consequently, for a function with three parameters, the associate chromosome will include an array with three elements. The boundary values of the parameters can be controlled by giving the lower and upper limits.

All functions under test were implemented in a class (named SUT) as static methods. The functions were manually injected with calls to a trace function that appends the covered path for the current input parameters.

In order to compare the results, a random test generator (class RND_G) was created. This runs the function under test using random generated data and records the covered path.

Figure 3 depicts the classes implemented for this solution. The class Program contains the main entry point that handles the test setup and running.

The parameters of the genetic algorithm are population size, crossover and mutation probabilities. The Chromosome class properties control the number of encoded parameters and the value limits.

The best individuals are selected for the next generation based on their fitness using the roulette wheel selection.

The results are presented in the next section.

4. Experimental Results

In order to test the algorithm, four well known functions were used as SUT:

1. Determines the solutions for quadratic equation. The function has three signed parameters;
2. Sort three numbers. The function has three signed parameters;
3. Triangle classification. The function has three signed parameters. The function validates against negative values;
4. Calculate GCD using Euclid's algorithm. The function has two positive parameters. The function includes a loop.

Two metrics (line of codes and cyclomatic number) of the functions under test are presented in Table 2.

Table 2. SUT information

SUT	LOC	CC
1	16	4
2	20	5
3	24	5
4	18	3

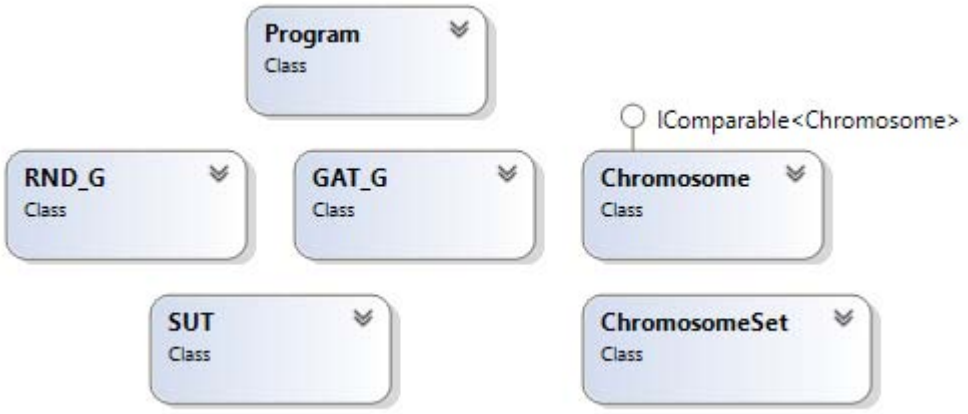

Figure 3. Proposed solution classes

Table 3 presents the evolution of several generations for the SUT 1 (to determine the solutions for quadratic equation). The population size was of 8 chromosomes, there were four paths to be covered. In Table 3 data set represents the generated input data, *f* is the value of fitness function, *p* is chromosome probability based on its fitness and *cp* represents the cumulative probability. *Path* represents path covered by the chromosome.

Table 3. Example of genetic algorithm evolution

Generation: 1, initial population				
Data set	**f**	**p**	**cp**	**Path**
-84, 93, 87	**1.0**	**0.1155**	**0.1155**	**ACEFH**
-12, 113, 39	1.0	0.1155	0.2309	ACEFH
43, 19, -10	1.0	0.1155	0.3464	ACEFH
14, 122, 62	1.0	0.1155	0.4619	ACEFH
78, 70, -119	1.0	0.1155	0.5774	ACEFH
126, -89, -36	1.0	0.1155	0.6928	ACEFH
27, 36, -75	1.0	0.1155	0.8083	ACEFH
-99, -65, -125	**1.66**	**0.1917**	**1.0000**	**ACEGH**
Generation: 704				
Data set	**f**	**p**	**cp**	**Path**
64, -113, 58	1.0	0.1111	0.1111	ACEGH
110, -20, 58	1.0	0.1111	0.2222	ACEGH
67, -17, -21	1.0	0.1111	0.3333	ACEFH
66, -85, -23	1.0	0.1111	0.4444	ACEFH
66, -85, -21	1.0	0.1111	0.5556	ACEFH
-62, -20, 58	1.0	0.1111	0.6667	ACEFH
102, -113, 58	1.0	0.1111	0.7778	ACEGH
0, -113, 58	**2.0**	**0.2222**	**1.0000**	**ACDH**
Generation: 3405, last population				
Data set	**f**	**p**	**cp**	**Path**
64, 33, 40				ACEGH
-64, 11, -1				ACEGH
86, -96, 126				ACEGH
0, 32, 126				ACDH
118, 33, 126				ACEGH
0, 0, -2	**3**	**-**	**-**	**ABH**
0, 32, 126				ACDH
0, 34, 126				ACDH

The highlighted lines in Table 3 (bold) represent the chromosome included in solution. The solutions were found in first generation (two), generation 704 (one) and the last one on the generation 3405.

The results for 100 runs of both genetic (GA-TDG) and random data generator (RND-TDG) are presented in Table 4. The GA-TDG was set up with the following parameters:

- Population size: 100
- Mutation probability: 0.8
- Crossover probability 0.1
- Number of tests: 100

The columns from Table 4 have the following meanings: SUT is the function under test, then, number of paths to be covered, and, for each type of TDG (GA and Random) are included average duration and the number of runs (average, minimum and maximum).

The averages were calculated for 100 runs of each generator.

As it can be seen, the number of paths is greater or equal to cyclomatic complexity number.

Figure 4 shows a part of a profiler report generated within the Visual Studio for our program.

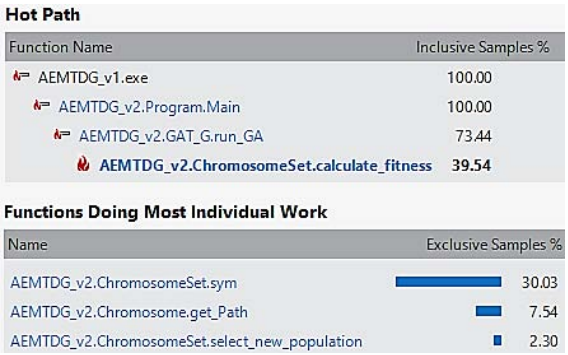

Figure 4. Profiler results

The most time consuming function is sym() that calculates the degree of similitude within the current population.

The GA-TDG results were recorded on a computer with the following configuration: Windows 8 64 bits operating system, Intel Core i7 processor and 4 GB or RAM.

Table 4. Program results

SUT	**No. of paths**	**GA**				**Random**			
		Avg. duration (sec.)	**Avg. runs**	**Min. runs**	**Max runs**	**Avg. duration (sec.)**	**Avg. runs**	**Min. runs**	**Max. runs**
1	4	0.06473	7075	300	42200	0.01328	63790.4	323	370835
2	7	0.00680	664	100	2600	0.00089	490.21	14	2321
3	5	0.23725	27720	200	131400	0.02439	136689.62	911	700044
4	8	0.00103	107	100	300	0.00091	57.75	10	185

5. Discussions

As it can be seen, the random data generator has better results in respect of timing due the reduced number of computations. On the other side, the GA performs slower due to fitness function.

The most important, in most of the tested cases, the GA-TDG runs the SUT fewer times than the random test data generator. For functions with higher complexity and processing effort, the number of runs is crucial for the testing performances.

The TDG GA can be tuned to work closer or even better than de random test generator. The parameters that can be changed are the population size and the crossover and mutation probability. For example, if the GA-TDG ran 100 times with a population of 8 chromosomes and the mutation probability 0.1 and the results were:

- average duration: 0.1653 seconds;
- average runs: 461.52;
- minimum runs: 16;
- maximum runs: 1528.

This shows that the results are slightly better than the similar results presented in Table 4, for the second SUT.

First generation is initialized randomly, so the GA-TDG is comparable as behavior with RND-TDG for a number of steps equals with the population size.

As depicted in Figure 5, for the first SUT, the GA-TDG was six times slower than the RND-TDG (0.06 sec. vs. 0.01 sec.). The ratio is high, but is the GA-TDG is still very fast.

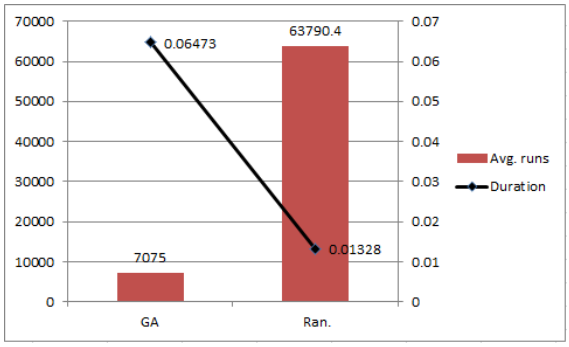

Figure 5. Compared results for average runs and duration

Regarding the runs, on average, the GA-TDG called the SUT 82 time less than the Random TDG (7075 runs vs. 63790 runs). This could impact the test duration for complex programs: a higher number of calls will lead to a longer duration.

The algorithm can be easily adapted for arrays and other complex structures. Also, even the functions under test have a lower complexity due to need to manually generate injection we are confident that the GA-TDG will behave well for complex functions.

In order to have a high usable system and avoid all associated risks as presented in [15] and [17], the proposed system further implementation has to take into account user-interface response time and user messages.

6. Conclusions and Future Work

Computer programs work as systems and their setting and control is preceded by software testing processes.

Genetic algorithms are successfully implemented for test data generator. Their performances are, in many situations, very good compared with random data generation.

The proposed solution presented in this paper helps to control and automate test data generation. The control is made by tuning the genetic algorithm parameters.

The paper presents the results of a test data generator module implemented using genetic algorithms. The module is part of a system that assures automatic control of test data generation. The presented GA-TDG finds solutions that cover all feasible target paths. As it can be seen, our test data generator has good performances reported to random TDG. The solution can be applied for practical projects, and is not only for experimental tests.

GA-TDG solution is suitable also for agile development, where unit testing has a very important role in these types of software development process. Also, this solution can be applied for larger projects, by generating data both for functions and the integrated modules.

The source code analyzer module has to be completed in order to automatically inject the SUT. Also, it will include a component that generates tree state transformations from CFG or SUT source code.

We will continue our research in order to improve the fitness function. Also, the research will aim to compare the results with other TDG

based on genetic algorithms. Due the fact that other authors use different software under tests, we plan to implement several solutions (fitness functions) and to use them against our solutions.

REFERENCES

1. BEIZER, B., **Software Testing Techniques – Second Edition**, Van Nostrad Reinhold, New York, 1990.
2. BORGELT, K., **Software Test Data Generation from a Genetic Algorithm**, Industrial Applications of Genetic Algorithms, Ed. By Charles L. Karr and L. Michael Freeman, CRC Press, Boca Raton, 1999, pp. 49-68.
3. IVAN, I., P. POCATILU, **Testarea software orientat obiect**, Inforec Publishing House, Bucharest, 1999.
4. MICHAEL, C. C., G. E. MCGRAW, M. A. SCHATZ, C. C. WALTON, **Genetic Algorithms for Dynamic Test Data Generation**, In Proc. of IEEE International Automated Software Engineering Conference (ASE97), Nov. 3-5, 1997, pp. 307-308.
5. MEYERS, G. J., **The Art of Software Testing**, Second Edition, John Wiley & Sons, New Jersey, 2004.
6. ROPER, M., **Software Testing**, McGraw-Hill Book, 1994.
7. POCATILU, P., T. MIHAI, D. CAZAN, **Test Data Generation Using Genetic Algorithms**, Proceeding of the Workshop on Evolutionary Algorithms, Bucharest, 2001.
8. POCATILU, P., T. MIHAI, **An Evolutionary Method for Test Data Generation**, Proc. of the Fifth Symposium on Economic Informatics, May 10-13, 2001, Bucharest, pp. 761-764.
9. PARGAS, R. P., M. J. HARROLD, R. R. PECK, **Test-Data Generation Using Genetic Algorithms**, Software Testing, Verification And Reliability, vol. 9(4), 1999, pp.263-282.
10. SRIVASTAVA, P. R., T.-H. KIM, **Application of Genetic Algorithm in Software Testing**, International Journal of Software Engineering and Its Applications, vol. 3(4), 2009, pp. 87-95.
11. NELSON, R. C., **Overview of Genetic Algorithms**, 1999. http://www.cs.rochester.edu/~nelson/courses/csc_173/genetic-algs/algorithm.html
12. VISOIU, A., **Deriving Trading Rules Using Gene Expression Programming**, Informatica Economica, vol. 15(1) 2011, pp. 22-30.
13. McCABE, J. T., **A Complexity Measure**, IEEE Transaction on Software Engineering, vol. SE-2, no. 4, Dec. 1976, pp. 308-320.
14. BOJA, C., L. BATAGAN, **Analysis of m-Learning Applications Quality**, WSEAS Transactions on Computers, vol. 8(5), 2009, pp. 767-777.
15. BALOG, A., C. PRIBEANU, **Developing a Measurement Scale for the Evaluation of AR-Based Educational Systems**, Studies in Informatics and Control, Vol. 18, No. 2, June 2009, pp. 137-148.
16. MALHOTRA, R., M. GARG, **An Adequacy Based Test Data Generation Technique Using Genetic Algorithms**, Journal of Information Processing Systems, vol.7(2), 2011, pp. 363-384.
17. SUDUC, A. M., M. BÎZOI, F. G. FILIP, **User Awareness about Information Systems Usability**, Studies in Informatics and Control, vol. 19(2), 2010, pp. 145-152.
18. KHAMIS, A. M., M. R. GIRGIS, A. S. GHIDUK, **Automatic Software Test Data Generation for Spanning Sets Coverage Using Genetic Algorithms**, Computing and Informatics, vol. 26, 2007, pp. 383-401.
19. HAUPT, R. L., S. E. HAUPT, **Practical Genetic Algorithms**, Second Edition, John Wiley & Sons, Inc., Hoboken, New Jersey, 2004.
20. SHIMIN, L., L. ZHANGANG, **Genetic Algorithm and its Application in the Path-oriented Test Data Automatic Generation**, Procedia Engineering, vol. 15, 2011, pp. 1186-1190.