Implementation of multiversion software based on an object-oriented approach

I V Kovalev, D I Kovalev, N S Chefonov, N A Testoedov and E N Golovenkin

1 Siberian Federal University, 79, Svobodny pr., Krasnoyarsk, 660041, Russian Federation
2 Reshetnev Siberian State University of Science and Technology, 31, Krasnoyarsky Rabochy Av., Krasnoyarsk, 660037, Russian Federation
3 Krasnoyarsk State Agrarian University, 90, Mira pr., Krasnoyarsk, 660049, Russian Federation
4 JSC Academician M. F. Reshetnev «ISS» Zheleznogorsk, Krasnoyarsk region, Russia

E-mail: nikita.chefonov@mail.ru

Abstract. The article discusses the concept of redundant programming and its implementation based on an object-oriented approach. Three components of multiversion programming, which are basic in the synthesis of fault-tolerant software, are identified. In comparison with the standard software life cycle, the basic procedures of the multiversion programming methodology are considered. The concept of abstract state of the multiversion software components version is based on the object-oriented approach.

1. Introduction

Three forms of redundancy can be distinguished: hardware, software, and temporary. Hardware redundancy should prevent hardware failures, while software and temporary redundancy should prevent accidental failures. Software redundancy includes the usage of special control programs and control codes. Temporary redundancy implies the repeated execution of programs for particular functions, the reliability of which should be especially high. As you know, the concept of redundant programming was introduced by A. Avizhenis [1, 2].

The multiversion of the execution of software modules implies the independent generation of N * 2 functionally equivalent programs (multiversions) in accordance with the identical initial specifications. For these N versions of programs, competitive execution tools are provided, during which the programs generate the comparison vectors (c-vectors) at the designated control points (cc-points). The comparison vectors’ components and control points were previously determined at the stage of the initial specifications [3-5].

2. Basic elements of multiversion software

Analyzing the results of recent researches in the field of the methodology under consideration, we can identify three components of multiversion programming, which are basic in the synthesis of fault-tolerant software [6-12].
1. The process of the initial specification and multiversion programming (N-version programming - NVP), which involves the guarantee of the independence and functional equivalence of N individual programming processes.

2. Result (multiversion software - N-version software or NVS) of an NVP process for which competitive execution tools with specified control points and c-vectors are available.

3. External support tools for the execution of software multiversions (N-version executive or NVX), which provide NVS execution and involve decision-making algorithms at control points.

All three elements of NVP are united by the goal of independent and competitive generation of N * 2 functionally equivalent software components, which will be executed and a decision will be made on the correct functioning of the software. These elements are completely unnecessary when only one version of the program is used. So, the concept of NVP (as an approach to creating fault-tolerant software) introduces such new areas of research in the field of software engineering as NVP, NVS, NVX, the definitions of which we have already considered.

In the process of analyzing the structure of multiversion software, it should be noted that a significant difference between multichannel hardware and fault-tolerant software is that a simple duplication of the same elements during design is effective against random errors of a physical nature in technical systems, but not effective for software failures. Simple copying is also copying currently undetected (“sleeping”) errors in programs, so N software components must be developed separately and independently, which corresponds to the concept of redundant software design.

Two main areas of fault-tolerant programming can be distinguished: multiversion programming - NVP and programming with a recovery block - RB. Common is the presence of two or more redundant components (in NVS they are called versions, and in the RB approach, they are alternatives combined with the acceptance test). The main difference is in the method by which a decision is made on the results of the programs. Obviously, combinations of basic schemes are possible, as well as an extension of the functions of the EE environment both for the NVP process (for example, a multilevel decision algorithm, a common or individual RB block for a multiversion program), and for the RB method (for example, cache memory for software component recovery, multilevel hierarchical acceptance test, etc.).

The implementation of the recovery element in both methods has a limitation: in RB, errors that will not be detected by the acceptance test (it is possible) will not lead to switching to recovery during software operation; in NVS, switching to recovery will be incorrect if most versions provide an erroneous result at the recovery point.

However, the presence of N>2 versions of redundant software components does not yet involve the stability of this software against failures, since the execution environment (EE) for NVS is required. Both individual software versions and the EE environment must meet the following requirements:

EE-environment should provide functions to support the execution of N>2 versions of software components in error control mode;

Individual specifications of software versions should determine the ways to achieve fault-tolerant execution that are supported in the EE environment;

The best NVP outcome should minimize the probability of not detecting (or not returning for RB) the erroneous execution of a multiversion software component, unlike a conventional software system.

After defining the main elements of multiversion programming, let’s note not less important design stages [6–9]. These stages relate, firstly, to the versions’ specification of NVS software systems (taking into account the requirements for their further competitive execution in the EE environment using external NVX execution support tools); These stages relate, firstly, to the specification of versions of NVS software systems (taking into account the requirements for their further competitive execution in the EE environment using external NVX execution support tools); secondly, to the immediate programming stage, taking into account that the main requirement of NVP is the maximum independence of the process outcomes and, accordingly, software versions; thirdly, to the creation of NVX-systems of the EE-environment for executing software components of multiversion software.
both with high reliability and with effective indicators of runtime, which is especially important for using NVS-system blocks in software of autonomous control systems operating in real time [13-16].

The solution to the first of these problems is based on the specification system application of multiversion software called "V-Spec", which defines:
- a function that must be performed taking into account restrictions on execution time, initial data, and the initial state of the software version;
- requirements for internal error control and testing module versions;
- requirements for a variety of programming;
- control points (cc-points) at which the decisive algorithm of the NVX environment will be used to control all versions;
- recovery points (r-points) at which the RB block is used to restore erroneously executed software versions;
- parameters of the decisive algorithm of the NVX environment used at each cc- and r-point;
- possible outcomes of execution in the NVX environment, including the case when the decisive algorithm is not able to make a decision.

Regarding NVX-systems supporting version execution, it can be noted that their main functions should provide the following:
- a decisive algorithm or many algorithms;
- input data sequences for all versions;
- intraversion communication;
- synchronization of version execution (if the hardware and software complex allows) under strict time constraints;
- "local" control of the execution of each version;
- "global" execution of multiversion software in conjunction with decisive functions for recovering erroneous versions by r-points (local RB-block as part of NVX);
- the interactive user interface of the NVX system.

3. Multiversion software life cycle
Let’s consider successively the procedures of the multiversion programming methodology in comparison with the standard SDLC [8, 9].

3.1 System Specification Phase - NVX Formation (NVS - Supervisor)
In this phase, the external runtime support environment (NVX) of the multiversion software should be determined taking into account the evolution of the entire system and the requirements for the user interface. There are three aspects to this step.

1. Choosing a way to implement the NVX environment and resource allocation.

   Obviously, the entire system architecture is formed at the phase of system specifications and, thereby, the software configuration, the number of versions and the way they interact are selected.

2. Maintenance support tools and toolkit.

   Existing NVX environments can be adapted and accommodated to new applications, it should be noted that the NVX environment can be organized in the form of both software and hardware (combinations are possible). The basic functions of NVX as an NVS supervisor include: decisive algorithm (or many algorithms); means of ensuring compliance with the initial conditions of all versions; interversion interaction; version synchronization and fulfillment of time limits; "local supervisor" for each version; general competitive execution and crucial functions for recovering erroneous versions; user interface for monitoring, entering control commands, collecting information for multiversion execution of programs.

3. Choice of hardware structure.

   Regarding this aspect, let’s only note that in some cases for the execution of the software NVS-system, a specially designed for these purposes processor (or several processors) may be required.
This is especially important if NVS is used in real-time systems with severe restrictions. A hybrid structure including fault-tolerant hardware is possible.

3.2 The phase of developing software requirements - the choice of ways to achieve and the degree of diversity of software versions

At this phase of the software life cycle, the main task is to avoid the coincidence of programming methods, potentially affecting the independence of multiversion errors. There are three stages of selection.

The distribution between random diversity and diversity by presenting established requirements.

If the first corresponds to the personal qualities of a programmer (way of thinking, programmer individualism), then the second, on the contrary, is a formal way to establish requirements for a programming language, ways to ensure reliability, etc.

Evaluation of the required diversity of the software design process (for most phases of SDLC)

In fact, there are four phases that contribute to diversity: specification, design, coding and testing (various programming languages, tools, algorithms, methodologies).

3. Diversity as design constraints.

It is installed through the last of the possible methods, which is specific for each software development, and a typical design constraint is cost, a strict sequence of work, stage independence, etc.

3.3 Software specification phase - installation of error detectors and recovery algorithms

The specification of versions of NVS-software is carried out using the "V-spec" [77], which fully establish the functional requirements for the software module and at this phase an error detector and recovery algorithm (s-point, s-vector, RB- block) must be selected. When these parameters are consistent in each version of the software, it is necessary to try to avoid factors that limit diversity, for example, use different types of "V-spec" that establish the same requirements.

3.4 Design and coding phase - maintaining the C'D protocol of NVS development

In this phase, multiprogramming of competitive software versions starts in accordance with the V-spec specifications. A strict protocol for interaction and documentation is defined (C'D protocol). One of the positive consequences of the protocol is that it provides a sufficient set of details when searching for the reasons for the erroneous execution of the NVS-component at a later stage of software operation.

3.5 Test phase - NVS pre-operation phase

At this stage, during sequential testing of \(N > 2\) versions, the software composition can be supplemented if the probability of erroneous execution is high due to the issuance of an equally incorrect result by most multiversion components. It is also possible to attract experts to determine the "model" version of the software based on the results of a test run.

3.6 Assessment and acceptance phase - fault tolerance assessment

Evaluation of attributes characterizing software fault tolerance implies analytical modeling, simulation, experiments (or combinations of these methods). It is necessary: to determine the criteria for evaluating fault tolerance; evaluate signs of diversity; make preliminary conclusions, for which two aspects are considered regarding the selection of an appropriate model for evaluating software reliability and determining the quantitative indicators of the evaluation method, respectively.

3.7 Maintenance and development phase - selection and implementation of the NVS maintenance method

The essence of the NVS support phase is characterized by the following specifics. Firstly, functionally, the NVX environment must be adequately guaranteed against failures and errors during
this operational phase. Parts of the NVS supervisor that are critical in terms of reliability can be protected by applying a multiversion methodology. Abnormal situations recorded during the maintenance process are the subject of further research. Secondly, the modification of the NVS-software system must follow the design paradigm of multiversion programming. When adding software functions, this can concern directly the multiversions, as well as the components of the NVX environment (decision algorithms, acceptance tests, etc.).

An analysis of the contents of these methodological procedures allows us to conclude that at present only the multiversion fault-tolerant programming approach that we are considering is a possible alternative to testing and proving the correctness of programs, providing a high level of reliability of critical component failures and errors. It is this methodology that provides a guarantee that errors in one of the versions will not lead to disruption of the management of complex objects, which are characterized by stringent requirements for reliability and autonomy of operation.

Finally, at the stage of supporting fault-tolerant software, during its development and modification, the main focus is on the development of V-Spec specifications using high-level languages, which allows direct management specialists to be involved in the development of multiversion components and to focus on the quality of reliable development requirements software. It is also significant that when modifying a software system, adherence to the multiversion paradigm is mandatory, and when developing and adding software functions, this applies to both multiversions and the components of the NVX environment.

4. Recovery of failed components in multiversion object-oriented programming
The object-oriented approach allows you to create a redundancy scheme that implements error detection, which allows you to not only control, but also restore failed versions of components, as well as prevent the spread of errors. To implement such mechanisms, a special class must be created, the methods of which will map the internal state of objects to a more abstract level [6].

The functioning of N-versioned software depends on the runtime, which is responsible for ensuring that the component versions are executed in parallel, synchronized, passed their execution results to the decisive block, etc. Using the concepts of an object-oriented approach, we can embed a certain multiversion environment directly into a system that will be transparent to the user and can be easily used for any component [17-19].

4.1 Approach to building software redundancy
Fundamental principles for the development of modern redundant software:
1) Redundant components must be developed using structural programming.
2) It must be possible to apply redundancy schemes recursively. This allows any sub-component of a system that is recursive for any component to be redundantly developed.
3) Another important point is the encapsulation of redundancy. Complex systems should be built so that component redundancy is hidden from external components. This means that the redundancy control, as well as the redundancy itself controlled in the software, must be hidden inside the components, which provide the application with only a normal interface.
4) Independence of component versioning. The scheme itself must support independent component development.
5) The control and support of redundancy should be separated as much as possible from the application code and components.

4.2 Software Redundancy and Object Oriented Approach
The usage of N-version programming in the development of classes and objects that are elements of the architecture of object-oriented software can make it quite simple to implement all the features listed above. This approach allows you to apply all the advantages of OOP when developing software without dividing the system structure into complex interfaces of interaction of system elements.
When developing N-versioned software, in the context of OOP, all elements of the scheme should be implemented in terms of OOP: redundant components, versions, decision blocks, control mechanism (manager or supervisor), input / output parameters, internal data representation. The main idea is to use class redundancy when developing them according to the same specification. Such classes are called DI-classes (Diversity-Implemented).

Object version developers use the same specifications. Versions are hidden inside DI classes. In addition, DI classes have hidden objects inside - a decision module and a manager (control mechanism).

![Diagram](image)

**Figure 1.** Representation of the DI class.

All DI classes must be inherited from the meta class, which implements all the mechanisms for constructing and controlling redundancy in the form of hidden class methods.

When the DI method of the object is called, the manager method calls this method in all N-versions for parallel execution, expects the results and sends them to the decision module - the method for comparison. The solver returns the correct result if there is a majority among the solutions. If all the results are different, then the manager method signals an error.

### 4.3 General principles for the implementation of multiversion redundancy

The first scheme for recovering components after a failure appeared a long time ago - CER (Community Error Recovery), proposed by Avizhenis [2]. The main idea was that the state of the correct versions can be used to restore failed versions. This idea has been used and developed for a long time. The main drawback of this idea is that it is impossible to roll back all versions of a component or at least one of them (failed).

Another approach is to use exception management within each failed version. But this does not guarantee that the states of all versions are the same without knowing the state of the correct version.

In CER, the correct version state is used to restore the failed version to a state in which it could not fail. This is a special case of "forward recovery" after failure (forward error recovery). The only important resemblance to the "restore back" is that the state of the component version is set to an already known state. In fact, this recovery "rolls back" the state of the component version forward, to
the first correct state obtained from the correct component. The main problem with this recovery is the complexity of creating display functions to convey the internal states of component versions.

In CER, redundancy units are not clearly defined, they are not structural units of the system - these are N independently developed programs that consist of modules of the same functionality executed in the same order. Built programs to the same specifications. This approach is not recursive, it has version restrictions. In this approach, version redundancy is not encapsulated. The number of control points should be the same for all versions, and they should include the same set of data types. The development of N programs has its limitations, for example, they must contain the same modules, the execution of which ends at the recovery points.

CER uses two-level error detection, but with some features. At recovery points (r-points) the same thing is done as at comparison points (cc-points). At comparison points, data arrays are compared - comparison vectors (cc-vectors), which should be the same for all versions of the component. It is not always known how to select these vectors. However, this data is very important because they are used for recovery. Recovery consists in installing the correct data in the comparison vector for the failed version. Choosing data sets that would serve both for detection and for version recovery is a quite difficult task.

4.4 Modern approaches to the implementation of multiversion redundancy
CER was the first of the approaches using two types of points: cc-points (control points) and r-points (restore points), in which version synchronization occurred - this is a two-level recovery. Comparison points are used not only for comparing comparison vectors (cc-vectors), but also for partial restoration of versions: each version sends a result to the decision module, including the comparison vector. The comparison vector is used for partial recovery since the data in this vector represents a subset of version states. Each version consists of several modules executed sequentially, each module has several control points inside.

Recovery points are installed between modules and are used to completely restore the version of a module or component. At these points, the final state of each version is displayed in an intermediate format common to all versions, so that the results can be compared and transferred to the failed version for recovery.

The concept of the abstract state of a version, presented in object-oriented terms, has recently been developed.

4.5 Object-oriented software component version recovery
The object-oriented scheme uses two-level failure tracking: both output variables and version states. The first ensures that no erroneous information is distributed outside the class. The second is used to prevent the spread of failure within the class. Here, the internal states of versions are compared through a special unified representation called the abstract state of the version. The version designer must override the DI method of the meta class, which calculates the current abstract state of the version and returns the result to the manager method.
The manager method transfers the abstract states of all versions to the decision module (DI meta class method). This assumes that the state of the failed version is encapsulated inside the version of the object, so this is enough to restore this object. External specifications are the same for all versions and it is assumed that they are correct and failures occur due to errors made when programming versions. It is assumed that the abstract mapping method of the version state is correct. The abstract state of a version can be represented as a lot of different data representing various aspects of a particular state of an object. This type of data is used by the manager method to control version recovery. Data is hidden inside the DI object. Their type depends on the specific application and can be executed as a class that includes only access methods. Therefore, only one instance of the class is sufficient for each DI object. The decisive method is quite simple, because it should not contain complex calculations - it compares only the outputs of versions that should be of the same type.

In some applications, manager methods can be made more flexible to implement more flexible version control policies and reduce development costs.

One of these policies is to compare the input states of versions, for example, in very complex functions, where it is likely that an error will occur during the calculation of such a function.

Another interesting policy of the calculations timing when the outputs are the same, and the version states are different is the usage of replicas during execution, when the manager method has already received confirmation from the decisive method about the majority and interrupts versions that were not executed until the end. It is used more often for uniprocessor hardware and to save processor time.

One more interesting combination of two types of results comparison is the analysis of input and output parameter values. For example, some methods may not contain outputs, and in this case the decision method is not called, so here we can compare the input and output values of the version states.

5. Conclusion
So, we can summarize that there are two main ideas on which the concept of abstract state of the multiversion software components version is based. Firstly, all versions of objects have common data states, because they are instances of the same class. The abstract state of a version is to abstract all instances. Secondly, the abstract state of versions can be used for various purposes in addition to restoring versions: measuring performance, rolling back actions (transactions) committed in the system, collecting various statistical information about the modeled object or process, etc.

In the process of developing display functions and abstract version states, some auxiliary results can be obtained:
1. Performing “background testing” of versions that are currently not used to compare their abstract states.
2. Save the abstract state of versions in the data storage and use failed versions for recovery.
3. “Rollback” all versions back and restart them (it may be useful when the decisive method has not revealed the majority).
4. Implementation of the “rollback” scheme.

Acknowledgments
The reported study was funded by Russian Foundation for Basic Research, Government of Krasnoyarsk Territory, Krasnoyarsk Regional Fund of Science to the research project: «Multiversion method for improving information reliability of environmental monitoring of thermal power plants» (No 18-48- 240007).

References
[1] Avizienis A and Lapri J 1986 Guaranteed computing: from ideas to implementation in projects THER 74(5) 8-21
[2] Avizienis A 1985 The N-Version approach to fault-tolerant software IEEE Trans. on Software Engineering SE11(12) 1491-501
[3] Kovalev I V 2014 Analysis of problems in the field of software reliability research: multi-stage and architectural aspect Vestnik of SibGA 3(55) 78-92
[4] Kovalev I V, Zelenkov P V, Karasev M V, Tsarev M Yu and Tsarev R Yu 2015 Model of the reliability analysis of the distributed computer systems with architecture "client-server" IOP Conf. Ser.: Mater. Sci. Eng. 70(1) 012009
[5] Kovalev I V, Zelenkov P V, Losev V V, Kovalev D I, Ivleva N V and Saramud M V 2017 Multiversion environment creation for control algorithm execution by autonomous unmanned objects IOP Conf. Series: Materials Science and Engineering 173 012025
[6] Kovalev I V and Younoussov R V 2002 Fault-tolerant software architecture creation model based on reliability evaluation Advanced in Modeling & Analysis 48(3-4) 31-43
[7] Levendel Y 1990 Reliability analysis of large software systems: Defect data modeling IEEE Trans. Software Engineering 16 141-52
[8] Lyu M R 1996 Handbook of Software Reliability Engineering IEEE Computer Society
[9] Lyu M R 1996 Software Fault Tolerance (John Wiley & Sons Ltd)
[10] Cherif A 1999 Improving the Efficiency of Replication for Highly Reliable Systems FastAbstract
[11] Hecht H 1979 Fault tolerant software IEEE Trans. Reliability R-28 227-32
[12] Silayeva T and Grosspietsch K-E 1995 An Innovative Method for Program Reliability Evaluation Euromicro ’95 Como (Italy)
[13] Kovalev I V, Losev V V, Saramud M V, Kovalev D I and Petrovskiy M O 2017 To the question on implementation of multi-version execution environment software of onboard autonomous unmanned objects by means of real-time operating system Vestnik of SibGA 18(1) 58-61
[14] Kovalev I V, Losev V V, Saramud M V, Kovalev D I, Petrovskiy M O and Brezitskaya V V 2017 To the question on implementation of multi-version execution environment software of onboard autonomous unmanned objects by means of real-time operating system Siberian Journal of Science and Technology 18(4) 744–7
[15] Saramud M , Kovalev I et al. 2018 Kuznetsov Software interfaces and decision block for the execution environment of multi-version software in real-time operating systems International Journal on Information Technologies & Security 1(10) 25-34
[16] Saramud M V et al. 2018 Application of FreeRTOS for Implementation of the Execution Environment of Real-time Multi-version Software International Journal on Information Technologies & Security 3(10) 75-82
[17] Saramud M V, Kovalev I V, et al. 2019 Comparison of fault-tolerant software models in a
simulation environment Information Technology 25(1) 20-5
[18] Saramud M V, Kovalev I V, Losev V V, Petrosyan M O and Kovalev D I 2019 To the question of applying the correctness calculation methods for assessing the ultimate reliability of multiversion models of fault-tolerant systems Devices and systems. Management, control, diagnostics 3 19-25
[19] Koltyshev A A, Shatrov A K, Kovalev I V, Losev V V and Chuvashova M N 2018 The implementation of measures for reduction an inertia of the interaction components of the heterogeneous environment for the automated process control system IOP Conf. Series: Materials Science and Engineering 450 042015