The object of research is means to increase computational effectiveness for automatic unit test generation process. It provides arguments for developing new method to achieve wider use of symbolic execution in commercial software development. The main task of the research is to create adaptive code clustering method that considers test generation complexity for structural source code elements and available computational resources that will increase effectiveness of computations. It is achieved by estimating test generation complexity and balancing the it for produced clusters during clusterization. As a result, proposed clustering method is adaptive to hardware and source code variability. It is shown that developed approach provides up to 30 % increase in computation effectiveness compared to clustering based on code structural properties alone for selected samples and up to 250 % in separate cases. This is caused by balanced estimated test generation complexity within generated clusters. It limits path explosion to expected levels that match computational resources for every cluster. Estimate of test generation complexity makes it possible to stop the computation when the spent time exceeds the corresponding complexity limit. Consequently, it makes it possible to prevent performing unnecessary computations. Proposed method makes it possible to use symbolic execution in commercial software development due to higher adaptability for source code and hardware variations. It will allow to reduce expenses on early-stage software testing and provide means for determining feasibility of symbolic execution for commercial projects

Keywords: unit testing, effective computations, dynamic code analysis, static code analysis

UDC 004.41

DOI: 10.15587/1729-4061.2023.286160

IMPROVING THE EFFICIENCY OF SYMBOLIC EXECUTION BY CLUSTERING THE INPUT DATA BASED ON THE COMPLEXITY OF TEST GENERATION

Roman Bazylevych
Doctor of Technical Sciences, Professor*
Andrii Franko
Corresponding author
Postgraduate Student*
E-mail: andrii.v.franko@lpnu.ua
*Department of Software
Lviv Polytechnic National University
S. Bandery str., 12, Lviv, Ukraine, 79013

Received date 01.06.2023
Accepted date 18.08.2023
Published date 30.08.2023

***How to Cite**: Bazylevych, R., Franko, A. (2023). Improving the efficiency of symbolic execution by clustering the input data based on the complexity of test generation. Eastern-European Journal of Enterprise Technologies, 4 (2 (124)), 17–25. doi: https://doi.org/10.15587/1729-4061.2023.286160*

1. Introduction

Automated test generation is one of the key tasks for advancing the software development industry. Solving this problem is necessary for the further development of means of generating software code and increasing the efficiency of the early stages of testing and development of software products. Heuristic algorithms [1] and artificial intelligence tools [2] are used to generate tests. One of the methods of automated unit testing is symbolic execution [1]. This method interprets the program code that is the object of the test, manipulating the value of the variables marked as symbolic in order to achieve all possible states of the test program execution. This makes it possible to detect errors related to memory usage, division by zero, and others corresponding to an unexpected state of the software [1].

The main limitation of the symbolic execution method is the high computational complexity of the test generation process. To achieve 100 % coverage of the code with the generated tests, it is necessary to bypass all possible ways of executing the program. This leads to the problem of explosive growth in the number of paths. As a result, their number grows exponentially depending on the existing conditions and cycles, which leads to a corresponding increase in the size of the data for processing by means of symbolic execution. Increasing the length of the path leads to the complexity of SMT/SAT tasks for generating input data for the test. The presence of infinite loops adds to the problem of stopping test generation [3].

The increase in computational complexity when scaling the functionality of the software results in a corresponding increase in the costs of using this testing method. This limits the practical application of symbolic execution to individual modules and programs with a minimal amount of functionality. The time spent on generating tests for software with a significant amount of functionality is sometimes measured in hours [4].

Considering the need to improve the efficiency of calculations for the method of symbolic execution, research aimed at achieving this goal should be considered relevant.

2. Literature review and problem statement

Work [4] describes the current state of symbolic execution, namely, the features of its application and its results for the Klee tool. These results demonstrate that test generation requires significant computing time and resources. To solve this problem, static analysis methods are used in [5] to simplify the abstract syntactic tree and optimize predicates for logical expressions. This increases the efficiency of calculations, but the possibility of their parallel execution remains. In the course of research [6] and similar, the increase in efficiency from the use of parallel computing is significant and allows for a limited scaling of the method of symbolic

execution. However, given the non-linear growth in the number of operations, simple speed-ups due to the use of more powerful computing systems and parallel computing cannot compensate for the explosive growth in the number of paths.

In study [1], it is proposed to divide the code into segments when analyzing loops and external calls. It also suggests using the unit testing approach for selected segments. This approach solves the problem of the exploding number of paths for loops but not for conditional statements contained in the program code. Also, a separate analysis of each cycle can lead to the problem of stopping automated unit testing. To solve the stopping problem, [3] suggests using an incremental approach to the symbolic analysis of cycles. This is achieved by analyzing the variables and selecting only those paths that lead to changing the results of the symbolic execution of the loop, which makes it possible to reduce the number of paths and partially solve the stopping problem for loops. In study [7], it is proposed to combine paths to reduce their number, which leads to a partial solution to the problem of explosive growth in the number of paths but does not guarantee its solution in all cases. In order to limit the explosive growth of the number of paths depending on the cycles and conditions, an approach [8, 9] aimed at using clustering to divide the software into modules by structural characteristics is proposed at the same time. This helps limit the growth of the number of paths by the number of conditions and cycles that belong to the selected cluster (module). This approach divides the set of functions as structural units of the software code into clusters. Clustering is based on the relationship of a feature to other features and data outside its local scope. This makes it possible to dynamically change the sizes of clusters by changing the similarity threshold for combining functions into clusters [8]. The division into clusters [8] based on structural characteristics is not adaptive to the characteristics of a specific computer system and to the peculiarities of the source code but requires the determination of a similarity threshold for clustering. This leads to the selection of such a value by the method of expert evaluation, and not on the basis of data on the capabilities of the computer system and the characteristics of the input data. That, in turn, leads to:

1) overhead costs in the form of memory allocation and freeing, reading data from the disk, and performing initialization when testing individual structural units;

2) the calculation time will grow exponentially (due to the explosive growth of the number of paths), as a result of which they will be ineffective.

For each specific combination of computing resources and software code, a set of possible divisions into clusters can be proposed. Each of them will give different calculation efficiency. This will be expressed in tests covering a greater percentage of the statements of the program code per unit of time. To improve the efficiency of calculations, you should choose the best options from the available set. Manipulation of the clustering parameters will allow the calculation to be adapted to the hardware and the given software code for testing. The task arises to estimate the complexity of generating tests for a cluster, taking into account variations in input data (program code for testing) and properties of the computing system.

The ability to estimate the complexity of generating tests for a selected computer system will make it possible to more widely implement the method of symbolic execution in commercial software development by increasing its adaptability and computational efficiency. It will also provide the following benefits:

1) estimation of the execution time of full testing of the program code by the method of symbolic execution, which makes it possible to quickly assess the feasibility of conducting this type of testing for selected input data with available computing resources;

2) the ability to balance the complexity of generating tests for the formed clusters after dividing the input data, which will make it possible to quickly obtain test results.

All this allows us to state that it is appropriate to conduct a study aimed at improving the efficiency of calculations for automated unit testing due to clustering based on the assessment of the complexity of test generation.

3. The aim and objectives of the study

The purpose of this study is to improve the computational efficiency of the symbolic execution method by increasing its adaptability to variations in input data and computing systems. This will reduce costs for the software testing process. The adaptability of the symbolic execution method to input data will make it more widely used in commercial software development.

To achieve the goal, the following tasks were solved:

– to devise a method for assessing the complexity of generating tests for a given

software code on available computing resources, which will provide an opportunity to limit the explosive growth of the number of paths;

– to construct a method for clustering the software code, which will ensure a balanced complexity of generating tests for each cluster.

4. The study materials and methods

4. 1. Basic hypothesis, research assumptions, materials and methods

The object of research is methods for improving the efficiency of calculations for automated unit testing. The main hypothesis assumes an increase in the computational efficiency of the symbolic execution method due to the clustering of the input data, which will limit the explosive growth of the number of paths according to the hardware capabilities of each cluster. To this end, the method of assessing the complexity of test generation and clustering based on it will be used, which should provide appropriate results. To apply the proposed method, a dynamic analysis of test generation time for hardware using samples will be conducted. To test the hypothesis, computational experiments will be conducted to generate tests for selected samples of software code and a comparison of the efficiency of calculations will be performed.

Sample programs were designed to evaluate the influence of the type and number of C language operators on symbolic execution. They consist of one condition containing as a logical expression symbolic variables combined with an operator of the C language. The dependence of the time required for automated testing of the condition and the number of operators of the selected type is evaluated. Such an approximation must be performed for the symbolic execution tool and the hardware platform as a unique combination. It can only be

valid within the limits close to the range of samples used. This makes it possible to evaluate the impact of logical expressions as a component of conditions and cycles that affect the time of generating tests for given computing resources.

Samples are used for computational experiments in order to determine the relationship between execution time and test generation complexity estimated by the proposed method for selected hardware. They contain a loop with a given number of iterations and an appropriate termination condition (the number of iterations varies from 100 to 2000 in steps of 100) that produces the appropriate number of paths. This will allow the estimated complexity to be related to the hardware platform resources.

Samples from GitHub were used to evaluate the efficiency of calculations. 10 samples representing utilities (parsers, console games, compilers) and Git code are used. To perform computational experiments, the Klee (USA) symbol execution tool was chosen [4]. It generates tests for the input program code written in the C language, which has been converted to byte code [10]. Before performing clustering, the capabilities of the computer system are determined. To do this, tests are generated on reference samples and polynomial approximation of the time spent on each type of operator in the C language is performed. Based on the obtained data and the complexity assessment method, the relationship between complexity and execution time is formed for a given computer system. Based on this ratio, the complexity limit for future clusters is chosen. To determine the maximum complexity, the test generation complexity is first calculated for all the software code to be tested. Then, the time corresponding to the obtained complexity is determined. If the obtained score corresponds to the expected time limits of the test (determined by the expert), then the determined maximum difficulty is used. If the resulting estimate of execution time is too high, it is divided by two and a new maximum complexity is determined. Based on it, clustering is performed, and the new execution time is determined as the sum of the execution times for all obtained clusters. This step is repeated until either the expected execution time satisfies the expert, or the testing is deemed inadvisable. The following utilities *cflow* (building a call tree) and *pycparser* (building and manipulating an abstract syntactic tree) were used to divide the program code into clusters. We implemented software that, using the specified utilities, divides the program code into clusters according to structural characteristics, evaluates the complexity of generating tests and balances the complexity for existing clusters. The complexity assessment is based on data on the capabilities of the computing system and static analysis of the code (data on the number of cycles, conditions and operators). Separate files are created from the formed clusters for their symbolic execution. Each of the generated files contains only the functions belonging to the corresponding cluster. Tests are generated for them, the generation time and operator coverage metrics are measured, and the ratio of time to coverage is calculated. The program code of each sample is divided into clusters and tested by the Klee tool. The calculation was performed on an Intel-I5-11320H processor (USA).

4. 2. Evaluating the effectiveness of calculations

Evaluation of the results of test generation is possible according to the parameters of the number of found defects and coverage metrics. Comparing defects requires analysis of each, so such results cannot be used to evaluate computational efficiency. There are several metrics of code coverage [11]. Path and condition coverage metrics are not comparable when the number of elements in the cluster changes. Only operator coverage by tests can be compared for different clustering options. Operator coverage is compared as percentage coverage multiplied by the ratio of the number of operators within the cluster to the total number. Thus, it is possible to compare the results of generating tests for different clustering options. To evaluate the efficiency of calculations, it is suggested to evaluate the coverage of operators achieved per unit of time. Such an assessment will make it possible to compare the efficiency of calculations for different clustering options using one parameter.

5. Results of improving the efficiency of calculations due to the clustering of input data based on the complexity of generating tests

5. 1. Results of the development of a method for assessing the complexity of generating tests for structural units of the program code

The first task concerns the development of a method for estimating the complexity of symbolic execution for a cluster. Available metrics do not reflect the complexity of generating tests [12]. The new complexity metric should reflect the number of operations that the symbolic execution tool would need to generate tests based on the sample code. This includes the operations necessary to cover all available paths and the selection of input data to satisfy the relevant conditions. The condition can be characterized by the number of operators and the size of the operands. For standard operators, operand size can be neglected because there are no operators in a typical programming language that perform a function on an array of data at once. Therefore, to evaluate the complexity, it is necessary to determine the number of paths and the number of operators.

To obtain the necessary information, it is suggested to perform an analysis of the instructions for each element (function, as a structural unit of the program code) in the cluster. This is achieved by bypassing the AST (Abstract Syntax Tree). It contains information about each condition, cycle, and the parameters used in them. The complexity of generating tests for a function that is an element of a cluster should be the sum of the complexity of all high-level conditions/loops (at the first level in its AST). The complexity of the node can be calculated using formula (1):

$$C = \sum_{i=1}^{n} c_i. \quad (1)$$

In formula (1), c_i is the estimated complexity of the high-level cycle/condition belonging to the node, n is the number of conditions and cycles at the top level of the abstract syntax tree of the function. The explosive growth in the number of paths is taken into account when calculating c_i for each cycle/high-level condition. Formula (1) also describes the complexity estimation for nested conditions/loops and can be applied recursively.

It is assumed that items with higher test generation complexity would require more CPU time to be processed by the symbolic execution tool. The number of paths to test increases with each additional condition or cycle contained in the cluster. A condition can be independent (when the condition depends on a unique set of variables) or dependent (when at

least one variable is used in more than one condition). Each additional independent condition multiplies the number of paths by two. Each dependent multiplies by a number from two to one. This is a consequence of the fact that some conditions may be mutually exclusive. For example, if one condition out of five is mutually exclusive to two conditions and does not apply to the other two, then the number of paths would be 24 instead of 32. Parsing and analyzing dependences between conditions requires symbolic execution. To avoid the recursive problem, the worst-case number of paths should be used (all conditions are independent of each other). As a result, the actual number of paths for which it is necessary to generate tests may be less than the estimated number.

A loop in conditional similarity increases the number of paths by at least two times. The first when the loop body was executed once and the second when the loop body was skipped. Loops increase the number of paths similarly to conditions, but the range of the multiplier can be from 1 to infinity, due to the large number of iterations. There can be a definite loop that always executes a fixed number of iterations, and an indefinite loop that depends on the symbolic value (potentially an infinite number of paths). Loops contribute to the increasing complexity of test generation more than conditions. If the number of iterations is undefined, a stalling problem occurs. To combat the deadlock problem, a limit on the number of paths through the loop body is proposed. The limit prevents long or infinite symbolic execution of the loop body at the expense of detecting potential errors. This happens due to the analysis of a limited number of paths, which does not affect the code coverage indicators. A symbolic execution of the loop could potentially detect only a few defects but process thousands of paths, resulting in corresponding time costs. The constraint introduced will allow the symbolic execution to analyze alternative paths. This will increase the code coverage metric of the tests and potentially increase the number of detected defects. Limiting the number of paths for the loop body must be defined by the user as a parameter of the test generation algorithm.

A final factor affecting the complexity of test generation is the statements and operands used in conditions and loops to control the flow of execution. A set of operators and conditions creates a SAT/SMT problem [4]. By traditional means, it is impossible to predict the execution time of heuristic algorithms for solving the above-mentioned problems. It is proposed to approximate with a polynomial function the test generation time for the condition containing the selected type of operator. Given the large number of possible combinations of operators, it is only possible to estimate the increase in execution time for their type. The proposal is based on the following assumptions:

1) one condition or loop usually uses only a small number of statements;

2) cases where SMT or SAT problems present in the code are insoluble for heuristics are rare;

3) clustering will limit the size of SMT and SAT tasks;

4) some operators are more complex for symbolic execution than others;

5) several nested conditions can be represented as a set of logical expressions, which are combined by the logical operator &&.

The above statements show that a practical and simple way to account for the impact of statements on the generation time of unit tests is to estimate the execution time for each statement type based on the number of times they are repeated in a condition. This model will ignore potential combinations of operators whose running time growth will differ from the approximated polynomials for each.

It is proposed to estimate the complexity of the condition/cycle according to the following formula:

$$c=\sum_{j=1}^{n} k_j\left(r_j\right)\cdot\left(2^l+a\cdot s\right). \quad (2)$$

In it, r represents the number of operators of type j. The function k represents an approximate polynomial that describes the expected time for the symbolic execution of an operator of type j, l is the number of conditions, a is the number of cycles, s is the limit of iterations per loop body. Such an estimate of the complexity of generating tests involves the product of the maximum amount of time for generating input data for the most difficult path by the maximum possible number of paths for a cluster element. This approach aims to identify and separate potential sites with high complexity. The complexity estimate is not the same as the execution time estimate. It cannot detect infinite loops or unsolvable SAT/SMT problems.

However, with each additional operator in the boolean expression and each additional condition/loop within the cluster, they are more likely to occur.

5. 2. Results of clustering method development based on estimation of complexity of generating tests

The previously proposed clustering algorithm [8] should be improved as follows:

1) grouping by location. This means that close clusters should be merged to reduce the number of clusters and to avoid clusters from individual elements;

2) division and exchange of elements between clusters to achieve balanced test generation complexity for each cluster.

The first proposed improvement requires tracking the structural relationships between cluster elements. A cluster with one element should be included in another cluster if it is structurally related to it. If it is connected to several clusters at the same time, the one with the largest number of structural connections is chosen. If it is the same, the selection is made randomly. This is shown schematically in Fig. 1. This is necessary to reduce the number of clusters and simplify the further process of their balancing.

Calculating the complexity of generating tests for each element of a cluster makes it possible to exchange elements between clusters for balancing [13]. For this purpose, it is proposed to apply the following approach:

1. Set limits on the complexity of generating tests for the cluster. Calculate it for each of the available clusters.

2. If the limit is exceeded, transfer the element for which the complexity of generating tests is the highest within the cluster to one of the neighboring clusters. The cluster with the largest number of connections to the required element will be selected. If the estimated test generation complexity exceeds the limit, the cluster with the next highest number of connections will be selected. This is shown in Fig. 2.

3. If there is no cluster that can accept the element, then it is separated to a new cluster. This is shown in Fig. 3.

4. The procedure of transferring elements to neighbors and dividing clusters is repeated iteratively.

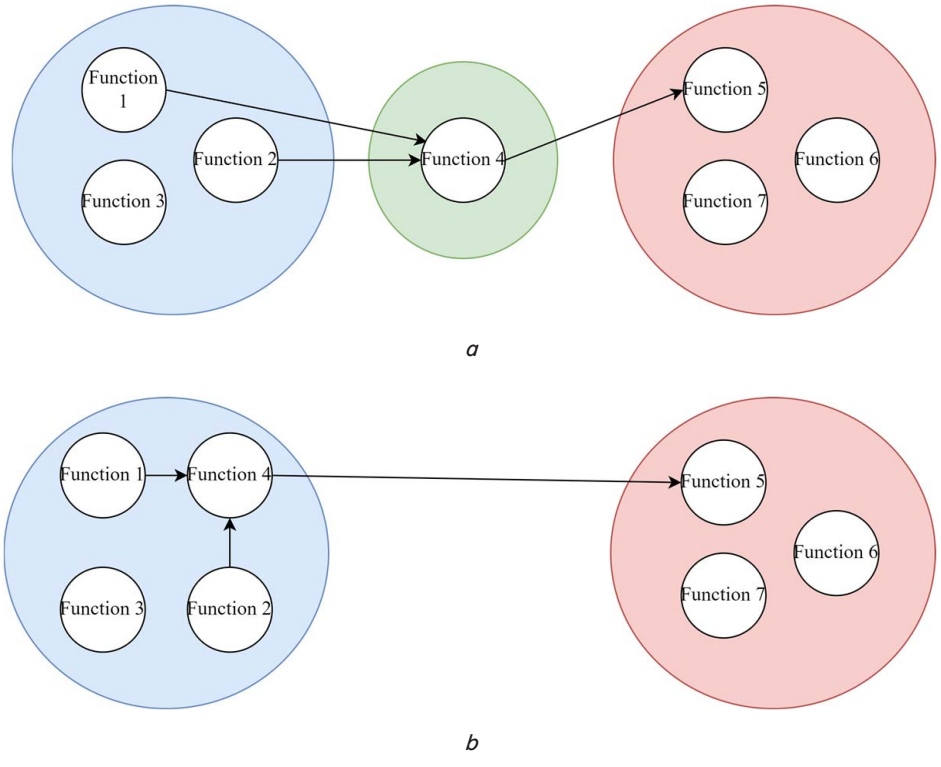

Fig. 1. Inclusion of a cluster containing one element into the cluster with the largest number of common structural connections: *a* — a cluster of one element exists; *b* — clusters are combined

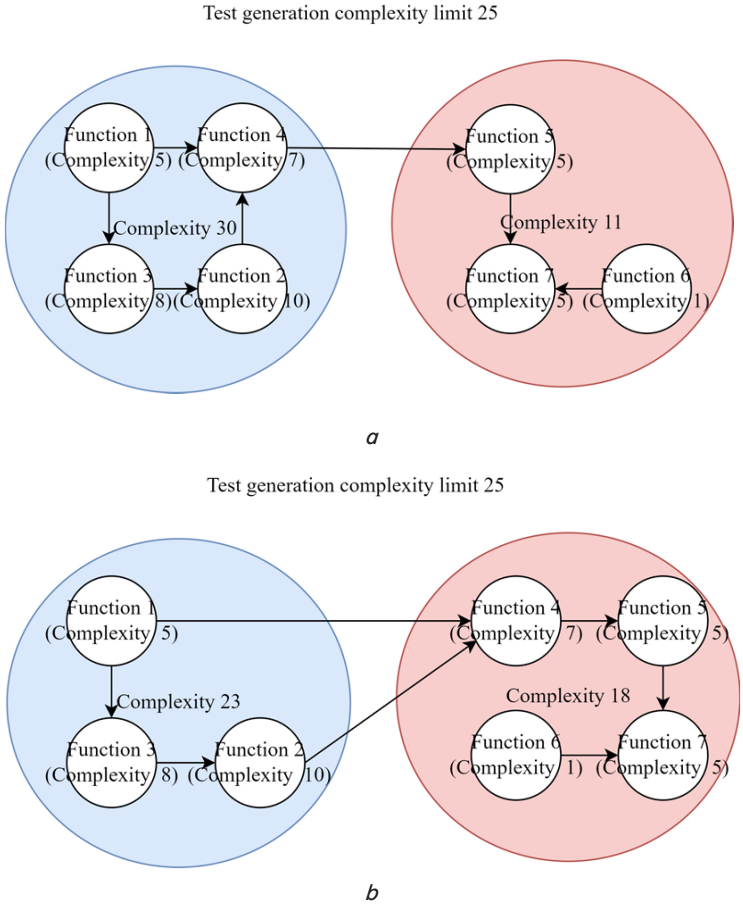

Fig. 2. Balancing the complexity of generating tests between two clusters: *a* — the complexity of generating tests is not balanced; *b* — the difficulty is balanced

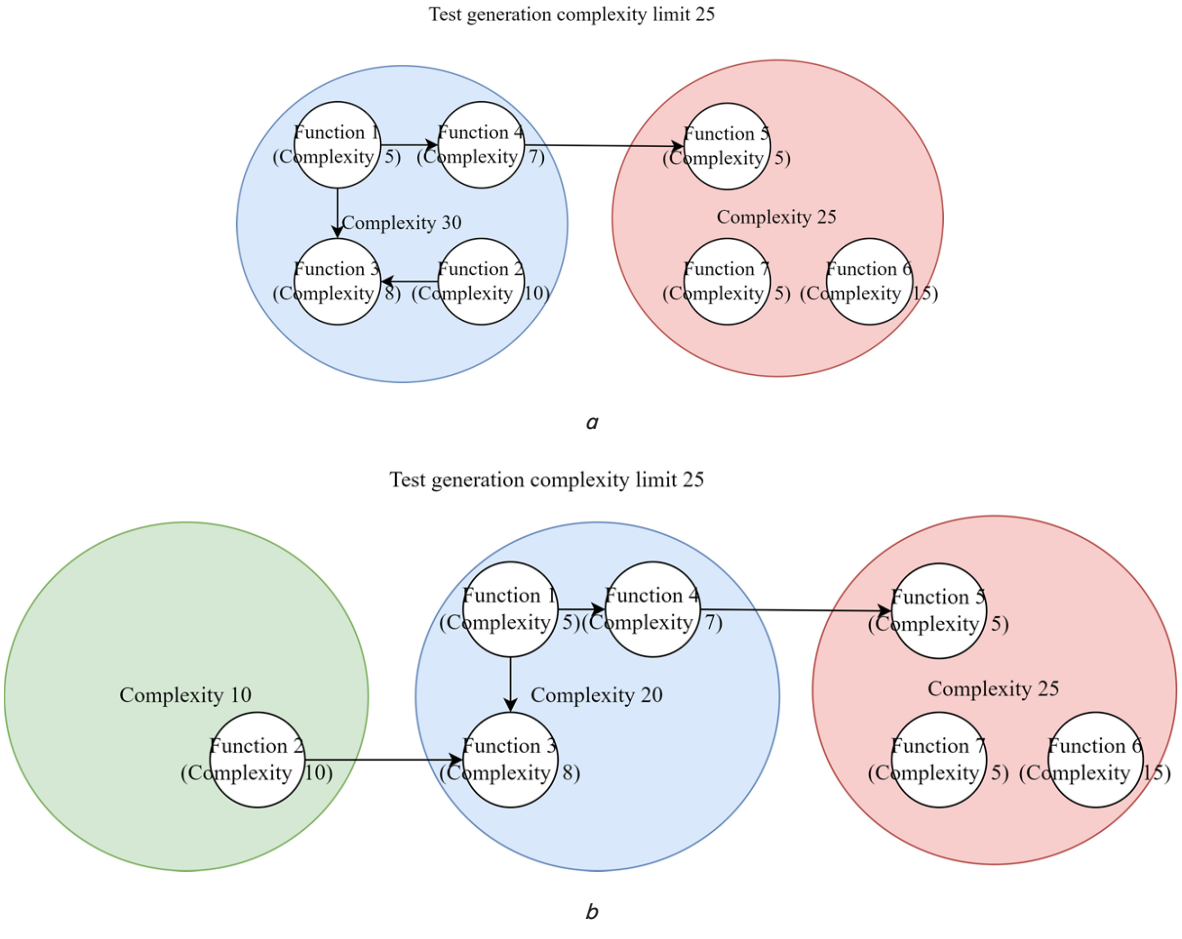

Fig. 3. Balancing the complexity of generating tests by creating a new cluster: *a* — the complexity of generating tests is not balanced; *b* — the complexity is balanced by the formation of a new cluster

The results of the assessment of the influence of the type and number of C language operators on the symbolic execution time are given in Table 1. According to the proposed methodology, operators were divided into equivalent groups. For each of them, an approximation of the execution time was carried out depending on the number of operators under the condition by way of dynamic analysis of the results of generating tests using Klee.

Table 1

C language operators and polynomials approximating them for Klee symbolic execution tool, Intell5-11300H processor in the range from 0 to 100 operators per condition

Operator	Type of polynomial	Polynomial
+, –,	Polynomial of the first power	$0.0012*x+0.015$
*, /, <<, >>	Polynomial of the first power	$0.016*x+0.037$
&, |, ||, &&, <, >, ==, !=, <=, >=	Polynomial of the second power	$0.00004*x^2+0.00012*x+0.004$
*,[],–>	Polynomial of the second power	$-0.1686*x^2+13.558*x+6.2909$
%	Polynomial of the second power	$0.851*x^2-1.85*x-7.71$

Symbolic execution for memory access operators can take a significant amount of time. This occurs when the variable used for indexing depends on character data. In this case, it is suggested to use a polynomial obtained for a similar case in the absence of dependence of the index on character data. The resulting score will limit the complexity and execution time for the cluster. This will solve the problem of stopping test generation for this case.

There is a task to determine the maximum complexity for the cluster. It depends on the characteristics of the computing system and the input data in the form of the software code under test. The complexity limit will be unique to the combination of software and hardware tested. By symbolically executing the loop with a variable number of iterations, the relationship between execution time and complexity is established. As a result, each cluster can have a time limit for execution. This will help solve the problem of «freeze» in symbolic execution.

Fig. 4 shows the relationship between test generation time and estimated test generation complexity when changing the number of paths in the program code. This parameter must be entered into the algorithm manually. According to the calculated ratio, the maximum complexity should be chosen based on the relationship with the symbolic execution time. It is proposed to be determined based on the capabilities of the computer system and the needs of the user. Complexity increases non-linearly from formula (2) through the use of conditions/loops. The measured execution time increases similarly. The maximum complexity is chosen to limit the non-linear growth of execution time. The maximum execution time for a cluster is defined as a constant by the user of symbol-

ic execution. Then the corresponding difficulty is chosen as the maximum difficulty.

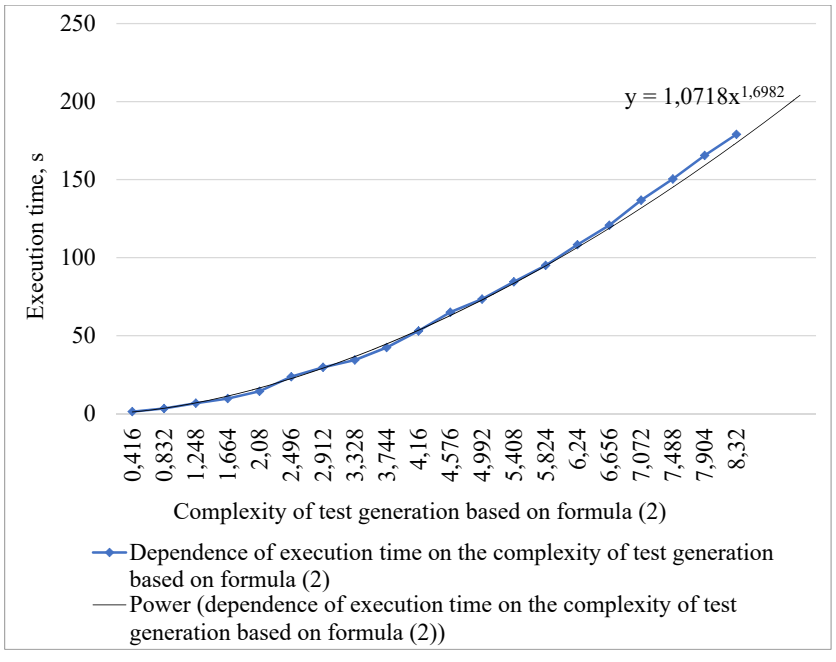

Fig. 4. Dependence between the time of generating tests by the Klee tool and the complexity of their generation and its approximation by a power function for the Intell5-11300H processor

The results of code execution with clustering by complexity in comparison with the previously proposed clustering method and standard symbolic execution are given in Table 2 below. The value corresponding to 21 minutes of execution time was chosen as the maximum difficulty. This value was obtained by expert evaluation. For the samples used, taking into account the number of clusters from 7 to 20, the maximum expected execution time would be from 147 to 420 minutes. This calculation time will allow the research to be performed, so the chosen value is acceptable.

Table 2

Comparison of the efficiency of calculations for generating tests using the Klee tool for the Intell5-11300H processor when using the proposed clustering method and other methods

Program code	Results of implementation	
	Method	Code coverage generated per minute
Git tool program code	Standard symbolic execution [4]	0.013 %
	Clustering by structural code elements [7]	0.018 %
	Clustering by structural code elements to balance the complexity of generating tests for clusters	0.023 %
Average for 10 code samples	Standard symbolic execution [4]	0.69 %
	Clustering by structural code elements [7]	1.24 %
	Clustering by structural code elements to balance the complexity of generating tests for clusters	1.64 %

The proposed methods make it possible to perform calculations 2.5 times more efficiently than standard symbolic execution, and 30 % more efficient than symbolic execution with code clustering by properties.

6. Discussion of results of clustering input data based on the assessment of the complexity of generating tests

Our results are explained by the application of the devised method for assessing the complexity of test generation and the clustering method based on it. Complexity assessment makes it possible to perform clustering of input data taking into account the features of the computing system and software code. This is shown in Fig. 4. There, the relationship is established between the complexity estimated by formula (2) using the data in Table 1 and execution time. The relationship is characterized by a power-law function, which allows estimation of execution time based on complexity for a given hardware platform. For each hardware platform combination, such calculations will be unique and may be performed using samples. This makes it possible to use the developed clustering method, which will be adaptive to computing resources and input data. The proposed clustering method in Fig. 1–3 has made it possible to improve the efficiency of calculations by 30–250 % compared to [8] and the usual symbolic execution.

As a result of evaluating the complexity of test generation, the problem of stalling for symbolic execution can be solved. The established correlation between execution time and test generation complexity for a combination of hardware platform and symbolic execution engine allows us to limit the explosive growth of the number of paths within clusters according to the input data. Thus, it is possible to set a time limit for the symbolic execution of each cluster, which will correspond to the features of the software code included in it. From the obtained results, it can be seen that in a significant number of cases, adaptability makes it possible to improve the efficiency of calculations for various input data. This makes it possible to reduce the overhead of calculations and avoid the increase in the calculation time of the exponential function.

In comparison with [8, 9], where only structural connections are taken into account, the complexity of generating tests for the distribution of elements between clusters is used. In comparison with studies [5, 6], a different approach is used to reduce the number of operations, and the proposed clustering makes it possible to limit the number of paths for analysis within the cluster. Compared to study [1], the selected segments refer to the entire program code, not only cycles, and the division is adaptive to the capabilities of the computer system. In comparison with 3, the stopping problem can be solved not only for loops but also for difficult to analyze logical expressions by means of symbolic execution.

In general, unlike the considered approaches, our study offers an adaptive approach to the problem of explosive growth in the number of paths, which uses data about the computing system and takes into account the peculiarities of the input data. The developed methods make it possible to improve the efficiency of calculations for symbolic execution tools, taking into account possible software variations. The devised method of assessing the complexity makes it possible to estimate how long it takes to generate tests and perform clustering of input data on this basis, which in turn increases the efficiency of calculations.

The use of the proposed methods has limitations. Options such as the maximum test generation complexity for the cluster make it possible to choose between quickly checking the code with symbolic execution tools and analyzing more paths to increase the probability of detecting defects. However, lowering the test generation complexity limit could lead to detection of states that would be impossible during program execution and marking them as defects. Computational performance improvements may differ from actual results due to the limited number of samples for hypothesis testing and the high variability of input data and computing systems. In some cases, symbolic execution of the entire program code, or using clustering and segmentation methods that are not adaptive to the peculiarities of the input data, can give better results. However, for widespread use in the commercial development of software products, adaptability is necessary.

The study has flaws. In particular, the use of computing resources for tasks other than test generation can affect the dynamic analysis of the system's capabilities and lead to an incorrect estimate of the complexity. The use of operator samples is designed for one type of operator only, so there may be deviations when evaluating a combination of operators based on this data.

The application of the methods reported in [3] will make it possible to improve the estimation of the complexity of generating tests for loops in the future. The use of parallel computing methods will make it possible to scale the approach. Improving the accuracy of runtime estimation based on the complexity of test generation and running computational experiments on more hardware platforms will improve the method.

7. Conclusions

1. The methods of dividing input data of symbolic execution and their preliminary processing were analyzed. It was determined that the use of static and dynamic analysis for this could improve the efficiency of symbolic execution. To improve the efficiency of calculations, it is proposed to use clustering of the software code based on the assessment of the complexity of generating tests. The proposed method is an improvement of the previously described clustering method for symbolic execution.

2. A method has been devised for evaluating the complexity of generating tests for software code by analyzing its instructions. We used static analysis of conditions, instructions, and loops along with dynamic analysis of test generation times for reference code samples. The maximum complexity is determined based on the correlation between execution time and complexity determined by dynamic analysis for a given hardware platform. This allows balancing the complexity of generating tests for selected modules. Symbolic execution of complexity-balanced clusters generates 20 % more code coverage per minute of execution than without balancing, with up to a 45 % efficiency gain compared to computing without input splitting.

Conflicts of interest

The authors declare that they have no conflicts of interest in relation to the current study, including financial, personal, authorship, or any other, that could affect the study and the results reported in this paper.

Funding

The study was conducted without financial support.

Data availability

The data will be provided upon reasonable request.

References

1. Le, W. (2013). Segmented symbolic analysis. 2013 35th International Conference on Software Engineering (ICSE). doi: https://doi.org/10.1109/icse.2013.6606567
2. Shen, S., Shinde, S., Ramesh, S., Roychoudhury, A., Saxena, P. (2019). Neuro-Symbolic Execution: Augmenting Symbolic Execution with Neural Constraints. Proceedings 2019 Network and Distributed System Security Symposium. doi: https://doi.org/10.14722/ndss.2019.23530
3. Yi, Q., Yang, G. (2022). Feedback-Driven Incremental Symbolic Execution. 2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE). doi: https://doi.org/10.1109/issre55969.2022.00055
4. Cadar, C., Nowack, M. (2020). KLEE symbolic execution engine in 2019. International Journal on Software Tools for Technology Transfer, 23 (6), 867–870. doi: https://doi.org/10.1007/s10009-020-00570-3
5. Vishnyakov, A., Fedotov, A., Kuts, D., Novikov, A., Parygina, D., Kobrin, E. et al. (2020). Sydr: Cutting Edge Dynamic Symbolic Execution. 2020 Ivannikov Ispras Open Conference (ISPRAS). doi: https://doi.org/10.1109/ispras51486.2020.00014
6. Singh, S., Khurshid, S. (2020). Parallel Chopped Symbolic Execution. Lecture Notes in Computer Science, 107–125. doi: https://doi.org/10.1007/978-3-030-63406-3_7
7. Păsăreanu, C. S., Kersten, R., Luckow, K., Phan, Q.-S. (2019). Symbolic Execution and Recent Applications to Worst-Case Execution, Load Testing, and Security Analysis. Advances in Computers, 289–314. doi: https://doi.org/10.1016/bs.adcom.2018.10.004

8. Bazylevych, R. P., FrankoA. V. (2022). Hierarchical model of automated test generation system. Scientific Bulletin of UNFU, 32 (4), 77–83. doi: https://doi.org/10.36930/40320412
9. Bazylevych, R., Franko, A. (2022). Input decomposition by clusterization for symbolic execution. 2022 IEEE 17th International Conference on Computer Sciences and Information Technologies (CSIT). doi: https://doi.org/10.1109/csit56902.2022.10000433
10. Poeplau, S., Francillon, A. (2019). Systematic comparison of symbolic execution systems. Proceedings of the 35th Annual Computer Security Applications Conference. doi: https://doi.org/10.1145/3359789.3359796
11. Mues, M., Howar, F. (2022). GDart: An Ensemble of Tools for Dynamic Symbolic Execution on the Java Virtual Machine (Competition Contribution). Lecture Notes in Computer Science, 435–439. doi: https://doi.org/10.1007/978-3-030-99527-0_27
12. Peitek, N., Apel, S., Parnin, C., Brechmann, A., Siegmund, J. (2021). Program Comprehension and Code Complexity Metrics: A Replication Package of an fMRI Study. 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). doi: https://doi.org/10.1109/icse-companion52605.2021.00071
13. Bazylevych, R., Palasinski, M., Bazylevych, L., Yanush, D. (2013). Partitioning optimization by iterative reassignment of the hierarchically built clusters with border elements. 2013 2nd Mediterranean Conference on Embedded Computing (MECO). doi: https://doi.org/10.1109/meco.2013.6601362