Multi-attribute decomposition technique for fault tolerant multiversion systems

I V Kovalev¹,²,³,⁴,⁶, N A Testoyedov⁴,⁵, D I Kovalev²,⁶, V V Losev⁴, M V Saramud¹,⁴, E N Golovenkin⁴,⁵ and I A Maksimov⁵

¹ Siberian Federal University, 79, Svobodny pr., Krasnoyarsk, 660041, Russia
² Krasnoyarsk State Agrarian University, 90, Mira pr., Krasnoyarsk, 660049, Russia
³ China Aviation Industry General Aircraft Zhejiang Institute Co., Ltd, China
⁴ Reshetnev Siberian State University of Science and Technology, 31, Krasnoyarsky Rabochy Av., Krasnoyarsk, 660037, Russia
⁵ JSC “Academician M F Reshetnev Information satellite systems”, 52 Lenin street, Zheleznogorsk, Krasnoyarsk region, 662972, Russia
⁶ Krasnoyarsk Science and Technology City Hall of the Russian Union of Scientific and Engineering Associations, 61, Uritskogo street, Krasnoyarsk, 660049, Russia

E-mail: kovalev.fsu@mail.ru

Abstract. The article proposes an original method for decomposition of fault-tolerant multiversion systems based on multi-attribute analysis. Decomposition into functional subsystems is carried out taking into account the generality of functions performed by software modules. In this work, the attribute of the importance of functions is determined based on the general function of the unmanned aerial vehicle and the set of tasks that it must perform. The division of software modules into types allows you to determine possible ways to improve reliability that are applicable to them. These are, for example, multi-version programming, recovery blocks, $t/(n-1)$-variant programming.

1. Introduction

When designing complex software systems [1-6], the problem arises to decompose the system according to certain criteria. As a possible solution, the authors proposed a methodology that allows, during development, to identify modules and elements of the developed (software) system that have different attributes.

Possible attributes can be individual elements suitable for redundancy - software modules that have different multiversions, functions performed by these modules, their types and phenomena that occur when they fail.

The division of software modules according to their functions is necessary to ensure highly reliable operation of the most important modules, and, consequently, functions.

Decomposition as a separation process allows us to consider any investigated system as complex, consisting of separate interconnected subsystems, which, in turn, can also be divided into parts. During decomposition, each division forms its own level.

The original system is at zero level. After its division, first-level subsystems are obtained. The separation of these subsystems or some of them leads to the emergence of subsystems of the second
level, etc.

Functional decomposition is based on the analysis of system functions. This raises the question of what the system does, regardless of how it works. The basis for the division into functional subsystems is the generality of the functions performed by the modules.

2. Problem statement
Within the framework of this article, the importance of functions is assumed based on the general function of an unmanned aerial vehicle (UAV) and the set of tasks that are assigned to it [7-9].

Moreover, one and the same module can perform several functions at once, and, therefore, it is taken into account how many functions are performed, and which of them is the most important.

When analyzing the importance of the onboard software (OBS) of the UAVs, it should be borne in mind that many software modules perform system tasks, and not applied tasks, which means that their importance should be given more.

Applied functions can be divided into major and minor. Such division is based on the fact that the termination of the main function leads to an accident and destruction of the UAV, while the termination of the secondary one leads to a malfunction in the operation of the vehicle, a malfunction that does not lead to an accident.

Software modules that perform system functions are assigned the highest priority, modules involved in the execution of the main function are assigned a high priority, and those that perform secondary functions are assigned a low priority.

The division of software modules into types allows you to determine possible ways to improve reliability that are applicable to them.

Possible types of modules can be modules suitable for different types of design with the use of redundancy: multiversion programming [10-13], recovery blocks [13-15], t/(n-1) - variant programming [16-18].

3. Application of the t/(n-1) algorithm for multiversion systems
Since the t/(n-1) algorithm is non-trivial, it is necessary to consider it in more detail. This algorithm was proposed by J. Xu of the University of Newcastle upon Tyne and is based on t/(n-1) diagnosability [16]. For simplicity, we will call it the t/(n-1) decision-making algorithm. The essence of the algorithm is not in voting all versions of the versions, but only in comparing some of them sufficient for making a decision.

Let us consider the example of a system with the number of versions N = 5 and the maximum number of errors t = 2, that is, consider the 2/(5-1) option. If the number of errors does not exceed t, the algorithm guarantees the selection of the correct variant from N version outputs. However, even with a large number of incorrect version outputs, the system will not necessarily select the wrong one. With a certain probability, the choice of the correct output will occur, but this is no longer guaranteed [19].

Let us consider the algorithm in more detail using the example of option 2/(5-1). The outputs of four versions are compared in pairs - 1 with 2; 2 with 3; 3 with 4. We get three results of comparisons $\omega_{12}$, $\omega_{23}$, $\omega_{34}$, equal to 0 if the outputs are the same and 1 if they differ.

Based on only these three outputs of the comparators, the algorithm decides to switch the output between outputs 1, 4 and 5 of versions. That is, versions 2 and 3 are used only for comparison, the values of their outputs are never used as the system output. More clearly, the scheme of work can be studied in figure 1.
Figure 1. Architecture of the $t/(n-1)$ algorithm for $n = 5$ and $t = 2$.

As can be seen from figure 1, the decision tree in the $t/(n-1)$ algorithm is relatively straightforward. In the case of five multi-versions, only the results of three paired comparisons of the outputs of the four versions are needed to make a decision (correct control of the output switch). The release value of the fifth version is not used for decision making. The output switch control logic based on the comparison results for $n = 5$ is shown in table 1.

Table 1. Possible selections based on comparator outputs for $n = 5$.

| $\omega_{12}$ | $\omega_{23}$ | $\omega_{34}$ | Presumably correct versions |
|---------------|---------------|---------------|-----------------------------|
| 0             | 0             | 0             | 1, 2, 3, 4                  |
| 0             | 0             | 1             | 1, 2, 3                     |
| 0             | 1             | 0             | 5                           |
| 0             | 1             | 1             | 1, 2                        |
| 1             | 0             | 0             | 2, 3, 4                     |
| 1             | 0             | 1             | 5                           |
| 1             | 1             | 0             | 3, 4                        |
| 1             | 1             | 1             | 5                           |

Based on figure 1 and table 1, we can conclude that, with relatively reliable versions, in most cases the comparators will return $(0; 0; 0)$ and the execution value of the first version will be output.

It can also be concluded that there is no need to execute the fifth version every time. And to do this only in the case of the corresponding values of the comparison results, when it is the result of the fifth version ($(0; 1; 0), (1; 0; 1), (1; 1; 1)$) that must be submitted to the output. This fact reduces the average load required by the execution environment of the multiversion software to work, since in the vast majority of cases 4 out of 5 versions will be calculated. The decision-making algorithm itself is also less resource-intensive compared to voting, especially with weighted modifications, where all versions are executed at each vote, classes are created and weights are calculated for each of them.

For $t/(n-1)$ with $n = 5$, only three simple binary output comparisons are needed.

Further, an unambiguous, a priori given selection of the output for one of eight possible combinations of the comparator outputs is carried out (table 1).

The new scheme of the algorithm is shown in figure 2 and table 2. As can be seen from the presented data, when adding an additional version, it is necessary to add a comparator. Consequently, we already have 4, not 3 Boolean outputs, necessary for making a decision. Accordingly, the number of possible combinations of comparators doubles - from 8 to 16.
| \( \omega_{12} \) | \( \omega_{23} \) | \( \omega_{34} \) | \( \omega_{45} \) | Presumably correct versions |
|--------------|--------------|--------------|--------------|---------------------|
| 0            | 0            | 0            | 0            | 1, 2, 3, 4, 5       |
| 0            | 0            | 1            | 0            | 1, 2               |
| 0            | 1            | 0            | 0            | 3, 4, 5            |
| 0            | 1            | 1            | 0            | 6                  |
| 1            | 0            | 0            | 0            | 2, 3, 4, 5         |
| 1            | 0            | 1            | 0            | 6                  |
| 1            | 1            | 0            | 0            | 3, 4, 5            |
| 1            | 1            | 1            | 0            | 4, 5               |
| 0            | 0            | 0            | 1            | 1, 2, 3, 4         |
| 0            | 0            | 1            | 1            | 1, 2, 3            |
| 0            | 1            | 0            | 1            | 6                  |
| 0            | 1            | 1            | 1            | 1, 2               |
| 1            | 0            | 0            | 1            | 2, 3, 4            |
| 1            | 0            | 1            | 1            | 6                  |
| 1            | 1            | 0            | 1            | 3, 4               |
| 1            | 1            | 1            | 1            | 6                  |

Let's consider the impact on the algorithm operation of adding another version, up to \( N = 6 \).

**Figure 2.** Architecture of the \( t/(n-1) \) algorithm for \( n = 6 \).

4. Discussion and conclusion

It can be concluded that the computational complexity \( t/(n-1) \) - of the algorithm is extremely sensitive to the number of versions. With each additional version added, the computational complexity doubles. Accordingly, with 2 versions, 4 times, with 3 additional versions, 8 times, etc. Since the main advantage of the \( t/(n-1) \) - algorithm is precisely its low resource intensity, its use is most justified when the number of versions does not exceed \( N = 5 \).

Let's return to the description of the proposed technique. The last considered attribute – the phenomenon, implies specific failure options. One of these specific failures can be a dangerous failure.

In the case of UAVs, the danger is primarily represented by abnormal contact with a surface or other body. To identify the causes of such failures, event structure diagrams are built that determine which
modules failures can lead to a dangerous phenomenon. Dangerous and safe failures of a particular software module are determined by whether this module affects the movement of the object. The difference in hazards in their influence can be expressed in the form of a certain value, coefficient. A value of C should be determined for each module, quantifying the danger posed by failure. A possible measure of this value for a module controlling a certain direction can be the speed of movement in that direction. That is, the higher the possible speed of movement, the more dangerous the failure becomes.

References

[1] Goloubeva O, Rebaudengo M, Reorda M S and Violante M 2006 Software-Implemented Hardware Fault Tolerance (Springer Science+Business Media LLC)
[2] Lipaev V V 2005 Maintenance and configuration management of complex software tools (SINTEG, Moscow)
[3] Lipaev V V 2005 Problems of development and quality assurance of large-scale software tools Programming 31(1) 73-7
[4] Lipaev V V 2015 Basic concepts, factors and standards that determine the quality of large-scale software tools (Direct-Media, Moscow-Berlin)
[5] Losev V V et al. 2020 J. Phys.: Conf. Ser. 1679 032099
[6] Losev V V et al. 2020 The research of multiversion methods for improving the fault tolerance of software systems for monitoring the parameters of technological processes IOP Conf. Ser.: Earth Environ. Sci. 548(3) 032042
[7] Kovalev I V et al. 2019 J. Phys.: Conf. Ser. 1399 055095
[8] Kovalev I V et al. 2019 J. Phys.: Conf. Ser. 1399 055100
[9] Kovalev I V, Saramud M V and Losev V V 2020 Simulation environment for the choice of the decision making algorithm in multi-version real-time system Information and Software Technology 120 106245
[10] Nasyrov I R and Stashkov D V 2015 Multiversion programming in automatic control systems for technological processes Priority research areas: from theory to practice 16 75-8
[11] Algirdas Avizienis 1995 The Methodology of N-Version Programming: Software Fault Tolerance (John Wiley & Sons)
[12] Avizienis A 1985 The N-Version Approach to Fault -Tolerant Software IEEE Transactions on Software Engineering SE-11(12) 1491-501
[13] Gruzenkin D V, Nivikov O S and Suchanova A V 2016 Multipersonal Software and Recovery Blocks - Two Ways to Protect Against Errors New Science: From Idea to Results 11-2 72-5
[14] Tzarev R Yu et al. 2015 Model of the formation of an optimal software system according to the recovery block scheme with coordination Modern problems of science and education 1(1) 297
[15] Tzarev R Yu 2015 Algorithm for the formation of a software system according to the recovery block scheme with coordination based on fuzzy logic Modern problems of science and education 2 178-9
[16] Xu J 1991 The t(n-1)-diagnosability and its applications to fault tolerance Fault-Tolerant Computing (The Twenty-First International Symposium) 496-503
[17] Xu J and Randell B 1997 Software Fault Tolerance: t/(n-1)-Variant Programming IEEE Transactions on Reliability 46(1) 60-8
[18] Losev V V et al. 2020 J. Phys.: Conf. Ser. 1679 052088
[19] Kovalev I V et al. 2017 Comparative Tests of Decision Making Algorithms for a Multiversion Execution Environment of the Fault Tolerance Software European Conference on Electrical Engineering and Computer Science - EECS (Bern) 211-7