Simulation environment for the choice of the decision making algorithm in multi-version real-time system

Igor V. Kovalev[a,b], *Mikhail V. Saramud*[a,b*], *Vasiliy V. Losev*[b]

[a]*Siberian Federal University, 79 Svobodny avenue, Krasnoyarsk, 660041, Russian Federation*
[b]*Reshetnev Siberian State University of Science and Technology, Krasnoyarsky Rabochy Av. 31, Krasnoyarsk, 660037, Russian Federation*

ARTICLE INFO

Article history:
Received 00 December 00
Received in revised form 00 January 00
Accepted 00 February 00

Keywords:
Multi-version
Fault tolerance
Quality assurance
Reliability
Voting algorithms
Simulation modeling

ABSTRACT

Context: Nowadays the most effective way to improve the reliability of software is an approach with the introduction of software redundancy - multi-version programming. The reliability of a multi-version system is determined not only by the reliability of the versions that make it up, but to a greater degree by the decision making algorithm.

Objective: Our objective is evaluation and selection of the most reliable voting algorithms in multi-version environments. In order to get this objective there is a need to check all the algorithms in the execution environment, simulating characteristic of the developed system. Thus, we obtain the characteristics of the quality of the algorithm operation in precisely those conditions in which it will work in the system that is developed.

Method: The article suggests weighted voting algorithms with a forgetting element, as well as modifications of existing voting algorithms. To be able to check the quality of their work, the simulation environment has been implemented that simulates the operation of the software multi-version execution environment.

Results: The article substantiates the use of the most reliable decision making algorithms in the decision block of the real-time operating system. A comparative analysis of decision making algorithms for the operation of the decision making block of the multi-version real-time execution environment has been carried out.

Conclusions: The software implementation of the simulation environment that implements the simulations of versions with given characteristics is considered, not only classical decision making algorithms, but also the author's modifications are investigated. The environment allows to obtain the quality characteristics of all implemented decision making algorithms with given system characteristics. The modeling results are considered, the dependence of the system reliability indicators on its input parameters is shown, a comparative analysis of various decision making algorithms based on the modeling results is made.

1. Introduction

Nowadays the most effective way to improve the reliability of software is an approach with the introduction of software redundancy - multi-version programming [1]. The reliability of a multi-version system is determined not only by the reliability of the versions that make it up, but to a greater degree by the decision making algorithm. Errors contained in separate versions are permissible, since they will be isolated by a correctly working decision making algorithm and will not lead to a system failure [2]. In their turn, the algorithms of the decision block will lead to the emergence of an erroneous exit of the designed system, which can lead to the failure of the entire system. Thus, it is necessary to pay maximum attention to improving the reliability of the algorithms used in the decision block of the multi-version environment (NVX).

To guarantee the quality of a complex system as a whole, we need to guarantee the quality of its component parts. In the case of fault-tolerant software based on software redundancy, we need to guarantee not so much the quality of the versions, but the decision block. This block selects the correct exit from the version answer collection. It is the quality of its

* *Corresponding author.* Tel.: +7-963-958-9649
E-mail address: msaramud@sfu-kras.ru
Peer review under responsibility of xxxxx.

http://dx.doi.org/10.1016/j.aebj.2014.10.012

work that is most critical for the fault tolerance of the designed system. Therefore, the main task of the system developers is to guarantee the quality of the decision block. At the same time, ensuring the quality of versions will mostly be the task of third-party developers, since the creation of a version to ensure diversification is transferred to various groups of developers [3].

2. Execution environment of the fault tolerant software

Considering the life cycle stage of fault-tolerant software design, we need to decide on the algorithms that form the basis of the decision making block. Currently, there are many algorithms that are used to determine the correct exit from the set of version responses. Most often, these are different voting algorithms, we will consider the most common ones: voting by an agreed majority, weighted voting by an agreed majority, fuzzy weighted voting by an agreed majority, median voting. We will also consider the t/(n-1) decision making algorithm and propose modifications of the existing voting algorithms and t/(n-1) decision making algorithm.

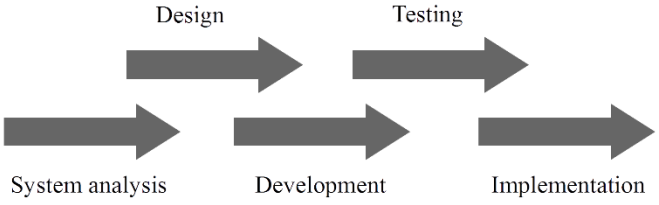

Fig. 1. Life cycle stages.

If you develop a system without evaluating the quality of the system at the early stages of the software life cycle (Figure 1), then assessing the quality at the testing stage of the resulting system, you can find that its quality indicators do not reach the required values. The reason for this is that the algorithms inherent in it will not allow the system to provide the required reliability characteristics for any software implementation.

This is a very dangerous situation, since it leads to reject the already developed product and return to the stage of system analysis to replace the algorithms with those that allow the system to achieve the required quality indicators. As it can be seen from Figure 1, this will actually lead to the repetition of all the work done.

Therefore, it is most reasonable to evaluate the characteristics of the system starting from the earliest stages of the life cycle [4], so that those which are surely able to fulfill the quality requirements for the decision system are always transferred to the next stage. Thus, we will get rid of the risk to reject the developed system due to the error in the choice of the algorithm in the early stages of the life cycle. This will not only guarantee the quality of the developed system, but also reduce the risks of material and temporary loss during its implementation.

We propose the guaranteeing method of the component quality of a fault-tolerant software - decision block in a multi-version execution environment at the design stage. This is achieved by choosing the known optimal algorithm with known characteristics of the system. Different decision making algorithms have their own strong and weak points. Some are more resistant to related faults, but do not work adequately with a large percentage of “inaccuracies”. Others, on the contrary, are resistant to both “inaccuracies” and relatively unreliable versions, but they make mistakes for each related fault, etc. Therefore, it is necessary to check all the algorithms in the environment simulating the characteristics of the developed system. Thus, we will obtain the characteristics of the algorithm operation quality in precisely those conditions in which it will work in our system.

In the case of development for embedded systems and controllers, all device software, from the operating system to the application software, is in fact one executable file. If an incorrect choice of algorithm is found at the operational stage, replacing even a small software component is a problem, since it leads to the need to recompile the entire project and replace the firmware on the device. This is not always possible, either because of the hardware features of the device, or its inaccessibility, in case when it is already put into operation.

In order to decide which way out of the multiple versions to recognize true and send to the output, different algorithms are used [5, 6]. The most common voting algorithms are:

Voting algorithm by an absolute majority - it is necessary that the absolute majority of versions vote for one option. For example - with 5 versions it is necessary that at least 3 versions vote for one option, otherwise we will assume that the correct answer cannot be chosen.

Voting algorithm by an agreed majority - it is necessary that more versions vote for one option than for others. Unlike the voting algorithm by an absolute majority, it is not necessary that the number of votes are more than half of the versions number. To make a decision it is enough that more versions vote for one option than for the others. In case that the same number of versions have voted for several options, any of them is chosen, since it is considered that they are equally “correct”.

The algorithm of fuzzy voting by an agreed majority - in contrast to the clear version, elements of fuzzy logic are added. A parameter such as the belonging coefficient equal to from 0 to 1 and the tolerance E is presented. In case the value of the version is equal to the value of the class, the coefficient is 1. If the version response differs from the class value by more than the E tolerance, then the coefficient is 0. If the version value is not equal to the class value, but is no further than the E tolerance from it, then the coefficient is in the range from 0 to 1. Thus, the coefficients of occurrence of all versions are added to the weights of classes, that is, one version can vote for several classes if their values differ by no more than tolerance E. The response (class) with the highest number of votes is recognized as correct.

Median voting - with this voting, the outputs of all versions are taken as erroneous, and their average value is taken as the output. This approach is used in cases when it is impossible to compare directly the outputs of versions, for example, when the outputs are the direction of movement, vectors, etc. There are various implementations of median voting. In our case, all the answers are sorted and the middle element is taken as the response.

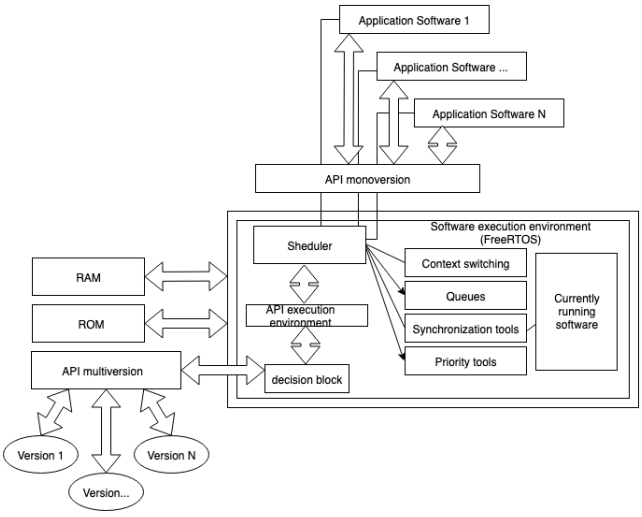

Fig. 2. Diagram of the execution environment of multi-version real-time software.

Figure 2 shows the structure of the real-time environment for multi-version software. It is exactly for the decision block of this system that it is necessary to choose the optimal decision algorithm. This execution environment and the technical solutions used in it are described in detail in the article [7].

3. Modifications of existing algorithms

The most dangerous type of fault is related [8]. The related fault is not a correct, but coinciding in value output of several versions. This error is the most dangerous, since it is extremely difficult to identify a failed version response if several votes are received for it. In order to increase the resistance to this type of error, we have proposed modifications of the voting algorithms by an agreed majority and fuzzy voting by an agreed majority. Modifications are in the introduction of a dynamic assessment of the reliability of each version or its "weight", as well as the forgetting element. For the first time, weighted voting algorithms with forgetting were presented at the European Conference on Electrical Engineering and Computer Science [9].

The weight is calculated as the sum of the correct answers of the version divided by their number. Technically it is implemented as follows - for each software version, a boolean queue of a given depth is created in the system. "0" is added to the queue in case if the decision block decides that the version gave the wrong answer and "1" if it gave the correct one. In case of fuzzy voting, if the version belonging coefficient to the winning class is > 0. That is, all versions that added weight to the class that won the vote will be marked as correct. Since the queue depth is fixed within the simulation, the new data will replace the old ones. That is, the queue works on the FIFO principle. This allows you to enter the forgetting element to the depth of the queue. For example, if the queue depth is 100, then the results of the version older than 100 votes will not be taken into account. The element of forgetting is needed to ensure the rapid response of the system to changes in versions behavior. In case when the versions significantly change their reliability when the input data stream changes, it is necessary to change quickly their rating for the most correct weighted voting [10].

The weight is determined by summing all the items in the queue. For example, if in queue of 1000 elements long 986 values "1" and 14 values "0" are written, the weight of this version will be 0.986.

With the software implementation one limitation is introduced - the weight of the version can not be equal to one. This situation may occur in practice - fairly reliable versions give the right response 100, 1000, 10000, etc. once in a row and without limitation, they would receive the entire queue of "1" (or TRUE), which would give them a rating of 1. Such situations should not be allowed, since in case of an incorrect response with this version, it will receive weight 1, while the correct response to the rest N-1 versions according to weight will only approach to 1 and lose the vote. In addition, analytically, the version reliability rating equal to 1 does not make sense. Since if we have an absolutely reliable software module, the sense of a multi-version system is lost.

4. T/(n-1) algorithm

It is also necessary to consider the decision making algorithm in multi-version systems proposed by Jie Xu from University of Newcastle upon Tyne, based on t/(n-1) diagnosability [11]. For simplicity, we will call it

t/(n-1) decision making algorithm. The main point of the algorithm is not in the voting of all the versions outputs, but only when comparing some of them sufficient for making a decision. We will consider the example of a system with the number of versions N = 5 and the maximum number of errors t = 2, that is, we will consider the 2/(5-1) option. When the number of errors does not exceed t, the algorithm guarantees the choice of the correct variant from the N outputs of the versions. However, even with a larger number of incorrect version outputs, the system will not necessarily choose the wrong one. With a certain probability, the choice of the correct exit will occur, though this is no longer guaranteed [12]. We will consider the algorithm in more detail on the example of 2/(5-1) option. Outputs of four versions are compared in pairs - 1st with 2nd, 2nd with 3rd, 3rd with 4th, we get three results of comparisons ω12, ω23, ω34, 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 on switching the output between outputs 1, 4 and 5 versions. That is, versions 2 and 3 are used only for comparison, the values of their outputs are never used as a system output. More visually the scheme of work can be studied in Figure 3.

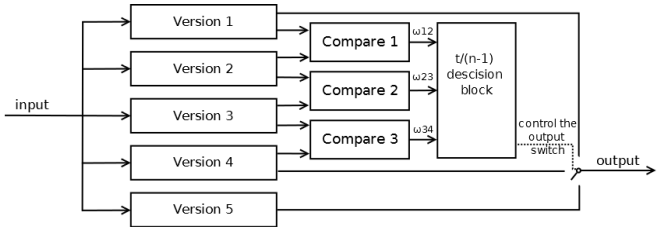

Fig. 3. The architecture of the t/(n-1) algorithm with n = 5 and t = 2.

As can be seen from Figure 3, the decision making flow in the t/(n-1) algorithm is relatively not complicated. In case of five multi-versions,

only the results of three pairwise comparisons of four versions outputs are necessary for making a decision (proper control of the output selector). The output value of the fifth version is not used for making a decision. The control logic of the output switch based on the results of comparisons with n = 5 is presented in Table 1.

Table 1
Possible kinds of choice based on comparator outputs for n = 5.

ω_{12}	ω_{23}	ω_{34}	**Conceivably 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 3 and Table 1, it can be concluded that with relatively reliable versions, in most cases comparators will return (0; 0; 0) and the execution value of the first version will be submitted to the output. It is also possible to conclude that there is no need to execute the fifth version each time. And do it only in case of corresponding values of the comparisons results when it is necessary to submit to the output exactly the result of the fifth version ((0; 1; 0), (1; 0; 1), (1; 1; 1)) to the output. This fact reduces the average load required by the execution environment of multi-version software for operation, since in most cases 4 out of 5 versions will be calculated. The decision making algorithm itself is also less resource-intensive compared to voting, especially with its weighted modifications where with each vote all versions are executed, classes are created and weights are calculated for each of them. For t/(n-1) with n = 5, only three simple comparison operations with binary output are necessary. Then an unambiguous, a priori given choice of output for one of eight possible values combinations of the comparators outputs is performed (Table 1).

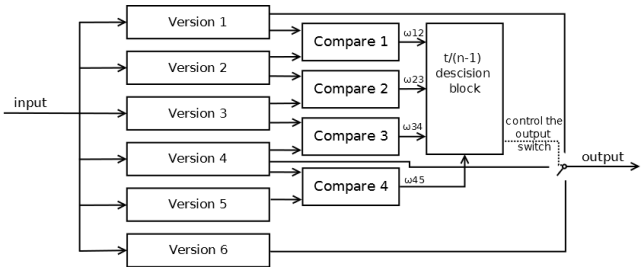

Fig. 4. The architecture of the t/(n-1) algorithm with n = 6 and t = 2.

We will consider the impact on the algorithm operation by adding another version up to N = 6. The new scheme of the algorithm operation is presented in Figure 4 and Table 2. As it can be seen from the presented data, when adding an additional version, you need to add a comparator. Consequently, we have already 4, not 3 boolean outputs, necessary for making a decision. Accordingly, the number of possible combinations of comparators is doubled from 8 to 16. It can be concluded that the computational complexity of the t/(n-1) algorithm is extremely sensitive to the number of versions. With each additional version added, 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 application is most justified when the number of versions does not exceed N = 5.

Table 2
Possible choice options based on the comparators outputs for n = 6.

ω_{12}	ω_{23}	ω_{34}	ω_{45}	**Conceivably correct versions**
0	0	0	0	1, 2, 3, 4, 5
0	0	1	0	1, 2, 3
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

5. The software implementation of the simulation environment

The program implements simulations of the required number of versions, which operate in accordance with the parameters specified on the form. The main window sets the number of versions from 3 to 9, the probability of error-free operation of each version for three consecutive data streams, the length of each data set (respectively, the total number of iterations is equal to three lengths), the probability of an related fault, the probability of inaccuracy, and the tolerance E for fuzzy logic.

At the beginning of the simulation, the function fills in an array of input values — version responses at each iteration. Data is generated in accordance with the specified parameters. Next, from this array, the generated version responses are transferred to the input of each algorithm. This solution is necessary to compare all algorithms on the same input data set. Since error generation occurs with given probabilities, in the case of generating version responses for each algorithm separately, they will receive for input data with different numbers of errors and the comparison will not be correct.

Changing the parameters of the versions operation in the simulation process for the three sets of input data is used to study the system reaction to a sharp change of the reliability of the versions. For example - version number 1 may have a reliability of 0.98 in the first set, in the second one the reliability drops to 0.61, and in the third one it increases again to 0.97.

In the environment 6 decision making algorithms are implemented, this is a classical voting by an agreed majority, a weighted voting by an agreed majority with forgetting, its fuzzy variant, t/(n-1) algorithm, its fuzzy modification with modified comparators and median voting.

We will consider the process in more detail. When voting by an agreed majority, the decision block receives the responses of the version simulations. If the output value does not coincide with the previously received ones, a new class is created with this value. If the value coincides with the value of the existing class, then the weight of this class is recalculated using the formula

$$P_{total} = P_{class} + (1 - P_{class}) * P_{version}$$

where P_{total} is the final weight of the class, P_{class} is the weight of the class before recalculation, $P_{version}$ is the weight of the version.

After the responses of all versions are processed, the weights of the created classes are compared. The class with the greatest weight is recognized as the winner and its value is sent to the exit. This is not always the class for which the largest number of versions voted. The weights of each version that voted for this class are important. After determining the class of the winner, the versions that voted for it get the weight "1" in the queue, and the versions that voted differently get "0".

In this model, there is no limit on the versions response time, since simulations run on the same algorithm and there is no point in comparing their resource intensity. In the case of the implementation of real software versions, rather than their simulations, as well as the well-known hardware limitations and requirements for the system reaction time, it makes sense to introduce such a restriction. This is necessary to make it possible to take into account the probability that resource-intensive versions will not have time to respond by the time of the vote, and will not be taken into account. These data are essential in the case of systems operating in real time [13]. In the case of time limits, even an absolutely reliable version that did not "have time" to provide a response in time will be considered as a failed one.

For fuzzy voting by an agreed majority after creating and evaluating all classes based on the outputs of the versions, the algorithm performs another iteration. Versions whose output value did not coincide with the value of the class, but differ from it no more than the tolerance E (therefore, class membership is > 0) also adds weight to the class. Class weights are recalculated using the formula.

$$P_{total} = P_{c\ lass} + (1 - P_{class}) * P_{version} * K_{belonging}$$

where $K_{belonging} = 1 - (|X_{class} - X_{version}|/E)$, X-values of the class and the version that gave the response not equal to the value of the class, but fitting into the tolerance E.

Simulations of versions in this software implementation generate 3 types of errors. In case of a random error, a failure is simulated in the module. Related fault - an admitted algorithmic miscalculation in several versions that will give errors with the same value for the same input. Inaccuracy - the response close to the correct one, removed from it not more than tolerance E, but not equal to it. This type of error simulates rounding errors with the lack of capacity, inaccuracy in digitizing the outputs of analog sensors, etc. That is, a situation where, algorithmically, the version worked correctly, but gave an inaccurate answer due to rounding errors, digitization, and lack of capacity. An error is generated with the probability specified for each version and each data stream. If an error is generated, a series of controls occurs. If this is not the first error in the current voting, then a related fault is generated with a given probability. The value returns that matches the value of the past error, this error simulates a coherent error. After that, with a given probability, an "inaccuracy" is generated - the output that does not match the value with the correct one, but differs from it by no more than the tolerance E returns.

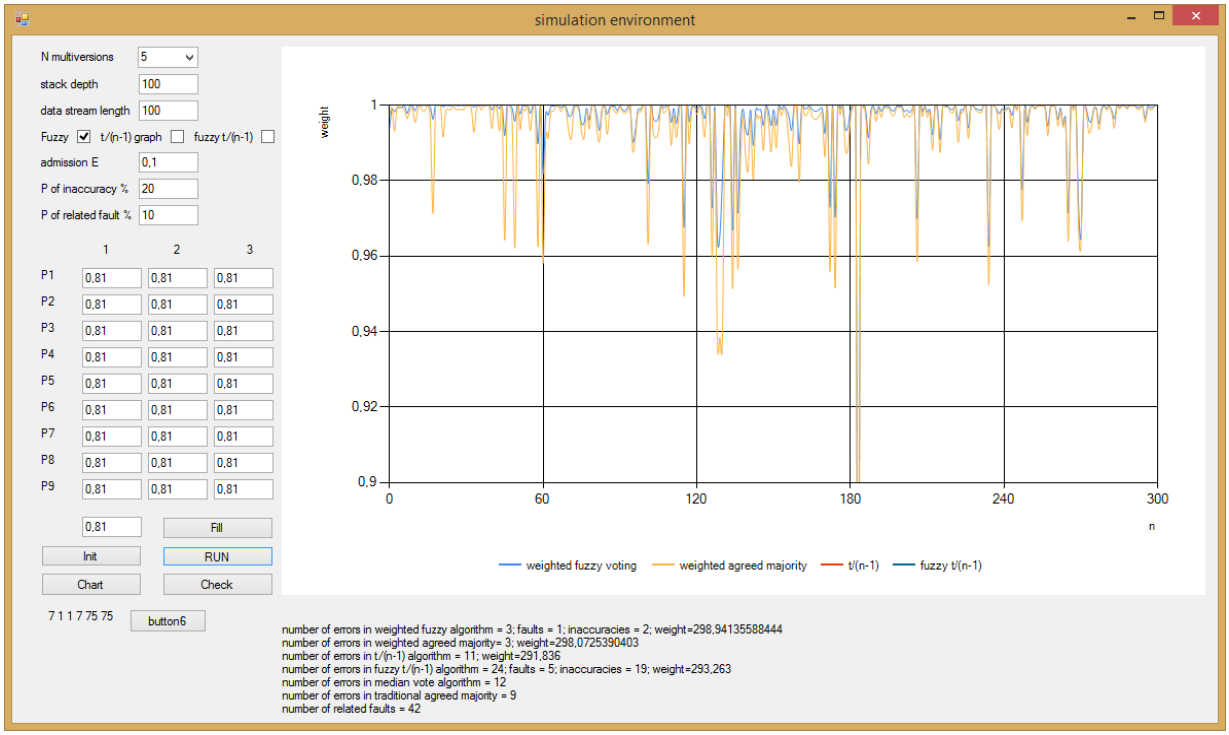

Fig. 5. Interface of simulation environment (t /(n-1) graphs are disabled)

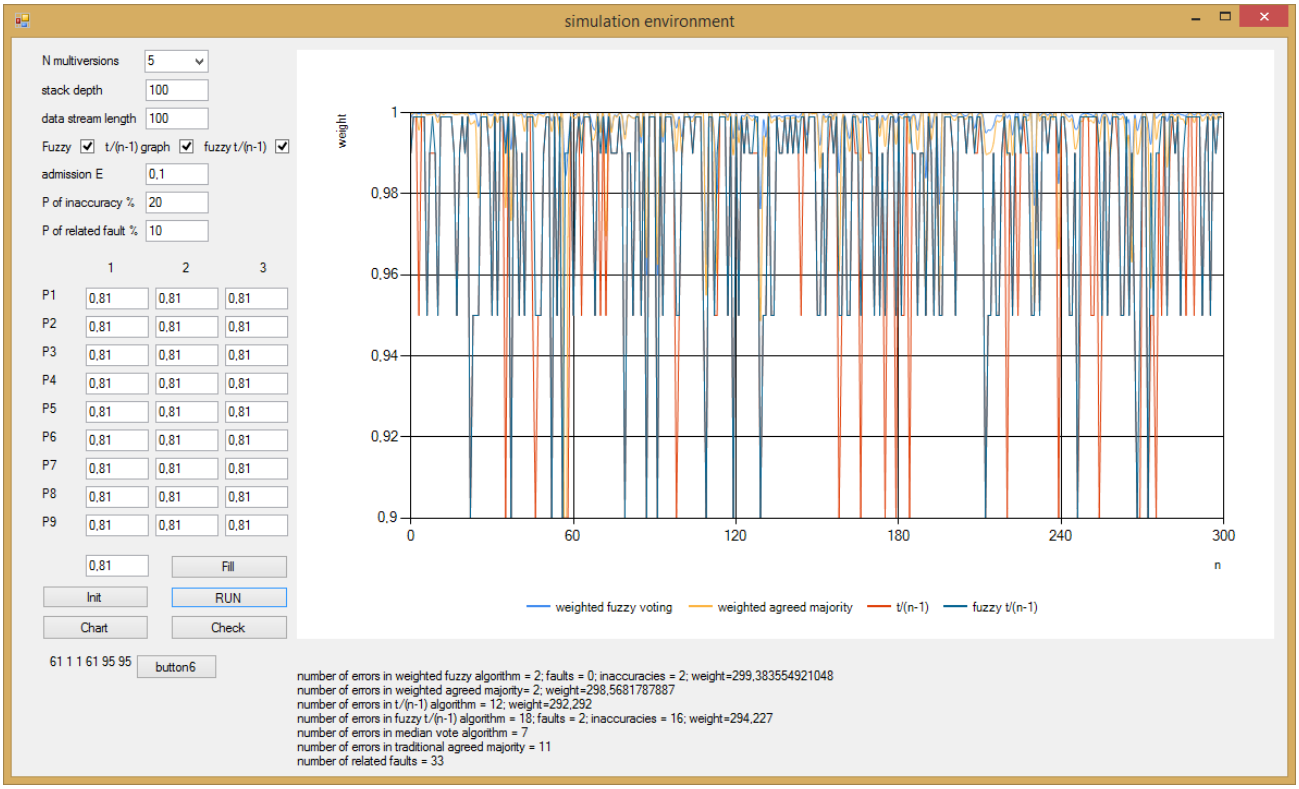

Fig. 6. Interface of simulation environment (all graphs are enabled)

If the described probabilities do not work, then a random error returns. It simulates a random failure in the current software version. According to the simulation results, it can be concluded that in the absence of "inaccuracies" there is no difference in the work of clear and fuzzy voting options. Random error is always different from the correct response by more than the value of the tolerance E. Therefore, it does not change the weight of classes during the second iteration of fuzzy voting. From these results, we can conclude that if there are no possible places of "inaccuracies" in the system (digitization of analog signals, lack of capacity in mathematical operations, etc.), then using a more resource-intensive fuzzy voting algorithm will not give advantages. But, if there is a probability that "inaccuracies" will occur, then a fuzzy algorithm will increase the reliability of the system. The only drawback of the work of the fuzzy algorithm is the same assigned weight to both the versions, which gave the ideally correct output, and the versions that are algorithmically correct, but have "inaccuracies".

For the possibility to debug algorithms, the form displays information about the last error in the output of the decision block - the iteration number, the output value and the weight of this class. Software implementation has the ability to build graphs of the weights of each version and the winning classes. The axes are iteration numbers and weight. For clarity, the ability to change the scale is implemented. For example, when studying weights of winning classes, their values do not fall below 0.9. On a scale from zero to one, graphs are a practically flat line in the area of one. For better perception in this case, the scale changes to a range from 0.9 to 1 along the axis of weight.

To be able to compare the weights sums of the t/(n-1) algorithm with weighted voting algorithms, we have introduced weights estimates for the t/(n-1) algorithm. They are based on the output of the comparators, since these are not real weights, but only an estimate. The estimate accepts one out of 4 discretely defined options (0.999; 0.985; 0.970; 0.950). For the t/(n-1) algorithm, the graphs are not informative and greatly complicate the perception of the graphs of the other algorithms. For this reason, the graphs of the t/(n-1) algorithms are made turned off for more convenient study of the remaining graphs. However, when turning off the graphs, stop of the counting of the sum weights (or their estimate) during the

* *Corresponding author.* Tel.: +7-963-958-9649
E-mail address: msaramud@sfu-kras.ru
Peer review under responsibility of xxxxx.

http://dx.doi.org/10.1016/j.aebj.2014.10.012

simulation does not happen. You can compare graphs kind in Figures 5 and 6.

According to the simulation results for each algorithm, the sum of errors for all iterations, the number of made “inaccuracies” for fuzzy algorithms, and the sum of the occurred related faults are displayed. If basing on the simulation results, there is a uniquely superior algorithm, the system displays a message about the selected optimal algorithm (Figure 7).

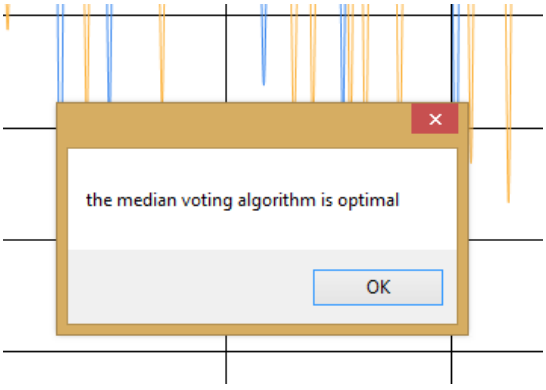

Fig. 7. The system's message about the selected optimal algorithm.

6. The simulation results

We will study the simulation results in the software implementation and consider the reaction of the algorithms to changes in the input parameters of the simulation.

To begin with, we will study the graphs of the versions weights introduced in Figures 8-10. Graphs are introduced for a queue with a depth of 100 and three consecutive data streams of 100 votes for each. In Figure 8, we can observe the operation of the system with all reliable versions (versions reliability in all data streams from 0.9 to 0.99). As you can see, the versions weights also vary within the minimum limits, without falling below the value of 0.9. The graphs also show the weight of the class that won the vote.

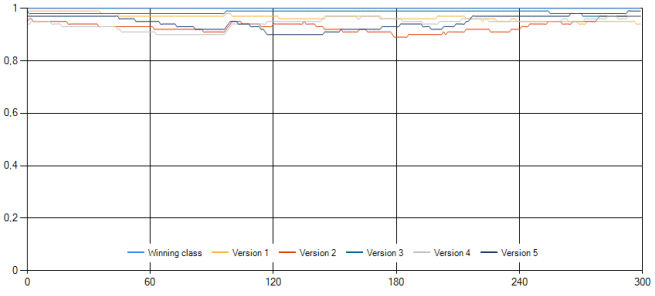

Fig. 8. The operation results of the simulation model of the implementation of the voting algorithm by an agreed majority (high reliability of all versions 0.9-0.99).

In Figure 9 we can observe the operation of the system with an average version reliability (version reliability in all data streams from 0.7 to 0.96). As you can see, the versions weights already vary within wide limits, but do not fall significantly below 0.7.

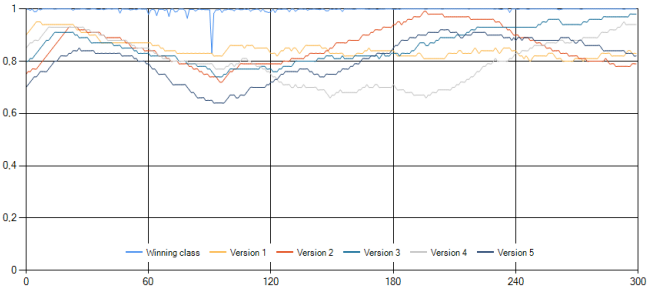

Fig. 9. The operation results of the simulation model of the implementation of the voting algorithm by an agreed majority (the average reliability of all versions is 0.7-0.96).

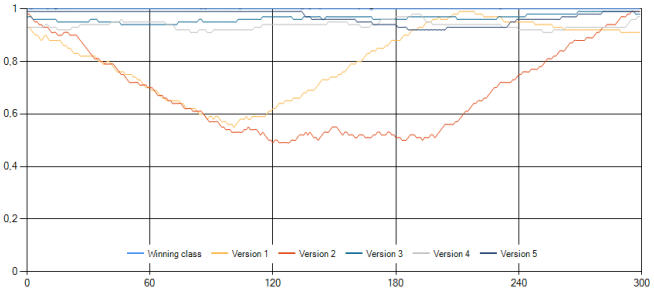

Fig. 10. The results of the simulation model of the implementation of the voting algorithm by an agreed majority (failure was found out in the first version of the first and second data sets and in the second version of the second data set with a 50% probability).

The graphs (Figures 9-10) show the reaction of the system to the changing of versions behavior when a reliable version begins to give errors. Or on the contrary, the low-weight version ceases to make mistakes (at the beginning of the experiment, all versions weights are relatively high). In Figure 10, the first version mistakes with a probability of 50% in the first and second data stream, in the third one it ceases to make mistakes. It can be concluded that the system responds fairly quickly to changes in the versions behavior. In the number of votes equal to the depth of the queue, the versions will receive estimated weights, which quite accurately correspond to their probabilities of a correct response.

Table 3
The results of the system performance at different amounts of multi-versions for unreliable versions with P = 0.7.

N		**3**	**4**	**5**	**6**	**7**	**8**	**9**
Clear weighted	Number of errors	48	27	14	6	3	1	0
	Amount of weights	268,40	281,96	293,40	295,82	298,47	298,74	299,71
Fuzzy weighted	Number of errors	45	28	15	8	4	2	0
	Amount of weights	280,92	288,74	296,66	298,28	299,19	299,61	299,91
Agreed majority traditional	Number of errors	89	36	16	13	5	3	2
Related fault		30	40	39	64	65	89	101

Table 4
Simulation results for different versions of reliability.

Specified version reliability		**0,65**	**0,7**	**0,75**	**0,8**	**0,85**	**0,9**	**0,95**
Clear voting by an agreed majority	Number of errors	19	13	7	4	2	1	0
	Amount of weights	285,02	292,16	296,53	298,70	299,68	299,92	299,99
Fuzzy voting by an agreed majority	Number of errors	21	14	10	4	3	1	0
	Of which failures	14	7	7	2	2	0	0
	Inaccuracies	7	7	3	2	1	1	0
	Amount of weights	291,99	295,76	298,05	299,30	299,81	299,97	299,99
t/(n-1) algorithm	Number of errors	43	25	17	6	4	1	1
	Amount of weights	286,71	288,56	290,55	292,55	293,87	295,96	297,78
Fuzzy modification of the t/(n-1) algorithm	Number of errors	58	32	31	11	8	8	5
	Of which failures	29	11	12	3	4	0	1
	Inaccuracies	29	21	19	8	4	8	4
	Amount of weights	289,07	290,83	292,49	293,74	295,19	296,51	298,35
Agreed majority traditional	Number of errors	29	27	10	6	4	0	0
Median voting	Number of errors	43	27	19	9	4	1	2
Related fault		58	50	39	38	20	16	8

Table 3 shows the simulation results with different number of multi-versions - from 3 to 9 (reliability of all versions in all streams equal to P = 0.7, queue depth 100, tolerance E = 0.1, inaccuracy probability P = 0.2, probability of related fault P = 0.1. Such a low reliability index of all versions as 0.7 is taken for clarity of the simulation results since with sufficiently reliable versions (P> 0.95) even with the number of multi-versions N = 3, the proposed voting algorithms do not pass to the output any single error 300 voting iterations [14]. The data obtained show how the reliability of the system as a whole increases, despite the use of extremely unreliable software modules. Despite the fact that each version generates an error in 30% of cases, in a system with 9 versions all 300 responses are correct. The algorithm of clear voting by an agreed majority always selects the correct response from the versions response pool. The fuzzy algorithm makes more mistakes, because sometimes the vote wins a class with a value close to the correct one (not more than E from it) but not equal to it. By simulating such a response is counted as erroneous, however, versions that add weight to the winning class still get 1 into queue, increasing their own weights. For this reason, despite the greater number of errors, the sum of the weights of the fuzzy algorithm is higher.

Not all considered decision algorithms are able to work with an arbitrary number of versions. For this reason, only voting algorithms are presented in this table, since they have the ability to work with any number of versions N≥3.

We will compare the reliability indicators of the t/(n-1) algorithm as compared with proposed by us weighted modifications of the voting algorithm by an agreed majority, its clear and fuzzy version and other algorithms. We will perform simulation in the simulation environment. The following model parameters are used: the number of iterations = 300, the same reliability of all five versions in all 300 runs, queue depth = 100 (for weighted algorithms), tolerance E = 0.1 (for fuzzy algorithms), inaccuracy probability P = 0.2, probability of related fault P = 0.1. We will change the version reliability value from 0.65 to 0.95 with a step of 0.05 and get the number of errors per 300 iterations for each algorithm. The simulation results are presented in table 4.

From the results presented in Table 4, it can be concluded that with relatively reliable versions, all algorithms provide error-free operation for 300 iterations. While each of the 5 versions gives in average 15 erroneous outputs per 300 iterations (with a reliability of 0.95 an imitation version will produce in average 5 errors per 100 iterations). The exception is a fuzzy modification of the t/(n-1) algorithm. The simulation results show that the fuzzy modification has extremely low resistance to inaccuracies occurrence. It did not miss a single failure. However, when generating inaccuracies in the first or fourth versions, fuzzy comparators return a match with neighboring versions, which gave an ideally correct answer and the system sends the result with inaccuracy to the output. When the reliability of versions decreases, the algorithms begin to miss errors on the system output, however, in different quantities. The most reliable was a weighted modification of the algorithm of agreed voting with forgetting. More errors of the fuzzy algorithm are explained by the cases when “inaccuracy” is chosen as the response (the value that is far from the correct one is not more than the E tolerance, but not equal to it). Similar responses are also counted as erroneous by the system. The t/(n-1) algorithm begins to yield significantly in reliability to the weighted voting algorithm by an agreed majority with relatively unreliable versions, since situations more often happen when more than t = 2 versions give the wrong answer. In such cases, the correct operation of the algorithm is not guaranteed. The results also show that the fuzzy version of the t/(n-1) algorithm gives more errors in general. However, the number of failures is even less than that of the basic algorithm, and most of the errors are inaccuracies.

After analyzing the obtained data, it can be concluded that the application of the t/(n-1) algorithm is possible with relatively reliable versions, when there will not be situations of simultaneous failure more than t versions. Its application will be justified in situations when it is necessary to reduce the computational load on the system.

Table 5
Simulation results with different probability of inaccuracy occurrence.

Probability of inaccuracy occurrence		5%	10%	20%	50%	100%
Clear voting by an agreed majority	Number of errors	4	2	2	2	3
	Amount of weights	298.15	299.10	298.63	299.07	298.85
Fuzzy voting by an agreed majority	Number of errors	5	3	3	4	6
	Of which failures	4	2	1	3	0
	Inaccuracies	1	1	2	1	6
	Amount of weights	298.63	299.41	299.32	299.78	299.99
t/(n-1) algorithm	Number of errors	10	6	8	9	11
	Amount of weights	291.69	293.23	292.10	292.49	292.48
Fuzzy modification of the t/(n-1) algorithm	Number of errors	12	7	14	34	48
	Of which failures	9	4	3	4	0
	Inaccuracies	3	3	11	30	48
	Amount of weights	292.16	293.94	293.80	295.75	299.48
Median voting	Number of errors	16	9	7	6	0
Agreed majority traditional	Number of errors	9	9	16	7	13
Related fault		33	24	37	38	30

Especially in cases where the comparators are simple to execute, and creating many classes each time and calculating their weights is too laborious. An example is pneumatic logic in hazardous industries. Comparators, logic, and an output switch can be assembled on pneumatic components. While implementing the voting algorithm will be almost impossible on pneumatic elements. However, in cases where relatively unreliable versions are used, or there is a high probability of related fault, its application is not desirable, since in such situations it shows less reliability in comparison with voting algorithms. As for the fuzzy modification - there is a sense in its application in systems where there is a probability of inaccuracies, passing of which is not critical for the system. Since the algorithm often allows the passing of inaccuracies, however, cuts failures better than the basic algorithm.

We will investigate the reaction of the system to changes in the probabilities of inaccuracies occurrence, we will take the reliability P = 0.8 for all versions in all data streams.

We will study the results introduced in Table 5. They show that with an increase in the probability of inaccuracy occurrence, fuzzy modifications of the decision making algorithms remain resistant to failures, although they allow these inaccuracies to pass [15]. For a reasonable choice of the most suitable algorithm, it is necessary to know how critical for the system the passing of exactly inaccuracies is. For example, for course correction systems with constant adjustment of inaccuracy with a slight deviation from the correct value will not adversely affect the operation of the system as a whole. While with the passing of failures, the system will significantly change the direction that can lead to disastrous consequences. In the case of systems that are unstable to inaccuracies, it is preferable to use classical variants of decision making algorithms. Since they better screen out inaccuracies, considering them equally incorrect, regardless of their proximity to the correct response, in contrast to fuzzy algorithms.

We will investigate the reaction of the system to a change in the probability of a related fault occurrence. This is of the most interest because it will show the stability of all studied algorithms to the most dangerous type of errors. We will take the reliability P = 0.8 for all versions in all data streams.

Table 6
Simulation results with different probability of related fault occurrence.

The probability of occurrence of a related error		5%	10%	20%	50%	100%	100% P=0.95
Clear voting by an agreed majority	Number of errors	1	4	7	8	13	0
Fuzzy voting by an agreed majority	Number of errors	2	4	7	9	13	0
	Of which failures	2	4	7	8	13	0
	Inaccuracies	0	0	0	1	0	0
t/(n-1) algorithm	Number of errors	9	7	10	17	13	1
Fuzzy modification of the t/(n-1) algorithm	Number of errors	13	8	11	21	13	13
	Of which failures	9	7	9	17	13	0
	Inaccuracies	4	1	2	4	0	13
Median voting	Number of errors	21	15	15	20	13	0
Agreed majority traditional	Number of errors	11	7	16	28	37	5
Related fault		31	38	64	164	308	61

The simulation results are introduced in Table 6. They show that the voting algorithms are much more resistant to related faults than the t/(n-1) algorithm. An additional column has also been added to the table, in which the behavior of the system with reliable versions (P = 0.95) is considered. This is done to demonstrate that with sufficiently reliable versions even 100% of the related faults do not affect the reliability of the system with the proposed modified voting algorithms. However, this is not the case with the t/(n-1) algorithm, which permits failures even with reliable versions. It is interesting to note that related faults do not affect the operation of the median voting algorithm, since it does not compare version responses for class weights change, and the collection of responses is simply sorted by the magnitude of the values. For median voting there is no difference whether versions give coinciding errors or not. We will consider an example - if 5 versions gave responses (3; 3; 6; 19; 19) and the responses “3” and “19” are related faults, it will still choose the response “6” as the average. The same feature of its work makes it resistant to ejections. Unlike the implementation of voting algorithms with averaging outputs, if one or several versions gives a

response that differs by several rates in any direction, this will not affect the operation of the algorithm.

7. Conclusion

The article gives recommendations on increasing the reliability of software at various stages of the life cycle, describes the specifics of a multi-version real-time execution environment. The scalability of the t/(n-1) algorithm for a different number of versions (N = 6) is shown. The dependence of the complexity of the algorithm on the number of versions is investigated.

The results of all simulations show a significant advantage of the proposed weighted voting algorithms with forgetting in comparison with the classical voting by the agreed majority. The data obtained using the developed simulation environment confirm the effectiveness of introducing version weights and the forgetting element.

The weighted voting algorithms with forgetting investigated in the article were tested in a multi-version real-time execution environment.

The results show the effectiveness of the proposed modifications of voting algorithms, as well as prove the possibility of creating the reliable system of unreliable software modules. Concerning the t/(n-1) algorithm, which is of interest as an alternative to the voting algorithms, its use is justified only in systems with significant limitations for computing resources (or other limitations, for example, the use of pneumatic logic in an explosive environment) and the use of fairly reliable versions. Since the algorithm shows the worst resistance to unreliable versions and different types of errors.

The algorithms investigated in the article can be used in real-time fault-tolerant control systems in various areas: autonomous unmanned objects, hazardous production, etc., where increased requirements are applied to reliability and safety parameters.

The proposed simulation environment is a useful tool for designing of a multi-version system, since it allows to obtain the characteristics of a decision block under the conditions in which it will work in the real system. This result is important for software development, especially for complex fault-tolerant systems, since it allows to obtain an assessment of the quality characteristics of the decision block at an early stage of development. This allows to choose obviously appropriate decision making algorithms. The choice of the decision making algorithm with guaranteed satisfying the requirements qualitative characteristics excludes the case of system
redeveloping due to the fact that only at the testing stage it turns out that the chosen algorithm is not able to meet the established requirements. This does not only guarantee the quality of the software product being developed, but also makes its implementation time more predictable.

Acknowledgements

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

REFERENCES

1. Eckhardt, D.E., and Lee, L.D. (1985). A Theoretical Basis for the Analysis of Multi-version Software Subject to Coincident Errors, *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, pp. 1511-1517.
2. Eduardo Liebl, Cristina Meinhardt, Paulo F. Butzen (2016). Reliability analysis of majority voters under permanent faults, 17th Latin-American Test Symposium (LATS), pp.180-180.
3. Luping Chen, John May (2016). A Diversity Model Based on Failure Distribution and Its Application in Safety Cases, 2014 Eighth International Conference on Software Security and Reliability (SERE) , pp.1-10.
4. Alberto A. Avritzer, André B. Bondi (2014). Developing Software Reliability Models in the Architecture Phase of the Software Lifecycle, 2014 IEEE International Symposium on Software Reliability Engineering Workshops, pp.22-23.
5. G. Latif-Shabgahi; S. Bennett (1999). 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, pp.113 - 120 vol.2
6. A. Karimi, F. Zarafshan, S.A.R. Al-Haddad, and A.R. Ramli (2014). A novel n-input voting algorithm for x-by-wire fault-tolerant systems, *The Scientific World Journal*, vol. 2014.
7. Saramud M.V., Kovalev I.V., Losev V.V., Kuznetsov P.A. (2018) Software interfaces and decision block for the execution environment of multi-version software in real-time operating systems, International Journal On Information Technologies And Security, № 1 (vol. 10), pp.25-34.
8. Luping Chen ; John H. R. May (2016). A Diversity Model Based on Failure Distribution and its Application in Safety Cases, IEEE Transactions on Reliability, vol. 65 , Issue 3, pp. 1149 – 1162
9. I. Kovalev, A. Voroshilova, V. Losev, M. Saramud, M. Chuvashova and A. Medvedev (2017). Comparative Tests of Decision Making Algorithms for a Multiversion Execution Environment of the Fault Tolerance Software, 2017 European Conference on Electrical Engineering and Computer Science (EECS), Bern, pp. 211-217.
10. Kovalev I.V., Zelenkov P.V., Losev V.V., Kovalev D.I., Ivleva N.V., Saramud M.V. (2017). Multi-version environment creation for control algorithm implementation by autonomous unpiloted objects [Electronic resource] // IOP Conf. Series: Materials Science and Engineering 173 012025.
11. J. Xu (1991). The t(n-1)-diagnosability and its applications to fault tolerance, Digest of Papers. Fault-Tolerant Computing: The Twenty-First International Symposium, pp. 496 – 503.
12. Jie Xu; B. Randell (1997). Software fault tolerance: t/(n-1)-variant programming, Software fault tolerance: t/(n-1)-variant programming, *IEEE Transactions on Reliability*, Volume: 46, Issue: 1, pp. 60 – 68.
13. Jebin V Thomas, R Ranjith, Radhamani V Pillay (2017). Guaranteeing fault tolerance in real time systems under error bursts, 2017 International Conference on Intelligent Computing, Instrumentation and Control Technologies (ICICICT), pp.1480-1484.
14. Brilliant, S.S., Knight, J.C., and Leveson, N.G. (1990). Analysis of Faults in an N-Version Software Experiment, *IEEE Transactions on Software Engineering*, vol.16,no.2, pp.238-247.
15. McAllister, D.F., Sun, C.E., and Vouk, M.A. (1990). Reliability of Voting in Fault –Tolerant Software Systems for Small Output Spaces, *IEEE Transactions on Reliability*, vol. 39, no. 5, pp. 524-534.