Optimal Resource Allocation for Reliability of Modular Software Systems

K. WORWA, J. STANIK
kazimierz.worwa@wat.edu.pl

Institute of Computer and Information Systems
Faculty of Cybernetics, Military University of Technology
Kaliskiego Str. 2, 00-908 Warsaw, Poland

Considerable development resources are consumed during the software-testing phase. The software development manager has to decide how to use the testing-resources effectively in order to maximize the software quality and reliability. The paper discusses a management problem to achieve a reliable software system efficiently during the module testing stage by applying a software reliability growth model. This model both describes the software-error detection phenomenon and represents the relationship between the cumulative number of errors encountered by software testing and the time span of the testing. As testing cost and software reliability are both important factors in the testing-resource allocation problems an investigation is performed in the paper to search for the optimal solution for modular software system with the objectives of maximising system reliability and minimising testing cost.

Keywords: modular software systems, software testing, software reliability.

1. Introduction

It is well known that a software development process consists of the following four phases: specification (including user requirements definition and software functional and non-functional requirements specification), design (structural design and detailed design), coding and testing. The reciprocal relationships between these phases, including phase sequence and development activities range, depend on assumed software life cycle model.

Practically, a software testing process consists of several testing stages including module testing, integration testing, system testing and installation testing. During the testing phase, software faults can be detected and removed. The quality of the tests usually corresponds to the maturity of the software test process, which in turn relates to the maturity of the overall software development process. In general, most popular and commercial software products are complex systems composed of a number of modules. Typically, module testing is the most time-critical part of testing to be performed. All the testing activities of different modules should be completed within a limited time, and these activities normally consume approximately 40−50% of the total amount of software development resources [14]. Therefore, project managers should know how to allocate the specified testing-resources among all the modules and develop quality software with high reliability. During the unit-testing phase, all the testing activities of different modules are competing for the limited testing-resource, i.e. testing time and testing cost. Thus, a critical problem is how to allocate the total available program testing-resource among program modules in an optimal way. There are many papers that investigate this problem, e.g. [6] has presented an overview of the methods that have been developed for solving various reliability optimization problems including testing resource allocation problems; in paper [9] two resource allocation problems for software-module testing are formulated and solved, considering the mean number of remaining faults in the software modules; paper [7] studied a dynamic resource allocation strategy for software module testing; in paper [13] the problem of determining an optimal testing strategy is investigated.

The optimal testing-resource allocation problem is important, but difficult in project planning and management. It has been studied extensively in the literature (see e.g. [2, 6, 7, 9, 14]). When only limited resources are available during testing of a software system, it is important to allocate the testing resources efficiently, so that the maximum reliability of the complete system is achieved.

Testing cost and software reliability are both important factors in the testing-resource allocation problems. This paper discusses a management problem to achieve a reliable software system efficiently during module testing stage by means of formulating and solving the testing-resource allocation optimization problem with the objectives of maximizing program reliability and minimizing the program testing cost. Section 2 introduces a formal model of a program testing process. This model considers the effect of testing-resources on the software reliability growth. Section 3 uses the program testing model to derive optimal allocation of the testing-resources for module testing. Finally, section 4 presents a numerical example to illustrate the method of determining an optimal program reliability structure.

2. Description of the Program Testing Process

We assume that the program under the testing process consists of several modules. Although there is no universally accepted definition of modularization, most programmers would conceive a module as a fragment of a program that carries out a specific function and may be used alone or combined with other modules of the same program. This usually implies that a module can be designed, implemented, and tested independently. Since a large-scale program always involves a substantial number of programmers working concurrently, a large program can be viewed as a collection of logically independent modules. A module is usually defined to perform a particular function. Let **M** mean a set of module numbers of the program under the testing, $\mathbf{M} = \{1,2,3,...,m,...,M\}$. Let us define the reliability of a module as the probability that the module performs the function correctly, i.e., the module produces the correct output and transfers control to the next module correctly. When a set of user input is supplied to the program, a sequence of modules will be executed. The reliability of the output will depend on the sequence of modules executed and the reliability of each individual module. We first assume that the reliabilities of the modules are independent. This means that errors will not compensate each other, i.e., an incorrect output from a module will not be corrected later by subsequent modules. Since errors do not compensate each other, the result of the execution of the program is correct, if and only if the correct sequence of modules is executed and in every instance of module execution the module produces the correct result. The reliability of a module, in general, is a function of many factors, and the study of the reliability function of a module is beyond the scope of this paper. However, if no modification is made on the modules and the user environment does not change, the reliability function of a module should remain invariant. Let R_m stand for the *m*-th module reliability function. We will assume that the module reliability functions can be determined and the following vector $\mathbf{R} = (R_1, R_2, R_3, ..., R_m, ..., R_M)$ will be called a program reliability structure.

We next assume that the transfer of control among program modules is a Markov process. This implies that the next module to be executed will depend probabilistically on the present module only and is independent of the past history. It is noteworthy that this assumption may not be valid for all types of programs. Although the assumption of Markovian behavior of control flow at the instruction level is questionable, experiments performed by researchers in memory management and scheduling have shown that this assumption may be valid at the macroscopic level, i.e. module level for a large number of programs [4, 8].

Let us represent the control structure of the program by a directed graph where every node *i* represents a program module and a directed branch (*i*, *j*) represents a possible transfer of control from *i* to *j*. To every directed branch (*i*, *j*) we will attach a probability p_{ij} as the probability that the transition (*i*, *j*) will be taken when control is at node *i*. If $p_{ij} = 0$, the branch (*i*, *j*) does not exist. This transition probability represents the branching characteristics of the decision point at the exit point of the module *i*. Without loss of generality, let us assume that the program graph has a single entry node 1 and a single exit node *M*. Let us consider every node in the graph as a state of the Markov process, with the initial state corresponding to the entry node of the program graph. Two states *C* and *F* are added as the terminal states, representing the state of correct output and failure, respectively. For every node *i*, a directed branch (*i*, F) is created with transition probability $1 - R_i$ representing the occurrence of an error in the execution of module *i*. Since errors do not compensate each other, a failure in *i* will ultimately lead to an incorrect program output, regardless of the sequence of modules executed afterwards. This phenomenon is represented by the transition to the terminal state *F*. The original transition probability between *i* and *j* is modified into $R_i \cdot p_{ij}$, which represents the probability that the execution of module *i* produces the correct result and control is transferred to module *i*. For the exit note *M*, a directed branch (M, C) is created with a transition probability R_M to represent correct termination at the exit node.

Assuming that the testing process is modelled by the above-described Markov process let $\overline{\mathbf{P}}(\mathbf{R})$ mean the transition matrix of this process.

$$\overline{\mathbf{P}}(\mathbf{R}) = \begin{array}{c} \\ C \\ F \\ 1 \\ \dots \\ m \\ \dots \\ M-1 \\ M \end{array} \begin{array}{c} \begin{array}{cccccccc} C & F & 1 & 2 & \dots & m & \dots & M \end{array} \\ \left[\begin{array}{cccccccc} 1 & 0 & 0 & 0 & \dots & 0 & \dots & 0 \\ 0 & 1 & 0 & 0 & \dots & 0 & \dots & 0 \\ 0 & 1-R_1 & 0 & R_1 p_{12} & \dots & R_1 p_{1m} & \dots & R_1 p_{1M} \\ \dots & \dots & \dots & \dots & \dots & & \dots & \dots \\ 0 & 1-R_m & 0 & R_m p_{m2} & \dots & R_m p_{mm} & \dots & R_m p_{mM} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 1-R_{M-1} & 0 & R_{M-1} p_{M-1,2} & \dots & R_{M-1} p_{M-1,m} & \dots & R_{M-1} p_{M-1,M} \\ R_M & 1-R_{M-1} & 0 & 0 & \dots & 0 & \dots & 0 \end{array}\right] \end{array}, \qquad (1)$$

According to earlier assumptions the matrix $\overline{\mathbf{P}}(\mathbf{R})$ has the form of (1). As we mentioned earlier vector $\mathbf{R}$ is the so-called program reliability structure. The first two rows and two columns of the matrix $\overline{\mathbf{P}}(\mathbf{R})$ represent absorbing states *C*, *F* of the Markov process.

The reliability of the program is, therefore, the probability of reaching the terminal state *C* from the initial state 1 of the Markov process. We can express the program reliability as a function of the reliability of the modules and frequency distribution of utilization of these modules. Using this approach, the reliability of the program is a function of both the deterministic properties of the structure of the program and the stochastic properties of the module failure behaviour and the utilization of these components by the user. Ideally, we would like a reliability model where the program structure, in terms of its modules, can be easily constructed, where the component reliabilities can be independently determined, and where a simple user profile can be measured easily by monitoring the dynamic behaviour of the program.

3. Optimization of the Testing-Resource Allocation Problem

The Markov reliability model uses the program flow graph to represent the structure of the system so that the structure can be easily obtained by analyzing the code. It uses functional modules as the basic components so that the component reliabilities are reasonably independent. It uses the branching characteristics among modules as the user profile so that they can be easily measured in the operational environment. It is noteworthy that similar structural models are often proposed in the literature, see e.g.[1, 3, 5, 8−10, 13]. The overall software reliability can be expressed in terms of individual module reliabilities and other parameters, such as visit statistics to each module.

Let $\mathbf{P}(\mathbf{R})$ be the matrix of order *M* obtained from $\overline{\mathbf{P}}(\mathbf{R})$ by deleting all the rows and columns correspond to the absorbing states *C* and *F*. According to earlier remarks reliability of the program is equal to the probability of reaching by the Markov process the terminal state *C* from the initial state *1*. Using a standard method of computing absorption probabilities in the Markov process reliability of the program can be computed as follows [1]

$$R(\mathbf{R}) = S_{1M}(\mathbf{R})R_M\,, \qquad (2)$$

where S_{1M} is an element of the matrix $\mathbf{S} = (\mathbf{I} - \mathbf{P}(\mathbf{R}))^{-1}$ that lies on the intersection of the first row and the *M*-th column, $\mathbf{I}$ is the identity matrix of order *M* and R_M is the reliability function of the *M*-th module.

It should be mentioned that this method of computing the program reliability coefficient as given in (1) can be quite inconvenient without the use of software that can perform the requisite symbolic linear algebra, although packages such as *Mathematica* [12].

The cost of a module is associated with a number of parameters such as the design and development time, and testing time, and can be assumed to be monotonically related to the value of module reliability. The module cost – reliability relation can be linear, exponential, and even random while maintaining the monotonic property. We assume that the cost of the program consisting of M components, denoted by $K(\mathbf{R})$ can be given by function of the form

$$K(\mathbf{R}) = \sum_{m=1}^{M} K_m(R_m) \qquad (3)$$

where $K_m = f_m(R_m)$, $m \in \mathbf{M}$, is the cost of the m-th module, if its reliability is R_m; is reasonable to assume that the cost function $K_m = f_m(R_m)$ depends monotonically on reliability R_m.

The management problem to achieve a reliable program efficiently during the module testing stage in the software development process can be formulated as an optimization problem with two objectives as follows:

$$\text{maximize} \quad R(\mathbf{R}) = S_{1M}(\mathbf{R})R_M, \qquad (4)$$

$$\text{minimize} \quad K(\mathbf{R}) = \sum_{m=1}^{M} K_m(R_m), \qquad (5)$$

subject to constraints

$$R(\mathbf{R}) = S_{1M}(\mathbf{R})R_M \geq R_{\min}, \qquad (6)$$

$$K(\mathbf{R}) = \sum_{m=1}^{M} K_m(R_m) \leq K_{\max}, \qquad (7)$$

$$\mathbf{R} = (R_1, R_2, R_3, ..., R_m, ..., R_M),$$

$$0 \leq R_m \leq 1, \quad m \in \mathbf{M}, \qquad (8)$$

where R_{min} is a minimal feasible value of a program reliability and K_{max} is a maximal feasible value of a program testing cost.

The testing-resource allocation problem (4)−(8) is a bicriteria optimization problem with nonlinear objective functions and nonlinear constraints. A solution of this problem can be obtained by using well known methodology of solving multiple optimization problems [11]. According to that methodology as a solution of the problem (4)−(8) we can determine:

- a dominate solution set
- a nondominate solution set
- a compromise solution set.

Taking into account that values of objective functions $R(\mathbf{R})$ and $K(\mathbf{R})$ are inverse (in sense that if a value $R(\mathbf{R})$ is increased, the value $R(\mathbf{R})$ is increased, too) it is reasonable to expect that the dominate solution set will empty. In such a situation a practically recommended approach is to determine a nondominated solution set. If this set is very numerous we can narrow it down by determining a so-called compromise solution [11]. The program reliability structure $\mathbf{R}^*$ that to be obtained after the solving the optimization problem (4)−(8) will both maximize the value of the program reliability function $R(\mathbf{R})$ and minimize the value of the program testing cost $K(\mathbf{R})$.

4. Numerical Example

The methodology of determination of the program reliability structure that has been presented can be illustrated by the following numerical example. Let Fig. 1 be the directed graph representing the control structure of a program with five modules where *1* represents the input module and *5* is the output module. The set of a program component modules is $\mathbf{M} = \{1, 2, 3, 4, 5\}$ and the program reliability structure is a vector $\mathbf{R} = (R_1, R_2, R_3, R_4, R_5)$.

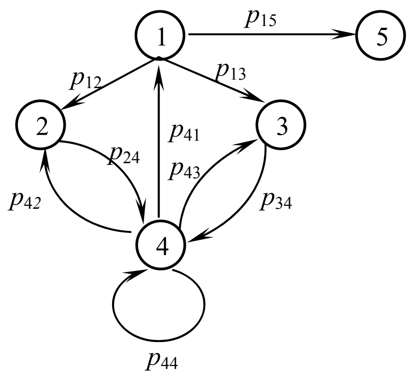

Fig. 1. A program control graph

Table 1 presents branching probabilities p_{ij} between the modules *i* and *j*.

Table 1. Values of branching probabilities between the modules *i* and *j*

$p_{11} = 0$	$p_{12} = 0.6$	$p_{13} = 0.3$	$p_{14} = 0$	$p_{15} = 0.1$
$p_{21} = 0$	$p_{22} = 0$	$p_{23} = 0$	$p_{24} = 1$	$p_{25} = 0$
$p_{31} = 0$	$p_{32} = 0$	$p_{33} = 0$	$p_{34} = 1$	$p_{35} = 0$
$p_{41} = 0.3$	$p_{42} = 0.2$	$p_{43} = 0.4$	$p_{44} = 0.1$	$p_{45} = 0$

Using (1) the value of the program reliability function can be determined as $R(\mathbf{R}) = S_{15}(\mathbf{R})R_5$, where quantity S_{15} can easily be computed by means of software packages *Mathematica* [12] as follows

$$S_{15} = \frac{L(\mathbf{R})}{M(\mathbf{R})}, \tag{9}$$

where:

$L(\mathbf{R}) =$

$= 0{,}1R_1 - 0{,}01R_1R_4 - 0{,}02R_1R_2R_4 - 0{,}04R_1R_3R_4$

and

$M(\mathbf{R}) = 1 - 0{,}1R_4 - 0{,}2R_2R_4 - 0{,}18R_1R_2R_4 +$

$- 0{,}4R_3R_4 - 0{,}09R_1R_3R_4$

where R_m is the *m*-th module reliability function, $m \in \mathbf{M} = \{1, 2, 3, 4, 5\}$.

Let the cost of the *m*-th module function $K_m = f_m(R_m)$, $m \in \mathbf{M}$, is of the form

$$K_m(R_m) = KS_m + \alpha_m e^{\beta_m R_m}, \quad m \in \mathbf{M} = \{1, 2, 3, 4, 5\} \tag{10}$$

where:

KS_m – a constant component of the *m*-th module testing cost,

α_m, β_m – shape coefficients of testing cost function of the *m*-th module; their values depend on module complexity, development technology, programmers experience, etc.

Let values of parameters KS_m, α_m, β_m, R_{min} and K_{max} be as in Table 2.

Table 2. Values of parameters KS_i, α_i, β_i, R_{min} and K_{max}

m	1	2	3	4	5
KS_m	1000,0	400,0	1500,0	2500,0	900,0
α_m	50,0	35,0	25,0	30,0	40,0
β_m	5,0	2,0	8,0	10,0	4,0
R_{min}	0,8				
K_{max}	160 000,0				

For the program control graph from Fig. 1 and parameter vales in Table 1 we will solve the bicriterial optimization problem (4)−(8). A feasible solution set for this problem is a set of module reliability structure vectors $\mathbf{R} = (R_1, R_2, R_3, R_4, R_5)$.

If we assume that $R_m \in [0.95, 1.00]$, $m \in \{1, 2, 3, 4, 5\}$, a cardinality number of this set is 6^5=7776. It is easy to check that the dominated solution set of the problem (4)−(8) is empty (it is because of objective functions $R(\mathbf{R})$ and $K(\mathbf{R})$ are inverse in sense their values). In that case practically recommended approach is to determine a nondominated solution set. The nondominated solution set of the bicriterial optimization problem (4)−(8) is plotted in Fig.2. Every point among points that are situated on the shadow contour is a nondominated solution of the bicriterial optimization problem (4)−(8). In order to narrow down this a nondominated solution set we will determine a compromise solution of this problem, i.e. such a solution belongs to the nondominated solution set that is nearest (in the sense of Euclidean distance) to the so-called ideal point [11]. For this reason both objective functions $R(\mathbf{R})$ and $K(\mathbf{R})$ determined by (6) and (7), respectively, will be normalized by means of the following formulae:

$$\overline{\mathrm{K}(\mathbf{R})} = \frac{K(\mathbf{R}) - K_{\min}(\mathbf{R})}{K_{\max}(\mathbf{R}) - K_{\min}(\mathbf{R})}, \tag{11}$$

$$\overline{\mathrm{R}(\mathbf{R})} = \frac{R(\mathbf{R}) - R_{\min}(\mathbf{R})}{R_{\max}(\mathbf{R}) - R_{\min}(\mathbf{R})}, \tag{12}$$

where:

$$K_{\min}(\mathbf{R}) = \min_{\boldsymbol{R} \in \boldsymbol{R}} K(\mathbf{R}), \quad R_{\min}(\mathbf{R}) = \min_{\boldsymbol{R} \in \boldsymbol{R}} R(\mathbf{R}) \tag{13}$$

$$K_{max}(\boldsymbol{R}) = \max_{\boldsymbol{R} \in \boldsymbol{R}} K(\boldsymbol{R}), \quad R_{\max}(\mathbf{R}) = \max_{\boldsymbol{R} \in \boldsymbol{R}} R(\mathbf{R}), \tag{14}$$

where $\boldsymbol{R}$ is a set of all feasible program reliability structure, i.e.

$\boldsymbol{R} = \{\mathbf{R} = (R_1, R_2, R_3, R_4, R_5) : 0 \le R_m \le 1,$

$m \in \mathbf{M} = (1, 2, 3, 4, 5\}\}$.

As a result of the normalization (10)−(11) both normalized objective functions $\overline{\mathrm{K}(\mathbf{R})}$ and $\overline{\mathrm{R}(\mathbf{R})}$ have values belong to range [0, 1].

It is easy to notice that in the example that is considered the ideal point $(\overline{\mathrm{K}^*(\mathbf{R})}, \overline{\mathrm{R}^*(\mathbf{R})})$ is of the form of $(\overline{\mathrm{K}^*(\mathbf{R})}, \overline{\mathrm{R}^*(\mathbf{R})}) = (0, 1)$.

Table 2 presents the results of determination of the optimisation problem (3)−(7). The row that corresponds to the vector $\mathbf{R}^* = (1.00, 1.00, 0.95, 1.00, 1.00)$ is the compromise solution of this problem, i.e. such a vector that is nearest to the ideal point (0, 1), where a distance function $\mathrm{d}[(\overline{\mathrm{K}(\mathbf{R})}, \overline{\mathrm{R}(\mathbf{R})}, (0, 1)]$ is of the form of

$$\mathrm{d}[(\overline{\mathrm{K}(\mathbf{R})}, \overline{\mathrm{R}(\mathbf{R})}, (0, 1)] = \sqrt{[\overline{\mathrm{K}(\mathbf{R})}]^2 + [\overline{\mathrm{R}(\mathbf{R})} - 1]^2}. \tag{15}$$

It is easy to notice that the vector $\mathbf{R}^* = (1.00, 1.00, 0.95, 1.00, 1.00)$ complies with the following condition

$$\mathrm{d}[(\overline{\mathrm{K}(\mathbf{R}^*)}, \overline{\mathrm{R}(\mathbf{R}^*)}, (0, 1)] = \min_{\boldsymbol{R} \in \boldsymbol{R}} \mathrm{d}[(\overline{\mathrm{K}(\mathbf{R})}, \overline{\mathrm{R}(\mathbf{R})}), (0, 1)] \tag{16}$$

The values of the objective functions $R(\mathbf{R})$ and $K(\mathbf{R})$ for the compromise solution $\mathbf{R}^* = (1.00, 1.00, 0.95, 1.00, 1.00)$
are R($\mathbf{R}^*$) = 0,8711 and K($\mathbf{R}^*$) = 158 481,43.

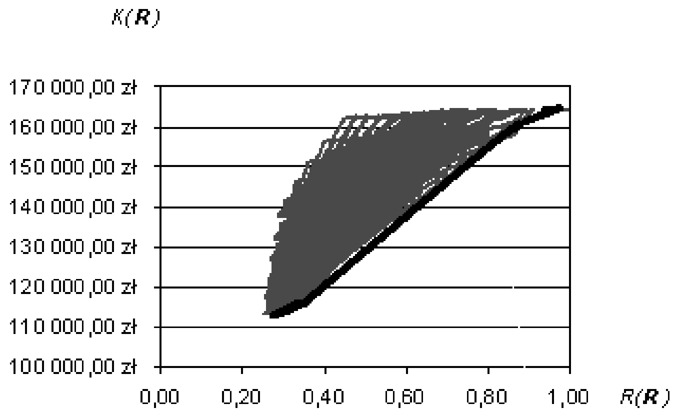

Fig. 2. The nondominated solution set of the optimization problem (4)−(8)

Table 2. The compromise solution of the optimization problem (4)−(8)

R_1	R_2	R_3	R_4	R_5	$\overline{\mathrm{R}(\mathbf{R})}$	$\overline{\mathrm{K}(\mathbf{R})}$	$\mathrm{d}[(\overline{\mathrm{K}(\mathbf{R})}, \overline{\mathrm{R}(\mathbf{R})}), (0,1)]$
1,00	1,00	0,99	0,99	0,99	0,5888	0,77	0,876889
1,00	1,00	0,99	0,99	1,00	0,5981	0,78	0,874058
1,00	1,00	0,99	1,00	0,95	0,7705	0,88	0,909792
1,00	1,00	0,99	1,00	0,96	0,7821	0,88	0,908320
1,00	1,00	0,99	1,00	0,97	0,7936	0,88	0,907053
1,00	1,00	0,99	1,00	0,98	0,8052	0,88	0,905995
1,00	1,00	0,99	1,00	0,99	0,8168	0,89	0,905147
1,00	1,00	0,99	1,00	1,00	0,8284	0,89	0,904515
1,00	1,00	1,00	0,95	0,95	0,2088	0,51	0,942643
1,00	1,00	1,00	0,95	0,96	0,2145	0,51	0,938661
1,00	1,00	1,00	0,95	0,97	0,2201	0,52	0,934730
1,00	1,00	1,00	0,95	0,98	0,2258	0,52	0,930854
1,00	1,00	1,00	0,95	0,99	0,2315	0,52	0,927034
1,00	1,00	1,00	0,95	1,00	0,2372	0,52	0,923273
1,00	1,00	1,00	0,96	0,95	0,2720	0,59	0,939384

5. Conclusions

This paper proposes a method of determining the optimal program reliability structure for the purpose of the program testing stage realization. The optimal program reliability structure that is to be obtained after solving the bicriteria optimization problem (4)−(8) both maximizes the value of the program reliability function $R(\mathbf{R})$ and minimizes the value of the program testing cost $K(\mathbf{R})$.

For formulating and solving the optimal testing-resource allocation problem, the following assumptions are made, which are valid in most cases:

- a software program under testing is composed of several modules, which are developed and tested independently during the unit-testing phase
- in the unit-testing phase, each program module is subject to failures at random times caused by faults remaining in the module and the failures of the modules are independent
- a module structure of a program under testing is known and a transfer of control among program modules follows a Markov process and the exchanges of controls among these modules are characterized according to the rules of a Markov process;

the next transfer of control to be executed is independent of the past history and depends only on the present module

- the probability of transition from one module to another is determined from the operational profile of a system
- the reliability of a module-based program depends on the architecture of the program and the reliabilities of their component modules
- the cost of a module function is monotonically related to the value of the module reliability.

Knowledge of the optimal reliability structure of the program under testing allows the software testing manager to rationally decide how to use the limited testing resources effectively in order to maximize the program reliability and minimize the program testing cost. On the basis of the optimal program reliability structure the manager knows how to allocate the specified amount of testing-resource expenditures for each program module to achieve both the maximal program reliability growth and minimal cost of the program testing.

The method of determining the program reliability structure that has been proposed was illustrated by the simple numerical example. For assumed module structure of a program, the optimal program reliability structure has been obtained as a compromise solution of the bicriteria optimization problem (4)−(8).

6. Bibliography

[1] R.C. Cheung, ”A user-oriented software reliability model”, *IEEE Transactions on Software Engineering*, Vol. SE-6(2), 1980.

[2] D.W. Coit, ”Economic allocation of test times for subsystem level reliability growth testing”, *IEEE Transactions on Software Engineering*, Vol 30(12), 1998.

[3] Y.S. Dai, M. Xie, K.L. Poth, B. Tang, ”Optimal testing-resource allocation with genetic algorithm for modular software systems”, *The Journal of Systems and Software*, No. 66, 2003.

[4] K. Goseva-Popstojanova, K.S. Trivedi, ”Architecture-based approach to reliability assessment of software systems”, *Performance Evaluation*, No 45, 2001.

[5] C.Y. Huang, J.H. Lo, ”Optimal resource allocation for cost and reliability of modular software systems in the testing phase”, *The Journal of Systems and Software*, No. 79, 2006.

[6] W. Kuo, V.R. Prasad, ”An annotated overview of system reliability optimisation”, *IEEE Transactions on Software Engineering*, Vol. 49, 2 (2000).

[7] Y.W. Leung, ”Dynamic resource-allocation for software-module testing”, *The Journal of Systems and Software*, Vol. 37, 2 (1997).

[8] J.D. Musa, *Software – Reliability--Engineered Testing Practice*. McGraw--Hill, New York, 1998.

[9] H. Ohtera, S. Yamada, ”Optimal allocation & control problems for software-testing resources”, *IEEE Transactions on Software Engineering*, Vol. 39(2), 1990.

[10] J. Rajgopal, M. Mazumdar, ”Modular operational test plans for inferences on software reliability based on a Markov model”, *IEEE Transactions on Software Engineering,* Vol. 28(4), 2002.

[11] J. Stadnicki, *Teoria i praktyka rozwiązywania zadań polioptymalizacji*. WNT, 2006.

[12] S. Wolfram, *The Mathematica book,* Cambridge University Press and Wolfram Media, Inc., 1996.

[13] K. Worwa, ”Estimation of he program testing strategy”, *Postępy Cybernetyki*, Zeszyt 3−4, 1995.

[14] S.T. Yamada, I.M. Nishiwaki, ”Optimal allocation policies for testing-resource based on a software reliability growth model”, *International Journal of Mathematical and Computer Modelling*, Vol. 22, 10−12 (1995).

Optimalizacja alokacji nakładów w procesie wytwarzania programu o znanej strukturze modułowej

K. WORWA, J. STANIK

W artykule przedstawiona jest metoda określania struktury niezawodnościowej programu, rozumianej jako wektor wskaźników niezawodności jego modułów składowych. Modelem rozpatrywanego programu jest graf przepływu sterowania, w którym prawdopodobieństwa uaktywniania poszczególnych modułów składowych w procesie wykonywania programu wynikają z tzw. profilu operacyjnego programu, charakteryzującego rzeczywiste środowisko jego pracy. Struktura niezawodnościowa wyznaczana jest w wyniku rozwiązania określonego zadania programowania matematycznego. Znajomość struktury niezawodnościowej programu umożliwia właściwe zaplanowanie nakładów czasowo-finansowych, wymaganych dla wytworzenia programu, spełniającego założone wymagania niezawodnościowe. Zastosowanie przedstawionej metody zilustrowane zostało przykładem liczbowym.

Słowa kluczowe: struktura modułowa programu, testowanie oprogramowania, niezawodność oprogramowania.