# SOFTWARE INTERFACES AND DECISION BLOCK FOR THE EXECUTION ENVIRONMENT OF MULTI-VERSION SOFTWARE IN REAL-TIME OPERATING SYSTEMS [1]

*Mikhail V. Saramud, Igor V. Kovalev, Vasiliy V. Losev, Peter A. Kuznetsov*

Reshetnev Siberian State University of Science and Technology, Krasnoyarsk
e-mails: msaramud@gmail.com, basilos@mail.ru
Russian Federation

**Abstract:** The article suggests a way to create fault-tolerant software systems, including control systems. The increase in fault tolerance is achieved by introducing software redundancy, namely multi-version programming. It is proposed to create a multi-version software execution environment based on the existing real-time operating system by developing a decision-making unit and two new programming interfaces that support its operation. The functions of the existing real-time operating system which provide the necessary functional, the queuing mechanism, and the flow control mechanism are considered. An example of multi-version execution of an applied problem is considered, UML - diagrams describing this process in the proposed system are given.

**Key words:** multi-version programming, execution environment, fault tolerance; reliability; voting algorithms.

## 1. INTRODUCTION

In control systems operating in the autonomous mode, without the possibility of physical intervention from man (space, deepwater work, work in conditions unfit for a human being), most of the control tasks are critical to the reliability and fault tolerance of their performance, since in the event of a failure caused by a software failure the loss of the managed object occurs. Thus, increasing the degree of fault tolerance of software that implements these tasks is relevant and requires the application of new or specific approaches to the development of software, such as the introduction of software redundancy, for example, the multi-version approach [1].

In order to increase the reliability of control systems based on real-time operating systems (RTOS), in particular FreeRTOS there arises the task to implement in its framework the execution environment of multi-version software to perform the most critical to reliability tasks [2].

The existing literature describes the principles of creating a multi-version software environment execution [3,4], analyzes the fault tolerance of operating systems [5], describes

[1] This work was supported by Ministry of Education and Science of Russian Federation within limits of state contract № 2.2867.2017/4.6

the mechanisms of data exchange and tasks control in FreeRTOS, however, the presented studies do not consider solutions that use the multi-version methodology of the software within the RTOS, options have not been offered for creating an execution environment for multi-version software based on RTOS, for example FreeRTOS, which will combine the advantages of RTOS and multi-version approach and significantly increase the fault tolerance of UAV management systems.

We propose a model applicable at the design stage of the planned for development control system of UAV, whose main task is to increase fault tolerance. The model is based on the use of RTOS as the basis for the software operation environment, supplemented with a decision block to provide the functioning of the execution environment of multi-version software and also the program interfaces (APIs) necessary for the interaction of the components of the projected system are described. The functioning of the API is based on the mechanisms of queues and task control in FreeRTOS. The result is the proposed fault tolerance model and the API needed for its operation.

## 2. DECISION BLOCK AND NECESSARY API

To implement the functionality that ensures the working capacity of the multi-version software execution environment within the framework of the FreeRTOS real-time operating system, it is necessary to develop a separate component of the decision block. Its functionality will be somewhat wider than the classical functions of the DB described in the literature, since its task will be not only to select the correct answer from the N versions.

The decision-making process should:

- provide all versions with the necessary input data;
- in case of limitation on the response time of versions, abort the execution of versions that did not have time to respond in a timely manner;
- decide directly on the correct output based on the collection of output versions;
- in case of applying weighted voting algorithms, evaluation of the correctness of the version outputs and corresponding change in their weights, by putting in their stacks the weights "1" for the versions that gave the correct answer and "0" for the versions giving the answer that does not coincide with the correct one, in case of clear voting, or, in case of fuzzy modifications of the decision-making (voting) algorithms, the versions that gave the answer different from the correct one by more than a given tolerance.
- clear the memory of the used data and complete the versioning processes if they are no longer needed (they will not be executed again).

The weight stack is a binary stack of a given length, working on the FIFO principle (First In First Out), it contains "1" and "0", indicating in which number of votes this version gave the correct answer, and in which - the error one. On the basis of the stack, at each vote, the evaluation of the reliability of versions or its "weight" is calculated, for example - at a stack depth of 100, if it contains 97 elements "1" and 3 elements "0", the weight of the corresponding version will be 0.97. The fixed depth of the stack provides an element of "forgetting", the results of the version work older than the stack depth are no longer taken into account in calculating its reliability (weight).

Due to introduction of the multi-version execution environment, it becomes necessary to formalize two more programming interfaces, in addition to the standard API mono-version application software. This is the interface between the versions and decision block (figure 1), which, for the most part, should be described in the V-spec (specification for the development of versions by various developers, ensuring the guarantee of their performance in the target

execution environment [6]), and the interface between the decision-making block and the RTOS scheduler, since the decision block must take precedence over other processes and the right to launch, suspend, and terminate the tasks of versions processes.

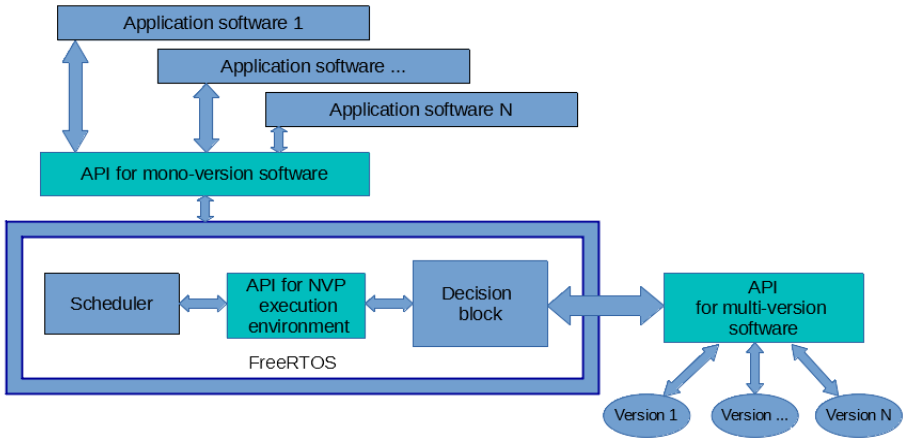


*Fig. 1. Structure of interaction of various program structures within the control system+*

In FreeRTOS for further process control, it is necessary to know handle (process descriptor), which is assigned when it is created. Therefore, it is more rational to run the version processes exactly from the decision block, saving their identifiers in the local variables, which allows the decision block to manage the created processes further using the standard FreeRTOS commands, passing in them the descriptor of the desired process as a parameter.

In Figure 2, you can observe a UML activity diagram showing the process steps in the multi-version execution of an application task. After the task is started, the decision block checks the availability of the necessary input data, as well as their correctness (data type, dimension, fitting the given range, etc., depending on the task). If the input data passes the test, the decision block starts version tasks and passes them the correct input data, waits for a response from all versions (checking the response queue). If there is an existing limit on the response time of versions, when this time expires, if not all versions gave the answer, then the decision is made basing on the responses received, and the process that failed to respond to the version ends compulsorily.

Depending on the chosen decision algorithm, it may require at least a certain number of responses. If you combine a constraint on the response time of versions and an algorithm that requires most of the version replies for its work, there may be a situation where less answers are received than the algorithm needs to work [5]. In this case, the decision block either returns an error informing about the impossibility to make a decision, or the answer is sent of any of the versions that had time to respond, depending on the implementation of the decision block. However, with all the answers of versions it is not always possible to choose the only correct variant, many decision algorithms have limitations, for example - for an absolute majority voting algorithm it is necessary that more than half of the versions coincide. If decision-making based on a set of output values is possible, then it is accepted, if not, then the decision block either returns an error, indicating that it is impossible to make a decision, or as the output is sent a response of either version, depending on the implementation of the

decision block. After determining the correct output, the version weights change according to the concurrence of their answers with the correct ones.

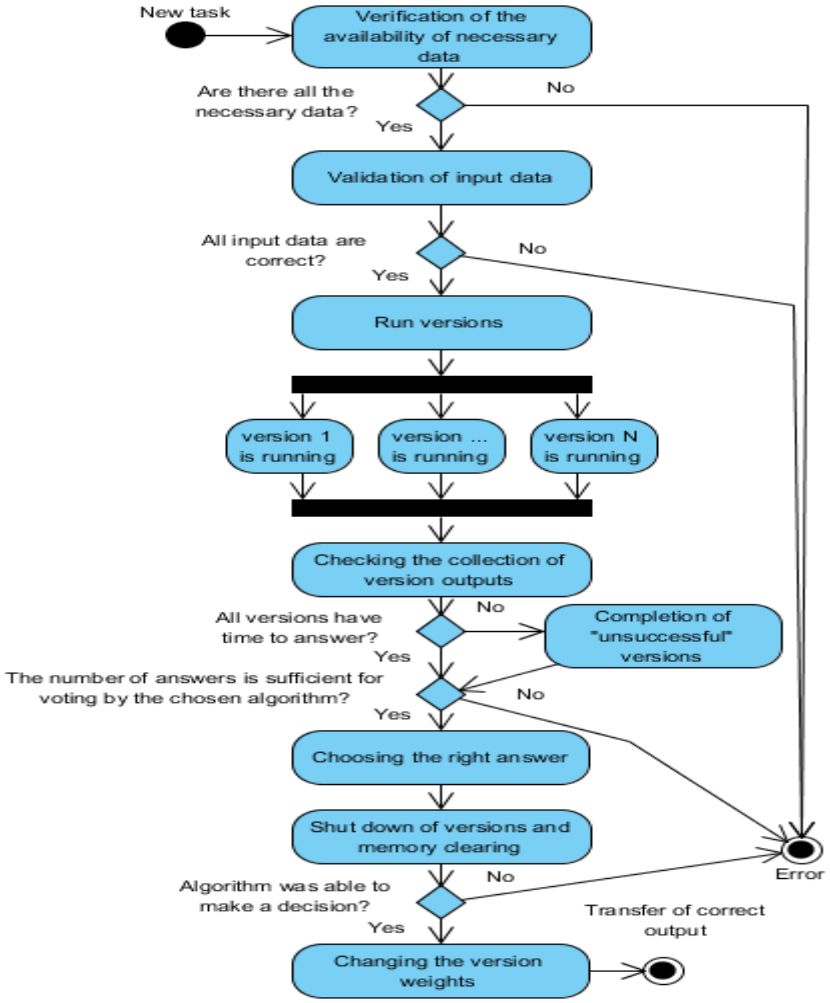


*Fig. 2. An activity diagram showing the process of multi-version execution of an application task within the framework of RTOS.*

### 2.1. Implementing the API through the queuing mechanism

A queuing mechanism is used for the safe and correct exchange of information between the FreeRTOS streams. A queue is a simple FIFO buffer (although there is also the possibility of writing to the top of the queue), which can store the number of elements not exceeding the specified queue size. The enrolling in the queue is a byte-by- byte copy of the data to the buffer, reading – it is the copying of data with the possibility of removing it from the queue. From the point of view of the functioning of the software system, queues are independent objects into which many processes can perform reading / write many operations without the risk of reading / writing the damaged data [5].

The queues are created by a function having the following form *xQueueCreate()*.From the point of view of the functioning of the created API, the following important parameters of the function should be noted: *uxQueueLength* - the length of the queue; *uxItemSize* - the size of the memory allocated to each object; *xQueueHandle* is the queue descriptor that allows you to access a specific queue.

The reading functions of the queue element have the following options: *xQueueReceive ()* - deletes the received item from the queue; *xQueuePeek ()* - leaves the read item in the queue.

Since the writing of specific versions can be entrusted to third-party developers, to diversify software versions, in order to avoid related faults [4], it is necessary to ensure the transmission of identifiers of the input and output queues in the version for processing them.

This task is carried out by the decision block. A special queue of the input data queue descriptor *BPRInQueue* is created. When a multi-version function is called, the identifier of that queue is placed in it, from where the versions should receive the original data. The decision block starts and reads the identifier from the descriptor queue. From the queue with this descriptor, the decision block receives the original data and analyzes it. Next, this identifier is passed to each version as an argument to *pvParameters* and used by it to obtain the original data. The result of the version work must be written to the data processing queue used by the decision block to select the correct result value. Therefore, the decision block must also pass its descriptor as a parameter. In the same way, the decision block receives a queue descriptor into which it should write down the output data of the version that gave the correct result. There is a descriptor queue of the output queue *BPROutQueue,* from which the decision block takes the output queue descriptor and writes the output data to the queue, which corresponds to it. Since queues become available to all threads when specifying the appropriate identifier, you should enter a monitoring mechanism to prevent the item from being withdrawn from the queue that does not correspond to the work algorithm. You should make sure that the software versions do not call the function to remove *the xQueueReceive* from the queues of input, output, or identity queues.

Before compiling the code for software versions, you should check the code content, monitoring the use of the xQueueReceive command and removing the cases of its inappropriate use.

**2.2. Implementation of the API through the stream control mechanism.**

Another task of the decision block will be the launching of separate versions for execution and monitoring of their execution. Previously, all versions are created in the form of ordinary functions in the C language.

Version tasks is started by calling the *function portBASE_TYPE xTaskCreate (),* which has the following important parameters: *vTaskCode* - pointer to a function that describes tasks; *usStackDepth* - the depth of the function stack; *pvParameters* - a pointer to the parameters that the function uses; *uxPriority* - task priority; *pvCreatedTask* is the stream descriptor used to further manage the stream being created.

The task ends with the function *vTaskDelete (),* with *xTaskHandle*- tasks descriptor, as the parameter, If it is not specified, the task that caused this function terminates.

To control the execution time of tasks, you use the scheduler management functions, for example *vTaskDelay (),* delaying the task execution for a certain time interval (parameter - *TicksToDelay)* or *vTaskSuspend (),* pausing the task with a descriptor (parameter -

*pxTaskToSuspend*) before giving the command *vTaskResume ()* corresponding descriptor as a parameter.

**2.3. An example of multi-version execution of an applied task.**

For a visual explanation of the interaction with the proposed programming interfaces, we consider the UML sequence diagram showing an example of the sequence of actions for multi-version execution of a problem with the number of versions N = 3.

In the process of inter-component interaction, the following actors participate: the scheduler is a standard element of FreeRTOS, responsible for switching between processes; decision block - the module developed by us with the purpose of executing versions and making the right decision based on a collection of their outputs; N-versions - functionally equivalent, but algorithmically different versions [8]; the source data queue is a queue containing the descriptor of the other queue, in which the necessary input data for the solution of the problem by versions is stored; the processing queue is the queue into which all versions place their responses in the form of a structure containing the response and the version number required by the decision block for making the decision; output queue is a queue containing the descriptor of the other queue into which the correct answer is to be placed.

It should be noted that in the case of unweighted decision-making algorithms, only the answers should be placed in the data processing queue, and the sequence is not important, since it does not matter which answer belongs to each version. However, in the case of applying weighted algorithms, we need to know which version gave the answer, since the answers will have different weights that affect the changes in the weights of the classes [9]. It is also necessary to know for changing the version weights by the decision result when comparing the winning response with the answers with these versions.

Let's consider the process of multi-version execution of an applied task, represented as a UML diagram of the sequence in Figure 3. Each task, multiverse performed, is launched from the body of the main program in accordance with the software algorithm.

Stages of the process:

1. The scheduler starts the decision block with the *xTaskCreate ()* function, if this is the first start, or passes it the control of the *vTaskResume ()* function if it was already started earlier and expected the item to appear in the raw data queue.

2. The decision block accesses the raw data queue, reads the identifier from the *vQueuePeek ()* function, and retrieves the original data to verify the suitability for processing, determines whether the data is sufficient, it’s type, range of values, if the data is suitable for the task, proceeds to the next step.

3. The decision block calls the *xTaskCreate ()* function for the correct versions, the scheduler launches the software version tasks.

4. The decision block itself is delayed, invoking the *vTaskDelay ()* function for a time interval equal to the version assigned for running the version in case of a runtime constraint, or "falling asleep", waiting for the results to appear in the data processing queue.

5. Versions begin their work by getting the source data from the source data queue by the *vQueuePeek ()* function.

6. Having worked out and received the resulting data, each version by the function *xQueueSendToFront ()* sends them to the processing queue.

7. At the end of the delay time, the scheduler transfers control to the decision block, the decision block takes the data from the processing queue, evaluates them and, based on the decision algorithm, chooses which result data is correct.

8. The received correct output data is written to the output queue.

9. In the case of applying weighted decision-making algorithms, the decision block writes the corresponding values to the weight stacks of the versions.

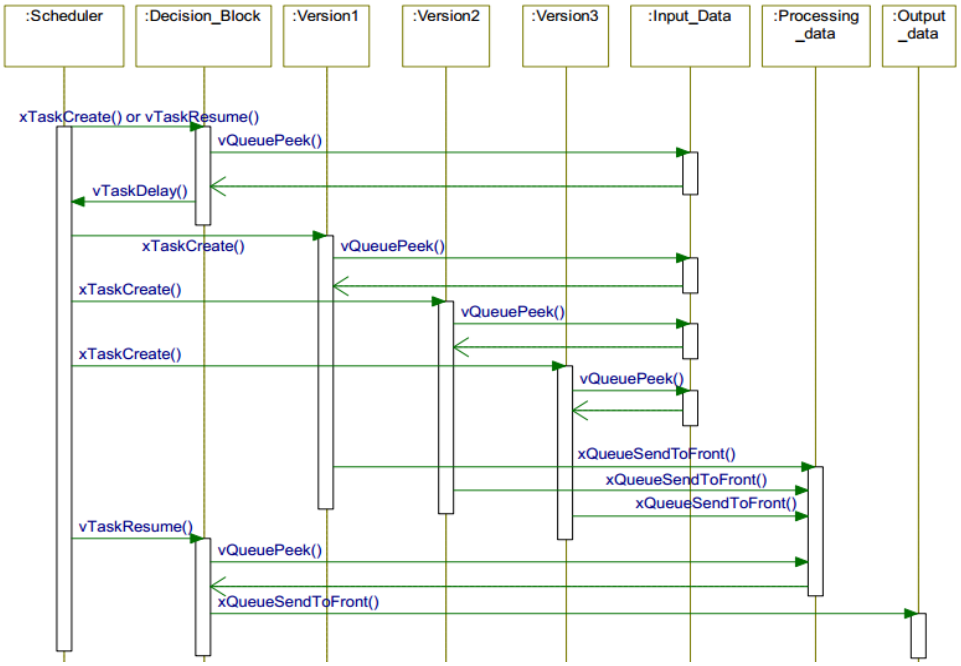


*Fig. 3. Sequence diagram of multi-version execution of an application task on successful completion of all versions.*

Also in Figure 4 we consider the case in which not all versions managed to answer at the set time interval, in our case - the second version.

The sequence of actions coincides with the previous case, except that when one or more versions failed to meet the specified time interval, the decision block calls the function *xTaskDelete ()* with the descriptor of this version and the scheduler completes it forcibly.

### 2.4. Software implementation of the decision block

Figure 5 shows the result of the simulation environment comparing various decision algorithms used in the system decision block with software redundancy. In this system are implemented: N-version programming with a weighted voting algorithm with forgetting and its fuzzy modification, the model of recovery blocks, N-Self Checking Programming (NSCP), Consensus Recovery Block, and t/(n-1) decision making algorithm. The system simulates the outputs of versions with the specified reliability characteristics (probability of random error, inaccuracy, interversion error), as well as the parameters of the algorithms themselves - the number of versions from 3 to 9 and the tolerance for fuzzy algorithms. The system runs a series of iterations (300 by default), in each of which, the version simulations give outputs same for all algorithms, and each algorithm makes a decision based on the same inputs and gives its answer, the imitating environment knows the right answer in advance, it calculates the number of algorithm errors at each iteration.

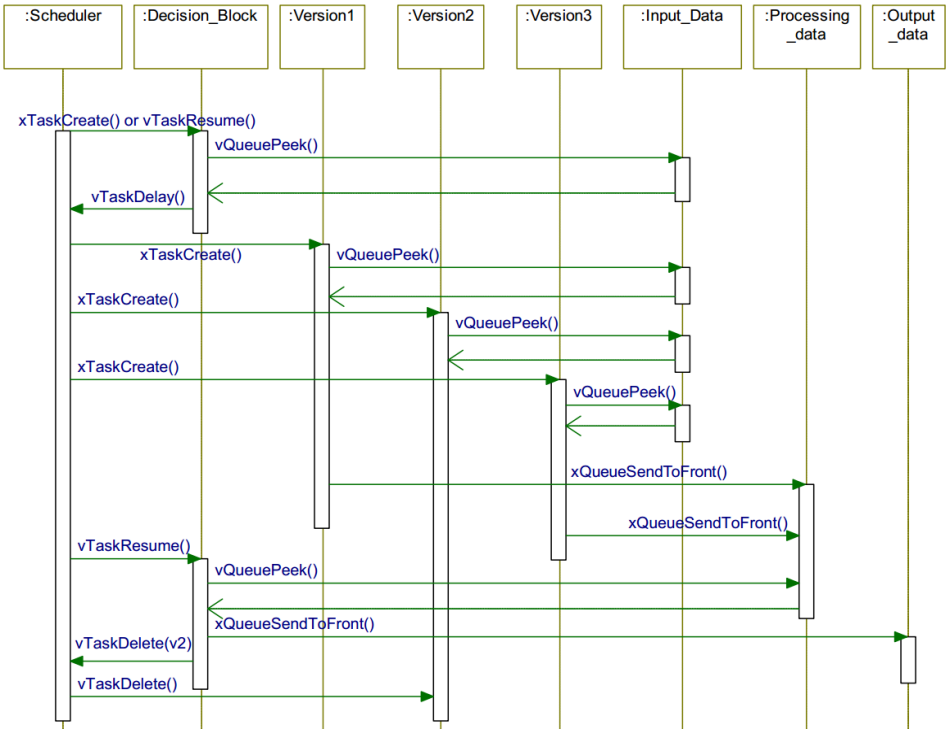


*Fig. 4. The sequence diagram of the process of multi-version execution of an applied task with one version that did not have time to answer.*

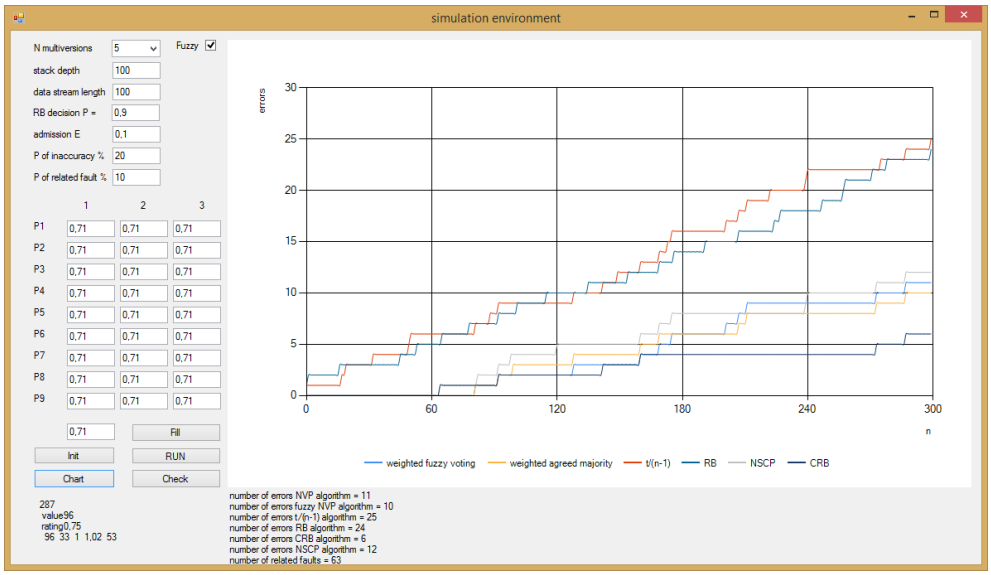


*Fig. 5. Software implementation of the simulation environment for execution of algorithms for the decision block.*

After the end of work, the environment gives the number of errors for each algorithm and builds the schedules of their distribution, as well as the number of interversion errors that occur are shown. An example of the execution of the system with low reliability of versions (p = 0.71) is presented for clarity, since at high parameters of reliability of versions algorithms are practically do not make mistakes, and graphs represent a horizontal line coinciding with the axis of abscissas.

The algorithms used in the decision block were implemented and tested in practice, as an X86 program in a Windows environment. In this case, the probabilistic and temporal characteristics, experimental data of delays and response time, when performing tasks in the real system, which are affected by the characteristics of the API, can be evaluated in practice only when the system is implemented in the target environment - RTOS. The reason for this is the API implemented by the queue and flow control mechanisms in FreeRTOS, whose operation will be affected both by the hardware characteristics and by the characteristics of the FreeRTOS itself - the size of the system time clock, etc.

## 3. CONCLUSION

This work shows that for implementation of the real-time multi-version software execution environment is most reasonable to use the real-time operating system, supplementing it with the necessary functionality, since the RTOS solves most complex problems, especially in regard to the real-time operation. The considered examples demonstrate the operation of the proposed scheme.

To implement the multi-version software described in the article, it is necessary to develop a decision block that must provide all versions with the necessary input data, in case of limitation on the version response time, to interrupt the execution of versions that did not have time to respond in a timely manner, directly decide on the correct output based on the collection of output versions, etc.

This module, working within the framework of the RTOS, for example - FreeRTOS, provides the runtime execution environment for multi-version software in real time, performing all the tasks necessary for this. This approach is much more rational, especially in terms of spent resources and development time, than creating a multi-version software execution environment from scratch, developing the functionality provided by existing operating systems on its own.

For the functioning of the developed unit, it is also necessary to develop two programming interfaces: between the decision block and the real-time operating system scheduler and between the decision block and the version tasks themselves.

We have confirmed the working capacity of the proposed model, but its real probabilistic and temporal characteristics become available for measurement and evaluation after the completion of the stages of development and testing of the system that being designed.

## REFERENCES

[1] Kovalev I.V., Zelenkov P.V., Losev V.V., Kovalev D.I., Ivleva N.V., Saramud M.V. Multi-version environment creation for control algorithm implementation by autonomous unpiloted objects [Electronic resource]. *IOP Conf. Series: Materials Science and Engineering* 173 (2017) 012025.

[2] Orges Çiço, Zamir Dika, Betim Çiço, High reliability approaches in cloud applications for business – reliability as a service (RAAS) model, *International Journal on Information Technologies & Security*, № 3, vol. 9, 2017, pp.3-18.

[3] L. Chen, A. Avizienis, N-Version Programming: A Fault-Tolerant Approach to Reliability of Software Operation, *Proc. Int. Symp. Fault-Tolerant Computing FTCS-8*, pp. 3-9, 1978.

[4] Lee, I., Tang, D., lyer, R.K., and Hsueh, M.C., Measurement-Based Evaluation of Operating System Fault Tolerance, *IEEE Transactions on Reliability*, vol. 42, no. 2, June 1993, pp. 238-249.

[5] McAllister, D.F., Sun, C.E., and Vouk, M.A., Reliability of Voting in Fault –Tolerant Software Systems for Small Output Spaces, *IEEE Transactions on Reliability*, vol. 39, no. 5,1990, pp. 524-534.

[6] Eckhardt, D.E., and Lee, L.D. A Theoretical Basis for the Analysis of Multi-version Software Subject to Coincident Errors, *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, 1985, pp. 1511-1517.

[7] Avizienis, A. The N-Version Approach to Fault -Tolerant Software, *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, December 1985, pp.1491-1501.

[8] Saramud M.V., Zelenkov P.V., Kovalev I.V., Kovalev D.I., Kartsan I.N. Development of methods for equivalent transformation of GERT networks for application in multi-version software [Electronic resource], *IOP Conf. Series: Materials Science and Engineering* 155 (2016) 012025.

[9] G. Latif-Shabgahi, S. Bennett. Adaptive majority voter: a novel voting algorithm for real-time fault-tolerant control systems, *Proceedings 25th EUROMICRO Conference. Informatics: Theory and Practice for the New Millennium*, vol. 2, 1999, pp.113-120.

[10] Richard Barry. *Using the FreeRTOS Real Time Kernel - Standard Edition* (FreeRTOS Tutorial Books), 2009 Richard Barry

***Information about the authors:***

**Mikhail Vladimirovich Saramud** - Engineer of research department, Reshetnev Siberian State University of Science and Technology. Areas of Research are fault-tolerant software, system analysis;

**Igor Vladimirovich Kovalev** – Professor at the Department of systems analysis and operations research, Reshetnev Siberian State University of Science and Technology. Areas of Research are fault-tolerant software, system analysis;

**Vasiliy Vladimirovich Losev** – Associate Professor at the Department of automation of production processes, Reshetnev Siberian State University of Science and Technology. Areas of Research are automation control systems, system analysis;

**Petr Anatolyevich Kuznetsov** – Senior Lecturer at the Department of automation of production processes, Reshetnev Siberian State University of Science and Technology. Areas of Research are automation control systems, system analysis, system programming.

**Manuscript received on 25 October 2017**