ECONTECHMOD. AN INTERNATIONAL QUARTERLY JOURNAL – 2014. Vol. 3. No. 2. 15–20

Variables state-based software usage model

D. Fedasyuk, V. Yakovyna, P. Serdyuk, O. Nytrebych

Software department, Lviv Polytechnic National University;
79013, Lviv, Bandery st. 28
e-mail: pavlo.serdyuk@gmail.com

Received May 15.2014: accepted June 20.2014

Abstract. This article describes a new mathematical software usage model, which includes the effect of the set of global and external variables values for further analysis of multi-test scenarios to improve the effectiveness of the testing software. This model is represented as a graph of transitions and a set of variables with respective sets of equivalence classes. The proposed approach is particularly relevant for computational algorithms with complex logic.

Key words: software testing process, usage model, test scenario, software reliability.

INTRODUCTION

In spite of continued research efforts in the testing and software reliability assessing fields developers have not received an effective tool for software reliability prediction before its implementation. A lot of models are proposed, but versatility has not been proven for any of them; moreover, as a rule, even a slight deviation from the typical values of the parameters for these models can lead to incorrect calculations due to a number of assumptions which are made during the modeling [1].

The main tool for software reliability analysis and improvements is its testing [2-4]. However, despite the large number of high-quality software testing techniques, even large-scale testing is unable to detect all the failures in the project, which results in significant growth of expenses on software quality assurance process with growth of the software size.

Today a series of issues that do not allow to predict software reliability accurately remain unsolved. For example, the decomposition of the software in the analysis of the characteristics of its reliability [5-6]; until now there is no clear definition of dependency of software reliability on the characteristics of the code. Also, many authors [7-9] indicate a misuse of size and complexity software metrics (Halstead, McCabe metrics) as the sole sources for the software reliability prediction in statistical models [10] for these models ignore the causal effects of human and other factors on defects in the software, because too big complexity of applications may be caused by various circumstances, including programmer's little experience in this field, etc.

Today a series of issues that do not allow predicting software reliability accurately remain unsolved. For example, the software decomposition in the analysis of the reliability characteristics [5-6]; there is no clear definition of software reliability depending on the characteristics of the code. Also, many authors [7-9] indicate a misuse of size and complexity software metrics (Halstead, McCabe metrics) as the sole source for the software reliability prediction in statistical models [10], because these models ignore the causal effects of human and other factors on defects in the software, because too much complexity applications may be caused by various circumstances, for example, programmer's little experience in this field, etc.

Software reliability assessment using code coverage metrics [11, 12] during software testing is currently the most common tool in practice. But in case when the test scenarios do not cover all the software functionality, then, despite the fact that the software will pass all tests, its reliability may be still low. Also, during the software development it is very difficult to keep track of the failures that occur as a result of multiple interrelated execution steps, because the number of such scenarios grows exponentially depending on the number of steps and the variability of their parameters. This type of relationship occurs through the use of external static or dynamic variables that are used in several steps of the software scenario. Software testing often does not cover such scenarios as the time allotted for the test is limited. Failures of this type are very expensive to correct, as they often are tracked at the regression testing stage, software deployment or support. According to research at Cambridge University, the yearly cost of bugs that occur in production software is about 312 billion dollars [13]. Also, researchers have shown dependency diagram (Fig. 1) for bugs' correction cost on different stages of the software development life cycle [14]. So it is clear that it is important to keep track of such scenarios at the software development stage, which will reduce the financial and human resources required.

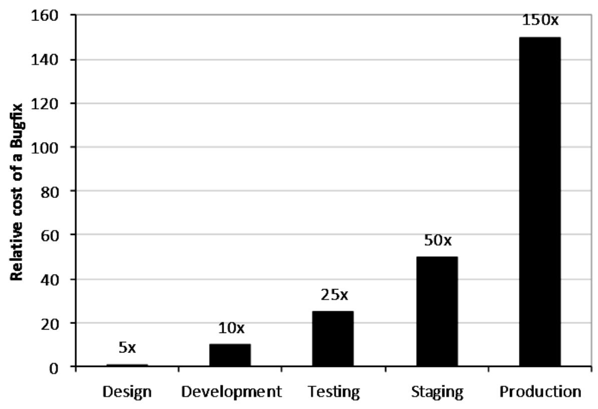


Fig. 1. Cost of fixing bugs at various stages of software delivery

In this research, we emphasize that the considered failures arise from long scenarios consist of several steps. And for them the most important aspect is usage of external variables and variables that are passed between components (as method arguments or other way), rather than the methods internal logic.

In addition, the importance of software variables usage consideration in test scenarios building process follows from the fact that many software reliability analysis models use metrics, which take into account the use of software variables. For example, Bieman J. and Kang B-K. offered class connectivity metrics that are based on direct and indirect connections between pairs of methods [15], which are implemented using the same variables. L. Ott and B. Mehra developed a class partitioning model [16], based on the variables of class instance. In addition, as it is known, complexity software metrics are divided into three main groups [17]: software size metrics; complexity metrics of software control flow; complexity metrics of software data flow. Metrics of the third group are based on an assessment of the use, configuration and data placement in the program; in particular this applies to global variables. This group includes Chapin metric [18], the essence of which is to evaluate a single program module by analyzing the usage nature of variables from I/O list; McClure complexity metric [19] based on the number of possible software implementation paths, the number of control structures and variables; metric of references to global variables [19].

Thus, the development of new methods and algorithms for software reliability assessment and testing is relevant and notable scientific task. Another important task during testing and software reliability estimation is taking into account the variables and their properties, but unfortunately, there is no model nowadays, which could investigate the effect of variables on the software reliability.

Software usage model, based on the analysis of software variables, is described in this paper. It is the basis of the algorithm for automated test scenarios generation that will improve the efficiency of software testing and investigate the influence of the characteristics of software on its reliability.

SOFTWARE USAGE MODEL BASED ON ITS VARIABLES

For building long software testing scenarios, which contribute significantly to the software cost, software usage model was developed. According to this, model considers failures that occur in later software development stages: regression testing, alpha and beta testing, software deployment. This is a consequence of a fact that such failures are usually caused by complicated scenarios, where non-standard initialization of certain variables, objects or components happens on the first steps and actual their usage takes place later on the following steps. In addition, these failures occur only while using a limited subset of variables values.

These scenarios are difficult to cover by the automated or manual testing due to the fact that each stage of a complex scenario can have a number of degrees of freedom, i.e. the possible subsets of the variables values set it uses. The corresponding number of all scenarios is the product of all degrees of freedom of each stage, which can be quite a large number. That is why some scenarios with certain subsets of variables values cannot be reproduced using the standard test scenarios.

The execution of such complex scenarios largely depends on the set of variables values that it uses. Variables in the software can be divided arbitrarily into 3 types:

1. Local – variables that are declared and used only within the method.
2. Global – public variables declared in the class or globally and can be changed by various.
3. Methods arguments – variables that are passed into the method, can be changed by method and transferred into other methods.

Failures, associated with the first type of variables, are not interesting for consideration because they do not affect the occurrence of failures in other methods. Other types of variables should be considered only during detailed multi-level methods analysis.

Each variable V^i is characterized by the number of equivalence classes E_j^i [20]. Equivalence classes are distinguished by choosing each requirement from software specification and splitting it into two or more groups. It should be noted that there are two types of equivalence classes: the correct equivalence classes representing the correct input values for variable and incorrect equivalence classes corresponding to all other possible states of the environment (i.e. wrong input values). This way one of the principles of software testing, need to focus on the wrong conditions, is being followed. For example, if the variables value according to specification is "an integer from 0 to 999 ", then there is one correct equivalence class (0 to 999) and two incorrect (values less than 0 and values bigger than 999).

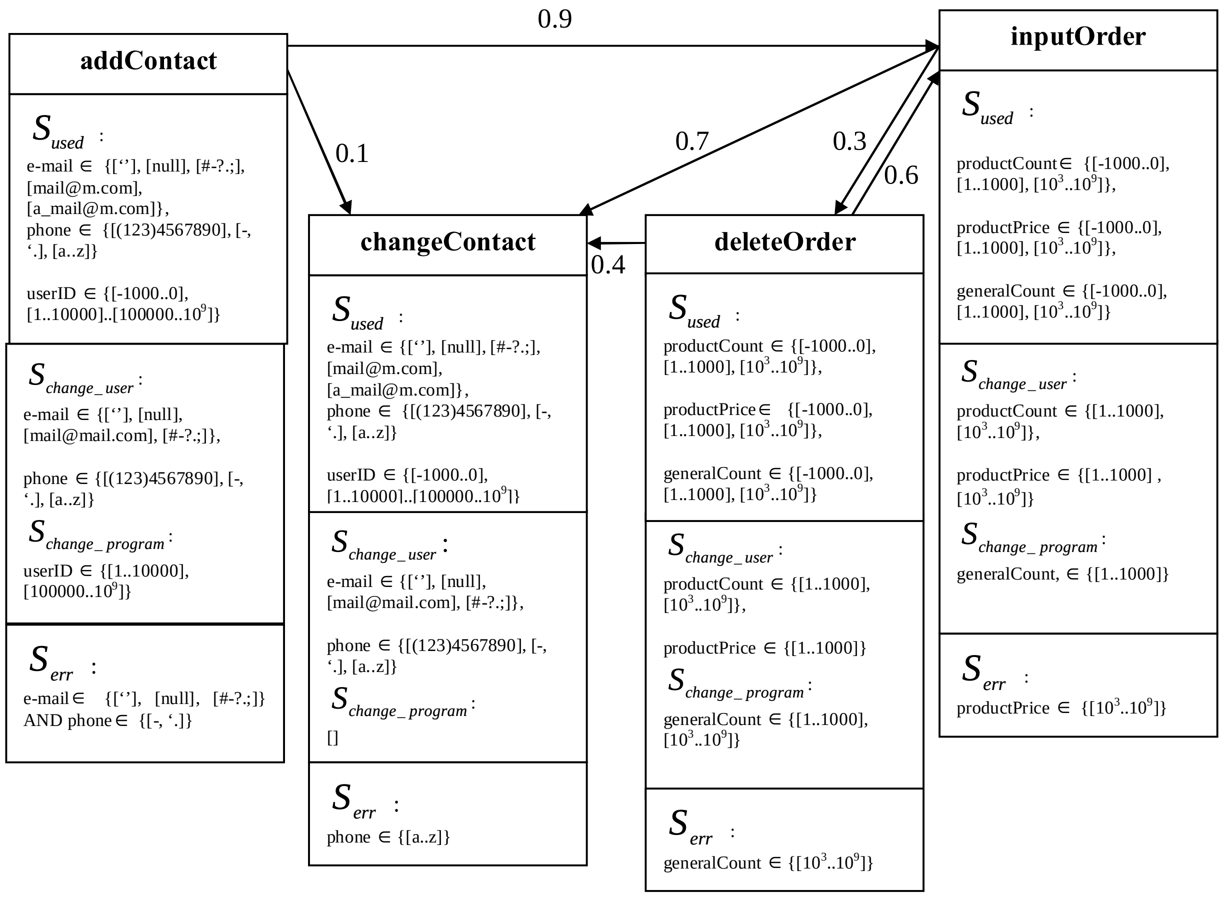
The software is represented in the form of a directed graph $G=\{S, P\}$, where S – set of software methods, P – set of transitions between the respective methods. An example of such a representation is shown in Fig. 3.

The process of software execution can be modeled by the passage paths of the graph, each node of which can be presented as a method that changes the value of the set of variables values (method arguments as well as global variables), and the edges will be responsible for the sequence of method call.

Each method S^i, that would fit the node has the following properties:

1. List of variables used by the method – S^i_{used}.
2. List of variables changed by the method S^i_{change}, and appropriate change probability. This set of variables is the union of the set of variables that can be changed by the user $S^i_{change\_user}$ (variables, tested with “black box” testing), and also variables that can be changed due to internal logic programs $S^i_{change\_program}$ (variables that can be tested only with “gray box” or “white box” testing).
3. List of variables and corresponding incorrect equivalence classes S^i_{err} that can cause failure in this method.
4. List of incident arcs Pi, which in turn contain the transition probability p_{ij} to other method S^j.

Let’s considering an example: given software that consists of the following methods: addContact, changeContact, inputOrder, deleteOrder. Software usage model, which is represented by a graph, is shown in Fig. 2. Nodes correspond to methods of application, and the edges weights reflect the transition probabilities between methods.



0.9
0.1
0.7
0.3
0.6
0.4
addContact
S_{used} :
e-mail ∈ {[‘’], [null], [#-?.;], [mail@m.com], [a_mail@m.com]},
phone ∈ {[(123)4567890], [-, ‘.], [a..z]}
userID ∈ {[-1000..0], [1..10000]..[100000..10^9]}
S_{change_user}:
e-mail ∈ {[‘’], [null], [mail@mail.com], [#-?.;]},
phone ∈ {[(123)4567890], [-, ‘.], [a..z]}
$S_{change\_program}$:
userID ∈ {[1..10000], [100000..10^9]}
S_{err} :
e-mail ∈ {[‘’], [null], [#-?.;]} AND phone ∈ {[-, ‘.]}
changeContact
S_{used} :
e-mail ∈ {[‘’], [null], [#-?.;], [mail@m.com], [a_mail@m.com]},
phone ∈ {[(123)4567890], [-, ‘.], [a..z]}
userID ∈ {[-1000..0], [1..10000]..[100000..10^9]}
S_{change_user} :
e-mail ∈ {[‘’], [null], [mail@mail.com], [#-?.;]},
phone ∈ {[(123)4567890], [-, ‘.], [a..z]}
$S_{change\_program}$:
[]
S_{err} :
phone ∈ {[a..z]}
deleteOrder
S_{used} :
productCount ∈ {[-1000..0], [1..1000], [10^3..10^9]},
productPrice ∈ {[-1000..0], [1..1000], [10^3..10^9]},
generalCount ∈ {[-1000..0], [1..1000], [10^3..10^9]}
S_{change_user}:
productCount ∈ {[1..1000], [10^3..10^9]},
productPrice ∈ {[1..1000]}
$S_{change\_program}$:
generalCount ∈ {[1..1000], [10^3..10^9]}
S_{err} :
generalCount ∈ {[10^3..10^9]}
inputOrder
S_{used} :
productCount ∈ {[-1000..0], [1..1000], [10^3..10^9]},
productPrice ∈ {[-1000..0], [1..1000], [10^3..10^9]},
generalCount ∈ {[-1000..0], [1..1000], [10^3..10^9]}
S_{change_user}:
productCount ∈ {[1..1000], [10^3..10^9]},
productPrice ∈ {[1..1000] , [10^3..10^9]}
$S_{change\_program}$:
generalCount, ∈ {[1..1000]}
S_{err} :
productPrice ∈ {[10^3..10^9]}

Fig. 2. Example of graph of software usage model

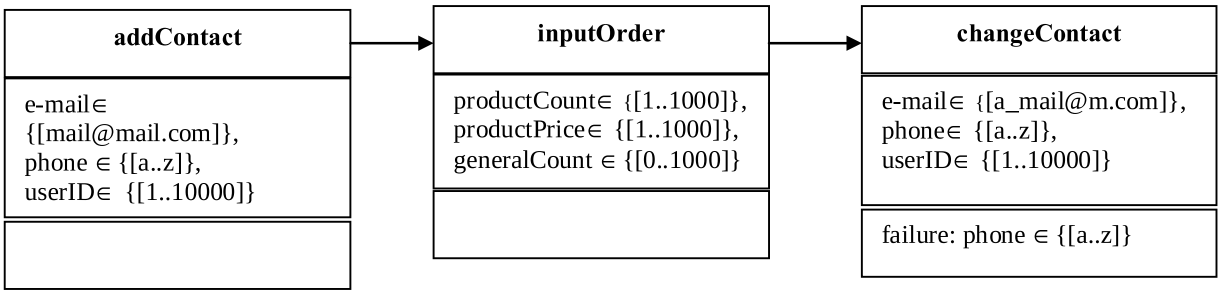

addContact
e-mail∈ {[mail@mail.com]},
phone ∈ {[a..z]},
userID∈ {[1..10000]}
inputOrder
productCount∈ {[1..1000]},
productPrice∈ {[1..1000]},
generalCount ∈ {[0..1000]}
changeContact
e-mail∈ {[a_mail@m.com]},
phone∈ {[a..z]},
userID∈ {[1..10000]}
failure: phone ∈ {[a..z]}

Fig. 3. An example of the scenario using three steps

Let's consider the example of a simple execution scenario of two steps, where changes on the first step result in a failure on the second (Fig. 3).

On the first step in the method addContact, phone value has changed to 1 (value from the first equivalence class), leading to the failure in the method changeContact.

Obviously, such scenarios can contain a larger number of steps. Testing all of these scenarios is not possible, because their number matches the product of all equivalence classes of all variables used in test scenarios. Still, their number can be reduced by the "white" box analysis.

CONSTRUCTION OF TEST SCENARIO BASED ON SOFTWARE USAGE MODEL

A sequence of software methods executions with certain variables values will be called a test scenario or test case. The test can be executed successfully, or terminate because of a failure in some method.

Today there are three most important methods of software testing, which are analyzed by our model [21]:

1. "Black box" methods, which consider system characteristics of the programs, ignoring their internal logical structure. In sense of presented model these methods use variables, which can be changed by end-user.
2. "White box" methods – a detailed study of the internal logic and structure of the software code when complete information about the source code of the software is available. Main difference for our model point of view that these methods use internal variables (for example, significant variables using unit testing).
3. "Gray box" methods, which allow testing the software with limited knowledge of the internal structure of the program, and are based on the use of information on major functional aspects of the software. These methods use variables, which can be changed by user, database variables and others (Fig. 4).

As noted in previous section, relationships and transitions between states are represented as a graph to build test scenarios. This graph can be built by the "white box" analysis of the code that best matches the real nature of the software, or by using a "black box" approach with logging of methods calls. The last one doesn't guarantee that all of the transitions will be included. Also, the construction of the required graph can be done using "gray box" methods, which means that some variables values in methods may be redundant, nonexistent in real software.

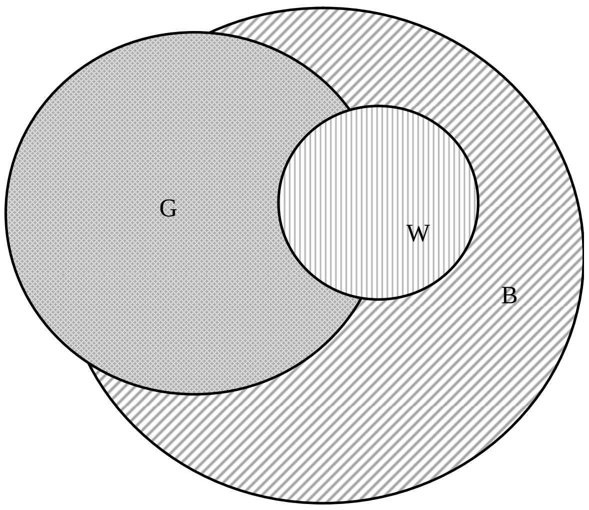


Fig. 4. Sets of transition in the graph during its creation by B – "black box", W – "white box", G – "gray box", U – the set of all transitions in the graph

Consider that each test scenario consist of several steps $T = \{T_0, T_{1, \ldots,} T_n\}$, where $T_i=\{S^i, V^i\}$, V^i – a set of variables and its values, which can be changed in method S^i Variables change its values on each step of the test scenario and failure can occur during test scenario execution.

To construct a set of test scenarios the following steps are used. Let's assume that distribution of components (methods) coverage by tests should be uniform on the zero iteration of the first scenario. The zero iteration T_0 of the first scenario starts with some components S^0 randomly with probability $p_0{}^1 = 1/N, N-$ number of all nodes of the graph. The probability of the next step T_{l+1} in the first scenario after step l defined as $p_{l+1}{}^1 = 1/\left|P(S^i)\right|$, where $\left|P(S^i)\right|$ – number of all transitions from component S^i, to ensure a uniform nodes coverage.

Let $Er^j(S^i)$ – count of failures that were found on *j*-th test scenario in the S^i component. $M(S^i)$ – the set of all numbers of nodes, having a transition from components S^i.

In next scenarios coverage is based on the density of failures obtained in the previous scenario. The *(k+1)-th* scenario starts from component S^0, such that: $p_0^{k+1} = \max \frac{Er^k(S^i)}{\sum_{j=1}^{N} Er^k(S^j)}, i = 1..n$. Nodes on the next iterations in *(k+1)-th* scenario are chosen with probability $p_{l+1}^{k+1} = \max \frac{Er^k(S^i)}{\sum_{j \in M(S^l)} Er^k(S^j)}, i \in M(S^l)$.

This choice provides an ability to choose nodes in proportion to the number of failures that occurred in the previous scenario. With built sets of test scenarios the software testing efficacy can be increased to improve performance reliability.

THE EXPERIMENTAL RESULTS

To illustrate the effectiveness of the proposed approach a research on dependency of number of failures (which describes the testing efficacy under equal conditions) on the nature of testing process (e.g. number of test scenarios) and the characteristics of the software product (the number of components) for different testing strategies had been conducted.

As an illustration, a software usage model was built for some test software with the following features: the number methods in software – 300, the total number of variables – 400, the average percentage of variable methods that can be changed by user – 20% and so on. Using the algorithm described above a set of test scenarios for such usage model was constructed, which were later executed to simulate the dependence of number of software failures on various parameters.

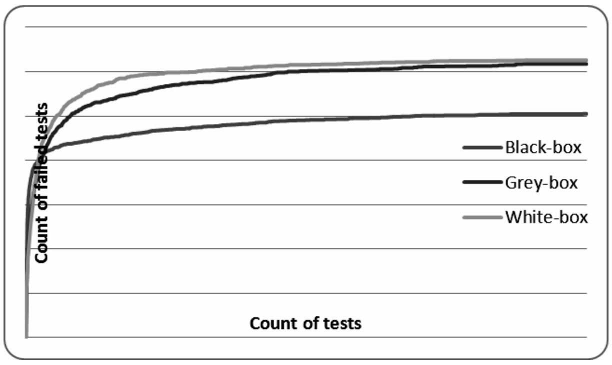


Fig. 5. The graph of dependence of number of passed tests on count of failures using three testing strategies

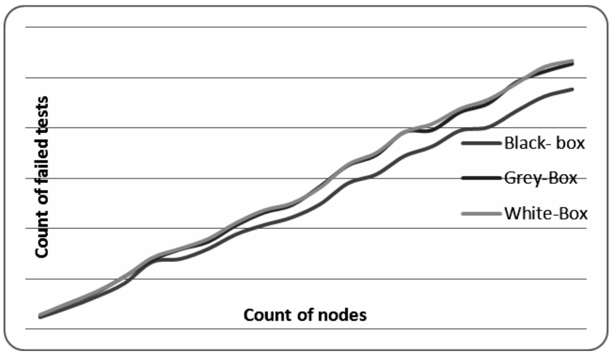


Fig. 6. Dependency graph of failures count and the number of methods using three testing strategies

In order to test the adequacy of the software usage model based on its variables a simulation of nature of dependency between the number of failures on the one side and software testing process characteristics and internal characteristics of the software on the other was conducted. Thus, in Fig. 5 a dependency of the number of failures on the number of test scenarios based on different strategies for particular test software is shown.

As shown in Fig.5 most failures were found during "white box" testing, and the least during "black box", which corresponds to the reality, since "white box" strategy takes into account the software architecture and features of its failures. The graph of dependency has been proven by many researchers in this area [22, 23].

Dependency graph of the number of failures on the number of software methods, depicted in Fig.6, shows a linear relationship between the software size (the number of methods in it) and the number of failures. As shown in Fig.6, the results, obtained using the proposed model, show almost linear dependency between the count of failures and the software size for all test strategies. The same dependency of the number of failures on the software size has been described in [24], which confirms the adequacy of the proposed software usage model. The calculated coefficients of determination R^2= 0.9971 for “black-box”, R^2= 0.9998 for “gray-box”, R^2= 0.9999 for “white-box” indicate a high correlation between the software size and the count of failures for particular testing strategy. It is also clear, that the smallest total number of failures, depending on the number of software components, has been obtained during “black-box” testing, meaning that “black box” testing efficacy drops with growth of software complexity – such testing is not able to detect all defects in a complex software. This conclusion corresponds well to empirical results obtained from the industrial software development practice, where the software size definitely correlates with its functionality and complexity.

CONCLUSIONS

This paper describes a new software usage model, which allow to investigate complex test scenario, taking into account the effect of method arguments, global and local variables, and considering failures that occur in the later stages of software development: regression testing, alpha and beta testing, software deployment. Also, a way to generate automated test scenarios based on formal software usage model is proposed, that will allow to increase the efficacy of software testing in software industry to assure a desired level of quality and reliability. On the basis of developed model the simulation of dependency of the count of failures on the number of test scenarios and the count of software methods has been performed for different testing strategies, which confirmed the validity of the developed software usage model and efficacy of automated generation of test scenarios. This approach also was probated on the software counting complex logic and demonstrated high efficacy in the case of computational algorithms with complicated relationships of methods and variables.

REFERENCES

1. **Pham H. 2006.** System Software reliability, Springer series in reliability engineering, 437.
2. **Lu Minyan and Chen Xuesong. 2000.** Software Reliability Testing and Practice, Testing and Control Technology, Volume 19, 509-512.
3. **Huang C.-Y. and Kuo S.-Y. and Lyu M.R. 2007.** An Assessment of Testing-Effort Dependent Software Reliability Growth Models, IEEE Transactions on Reliability, Volume 56, Issue 2, 198-211.
4. **Rafi M. and Rao K. 2010.** Software Reliability Growth Model with Logistic-Exponential Test-Effort Function and Analysis of Software Release Policy, International Journal on Computer Science and Engineering, Volume 2, Issue 2, 387-399.
5. **Jensen F. and Thoft C. 1992.** Application of linear decomposition technique in reliability-based structural optimization, System Modelling and Optimization Lecture Notes in Control and Information Sciences, Volume 180, 953-962.
6. **Zhang X.-L., Huang H.-Z. and Yu Liu. 2009.** Hierarchical decomposition for optimal reliability allocation. Reliability and Maintainability Symposium, 124-128.
7. **Meenakshi Kandpal and Anmol Kandpal. 2012.** Critical Analysis of Traditional Size Estimation Metrics for Object Oriented Programming, International Journal of Computer Applications, Volume 58, Issue 13, 39-45.
8. **Fenton N. and Neil M. 1999.** A Critique of Software Defect Prediction Models. IEEE Transactions on software engineering, volume 25, Issue 5, 675-689.
9. **Kamaljit Kaur, Kirti Minhas, Neha Mehan and Namita Kakkar. 2009.** Static and Dynamic Complexity Analysis of Software Metrics. World Academy of Science, Engineering and Technology, Volume 56, 159-161.
10. **Bobalo Y.Y., Volochyy B.Y., Lozynskyy O.Y., Mandzyy B.A., Ozirkovskyy L.D., Fedasyuk D.V., Sherbovskyh S.V. and Yakovyna V.S. 2013.** Mathematical models and methods for reliability analysis of radio-electronic, electrical and software systems: a monograph, Lviv: Publishing House of Lviv Polytechnic National University, 300. (in Ukrainian)
11. **Mei-Hwa Chen and Michael R. Lyu. 2001.** Effect of Code Coverage on Software Reliability Measurement. IEEE Transactions on Reliability, Volume 50, Issue 2, 165-170.
12. **Rao D.N. and Srinath M.V. and Hiranmani B.P. 2013.** Reliable code coverage technique in software testing, International Conference on Digital Object Identifier: 10.1109/ICPRIME, 157 – 163.
13. **McConnell S. 1996.** Software quality at top speed, Software Development, Volume 4, Issue 8, 38 – 42.
14. **Barry Boehm. 2007.** Equity Keynote Address.
15. **Bieman J. M. and Kang B-K. 1995.** Cohesion and Reuse in an Object-Oriented System, Proc. ACM Symposium on Software Reusability (SSR'95), 259-262.
16. **Ott L., Bieman J. M., Kang B-K. and Mehra B. 1995.** Developing Measures of Class Cohesion for Object-Oriented Software, Proc. Annual Oregon Workshop on Software Merics (AOWSM'95), 11.
17. **Izosimov A.V. and Ryzhko A.L. 1989.** Metrics assessment of software quality, Mai. (in Russian)
18. **Chapin N. 1989.** An Entropy Metric For Software Maintainability System Sciences, Proceedings of the Twenty-Second Annual Hawaii International Conference, Volume II: Software Track, 522–523.
19. Software Complexity Metrics [Internet source] – available: http://metrix.narod.ru/page1.htm [rus]
20. **Stepanchenko I.V. 2006.** Software testing methods, VolgGTU, Volgograd, 74 (in Russian).
21. **Mohd Ehmer Khan. 2010.** Different Forms of Software Testing Techniques for Finding Errors, IJCSI, Volume 7, Issue 3, 11-16.
22. **Omar Shatnawi. 2009.** Discrete Time NHPP Models for Software reliability growth phenomenon, The International Arab Journal of Information technology, Volume 6, Issue2, 124-131
23. **Shaik M. and Shaheda R. 2011.** Software reliability Growth model with bass diffusion test- effort function and analysis of software release policy, International Journal of Computer Theory and Engineering, Volume 3, Issue 5, 671-680.
24. **El Emam K. 2000.** A methodology for validating software product metrics, Tech. rep. NCR/ERC-1076, National Research Council of Canada, Ottawa, Ontario, Canada.