THE MATHEMATICAL STATEMENT FOR THE SOLVING OF THE PROBLEM OF N-VERSION SOFTWARE SYSTEM DESIGN*

I V Kovalev, D I Kovalev, P V Zelenkov, A A Voroshilova
Siberian State Aerospace University Named after Academician M.F. Reshetnev
31 "KrasnoyarskiyRabochiy" prospect, Krasnoyarsk, 660037, Russia.

E-mail:zelenkov@sibsau.ru

Abstract. The N-version programming, as a methodology of the fault-tolerant software systems design, allows successful solving of the mentioned tasks. The use of N-version programming approach turns out to be effective, since the system is constructed out of several parallel executed versions of some software module. Those versions are written to meet the same specification but by different programmers. The problem of developing an optimal structure of N-version software system presents a kind of very complex optimization problem. This causes the use of deterministic optimization methods inappropriate for solving the stated problem. In this view, exploiting heuristic strategies looks more rational. In the field of pseudo-Boolean optimization theory, the so called *method of varied probabilities* (MVP) has been developed to solve problems with a large dimensionality.

1. Introduction

Development of high-reliable fault-tolerant systems is an interesting engineering problem having not only technical meaning but also social importance. Systems of this kind determine the stability in social and technical environments, and multiple examples of such systems' crashes prove the strong need for more reliable constructions which can be realized through the use of up-to-date methods and approaches.

The rapid progress of computer technique of late years has made the software an essential part of any complex automated system. The reliability of software component may determine the reliability of whole the hardware-software system. That's why during last years large attention is paid to the development of the methodologies of designing high-reliable software complexes [1-5].

Practically, multi-channel tools increasing the system reliability at the expense of a multiple duplication of certain structure elements are very much in evidence. This approach has given a good account of itself in the designing of hardware parts of complex systems. The use of this methodology leads to a sizable decreasing of appearance probability of random errors having the physical nature. In turn, this approach is not an influence on software reliability, since it doesn't trace so called *dormant* (or sleeping) errors which could arise while writing the program code by a stated specification [6].

* The research was done with the financial support of the Ministry of Education and Science of the Russian Federation in accordance with the agreement № 14.574.21.0041, unique identifier RFMEFI57414X0041

The multi-version programming, as a methodology of the fault-tolerant software systems design, allows successful solving of the mentioned tasks. The idea of multi-version programming has been introduced by A. Avizienis in 1977 [7]. The term N-version programming (NVP) used in the literature is of equal meaning and often takes place in papers on the observed methodology. A. Avizienis introduced NVP as *an independent generation* of N≥2 functionally equivalent software modules from the same *initial specification*. The *concurrent execution tools* are provided for such the modules. In cross-check points (cc-points) software modules generate cross-check vectors (cc-vectors). The components of the cc-vectors and the cc-points are to be determined in the specification set.

The use of N-version programming approach turns out to be effective, since the system is constructed out of several parallel executed versions of some software module. Those versions are written to meet the same specification but by different programmers. Where, the writing process of each version of concrete software module in any way must not intersect with or depend on another version code writing. This is done to avoid the presence of same *dormant* (or sleeping) errors in separate software designs. This kind of errors is typical for software components.

The problem of developing the optimal structure of an N-version software system (NVS) is the following: to choose a set of software modules, so as to provide the highest reliability for the system subject to the budget constraint. Since a description of any possible system configuration is made through such the positioning of its components, we can say that an observed problem has the binary essence [8]. Moreover, the existing theory of pseudo-Boolean functions and their optimization contains strong tools for solving problems of this kind [9]. And that fact makes the use of binarization algorithms more affordable.

The process of a problem binarization consists in setting relationships between the system states and the binary space elements. In the case of our system model, we need to determine some Boolean vector the elements of which will characterize the system structure. Each element of such the Boolean vector will signify either presence or absence of corresponding system component [10].

In that way, before starting to describe the exact process of binarization, all the necessary terms should be coined and the presented system model should be overviewed in details.

2. Optimization Model for Structuring NVS

The structure of N-version software system is determined consisting of a set of tasks (a set $\mathbf{I}, \operatorname{card}\mathbf{I} = I$). All the tasks are divided into classes, i.e. a set of task classes is introduced as well ($\mathbf{J}, \operatorname{card}\mathbf{J} = J$).

To solve the tasks belonging to a certain class, there is a software module, which can be realized by any of its versions. Thus, $\mathbf{K}, \operatorname{card}\mathbf{K} = J$ – the set of software modules. Let us introduce the vector $\mathbf{S} = \{S_j\}\,(j = \overline{1,J})$, each component of which is equal to a number of module versions (S_j – the number of versions of module solving a task of class *j*) [11].

To describe the task belonging to particular classes, in [12] the authors define sets of tasks for every task class. That is introduced as two-dimensional array in programming terms. Since the numbers of tasks belonging to different classes are not equal, that may cause some difficulties when translating the analytic expressions into a program code.

Here, it is proposed to use only one set the capacity of which is equal to the number of tasks in a system. And each element of this set is equal to the number of class a task belongs to. So, the set **B**, card $\mathbf{B} = I$ is the set of class membership of tasks, i.e. the element B_i of the set **B** presents the number of class the *i*-th task belongs to.

Using the introduced notations, lets us determine a common analytic form of the number of versions solving *i*-th task. If the element B_i of the set **B** is the number of class the *i*-th task belongs to, then an element of the set **S**, the index number of which is equal to B_i, determines the number of versions in a module solving *i*-th task. Therefore, this number can be written like this S_{B_i}.

Basing on that, we will introduce the Boolean variables X_s^i to describe the control implication of different module versions:

$$X_s^i = \begin{cases} 1, & \text{the } s-\text{th } (s=\overline{1,S_{B_i}}) \text{ version of module } B_i \text{ is used} \\ & \text{to solve the } i-\text{th } (i=\overline{1,I}) \text{ task,} \\ 0, & \text{the } s-\text{th } (s=\overline{1,S_{B_i}}) \text{ version of module } B_i \text{ is } \mathbf{not} \\ & \text{used to solve the } i-\text{th } (i=\overline{1,I}) \text{ task.} \end{cases} \quad (1)$$

Expanding the introduced variables into the *implication vector* is the head moment in applying pseudo-Boolean optimization methods to the considered systems design.

Since a vector component number is specified by only one index and we deal with two-index variables, it is necessary to establish an algorithm forming an implication vector and an algorithm determining the component indices of this vector. Following section contains the algorithms to convert a problem of optimal structure design for N-version systems to a problem of pseudo-Boolean optimization and vice versa.

2.1. Conversion Algorithms

The algorithm of an implication vector forming acts in the following way (see the scheme on fig. 2). The first component of an implication vector **X** describes the first version of a module to be involved in the solving of the first system task. If the software module which solves the first task has more than one version, the next component of vector **X** characterizes the second version of the first task module. In this way, all the versions of all software modules are overhauled.

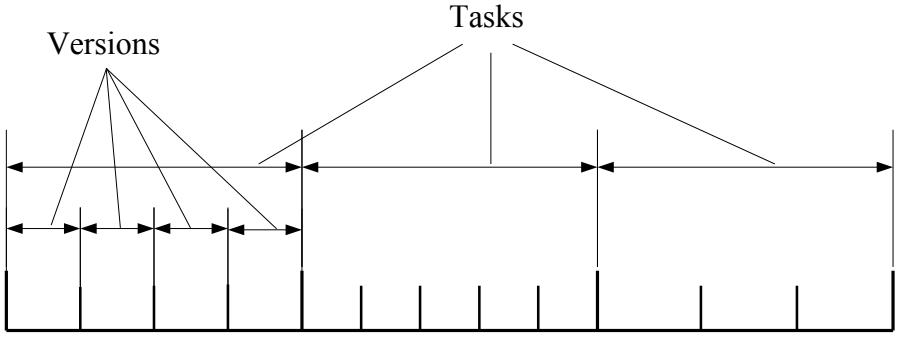

Figure 1. An example of the implication vector.

Hence, in order to determine the number of a vector **X** component, being aware of corresponding number of a task *i* and a number of version *s*, it is necessary to sum the number of versions in the modules solving the first (*i*-1) tasks and to add *s* to obtained value.

Analytically this conversion appears as follows:

$$pos = \begin{cases} s & , if \ \ i = 1, \\ \sum_{j=1}^{i-1} S_{B_j} + s & , if \ \ i > 1. \end{cases} \tag{2}$$

In order not to recalculate the first sum (the number of versions in the first (*i*-1) tasks) every time when optimizing a system, it would be better to count those sums depending on different *i* and to memorize them in an index vector:

$G_i = \sum_{j=1}^{i} S_{B_j}$, or in recurrent form

$$G_i = \begin{cases} S_{B_1} & , if \ \ i = 1, \\ G_{i-1} + S_{B_i} & , if \ \ i > 1. \end{cases} \tag{3}$$

Therefore, the value of the *i*-th component of vector **G** equals the number of versions in modules solving the tasks from the first up to the *i*-th. It results from this that the value of the last vector **G** component is equal to *n* – the implication vector dimensionality, i.e.

$$\sum_{i=1}^{N} S_{B_i} = n .$$

Once the index vector is introduced, the analytic record of a calculation of the implication vector component number takes the form of the following:

$$pos = \begin{cases} s & , if \ \ i = 1, \\ G_{i-1} + s & , if \ \ i > 1. \end{cases} \tag{4}$$

The reverse conversion task (a conversion of the implication vector component number to the numbers of task and version) consists of the consecutive determining of *i* and *s*. The flowchart of this algorithm is show on the figure 2.

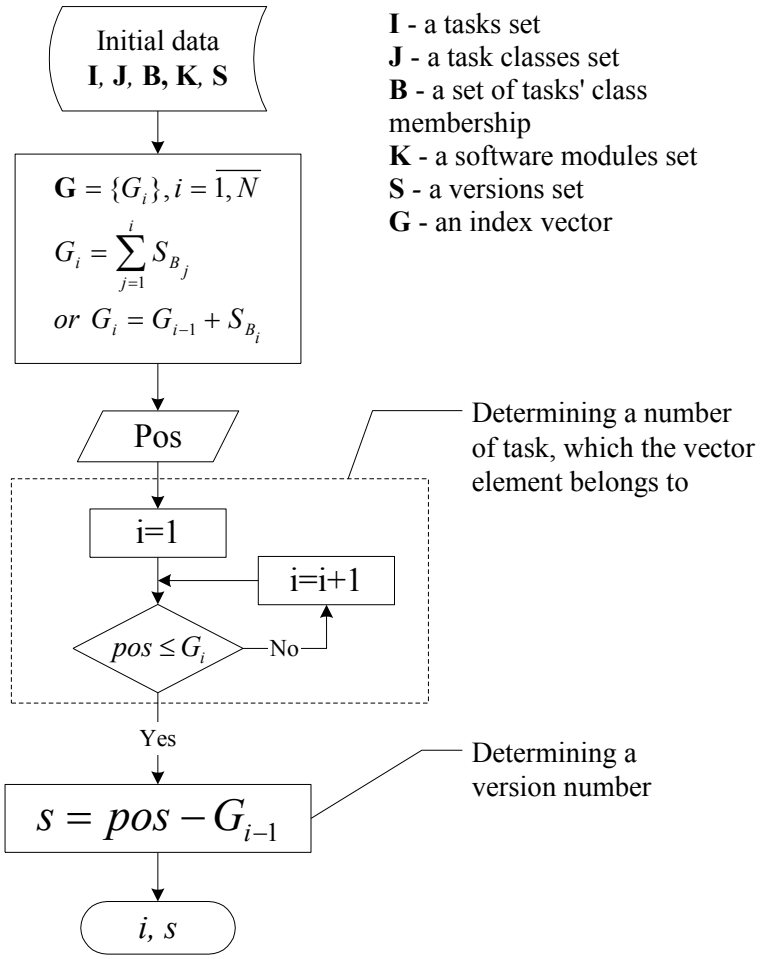

Figure 2. A conversion of the implication vector component number into the numbers of task and version.

Since the *i*-th element of the index vector equals the number of versions in modules solving the tasks from the first up to the *i*-th, the task number is determined by comparing the index of the implication vector component with the elements of the index vector. The comparison is being made from the first element of the index vector till the last one sequentially. And when the value of the parameter *pos* turns out to be less than or equal to the value of the index vector component, the required task number takes the value of this element number.

Then subtracting the number of versions in all the tasks from the first up to the (*i*-1)-th (equal to G_{i-1}) from *pos* we get a version number of software module solving the *i*-th task corresponding to *pos*.

The two presented algorithms [12] are the core of applying binary approach to solve the stated problem. Thus, having received the tools for a problem conversion, it became possible to use the methods developed within the confines of pseudo-Boolean optimization study. Some the features of the considered problem are discussed in the following section. Basing on this the conclusions about the relevant methods is made.

2.2. The mathematical Statement of the Problem

The converting algorithms considered above allow to describe the NVS structure in a form of a Boolean vector. As it was noticed previously, the optimal design of control system is held subject to different parameters: the reliability (it should be as big as possible), the cost (it should be as small as possible or at least it shouldn't exceed some limit), the allocation & scheduling and so on [14].

In terms of optimization theory, a system reliability function of a system structure is nothing else but an objective function. And conditions imposed on the system structure are the constraints set to limit the objective function domain [13]. Since we are able to associate a system structure with a Boolean vector, an objective function is a pseudo-Boolean one. And an optimization problem becomes a pseudo-Boolean one too.

In the framework of the presented model we will use a system reliability function as the objective function and the system cost will be the constraint imposed on the system [15-19]. In analytic form this problem can be written as follows:

$$\max R = \prod_{i=1}^{I} R_i ,$$

where $$R_i = 1 - \prod_{s=1}^{S_{B_i}} (1 - R_{B_i s})^{X_s^i}$$

subject to

$$\sum_{i=1}^{I} \sum_{s=1}^{S_{B_i}} X_s^i \cdot C_{B_i s} \le B .$$

Here, $R_{B_i s}$ and $C_{B_i s}$ are the reliability and the cost of the software version s from module which solves the task of class B_i

3. Conclusion

The problem of structuring an N-version software system is specified by the binary character, what made it plausible to apply the methods of pseudo-Boolean optimization. Within the limits of the discrete optimization a set of methods and algorithms has been proposed. Thus, having received the tools for a problem conversion, it became possible to use the methods developed within the confines of pseudo-Boolean optimization study.

References

[1] Laprie J-C et al 1987 Hardware- and Software-fault tolerance: definition and analysis of architectural solutions *Proceedings of the IEEEp* pp. 116-121.

[2] Anderson T, Barrett P A, Halliwell D N, Moudling M L 1985 An evaluation of software fault tolerance in a practical system *Proc. Fault Tolerant Computing Symposium* pp. 140-145.

[3] Scheer S, Maier T 1997 Towards dependable software requirement specifications *In: Daniel, P. (ed.) Proceedings of SAFECOMP* (New York)

[4] Kovalev I 1994 Optimization-based design of software of the spacecraft control systems *In: "Modelling, Measurement and Control* (AMSE Press), *B* vol.56 **1** pp. 29-34.

[5] Kovalev I V, Engel E A, Tsarev R Ju 2007 Programmatic support of the analysis of cluster structures of failure-resistant information systems *Automatic Documentation and Mathematical Linguistics* vol.41 **3** pp. 89.

[6] Keene S J 1994 Comparing hardware and software reliability *Reliability Review* **14(4)** pp. 5-21.

[7] Avizienis A 1995 The methdology of N-version programming *In: Software fault tolerance* (edited by M.R. Lyu, Wiley) pp. 23-47.

[8] Antamoshkin A, Schwefel H P, Torn A, Yin G and Zilinskas A 1993 System analysis, design and optimization. *Ofset Press* (Krasnoyarsk) 312 p.

[9] Antamoshkin A N et al 2013 Random search algorithm for the p-median problem *Informatica* **3(37)** pp. 127–140.

[10] Kazakovtsev L, Stanimirovic P, Osinga I, Gudima M and Antamoshkin A 2014 Algorithms for location problems based on angular distances. Advances in Operations Research. Articale ID 701267. 12 pages. - http://www.hindawi.com/journals/aor/raa/701267/

[11] Kovalev I 1998 Optimization problems when realizing the spacecrafts control *In: Advances in Modeling and Analysis* vol. 52 **1-2** pp. 62-70.

[12] Kovalev I V, Dgioeva N N, Slobodin M Ju 2004 The mathematical system model for the problem of multi-version software design *Proceedings of Modelling and Simulation, MS'2004 AMSE International Conference on Modelling and Simulation*, (MS'2004. AMSE, French Research Council, CNRS, Rhone-Alpes Region, Hospitals of Lyon. Lyon-Villeurbanne).

[13] Kovalev I, Grosspietsch K-E 2000 Deriving the optimal structure of N-version software under resource requirements and cost *Timing Constraints* (Proc. Euromicro' 2000, Maastricht, 2000, IEEE CS Press) pp. 200-207.

[14] Kovalev I et al 2013 The minimization of inter-module interface for the achievement of reliability of multi-version software *Proceedings of the 2013 International Conference on Systems, Control and Informatics* (SCI 2013), Venice, Italy, September 28-30 pp. 186-188.

[15] Ashrafi N et al 1994 Optimal design of large software-systems using N-version programming *IEEE Trans. on Reliability* vol.43 **2** pp. 344-350.

[16] Ashrafi N, Berman O 1992 Optimization models for selection of programs, considering cost and reliability *IEEE Trans. on Reliability* vol.41 **2** pp. 281-287.

[17] Ashrafi N, Berman O 1993 Optimization models for reliability of modular software systems *IEEE Trans. on Software Engineering* vol.19 **11** pp. 1119-1123.

[18] Kovalev I 1995 Optimal base software composition of the spacecrafts control system *In: "Advances in Modelling and Analysis, C*" (AMSE Periodicals) vol.47 **3** pp. 17-26.

[19] Kovalev I V et al 2002 Fault-tolerant software architecture creation model based on reliability evaluation *Advanced in Modeling & Analysis* (Journal of AMSE Periodicals) vol.48 **3-4** pp. 31-43.