Designing of a Real Time Software Fault Tolerance Schema based on NVP and RB Techniques

Omar Anwer Abdulhameed
Al-Mansour University College, Iraq
Computer Technology Engineering Dept.

Noor Kareem Jumaa
Al-Mansour University College, Iraq
Computer Technology Engineering Dept.

ABSTRACT
Software fault tolerance is an important criterion for the dependable systems, especially in real time and critical systems. There are few techniques that are used to implement fault tolerance in software, such as the most two common techniques: "N-Version Programming" and "Recovery Block", also there are other driven techniques from these two techniques, as well as, other supporting methods like Exception Handling. Development programs must consider the development risks associated with using conventional software fault tolerance techniques that theoretically can outcome in a better system; but also, that could drive the entire effort of the development to fail because of the design team inability to manage the system complexity within a reasonable cost and time frame. Also, these conventional techniques cannot always guarantee producing a correct or an acceptable output. So, this paper proposes is to design a fault tolerance technique that consists of two layers: the first layer is the special layer that derived from the other known techniques in a way that use the positive characteristics of these techniques, with the consideration of keeping the complexity of the system in minimum degree. This layer can be named the ***2-Version Software with Acceptance Test Support***. The other layer is the general layer that can be used with the software fault tolerance technique that proposed in the first layer or with any other "software fault tolerance" techniques. The second layer propose is the design of a software fault tolerance mechanism that concerns on the use of unusual (intelligence) ways for system recovering from design faults, also allowing the system operator to interfere in the process of system recovering. The developed mechanism will be used to support the operation of the conventional "software fault tolerance" techniques.

General Terms
Real Time Systems, Critical Systems, Fault Tolerance, Exception Handling.

Keywords
Software Fault Tolerance Techniques, N-version programming, NVP, Recovery Blocks, RB, N-version software, NVS.

1. INTRODUCTION
Software Fault Tolerance means that "the software is designed so that faults in the delivered software do not result in system failure". The basis for accepting the faults is that if and when the system response fails, it is economy (cheaper) to pay for the consequences of failure rather than the discovering and removing the faults before delivery of the system [1, 2].

Design of systems with "fault tolerance" capabilities to satisfy the requirements of a particular application is a complex process and loaded with experimental and theoretical analysis in order to find the most appropriate trade-offs within the design space. Properties of a system to be considered include: dependability (i.e. availability, reliability, maintainability, and etc.), failure modes, performance, environmental resilience, cost, weight, volume, design effort, power, and verification effort. "In addition to these, development programs must also weigh in the development risks associated with using technologies that in theory could result in a better system but that could also drive the whole development effort to failure due to the inability of the design team to manage the complexity of the system within a reasonable time frame". [3]

Also, these conventional techniques cannot always guarantee producing a correct or an acceptable output. There are some situations where these techniques are unable to produce or select the right output, and this will lead to the system failure.

The argument here is how to implement fault tolerance in a software system without increasing the overall system complexity in a way that may decrease the system reliability and how to overcome the disadvantages and the performance lacks that exist in the conventional software fault tolerance techniques.

This paper proposes the design of a fault tolerance technique that consists of two layers. The first layer is the special layer that derived from the other known techniques in a way that exploit the effective characteristics of these techniques, with the consideration of keeping the complexity of the system in minimum degree. This layer can be named the 2-Version Software with Acceptance Test Support. The other layer is the general layer that can be used with the software fault tolerance technique that proposed in the first layer or with any other software fault tolerance techniques. The second layer propose the design of a software fault tolerance mechanism that concerns on the use of unusual (intelligence) ways for system recovering from design faults, also allowing the system operator to interfere in the process of system recovering. The developed mechanism will be used to support the operation of the conventional software fault tolerance techniques.

So, before discussing the proposed technique, a review of the basic conventional software fault tolerance techniques will be presented including those that are accommodated in the proposed design. Also, the disadvantages and the performance lacks that exist in these techniques are exposed.

The rest of this paper is organized as follow: section 2 provide a literature survey of the related works, section 3 and its

subsections review the techniques of the fault tolerance, an additional consideration has been discussed in section 4, section 5 and its subsections present the details of the proposed fault tolerance technique, and section 6 discuss the concluding remarks.

2. RELATED WORKS

In [4], Jashan Deep and Dr. Rajiv Mahajan published a survey on software fault tolerance in parallel computing which surveys various software fault tolerance techniques and methodologies. Their research focuses on both RB and NVP techniques and their cost as fault tolerance techniques.

The researchers of [5] have been discuss the architecture of software fault tolerance techniques. The ”present the logical vehicle that permits reasoning on the equivalence or the compatibility of the various expressions of fault tolerance properties at various abstraction levels”.

The researcher of [6], discuss the techniques of the software fault tolerance. The research surveys the recovery blocks, single version programming, N-version programming, multi-version programming, and the combinational of N-version with recovery block techniques.

In [7], the N-Version Programming (NVP) is the main concern of the research. The researchers used the VPN to design a version of six language N-Version Programming project for fault tolerance flight control software.

3. SOFTWARE FAULT TOLERANCE TECHNIQUES

Fault tolerance techniques that applied to software can be categorized into to two classes: “*single version”* and “*multi-version”* software techniques. Single version fault tolerance techniques concern on developing the “fault tolerance” of a single piece of software (module) by addition of mechanisms into the design in order to improve the error detection, containment of errors, and handling of the errors caused by the activation of faults design [5, 6].

“Multi-version fault tolerance techniques concern on using multiple versions (or variants) of a piece of software (module) in a structured way to ensure that design faults in one version will be covered by other versions in such a way that do not cause system failures”. A basic typical of the software fault tolerance techniques is that they can, by principle, be applied at any level of a software system: process, procedure, full application program, or the entire system with the operating system. Also, the techniques can be applied selectively to those modules who most potentially to have design faults due to their complexity [5, 6].

So, a mainly review to the two basic software fault tolerance techniques are discussed in the following subsections since, these techniques can be accommodated in the proposed design. The two techniques are: *N-Version Programming* and *Recovery Blocks*.

3.1 N-Version Programming (NVP)

The N-version programming is defined as the “independent generation of $N \geq 2$ functionally equivalent programs from the same initial specification”. The N programs possess all the necessary attributes for concurrent execution, during which comparison vectors (“c-vectors”) are generated by the programs at certain points. The program state variables that are to be included in each c-vector and the cross-check points (“cc-points”) at which the c-vectors are to be generated are specified along with the initial specification (see Figure 1). In another words, N-Version programming is a multi-version technique. Using a common specification, the software system implemented in a number of different versions by different teams [2, 4, 8, 9].

These versions are executed in parallel on separate computers, and it can be used effectively in single-processor applications. All N results are sent to an output checker which [2, 4, 8]:

- Compares all results and votes on the comparison, then
  - If all agree, outputs the result, otherwise
  - Selects the result agreed by the majority and outputs this value (normally there are N – 1 agreement at any one time).

This technique can tolerate software bugs that affect a minority of versions, but cannot tolerate correlated fault (reason for failure is common to two or more modules) [10]. Building and using N-version programming requires three major efforts [11]:

- To lay down the member versions of the N-version programming unit, including all of the features that are needing to be embedded into the N-version execution environment.
- To describe and execute the N-version software process in a way that maximize the independence of the programming efforts.
- To design, build, and dimensions the system of N-version execution environment for a very reliable and time-efficient execution of -version programming elements.

3.2 Recovery Blocks (RB)

The Recovery Blocks technique is one of the common and earliest developed techniques for multi-versions software fault tolerance. This technique is based on combining the basic concepts of checkpoint and restart mechanism with multiple versions of software components (modules). The main issue in this technique is to use different developing methods and approaches (i.e., different algorithms, different programming languages, etc) in building the multiple versions, so as to try to ensure that at least there is one of the versions (alternate modules) be able to tolerate the component (module) or the system failure [8].

The approach of recovery block, attempts to prevent residual software faults from impact on the system environment; also, it is aiming to provide fault-tolerant functional modules which may be nested within a sequential program. The usual syntax is as follows (see Figure 2) [4]:

Checkpoints are shaped or created before version executes. After a version fails, checkpoints are needed to recover the state to make available a valid operational starting point for the next version if an error is detected. The acceptance test (performed on exit from a primary or alternate block to validate its actions), “need not be an output-only test and can be implemented by various embedded checks to increase the effectiveness of the error detection. Also, because the primary version will be executed successfully most of the time, the

alternates could be designed to provide degraded performance in some sense (e.g., by computing values to a lesser accuracy)".

"Like data diversity, the output of the alternates could be designed to be equivalent to that of the primary, with the unsuccessfully, the component must raise an exception to communicate to the rest of the system its failure to complete its function. Note that such a failure occurrence does not imply a permanent failure of the component, which may be reusable after changes in its inputs or state. The possibility of coincident faults is the source of much controversy concerning all the multi-version software fault tolerance techniques" [3,12].

One of the most difficult (and critical) aspects of the recovery blocks is the design of the acceptance tester. Three different methods may be used [9]:

- Test result against *pre-defined* values (e.g. checking that values lie within valid ranges).
- Test result against *predicted* values. In dynamical systems, for example, the maximum rates of change of parameters can be used for this purpose. Using this, the maximum possible change of parameter values in any time interval can be predicted. The actual value produced by the algorithm should not exceed the predicted amount.
- Using the output value, compute the input values which should have produced this output. Compare these with the checkpoint values to see if they agree. This technique (an inverse or 'reverse' algorithm check) can be applied in general to control and signal processing algorithms where time isn't a problem.

Note that the knowledge of system and/or software attributes must been known in order to form acceptance tests. definition of equivalence being application dependent. Actual execution of the multiple versions can be sequential or in parallel depending on the available processing capability and performance requirements. If all the alternates are tried

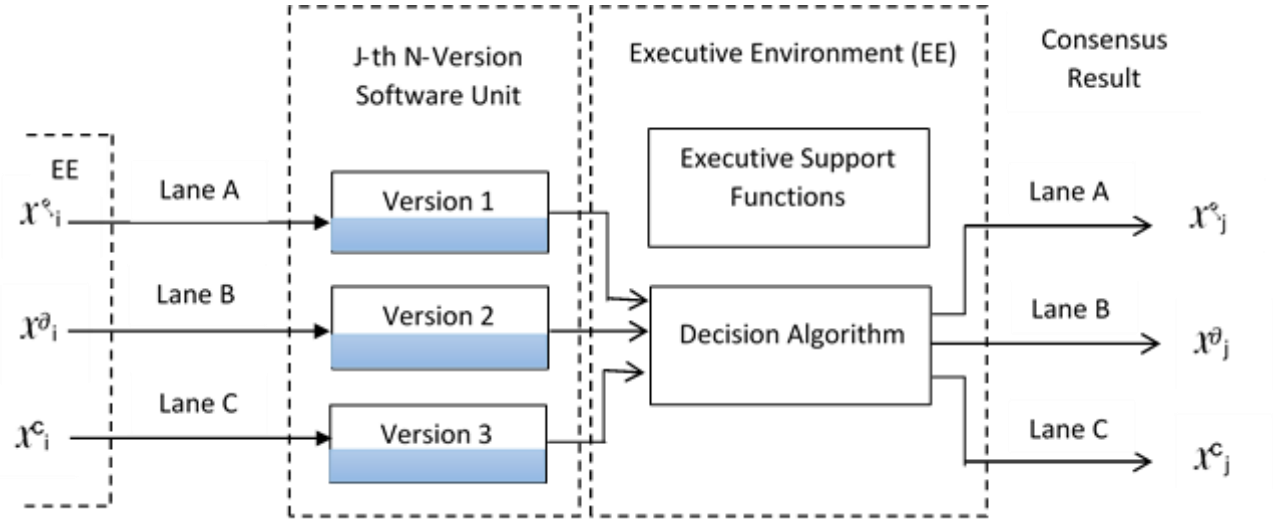

Fig. 1: The N-version software (NVS) model with n = 3

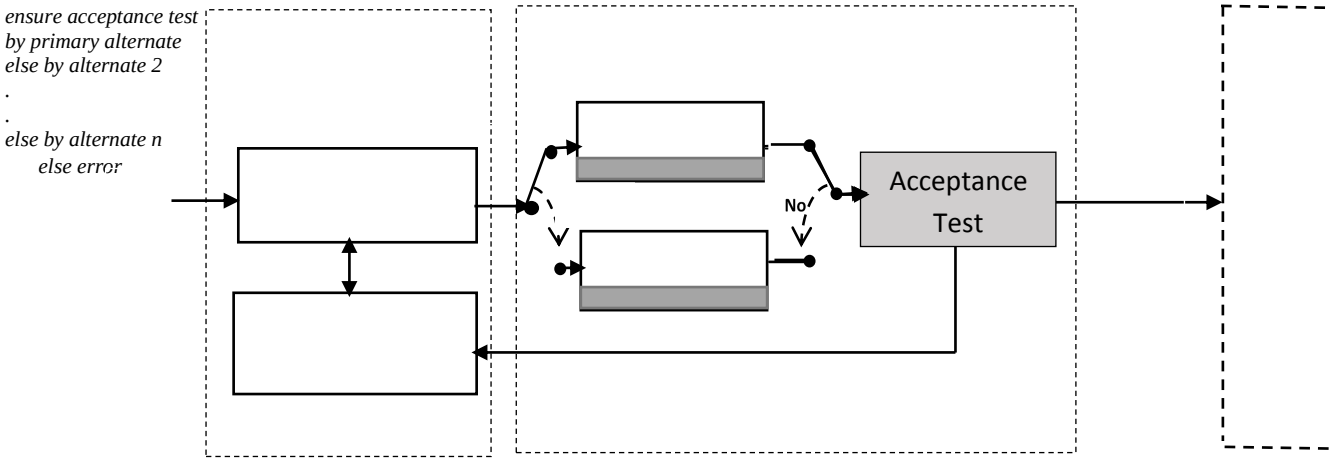

Fig. 2: The recovery block (RB) model

3.3 Other Derived Techniques

As mentioned before, there are numerated software fault tolerance techniques that are derived and developed from those two techniques (N-Versions Programming and Recovery Blocks) after applying some integration and updating processes on them. These techniques as follows:

3.3.1 N Self-Checking Programming

This technique is based on the using of multiple software versions that are combined with modified models of the Recovery Blocks and N-Version Programming. Hence, two models of this technique can be developed. The first one is the N Self-Checking programming using acceptance tests. The second model of the N Self-Checking programming is N Self-Checking programming using comparison for each pair of versions to detect errors [13].

3.3.2 Consensus Recovery Blocks

This technique is based on combining N-Version Programming technique and Recovery Blocks technique to gain better reliability than that achieved by using each technique alone. In other words, the Consensus Recovery Blocks technique is an integration of N-Version Programming technique and Recovery Blocks technique to try to overcome the shortages and difficulties those are embedded in each of those techniques when they designed and used as individuals [7, 13].

3.3.3 t/(n-1)-Variant Programming

Selection logic design is based on the theory of system _level fault diagnosis. Essentially, a t/(n-1)-VP architecture consisting of n variants and use the t/(n-1) diagnosibility measure to separate the faulty units to a subset of size at most (n-1) assuming that there are at most t units are faulty. Therefore, at least one non-faulty unit exists such that its output is correct and can be used as the result of computation for the module [3].

3.4 Exception Handling

Different definitions are available of exceptions. For example, an exception may be defined as an error or an event that occurs infrequently or unexpectedly, also an exception defined as an abnormal event or abnormal response from inside a module that indicates the detection of errors in the module. Exception handling is the immediate response and consequent action taken to handle one or more exceptions [14].

Although exception handling is one of the most powerful techniques for handling run-time errors, unfortunately there are many languages that not embed or support the exception construct. In such cases a work-around is needed to overcome this shortage [5, 14].

It makes good sense to always define pre- and post-conditions for encapsulated operations. The pre-condition may be translated into code to act as acceptance tests. If there is a test failure an exception may be raised. Alternatively, it may be sufficient to return an error indication to the calling unit. Post-conditions can be used to specify expected results when carrying out unit testing [5].

There are many requirements that must be considered in the design of a system that supplied with embedded feature of exception handling. Such as the possible events activating the exceptions, the effects of those events on the system, and the selection of appropriate recovery actions. For a software module, there are three classes of exception that activating events as follows: [5, 14]

- *Interface Exceptions*: These exceptions are activated by the self-protection mechanisms of a module when it detects an unacceptable service request. Then these exceptions will be handled by the module that requested the service.
- *Local Exceptions*: These exceptions are activated or triggered by the error-detection mechanisms of a module when it detects an error in its own interior operations. Then, these exceptions will handle by the module's fault tolerant capabilities.
- *Failure exceptions*: These exceptions are activated by a module when it detects an error but it faults processing mechanism did not have the ability to handle this error in correct way. So, failure exceptions inform the module ask for the service that some other means must be found to execute its function.

The concept of error containment and isolation is very important and must be considered in the design of a system that embed and support exception handling features, because this will lead to the design of effective exception handlers. These exception handlers must be supported with a proper design of system structure, actions, and error detection mechanisms in order to enclose and isolate the effects of errors within a particular set of interacting components at the moment the error is detected.

4. ADDITIONAL CONSIDERATIONS

There are some critical issues in the use of the software fault tolerance techniques that are reviewed above, such as:

- Research has demonstrated that the arguments for reliability through diversity (N-version programming) are not always valid. When developing software from the same specification, different teams made the same mistakes. Software redundancy did not give the theoretically predicated increase in system reliability. Furthermore, if the specification is incorrect, all versions will include the common specification errors. This does not mean that N-version programming is useless. It may reduce the absolute number of failures in the system. N-version programming gives increased confidence but not absolute confidence in the system reliability [3].
- There are not many but important differences between the N-versions technique and recovery block technique, as follows: [3, 7, 13]
  - The main difference between these two techniques is that the N-versions technique usually uses one generic decision algorithm (voter or decider), while the recovery blocks technique uses an acceptance tester (adjudicator) for each module which is application dependent.

  - The other important difference is that, in the beginning and also the conventional case, the multiple versions in the recovery blocks are executed on sequential manner. Later, the recovery block technique has been extended to include concurrent execution of the various alternatives. The N-versions programming is designed always to execute the multiple versions in parallel on separate processors.
  - So, in a sequential "retry system", the cost in time of trying multiple alternatives may be very expensive, especially for real-time system applications. Conversely, concurrent systems need the expense of N-way hardware and a communications network to connect them.

- The both techniques (N-versions technique and recovery block technique) have the advantages and disadvantages of the engineering trade-offs, especially economic costs, involved with developing. It is very important issue for the engineer to consider these costs when deciding the best technique to be implemented in his project [3].
- There is an argument that there is a difference between fault tolerance and exception handling. "The difference between fault tolerance versus exception handling is that exception handling deviates from the specification and fault tolerance attempts to provide services compliant with the specification after detecting a fault. This is an important difference to realize between trying to construct robust software versus trying to construct reliable software. Reliable software will accomplish its task under adverse conditions while robust software will be able to indicate a failure correctly", (hopefully without the entire system failing) [14].
- Theoretically, according to the reliability concepts, this combined approach (e.g., Consensus Recovery Blocks technique) has the likelihood of producing a more reliable piece of software, but it will be much more complex than either of the individual techniques. Hence, there should be a consideration that the added complexity could work against the system design in such a way that makes the design less reliable [3, 14].
- Although the previous mentioned techniques have many important advantages but also they include critical disadvantages, such as: [3, 7, 13]
  - in recovery blocks technique: the complexity of designing appropriate acceptance tests, and late results.
  - in N-version programming technique: concurrent systems need the expense of N-way hardware and a communications network to connect them, and there is a possibility that all the versions have different outputs.
  - in N self-checking programming technique, if one of the versions produces a result which is only slightly different from the other the acceptance test may not be able to determine that it is incorrect.

5. The Proposed Technique

A software fault tolerance technique that consists of two layers has been proposed in this paper. The layers are: the *general layer* and the *special layer*.

5.1 The General Layer

The general layer is the first layer in the proposed technique. This layer is a modified fault tolerance technique which its design and development will depend on accommodating the three software fault tolerance techniques previously mentioned in section 2 (N-Version Programming, Recovery Blocks and Exception Handling techniques). The architecture of the proposed system is consisting of the following components (see Figure 3):

- The *N-version software (NVS) model* with n = 2,
- The *Acceptance Tester* that will be taken from the recovery block technique, and,
- The *Exception handler* for local exception triggering events.

The two versions of the software will run in parallel. In each stage of processing there is a voter that will checks the truth of the stage's output, by comparing the results of the two versions and votes on the comparison. If the results of the two versions agree, then outputs the result, otherwise uses the acceptance tester to select the proper result. This means that the proposed system will include in its structure an acceptance tester that will activated (enabled) only when there is disagree in voting results.

The idea behind using only two versions of N-version software model is to decrease the system's complexity to minimum degree. The general concept is that "using more of software versions will increase the possibility of achieving higher reliability to the implemented system", but this thing may increases the system complexity in a way that affect its reliability in passive way.

An exception handling method is added to each software version, to support the dependability of our proposed system. The exception handlers in each version are signalled by a module as soon as its detection mechanisms of error find an error in its own interior operations. These exceptions should be handled by the module's fault tolerant capabilities.

The general layer can be named the ***2-Version Software with Acceptance Test Support***. This layer can also be implemented by using the other software fault tolerance techniques, therefore it termed as general layer.

There is a possibility that the first layer of the proposed technique cannot select or produce a correct or an acceptable output. This situation can be occurring when the two versions have different outputs and the acceptance test may not be able to determine which output is incorrect when one of the versions produces a result which is only slightly different from the other. To overcome this problem, the system will switch to the second layer to handle the output in order to produce a correct or an acceptable output.

5.2 The Special Layer

The second layer is called the special layer because its implementation depends on developing a specific mechanism that differs from the conventional software fault tolerance techniques but this mechanism will be apply to support the functionality of these techniques.

As mentioned before, when the first layer fails to produce a correct or an acceptable output, the system will switch to the second layer to overcome this problem by using unconventional ways to produce the appropriate output.

The unconventional ways that will be adopted in the development of the second layer mechanism are as follows:

- Assigning dynamic weights for each software version that exists in the first layer. The weights will be generated from normal cases when the acceptance tester is able to determine the correct or the acceptable output. There is a counter associated with each version's weight, where this counter will be incremented in each time the output of the related version is selected by the acceptance tester and decremented in each time the output of the other version is selected. In the same way the counter of the other version will work. Also, there is another counter that its weight will be incremented in each time the acceptance tester succeeds in selecting the appropriate output; this weight will be called the Total weight. The following formula can give a confidence measurement for the version:

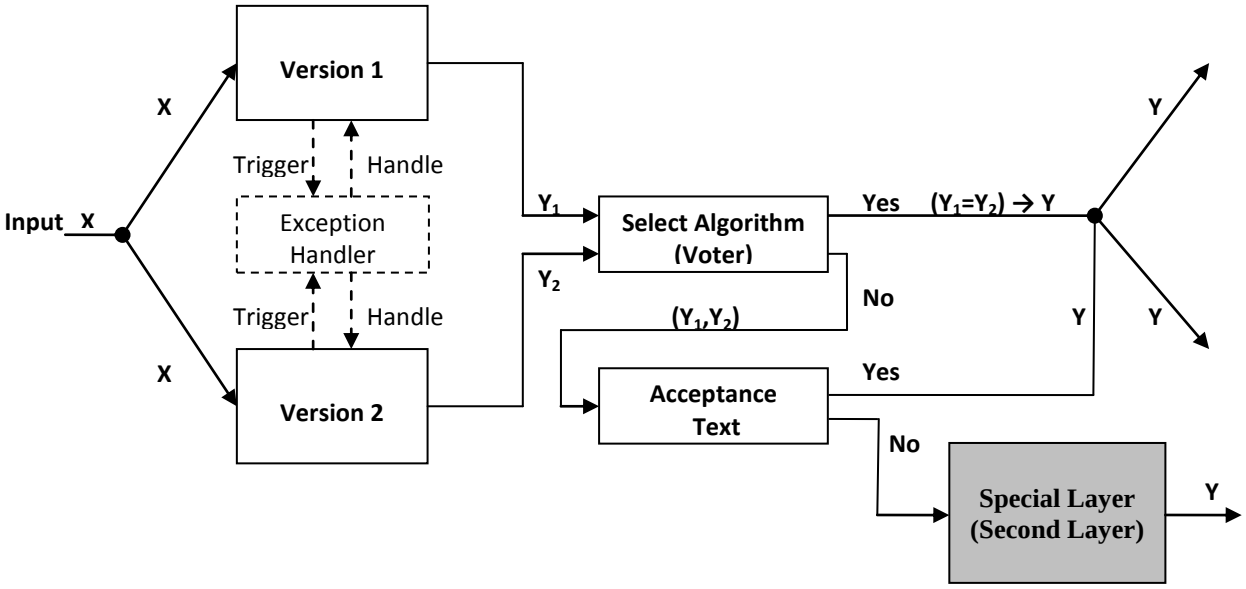

Fig. 3: The proposed technique (General Layer)

$$Confidence\ of\ a\ version = \frac{version\ weight}{Total\ \ weight} \times 100$$

- An archive file must be created to contain the outputs that generated from normal cases when the voter or the acceptance tester approves these outputs as correct or acceptable outputs. Also, this file must contain the correspondent inputs for these outputs and the identification of the version that produce the appropriate output.
- The proposed mechanism can produce the appropriate output according to the following two criteria:
  - It can use history of previous results to produce an expected output value by selecting the version output value that is closest to the expected value. Thus, a Divergence factor must be calculated depending on the following formula:

$$Divergence = \left| \ version\ output - \exp ected\ \ output \ \right|$$

  - It can use the value of the confidence factor of a version output as a second criterion to support the process of selecting appropriate output.
- So, the procedure of selecting the appropriate output will apply the following steps:
  - If the minimum divergence value and the maximum confidence value are associated with same version, then adopt this version output as the appropriate output.
  - If the minimum divergence value and the maximum confidence value are associated with different versions, then select the version with the minimum difference value and adopt its output as the appropriate output. The difference value can be calculated depending on the following formula:

$$Difference = \left| \ Divergence - Confidence \ \right|$$

6. CONCLUDING REMARKS

Balancing between system complexity and reliability is an important issue that must be considered in the system design to get more reliable and less complex system to achieve dependable software. For this reason, the designed algorithm was proposed using N-version software technique with only two versions to minimize the complexity of the system, but on other hand, creating dependable versions must be guaranteed. That can be achieved through trying to prevent the incident of common faults between the system versions. Usually, these common faults occur because of the design of the two software versions will depend on the same faulty system specifications. So, to avoid this type of faults, the following issues must be considered:

1- Using two different sources of system specifications to develop each software version.

2- Using two different teams to develop the system, each team will be responsible about developing a version depending on its own system specification source that differ fro the other team.

3- Using different programming languages, compilers, paradigms, data structure… etc.

But there is no guarantee that the common faults can be recovered totally. So, a proposition that the implementation of another layer in the system to support the first layer when it is fail to produce appropriate output. One of the ways that used in the second layer is generating confidence weights to the versions.

There is a previous work that proposes the use of genetic algorithms or neural networks for implementing the voter in such a way that performance is associated to the application and the particular characteristics of the software versions.

As known fact, the implementation of neural networks or genetic algorithms will increase the complexity of the system, considering that the system complexity is already increased due to the application of software fault tolerance mechanism on it. Therefore, an algorithm for proposed more simple method for weight generation that did not increase the system complexity in obvious way.

7. REFERENCES

[1] Ian Sommerville, "Dependable Software Development", pp 3 – 4, 2000.

[2] Ian Sommerville, "Software Engineering", 6th Edition, Addison Wesley, 2001.

[3] Wilfredo Torres-Pomales, "Software Fault Tolerance: A Tutorial", NASA, pp 8 – 9, 2000.

[4] Jashan Deep and Dr. Rajiv Mahajan, "A Survey on Software Fault Tolerance in Parallel Computing", International Journal of Engineering Sciences & Research Technology (IJESRT), ISSN: 2277-9655, 2013, Iraqi Virtual Scientific Library (IVSL).

[5] Titos Saridakis and Valerie Issarny, "Fault Tolerance Software Architectures", Institute National De Recherche En Informatique Et En Automatique (INRIA), 1998, Iraqi Virtual Scientific Library (IVSL).

[6] Dr. K. C. Joshi, "Techniques of Software Fault Tolerance", International Journal of Computer Science & Engineering Technology (IJCSET), Vol. 3, No. 4, 2012, Iraqi Virtual Scientific Library (IVSL).

[7] Michael R. Lyu, Jia-Hong Chen, and Algridas Avižienis, "Experience in Metrics and Measurements for N-Version Programming", Citeeer^{x}, Iraqi Virtual Scientific Library (IVSL).

[8] Michael R. Lyu, "Software Fault Tolerance", Wiley, pp 23 – 27, 1995.

[9] Jim Cooling, "Software Engineering for Real – Time Systems", Addison Wesley, 2003.

[10] Arun Somani & Nitin Vaidya, "Fault Tolerance", IEEE Computer, pp 7 , 25, 1997.

[11] Wanlei Zhou, " Fault Tolerant Computing - Study Guide ", Deakin University, pp 66, 2001.

[12] M.Soneru, "Fault Tolerance", CS-550[SaS], pp 10. Zaipeng Xie, Hongyu Sun and Kewal Saluja, "A SURVEY OF SOFTWARE FAULT TOLERANCE TECHNIQUES", University of Wisconsin-Madison/Department of Electrical and Computer Engineering 1415 Engineering Drive, Madison WI 53706 USA.

[13] Jie Xu & Brian Randell, "Exception Handling and Software Fault Tolerance", DSN-2000 Tutorial 4, pp 6, 2000.

[14] Chris Inacio, "Software Fault Tolerance", 18-849b Dependable Embedded Systems, 1998.