Researching of methods for assessing the complexity of program code when generating input test data

Konstantin Serdyukov
Novosibirsk State Technical University
Novosibirst, Russia
zores@live.ru

Tatyana Avdeenko
Novosibirsk State Technical University
Novosibirst, Russia
tavdeenko@mail.ru

***Abstract*—This article proposes a comparison of methods for determining code complexity when generating data sets for software testing. The article offers the results of a study for evaluating one path of program code, the work is not finished yet, it will be further expanded to select data for testing many paths. To solve the problem of generating test data sets, it is proposed to use a genetic algorithm with various metrics for determining the complexity of program code. A new metrics is proposed for determining code complexity based on changing weights of nested operations. The article presents the results and comparison of the generated input test data for the passage along the critical path. For each metric considered in the article, conclusions are presented to identify specifics depending on the selected data.**

Keywords—test data generation, genetic algorithm, metrics of asserting code complexity

I. INTRODUCTION

Software engineering is a comprehensive, systematic approach to the development and maintenance of software. When developing programs, the following stages are most often distinguished - analysis, design, programming and testing. At the stage of analysis, software requirements are determined and documentation is performed. At the design stage, the appearance of the program is detailed, its internal functionality is determined, the product structure is developed, and requirements for subsequent testing are introduced. Writing the source code of a program in one of the programming languages is done at the programming stage.

One of the most important steps in developing software products is testing. Important goals of testing are the compliance of the developed program with the specified requirements, adherence to logic in the data processing processes and obtaining correct final results. Therefore, for testing it is very important to generate input test data, on the basis of which the program will be checked for errors and compliance with specified requirements. To esteem the quality of the input data a code coverage indicator is used, that is percentage of the entire program can the test sets "cover". It is determined by the ratio of the tested operations to the total number of operations in the code.

Some software code testing processes are improving quite slowly. The development of most types of test scenarios is most often done manually, without the use of any automation systems. Because of this the testing process becomes incredibly complicated and costly both in time and in finances, if you approach it in all seriousness. Up to 50% of all time costs can be spent on testing some programs.

One of the main goals of testing is to create a test sets that would ensure a sufficient level of quality of the final product by checking most of the various paths of the program code, i.e. would provide maximum coverage. Nevertheless, the task of finding many paths itself consists of several sub-tasks, the solution of which is necessary to find high-quality test sets. One of the local problems that can be solved to find a test set is to determine one of the most complex code paths.

For the most part, validation and verification of software products is difficult to optimize. It is especially difficult to automate the generation of test data, which for the most part is done manually.

Nevertheless, there are many studies using non-standard algorithms to solve the automation problem. For example, in [1] it is proposed to use a Constraint-Based Algorithm for the Mort system, which uses error testing to find input test data. Test data is selected in such a way as to determine the presence or absence of certain errors.

Quite often, genetic algorithms are used in one way or another to solve this problem. The article [2] compares different methods for generating test data, including genetic algorithms, a random search method and other heuristic methods.

In [3] to solve the problem, it is proposed to use Constraint Logic Programming and Symbolic Execution. In [4], the Constraint Handling Rules are used to help in manual verification of problem areas in the program.

Some researchers use heuristic methods to automate the testing process using a data-flow diagram. Studies of automation methods using this diagram were done in articles [5, 6, 7, 8]. In [5] it is proposed to additionally use genetic algorithms to generate new input test data sets based on previously generated ones.

In articles [9, 10] it is proposed to use hybrid methods for generating test data. In [9], an approach is used that combines strategies of Random Strategy, Dynamic Symbolic Execution and Search-Based Strategies. The article [10] proposes a theoretical description of the search method using the genetic algorithm. The approaches to search for local and global extrema on real programs are considered. A hybrid approach for generating test data is proposed - a Memetic Algorithm.

The approach in [11] uses a hybrid intelligent search algorithm to generate test data. Proposed approach center on the methods of Branches and Borders and the Hill Climbing to improve intellectual search.

There are also studies using machine learning, for example, in [12]. It proposes a method using a neural network

and user-customizable clustering of input data for sequential learning.

Novelty Search can also be used to generate test data. In the article [13] it is proposed to use this approach to evaluate large spaces of input data and is compared with approaches based on the genetic algorithm.

The possibilities of generating test data for testing web services are also being investigated, for example, in the WDSL specification [14].

For the convenience of generating test data, UML diagrams are also used [15, 16]. The articles suggest using genetic algorithms to generate triggers for UML diagrams that will allow to find a critical path in the program. The article [17] proposes an improved method based on a genetic algorithm for selecting test data for many parallel paths in UML diagrams.

In addition to UML diagrams, the program can be described as a Classification-Tree Method developed by Grochtmann and Grimm [18]. In [19] the problem of constructing trees is considered and an integrated classification tree algorithm is proposed, and in [20] was investigated the developed ADDICT prototype (short for AutomateD test Data generation using the Integrated Classification-Tree methodology) for an integrated approach.

This article proposes a comparison of different methods for evaluating code complexity for generating test data. The article is structured as follows. Section 2 introduces terminology and provides basic information on the genetic algorithm. The third section sets the problem to be solved and introduces one of the methods for assessing code complexity. Section 4 proposes the results of the operation of the input data generation method using the introduced code estimation method. In section 5 there is comparing of different code evaluation methods.

II. GENETIC ALGORITHM

Formally, the genetic algorithm is not an optimization method, at least in the understanding of classical optimization methods. Its purpose is not to find the optimal and best solution, but to find close enough to it. Therefore, this algorithm is not recommended to be used if fast and well-developed optimization methods already exist. But at the same time, the genetic algorithm perfectly shows itself in solving non-standardized tasks, tasks with incomplete data or for which it is impossible to use optimization methods because of the complexity of implementation or the duration of execution [21, 22].

A genetic algorithm is considered to be completed if a certain number of iterations is passed (it is desirable to limit the number of iterations, since the genetic algorithm works on the basis of trial and error, which is a rather lengthy process), or if a satisfactory value of the fitness function was obtained. As usual a genetic algorithm solves the problem of maximizing or minimizing and the adequacy of each solution (chromosome) is evaluated using the fitness function.

The genetic algorithm works according to the following principle:

Initialization. A fitness function is introduced. An initial population is being formed. In classical theory, the initial population is formed by randomly filling each gene in the chromosomes. But to increase the rate of convergence of the solution, the initial population can be specified in a certain way, or random values can be analyzed in advance to exclude definitely inappropriate ones.

Population assessment. Each of the chromosomes is evaluated by a fitness function. Based on the given requirements, chromosomes get the exact value of how well they correspond to the problem being solved.

- Selection. After each of the chromosomes has its own fitness value, the best chromosomes are selected. Selection can be done by different methods, for example, from the sorted in order first n chromosomes are selected, or only the most suitable, but not less than n, etc.
- Crossing. [23]. The first is a significant difference from standard optimization methods. After selection of chromosomes suitable for solving the problem, they crossing. Random chromosomes from all the "chosen ones" randomly generate new chromosomes. Crossing occurs on the basis of the choice of a certain position in two chromosomes and the replacement of parts of each other. After the required number of chromosomes is generated to create a population, the algorithm proceeds to the next step.
- Mutation. [24]. Also the step specific to GA. In a random order, a random gene can change values to a random one. The main point in a mutation is the same as in biology - to bring genetic diversity into a population. The main goal of mutations is to obtain solutions that could not be obtained with existing genes. This will allow, firstly, to avoid falling into local extremes, since a mutation can allow the algorithm to be transferred to a completely different branch, and secondly, to “dilute” the population in order to avoid a situation where in the whole population there will be only identical chromosomes that will not generally move towards a solution.

After all the steps have been passed on, it is defined whether the population has reached the desired accuracy of the decisions or has come to limit the number of populations, and if so, the algorithm stops working. Otherwise, the cycle with the new population is repeated until the conditions are achieved.

III. PROBLEM DESCRIPTION

The use of genetic algorithms in the testing process allows to find the most complex parts of the program in which the risks due to errors are greatest. Evaluation occurs due to the use of the fitness function, the parameters of which are the weights of each passable operation. Definition of weights, i.e. the complexity of the program code, occurs due to various metrics used depending on the requirements for the input sets.

The task of generating input test data consists of three subtasks:

1. Search for input data for passing along one of the most complex code paths. Difficulty is determined by the chosen metric for code evaluation;
2. The exclusion or reduction of the weights of operations on the path for which the data were selected, based on the fitness function for other paths;

3. Generation of input test data for many paths of program code.

The limit on the number of sets of input data is established after the development stage and will allow to concentrate on certain paths.

The whole algorithm is performed cyclically - the procedure for searching for input data for one path is started, after which operations in this path are excluded from further calculations and the data search for one path is started again.

As one of the ways to determine the complexity of the code, an method is proposed that works as follows:

- The first operation is assigned a weight of, for example, 100 units.
- Each subsequent operation is also assigned a weight - if there are no conditions or cycles, the weight is taken in accordance with the previous operation.
- Conditions share the weight in accordance with the rule - if the condition contains only one branch (only if ...), then the weight of each operation is reduced by 80%. If the condition is divided into several branches (if ... else ...), then the weight is divided into equal parts - for two branches 50% / 50%, for three 33% / 33% / 33%, etc.
- The weights of operations in the cycle remain, but can also be multiplied by certain weights, if necessary.
- All nested restrictions are summed, for example, for two nested conditions the weight of operations will be 80% * 80% = 64%

Assigned weights can be used to develop test cases using genetic algorithms, that is, to assess how much calculated weight assigned on one or another branch for certain values of input parameters.

For convenience, we introduce the following notation:

X - data sets;

F (X) is the value of the fitness function for each data set depending on the calculated values of the weights.

The challenge is to maximize the objective function, i.e. F (X) → max

IV. The results of the method

In accordance with the previously proposed option for assessing the complexity of program code, this method is being finalized to better meet real requirements. Weights are considered in accordance with the operability of the program, in other words, the more iterations the program performs, the more weight the initial test version will have.

The first population is formed by random values. Each population contains 100 chromosomes. The total number of the population is also 100. Due to this, a sufficient number of different options will be formed and the best ones will be selected. Table 1 presents the test results.

In each of the tests, at least two different versions of the data were generated, in which the considered program code will work the most times, which means that it will go the greatest number of times in different ways. In addition, you can see certain patterns in the results - the first value is always maximum (random values were limited to 100), the second value is less than the first, but more than the third.

TABLE I. COMPARISON OF RESULTS

Population	Test 1	Test 2	Test 3	Test 4
0	1: 78, 23, 35 2: 62, 36, 95 3: 52, 35, 27 4: 17, 77, 73 5: 75, 9, 96	1: 97, 3, 6 2: 82, 77, 64 3: 24, 47, 57 4: 90, 13, 82 5: 81, 69, 24	1: 92, 97, 28 2: 38, 66, 52 3: 63, 76, 64 4: 7, 24, 56 5: 57, 48, 8	1: 15, 67, 26 2: 32, 27, 83 3: 37, 52, 64 4: 70, 49, 64 5: 67, 29, 94
20	1: 95, 64, 54 2: 95, 64, 29 3: 95, 64, 54	1: 97, 80, 4 2: 97, 80, 53 3: 97, 80, 28	1: 99, 13, 10 2: 99, 13, 11 3: 99, 13, 11	1: 99, 71, 45 2: 99, 71, 15 3: 99, 71, 3
50	1: 95, 64, 54 2: 95, 64, 29 3: 95, 64, 54	1: 97, 80, 29 2: 97, 80, 4 3: 97, 80, 53	1: 99, 13, 10 2: 99, 13, 11 3: 99, 13, 11	1: 99, 71, 60 2: 99, 71, 3 3: 99, 71, 3
Result (100)	**1: 95, 64, 54 2: 95, 64, 29**	**1: 97, 80, 4 2: 97, 80, 29**	**1: 99, 13, 10 2: 99, 13, 11**	**1: 99, 71, 60 2: 99, 71, 45**

V. Comparison of methods for assessing the complexity of program code

For the researching, several tests of the algorithm with four different metrics were carried out - a modified metric, the logic of which was described in Section 3, SLOC metrics for evaluating the number of lines of code, ABC metrics and Jilb metrics.

The metric SLOC (abbr. Source Lines of Code) is determined by the number of lines of code. This metric takes into account only the total number of lines of code in the program, which makes it the easiest to understand. In this case, the number of lines refers to the number of commands, and not the physical number of lines.

The ABC metric, or Fitzpatrick metric, is a metric that is determined based on three different indicators ABC = $<n_a, n_b, n_c>$. The first indicator n_a (Assignment) is allocated for lines of code that are responsible for assigning variables a certain value, for example, int number = 1. The indicator n_b (Branch) is responsible for using functions or procedures, that is, operands that work out of sight of the current program code. The indicator n_c (Condition) calculates the number of logical operands, such as conditions and loops. The metric value is calculated as the square root of the sum of the squared values of n_a, n_b and n_c.

$$F = \sqrt{n_a{}^2 + n_b{}^2 + n_c{}^2} \quad (1)$$

It is noteworthy that one line of code can be taken into account in different parameters, for example, when assigning a variable the value of a certain function (double number = Math.Pow (2, 3) is assigned both in na and nb). The disadvantages of this metric include the possible return of a zero value in some parts of the code.

The Jilb metric is aimed to determine the complexity of program code depending on its saturation with conditional operands. This metric is useful for determining the complexity of program code, both for writing and for understanding it:

$$F = cl/n, \quad (2)$$

where cl – the number of conditional operands, n – the total number of lines of code.

For testing, code is used with many different paths, where one is critical which has the largest number of operations. The selection of input data for this path will be a solution to the subtask and from this data it will be possible to determine

how accurately the data is selected. The critical path will be reached if the 1st and 3rd values from the selected data are greater than 50 and 1 value is less than 3.

The following genetic algorithm settings are used to generate input test data:

- Number of generations – 100
- Number of populations in one generation – 100

Range of received data values – (0, 100)

A. Results using the metric proposed in the article

An algorithm with this metric selects data with a priority of operations of a higher level. As a result (99th generation), two data sets were obtained - (70, 9, 78) and (75, 67, 82). Both sets go along the longest code path, which is the solution to the subtask. Table 2 presents the first 10 options in each of the generations.

TABLE II. RESULTS OF THE PROPOSED METRIC

Modified metric			
Variant\Gen.	0	1	99
1	(70, 9, 78) = 164 100	(75, 67, 82) = 164 100	(70, 9, 78) = 164 100
2	(75, 67, 82) = 164 100	(61, 29, 94) = 164 100	(70, 9, 78) = 164 100
3	(61, 29, 94) = 164 100	(63, 52, 87) = 164 100	(75, 67, 82) = 164 100
4	(63, 52, 87) = 164 100	(63, 49, 83) = 164 100	(75, 67, 82) = 164 100
5	(63, 49, 83) = 164 100	(70, 9, 78) = 164 100	(75, 67, 82) = 164 100
6	(5, 68, 90) = 96 382	(63, 52, 87) = 164 100	(75, 67, 82) = 164 100
7	(60, 37, 3) = 32 500	(70, 9, 78) = 164 100	(75, 67, 82) = 164 100
8	(12, 80, 49) = 16 000	(70, 9, 78) = 164 100	(70, 9, 78) = 164 100
9	(47, 12, 17) = 16 000	(70, 9, 78) = 164 100	(70, 9, 78) = 164 100
10	(53, 35, 76) = 16 000	(61, 29, 94) = 164 100	(75, 67, 82) = 164 100

Can be seeing that the algorithm works quite efficiently and already in the first generation the data was selected for the critical path.

B. SLOC metric

TABLE III. RESULTS OF METHOD WITH SLOC METRIC

SLOC metric			
Variant\Gen.	0	1	99
1	(64, 14, 96) = 6 411	(68, 50, 94) = 6 411	(63, 72, 91) = 6 411
2	(68, 50, 94) = 6 411	(80, 70, 88) = 6 411	(68, 50, 94) = 6 411
3	(80, 70, 88) = 6 411	(65, 81, 89) = 6 411	(68, 50, 94) = 6 411
4	(65, 81, 89) = 6 411	(63, 72, 91) = 6 411	(63, 72, 91) = 6 411
5	(63, 72, 91) = 6 411	(74, 83, 76) = 6 411	(80, 70, 88) = 6 411
6	(74, 83, 76) = 6 411	(64, 69, 91) = 6 411	(68, 50, 94) = 6 411
7	(64, 69, 91) = 6 411	(69, 88, 85) = 6 411	(68, 50, 94) = 6 411
8	(69, 88, 85) = 6 411	(64, 14, 96) = 6 411	(63, 72, 91) = 6 411
9	(5, 39, 72) = 3 618	(63, 72, 91) = 6 411	(63, 72, 91) = 6 411
10	(2, 67, 73) = 3 618	(68, 50, 94) = 6 411	(80, 70, 88) = 6 411

This metric is the simplest from the point of view of implementation, it takes into account only the total number of lines of code. The results are presented in table 3. The algorithm with this metric picked up 3 sets - (63, 72, 91), (68, 50, 94) and (80, 70, 88). All three satisfy the conditions for passing along the critical path.

As with the previous metric, the algorithm in the first generation picked up suitable data.

C. ABC metric

This metric takes into account more variations of the values, such as assigning values to variables, logical checks and function calls. The algorithm with the ABC metric picked up 2 options for the input data that pass along the critical path - (69, 46, 78) and (77, 36, 98). The remaining results are presented in table 4.

TABLE IV. RESULTS OF METHOD WITH ABC METRIC

ABC metric			
Variant\Gen.	0	1	99
1	(95, 27, 97) = 6 351	(77, 36, 98) = 6 351	(69, 46, 78) = 6 351
2	(77, 36, 98) = 6 351	(69, 46, 78) = 6 351	(77, 36, 98) = 6 351
3	(69, 46, 78) = 6 351	(61, 65, 95) = 6 351	(69, 46, 78) = 6 351
4	(61, 65, 95) = 6 351	(95, 27, 97) = 6 351	(69, 46, 78) = 6 351
5	(5, 67, 92) = 3 538	(61, 65, 95) = 6 351	(69, 46, 78) = 6 351
6	(5, 87, 95) = 3 538	(95, 27, 97) = 6 351	(69, 46, 78) = 6 351
7	(1, 35, 60) = 3 538	(61, 65, 95) = 6 351	(69, 46, 78) = 6 351
8	(1, 70, 53) = 3 538	(69, 46, 78) = 6 351	(77, 36, 98) = 6 351
9	(60, 30, 12) = 768	(69, 46, 78) = 6 351	(69, 46, 78) = 6 351
10	(60, 49, 73) = 768	(69, 46, 78) = 6 351	(69, 46, 78) = 63 51

D. Jilb metric

Unlike previous metrics, this one takes into account the absolute complexity of the program, which is calculated by dividing the number of cycles and conditions by the total number of operations on the way. The complexity of the program is determined in a completely different way, which led to the fact that the input data was selected for a different path. The results are presented in table 5.

TABLE V. RESULTS OF METHOD WITH JILB METRIC

Jilb metric			
Variant\Gen.	0	1	99
1	(75, 51, 3) = 100	(62, 25, 41) = 100	(78, 45, 21) = 100
2	(92, 33, 11) = 100	(94, 22, 35) = 100	(63, 36, 10) = 100
3	(94, 22, 35) = 100	(98, 51, 12) = 100	(75, 51, 3) = 100
4	(98, 51, 12) = 100	(80, 42, 20) = 100	(80, 42, 20) = 100
5	(80, 42, 20) = 100	(78, 45, 21) = 100	(78, 45, 21) = 100
6	(78, 45, 21) = 100	(80, 59, 8) = 100	(63, 36, 10) = 100
7	(80, 59, 8) = 100	(5, 40, 27) = 100	(80, 42, 20) = 100
8	(5, 40, 27) = 100	(99, 38, 29) = 100	(78, 45, 21) = 100
9	(99, 38, 29) = 100	(62, 25, 41) = 100	(75, 51, 3) = 100
10	(62, 25, 41) = 100	(63, 36, 10) = 100	(80, 42, 20) = 100

The data obtained are very different both with other metrics and within the metric. This is due to the features of the tested code - it has one common loop, within which there is one common condition. If this condition is not met, then none of the operations, except the cycle and conditions, will be taken into account when calculating the metric. A value of 100 indicates that among all operations on the path, all are cycles or conditions, i.e. formally selected input data are options when the first condition was not met and other operations were not taken into account.

VI. Conclusion

Evolutionary methods work in such a way as to find the best solutions to problems that are impossible or too costly to solve with standard optimization methods. They do not always work quickly or efficiently, but in problems with non-standard approaches it shows superiority.

The introduction of various metrics for calculating the fitness function made it possible to add a method for generating input test data of greater variability and the ability to introduce new data requirements. Each metric is focused on specific code parameters and can be used when data must be selected in accordance with certain requirements. In addition, in the case when the metric does not select data efficiently, it is possible to use other metrics that can overlap each other's shortcomings.

All analyzed metrics, with the exception of the Jilb metric, generated several data sets for the critical path that was originally selected. It is noticeable that metrics for a small code of 130 lines with several code paths successfully select data in the first generation, which indicates a rather high convergence rate of the algorithm. In subsequent generations, various options are sequentially eliminated.

The conducted studies allow to propose a new method for generating test data based on the genetic algorithm, in which the fitness function will be formed not on the basis of one of the known metrics for assessing code complexity (as in this paper), but on the basis of a hybrid metric, which is a weighted sum of the indicators present in metrics considered in this paper. It also seems promising in terms of increasing the degree of code coverage by creating an effective mechanism for regulating (increasing and decreasing) the weights of operations in the fitness function while increasing the nesting level of the code section.

In the future, it is planned to expand ways to determine the complexity of the code. In addition to using metrics directly, it is planned to develop a method for taking into account indicators of the number of operations, functions, conditions and cycles with different weights. It is also possible to establish the degree of reduction or increase in the weights of operations at different levels of nesting. This will allow you to set the priority for input generation when certain requirements arise.

Acknowledgment

The reported study was funded by RFBR, project number 19-37-90156. The research is supported by Ministry of Science and Higher Education of Russian Federation (project No. FSUN-2020-0009)

References

[1] A.D. Richard and A.O. Jefferson, "Constraint-Based Automatic Test Data Generation," IEEE Transactions on Software Engineering", vol. 17, no. 9, pp. 900-910, 1991.

[2] P. Maragathavalli, M. Anusha, P. Geethamalini and S. Priyadharsini, "Automatic Test-Data Generation for Modified Condition. Decision Coverage Using Genetic Algorithm," International Journal of Engineering Science and Technology, vol. 3, no. 2, pp. 1311-1318, 2011.

[3] C. Meudec, "ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution," Software Testing Verification and Reliability, 2001.

[4] R. Gerlich, "Automatic Test Data Generation and Model Checking with CHR," 11th Workshop on Constraint Handling Rules, 2014.

[5] M.R. Girgis, "Automatic Test Data Generation for Data Flow Testing Using a Genetic Algorithm," Journal of Universal Computer Science, vol. 11, no. 6, pp. 898-915, 2005.

[6] E.J. Weyuker, "The complexity of data flow criteria for test data selection," Inf. Process. Lett., vol. 19, no. 2, pp. 103-109, 1984.

[7] A. Khamis, R. Bahgat and R. Abdelazi, "Automatic test data generation using data flow information," Dogus University Journal, vol. 2, pp. 140-153, 2011.

[8] S. Singla, D. Kumar, H. M. Rai and P. Singla, "A hybrid pso approach to automate test data generation for data flow coverage with dominance concepts," Journal of Advanced Science and Technology, vol. 37, pp. 15-26, 2011.

[9] Z. Liu, Z. Chen, C. Fang and Q. Shi, "Hybrid Test Data Generation," State Key Laboratory for Novel Software Technology, ICSE Companion Proceedings of the 36th International Conference on Software Engineering, pp. 630-631, 2014.

[10] M. Harman and P. McMinn, "Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search," IEEE Transactions on Software Engineering, vol. 36, no. 2, pp. 226-247, 2010.

[11] Y. Xing, Y. Gong, Y. Wang and X. Zhang, "Hybrid Intelligent Search Algorithm for Automatic Test Data Generation," Mathematical Problems in Engineering, 2015.

[12] C. Paduraru, and M.C. Melemciuc, "An Automatic Test Data Generation Tool using Machine Learning," 13th International Conference on Software Technologies (ICSOFT), pp. 472-481, 2018.

[13] M. Boussaa, O. Barais, G. Sunyé and B. Baudry, "Novelty Search Approach for Automatic Test Data Generation," 8th International Workshop on Search-Based Software Testing, 2015.

[14] M. Lopez, H. Ferreiro and L.M. Castro, "DSL for Web Services Automatic Test Data Generation," 25th International Symposium on Implementation and Application of Functional Languages, 2013.

[15] C. Doungsa-ard, K. Dahal, A.G. Hossain and T. Suwannasart, "An automatic test data generation from UML state diagram using genetic algorithm," IEEE Computer Society Press, pp. 47-52, 2007.

[16] S.Sabharwal, R. Sibal and C. Sharma, "Applying Genetic Algorithm for Prioritization of Test Case Scenarios Derived from UML Diagrams," IJCSI International Journal of Computer Science Issues, vol. 8, no. 3-2, 2011.

[17] C. Doungsa-ard, K. Dahal, A. Hossain and T. Suwannasart, "GA-based Automatic Test Data Generation for UML State Diagrams with Parallel Paths," Part of book Advanced design and manufacture to gain a competitive edge: New manufacturing techniques and their role in improving enterprise performance, pp. 147-156, 2008.

[18] M. Grochtmann and K. "Grimm, "Classification trees for partition testing. Software Testing," Verification and Reliability, vol. 3, no. 2, pp. 63-82, 1993.

[19] T.Y. Chen, P.L. Poon and T.H. Tse, "An integrated classification-tree methodology for test case generation," International Journal of Software Engineering and Knowledge Engineering, vol. 10, no. 6, pp. 647-679, 2000.

[20] A. Cain, T.Y. Chen, D. Grant, P.L. Poon, S.F. Tang and T.H. Tse, "An Automatic Test Data Generation System Based on the Integrated Classification-Tree Methodology," Software Engineering Research and Applications, Lecture Notes in Computer Science, vol. 3026, 2004.

[21] K.Serdyukov and T. Avdeenko, "Investigation of the genetic algorithm possibilities for retrieving relevant cases from big data in

the decision support systems," CEUR Workshop Proceedings, vol. 1903, pp. 36-41, 2017.

[22] R.S. Praveen and K. Tai-hoon, "Application of Genetic Algorithm in Software Testing," International Journal of Software Engineering and Its Applications, vol. 3, no. 4, pp. 87-96, 2009.

[23] W.M. Spears, "Crossover or mutation?" Foundations of Genetic Algorithms, vol. 2, pp. 221-237, 1993.

[24] H. Mühlenbein, "How genetic algorithms really work: Mutation and hillclimbing," Parallel Problem Solving from Nature, vol. 2, 1992.