Recursive algorithm for exhaustive search of possible multiversion software realizations with the choice of the optimal versions set

Roman Yu. Tsarev[1], Denis V. Gruzenkin[1] and Galina V. Grishina[1]

[1] Siberian Federal University, Russia
tsarev.sfu@mail.ru
gruzenkin.denis@good-look.su
ggv-09@inbox.ru

Abstract. N-version software is used all over the world as one of the approaches that can provide with the high level of reliability and software fault tolerance. The application of redundant module versions of software allows to obtain a correct result even if there is an error in the separate module versions. However, the program redundancy that can increase software reliability needs extra resources. It results in an optimization problem. There is a necessity for a certain variant of multiversion software realization i. e. such a modules versions set is required that demands less resources and guarantees high level of reliability simultaneously. The exhaustive search of all possible multiversion software realizations is carried out by the recursive algorithm proposed in the article.

Keywords: multiversion software, N-version software, reliability, optimization, exhaustive search, recursion, recursive algorithm.

1 Introduction

The problem of software reliability has been in existence as long as software exists. There is a demand for the software that guarantees a high level of reliability. It forces software designers to resort to such methods and tools that allow creating error and fault tolerance software. Since the beginning of 1960s when the software industry started to develop, a vast amount of approaches and methods of assessment and increasing of software reliability have been suggested [1].

There are some of the approaches that can be distinguished. They are based on time, information and program redundancy. The introduction of redundancy enables both to increase reliability and to guarantee fault tolerance of software. The program redundancy is implemented by two main approaches. They are N-version programming [2], [3] and recovery blocks [4], [5].

N-version programming has successfully proved itself particularly in such spheres as fault-tolerant control software for communications satellite system [6], railway interlocking systems [7], producing an architectural framework to automate and en-

hance application security [8], developing a N-version programming-based protection scheme for microgrids [9], web services systems [10].

The idea of N-version programming can be understood in the following way. The developing software has to solve a certain problem. The solution of this problem is the achievement of a certain goal. The problem is divided into some subtasks and the goal is achieved by finding solutions to them. On the conceptual level every subtask is solved by an appropriate module. The module is realized by means of several versions (multiversions) in order to ensure high reliability and fault tolerance of software. So, the modules and the software as a whole are becoming multiversion.

The introduction of the program redundancy in the form of the redundant modules versions ensures high reliability. It happens due to the fact that if one (or several) of the modules versions returns an incorrect result, the other versions return correct results nevertheless. When all results are obtained, it is necessary to analyze them and choose the one that is correct. It will be sent to all versions of the next module as input data. The analysis is carried out by a decision-making unit. The process of decision-making is usually realized by a voting algorithm [11], [12]. During the implementation of N-version software, the voting algorithm is implemented after every modules versions implementation. This algorithm defines the correct result of the operation of the whole module (i.e. all its versions).

The main problem of the application of any type of redundancy is the requirement for extra resources. The problem is connected with optimizing that could correspond to a higher level of reliability and at the same time to less amount of resources [13]. While developing N-version software the problem is defined as the choice of a modules versions set that could achieve the goals. The formal description of the problem is presented below.

2 The generation model of optimal versions set of N-version software

The conventional signs that are used are as follows

n – the number of subtasks that are required to be solved to achieve the goal;

i – a subtask number, $i = 1, 2, \ldots, n$;

m_i – the number of multiversions that are available for the solution of i-subtask;

j – an available multiversion number for the solution of i-subtask, $j = 1, 2, \ldots, m_i$;

R_{ij} – the reliability (the possibility of no-failure operation) of j-multiversion, solving i-subtask;

N_i – the set of all multiversions subsets with the power range from 1 to m_i;

N_i^* – the multiversions subset, $N_i^* \in N_i$;

$|N_i^*|$ – power N_i^*;

RN_i^* – the N_i^* reliability during i-subtask solution. It is equal to the possibility that no less than $|N_i^*|/2$ multiversions from a large number of multiversions N_i^* return a similar result;

v_i – the voting algorithm that analyzes the results of i-subtask solution, $i = 1, 2, \ldots, n$;

R_{vi} – the voting algorithm reliability v_i, $i = 1, 2, ..., n$;
C_{ij} – the cost of j-multiversion that solves i-subtask;
C_{vi} – the cost of the voting algorithm development v_i, $i = 1, 2, ..., n$;
C_i – the total cost of multiversions selected for the i-subtask solution;
x_{ij} – Boolean variable is equal to1, if j-multiversion is selected for i-subtask solution, and it is equal to 0 in an opposite case.

The selection problem of the optimal versions set of N-version software can be presented by dual-purpose nonlinear task of integer programming with Boolean variables:

$$\max \quad f_1(x) = \prod_{i=1}^{n} R_i R_{vi} \tag{1}$$

$$\min \quad f_2(x) = \sum_{i=1}^{n} \sum_{j=1}^{m_i} C_{ij} x_{ij} + \sum_{i=1}^{n} C_{vi} \tag{2}$$

on conditions that:

$$\sum_{j=1}^{m_i} x_{ij} \geq 1, i = \overline{1,n},$$

$$R_i = \sum_{N_i^* \in N_i} \left[\prod_{j \in N_i^*} x_{ij} \prod_{k \in N_i - N_i^*} (1 - x_{ik}) \right] RN_i^*,$$

$$RN_i^* = \sum_{j=|N_i^*|/2}^{|N_i^*|} \left[\sum_{M \in N_i^* || M| = j} \left\{ \prod_{k \in M} R_{ik} \prod_{l \in N_i^* - M} (1 - R_{il}) \right\} \right],$$

The objective function (1) maximizes the software reliability while the objective function (2) minimizes the software cost. As a rule, these two purposes conflict with each other.

The software cost includes the cost of every multiversion or decision-making unit realizing the voting algorithm only once. So, if one and the same voting algorithm is applied after several subtasks solutions, its cost is included only once.

The realization variants of N-version software can be a solution to this problem, i. e. a certain modules versions set (or a version set) of N-version software. It is possible to choose the variant after the exhaustive search of all possible variants of N-version software. In order to solve the task the algorithm is proposed.

3 The exhaustive search algorithm of all possible realizations of N-version software

If the i-subtask is solved by the i-module realized in the form of some multiversions then the number of i-module multiversions is equal to m_i. The i-module multiversions set is presented by the series $x_{i1}, x_{i2}, \ldots, x_{imi}$, where x_{ij} is equal to 1, if j-multiversion is used in the i-module of N-version software and it is equal to 0 in the opposite case (see Fig. 1).

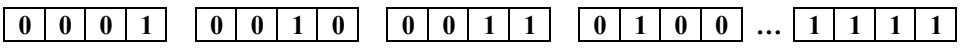

Fig. 1. Possible module version sets realized in four multiversions

There is an algorithm which is used to generate module version sets (Listing 1). The input data for this algorithm is the natural number N that varies in the range from 1 to 2mi – 1. Every N number in this range corresponds to one of the i-module version sets. The number of i-module version sets that is equal to 2mi – 1 is exhaustive. The i-module version sets is stored in a binary form in a one-dimensional array Bin, whose size is equal to mi.

```
assign for record the rightmost cell of the array Bin
execute in a cycle:
  record in the array Bin record cell the remainder of N
on division by 2;
  assign for record the cell that precedes the current
record cell;
  divide N by 2;
until the integer part from division is equal to 0;
record zeros in the remaining cells of the array Bin.
```

Listing 1. A binary array generation algorithm corresponding to the module version set

An exhaustive search algorithm of different realizations of N-version software has been developed. The algorithm is presented as a recursive function in pseudolanguage (Listing 2). On the basis of multiversions multitude the algorithm allows to consider all possible variants of the modules version sets of N-version software.

```
recursive_function (module number i, binary array Alt)
  for natural number N from 1 to 2mi – 1 execute in a cy-
cle:
    generate a binary array Bin for the current value N;
    copy the array Bin values in the i-line of the array
    Alt;
    if i-module is not the last then
      call for the recursive function (with the number of
      the next module (i+1) and the binary array Alt);
    carry out a required action on the current variant;
  the end of the cycle;
the end of the recursive function.
```

Listing 2. The algorithm of the recursive function performing an exhaustive search of all possible version set variants of N-version software.

The first function call is accompanied by sending the module number i = 1 as the first argument of the function. The current variant of the modules version set of N-version software is stored in a two-dimensional binary array Alt with the size n by max mi, i = 1, 2, ..., n.

The cycle is executed in the recursive function and the last line of the cycle implies any required actions that can be carried out on the obtained variant of version set of N-version software. There are some examples of such actions. They are the calcula-

tion of the reliability (1) or the cost (2) of the current variant of N-version software generation, the record of both the obtained variant version set and the values of characteristics corresponding to the variant into separate arrays for further application without the recurrence of the exhaustive search.

Fig. 2 shows the example of the recursive function execution during the exhaust search of different realizations of N-version software consisting of six modules that are realized by the following number of multiversions: $m_1 = 5$, $m_2 = 4$, $m_3 = 4$, $m_4 = 3$, $m_5 = 4$, $m_6 = 5$.

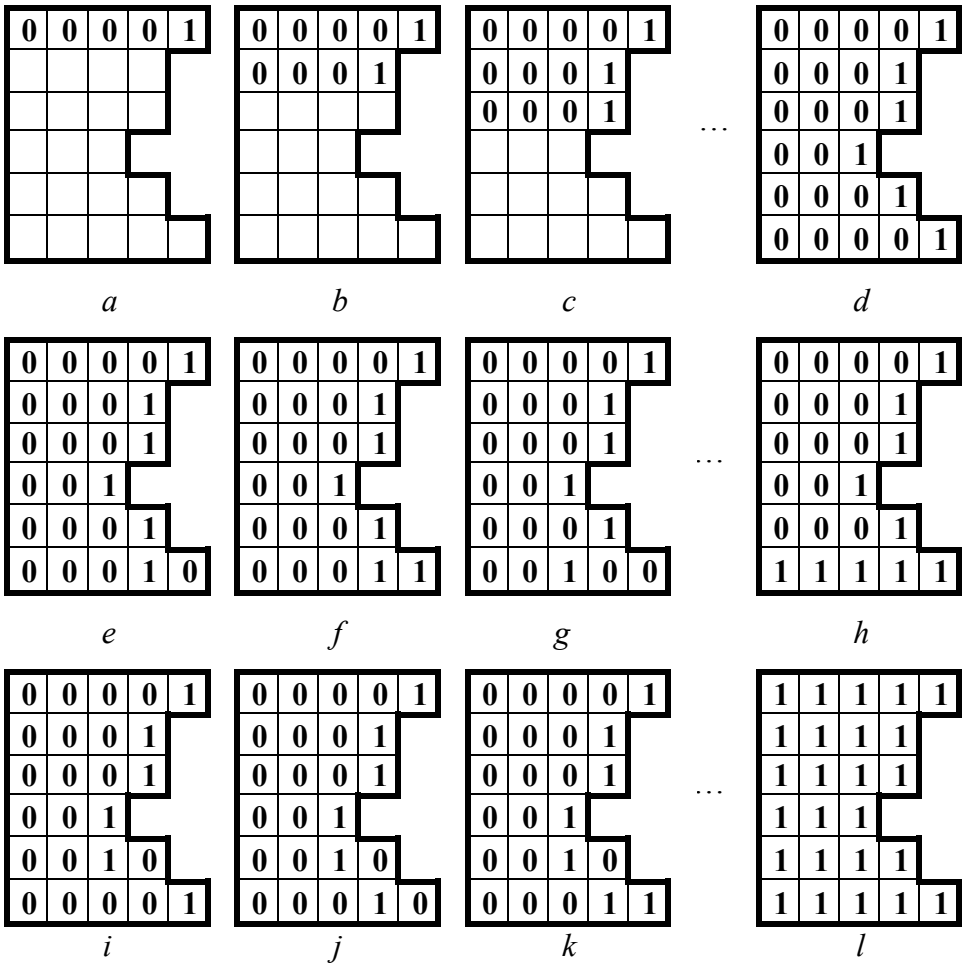

Fig. 2. Different realizations of N-version software

Every picture in Fig. 2 corresponds to one of the version sets of N-version software. The lines mean modules, the number of cells in a line mean the maximum possible number of the current module, the values in cells mean the values of the variable x_{ij} that reflects whether the current module version is applied ($x_{ij} = 1$) or not ($x_{ij} = 0$).

Fig. 2a-2d show the generation of the first variant of realization for every module from one multiversion according to the algorithm in listing 1. A new recursive function copy is called for every module. Fig. 2d shows the first variant of the version set of N-version software.

After that the sixth module version set is searched in the last recursive function copy (Fig. 2e-2h).

Then there is a return to the previous recursive function copy and another version is selected for the fifth module. The fifth line in Fig. 4i corresponds to this case. And again a new recursive function copy is called and the sixth module multiversions are searched (Fig. 2i-2k).

The last possible variant of the version set of N-version software is in Fig. 2l. All available multiversions are selected.

So, the proposed recursive algorithm allows to make a complete exhaustive search of all possible realizations of N-version software.

4 Conclusion

N-version software has a high level of fault tolerance and reliability due to the realization of program redundancy principle. In practice, reliable software modules are realized as a series of functionally equivalent versions. A software designer can include one or another module version into N-version software. The selection of versions is caused by the necessity to ensure a high level of reliability and to reduce the applied resources. The proposed recursive algorithm enables to make the exhaustive search of all possible realizations of N-version software that allows a decision maker to select an optimal variant.

References

1. Sommerville I. Software engineering, 9th edn. Addison-Wesley, Wokingham, England/ Reading (2010).
2. Avizienis A., Chen L. On the implementation of N-version programming for software fault-tolerance during program execution // Proc. IEEE Comput. Soc. Int. Comput. Software & Appl. Conf., COMPSAC ‘77. – 1977. – P. 149–155.
3. Gruzenkin, D.V., Chernigovskiy, A.S., Tsarev, R.Y. N-version Software Module Requirements to Grant the Software Execution Fault-Tolerance (2018) Advances in Intelligent Systems and Computing, 661, pp. 293-303.
4. Randell B., Jie X. The Evolution of the Recovery Block Concept // Software Fault Tolerance, Michael R. Lyu (ed.), Wiley. – 1995. – P. 1–21.
5. Kaur, R., Arora, S., Jha, P.C., Madan, S. Fuzzy multi-criteria approach for component selection of fault tolerant software system under Consensus Recovery Block Scheme (2015) Procedia Computer Science vol. 45 no. C, pp. 842-851.
6. Kulyagin, V.A., Tsarev, R.Y., Prokopenko, A.V., Nikiforov, A.Y., Kovalev, I.V. N-version design of fault-tolerant control software for communications satellite system (2015) 2015 International Siberian Conference on Control and Communications, SIBCON 2015 - Proceedings, art. № 7147116.

7. Eriş, O., Yildirim, U., Durmuş, M.S., Söylemez, M.T., Kurtulan, S. N-version programming for railway interlocking systems: Synchronization and voting strategy (2012) IFAC Proceedings Volumes (IFAC-PapersOnline), pp. 177-180.
8. Malaika, M., Nair, S., Coyle, F. N-Version architectural framework for application security automation (NVASA) (2014) CrossTalk, 27 (5), pp. 30-34.
9. Hussain, A., Aslam, M., Arif, S.M. N-version programming-based protection scheme for microgrids: A multi-agent system based approach (2016) Sustainable Energy, Grids and Networks, 6, pp. 35-45.
10. Peng, K.-L., Huang, C.-Y., Wang, P.-H., Hsu, C.-J. Enhanced N-Version programming and recovery block techniques for web service systems (2014) International Workshop on Innovative Software Development Methodologies and Practices, InnoSWDev 2014 - Proceedings, pp. 11-20.
11. Durmuş, M.S., Eriş, O., Yildirim, U., Söylemez, M.T. A new voting strategy in Diverse programming for railway interlocking systems (2011) Proceedings 2011 International Conference on Transportation, Mechanical, and Electrical Engineering, TMEE 2011, статья № 6199304, pp. 723-726.
12. Rezaee, M., Sedaghat, Y., Farmad, M.K. A confidence-based software voter for safety-critical systems (2014) Proceedings - 2014 World Ubiquitous Science Congress: 2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing, DASC 2014, статья № 6945688, pp. 196-201.
13. Gruzenkin, D.V., Grishina, G.V., Durmuş, M.S., Üstoğlu, I., Tsarev, R.Y. Compensation model of multi-attribute decision making and its application to N-version software choice (2017) Advances in Intelligent Systems and Computing, 575, pp. 148-157.