A Proposed System for Real Time Adaptive N Version Programming

Pooja Yadav
Bharat Electronics Limited,
Ghaziabad – 201010,
Uttar Pradesh, India.

Shilpa Singh
Bharat Electronics Limited,
Ghaziabad – 201010,
Uttar Pradesh, India.

Ajay More
National Informatics Center
New Delhi,
India.

ABSTRACT
N-version programming is a fault tolerance technique that depends on a generic decision algorithm to determine a consensus result from the results delivered by two or more member versions of the software. In N-version programming, N teams of developers work independently on N unique but equivalent implementations of the same program. The major objectives of the NVP process are to maximize the independence of version development and to employ design diversity in order to minimize the probability that two or more member versions will produce similar erroneous results that coincide in time for a decision (consensus) action. But this fault-tolerance technique has been criticized for its statistical assumptions and high cost. A solution is proposed in which there are N versions of the software out of which t versions implement only a subset of the entire functionality which is highly critical while (N – t) versions implement the entire functionality. One of the biggest hurdles in using N version programming for fault tolerance is its high implementation cost. This proposed technique minimizes the cost of implementation while improving the efficiency and reliability of the system.

Keywords
N version programming, fault tolerance, reliability, real time systems

1. INTRODUCTION
Multi-version or N-version programming [1, 2] has been proposed as a method of providing fault tolerance in software where high reliability is a major concern. The approach requires independent development of multiple versions (i.e. ‘‘N’’) of a software for some application. These versions are executed in parallel in the application environment; each receives identical inputs and each produces its output which is independent from all other versions. The outputs are collected by a voter or decision software and, in principle; they should all be the same. In practice there may be some disagreement. If this occurs, the results of the majority (assuming there is one) are assumed to be the correct output, and this is the output used by the system.

The basic technique for NVP as described in [3] is as follows-

i. The basic functional units of the software system consist of N parallel independent versions of programs with identical functionality: version 1, version 2. . . version N.
ii. The system input is given to all the N versions.
iii. The individual output for each version is fed to decision software.
iv. The decision software determines the system output using a specific decision algorithm.

Refer Fig.1 for block diagram of N-version programming

1.1 Example used in the Proposed System for N Version Programming
The proposed system will be explained through the example of missile control system. The missile control system will implement the following functionalities

1. Implementation of missile firing order.
2. Automatic positioning of missile according to direction of target.
3. Display of planned missile trajectory on map according to latitude and longitude of source and destination.
4. Generation of Air space clearance for the missile which means there should be no civil or military aircraft (which is not the target) in the path of the missile.
5. Whether status monitoring.
6. Entry of location and parameters of other weapons in the database.
7. Suggestion of alternate Weapon solutions.
8. Fuel status entry.
9. Payload entry (type and quantity).
10. Entry of on duty officer.
11. Configuration of duty timing of officer.

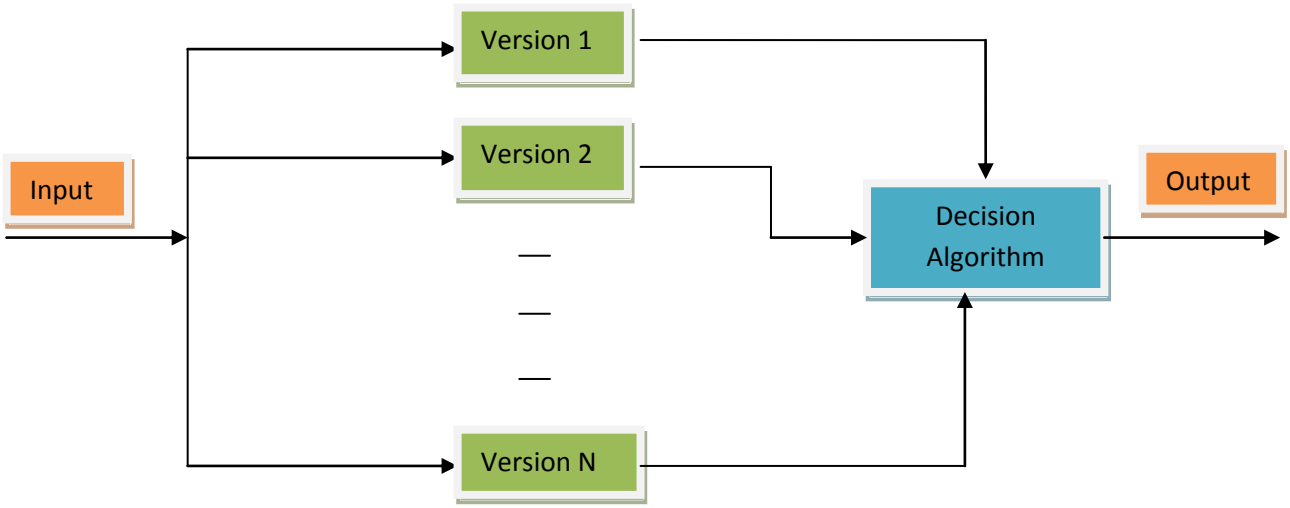

Figure 1: Block Diagram of N version programming technique

Here, the first four functionalities i.e. implementation of missile firing order, automatic positioning of missile, display of missile trajectory and generation of air space clearance are highly critical while the others are non critical. These highly critical functionalities must be separated from non critical functionality so that even if there is a system failure due to some unwanted condition in the non critical functionality the critical functionality is still operational.

In this implementation of N Version programming the software containing all the eleven functionalities will be replicated (N - t) times using different design techniques. Also there will be t replicas of the software containing only the first four functionalities which are highly critical.

This will save the software implementation cost as well as hardware cost while improving the efficiency and reliability of the software. Also, a cost comparison between the old system which replicates the software containing the entire functionalities N times and the newly proposed system will be shown.

2. A DESIGN PARADIGM FOR N VERSION PROGRAMMING IN REAL TIME SYSTEMS

All N versions of the software have to be developed independently by separate development teams. "Independent development of programs" here means that the programming efforts are carried out by N individuals or groups that do not interact with each other regarding the programming process. Wherever possible, different algorithms and programming languages are used in each effort. First, the initial specifications of the software are developed. The initial specification is a formal specification in a specification language. The goal of the initial specification is to state the functional requirements of the software completely and unambiguously, while leaving the widest possible choice of implementations to the N programming efforts [4].

A major observation concerning N-version programming is that its success as a method for on-line tolerance of software faults depends on whether the residual software faults in each version of the program are distinguishable (i.e., no two versions of the software exhibit identical faults). The NVP approach was motivated by the "fundamental conjecture that the independence of programming efforts will greatly reduce the probability of identical software faults occurring in two or more versions of the program" [4].

Diversity of implementation should be stated in the specifications of all N versions of the software. Diversity may be specified in one or more of the following elements of the NVP process: training, experience, and location of implementing personnel; application algorithms and data structures used; programming languages used for implementation; software development life cycle used (for example one version may be developed using waterfall model, other may be developed using spiral model and some other may use evolutionary model); programming tools and environments; testing methods and tools. The purpose of such required diversity is to minimize the opportunities for common causes of software faults in two or more versions (e.g., compiler bugs, ambiguous algorithm statements, etc.), and to increase the probabilities of significantly diverse approaches to version development. It is also possible to impose differing diversity requirements for separate software development stages, such as design, coding, testing etc [2].

Also, special dedicated hardware processors might have to be implemented or procured in advance for the execution of NVS systems, especially when the NVS supporting environments need to operate under certain stringent requirements (e.g., timing constraints in real time systems accurate supervision, efficient CPUs, etc.). Diversity in version implementation can also attained by using hardware of different specifications for each version. The options of combining software and hardware diversity for a hybrid configuration could also be considered [5], [6], [7].

The objectives of introducing diversity in implementation are [8]:

1. To reduce the possibility of same oversights, mistakes, and inconsistencies in the process of software development and testing in all versions;
2. To eliminate most perceivable causes of related design faults in the independently generated versions of a program, and to identify causes of those which slip through the design process;

3. To minimize the probability that two or more versions will produce similar erroneous results that occur at the same time.

The above objectives are highly essential for real time systems as the accurate output must be obtained under highly stringent timing constraints. There may be severe consequences if the output misses the deadline.

3. PROPOSED SYSTEM

N Version programming technique can be modified in the following manner to suit the requirements of real time systems and also to maximize reliability and minimize cost.

In the proposed system, it is assumed that the amount of processing required for implementation of critical functionality in software is much less than the amount of processing required for non critical functionality.

Refer Fig.2 for block diagram for proposed system.

Total no. of software versions = N

No. of versions containing critical functionality only = t

No. of versions containing complete (critical + non critical) functionality = N – t

Also, t > (N - t), as there must be more number of replicas for critical functionality. There should be some replicas for complete (critical + non critical) functionality also so that the system functionality does not degrade even if some of the software versions (critical or non critical) fail. At the same time the number of complete versions is kept much less than the number of critical versions in order to save cost but retain the reliability which would have been achieved by N complete versions.

Here the t versions of the software containing critical functionalities also receive the same inputs as the (N – t) versions containing critical + non critical functionality. But the t versions of the software discard the inputs which are not required for critical functionality.

3.1 Decision Algorithm for Proposed System

The decision algorithm for the proposed system is described below. This decision algorithm is more suitable for real time systems where strict timing constraints need to be followed. For explaining this algorithm a variable “Critical_flag” is used for determining if the output required is critical. If Critical_flag is equal to 1 the output required is critical otherwise the output is non critical. Also, a variable “flag” is used to determine if the critical output has already been obtained. Initially the value of flag is 0. It is set to 1 whenever the critical output is obtained otherwise it remains 0. It is assumed that the actual deadline of completing the critical task (for example implementation of missile firing order) is *b units of time*. This algorithm assures that this critical task is completed in *a units of time* where a is slightly less than b. This is done to ensure that the system never misses the deadline.

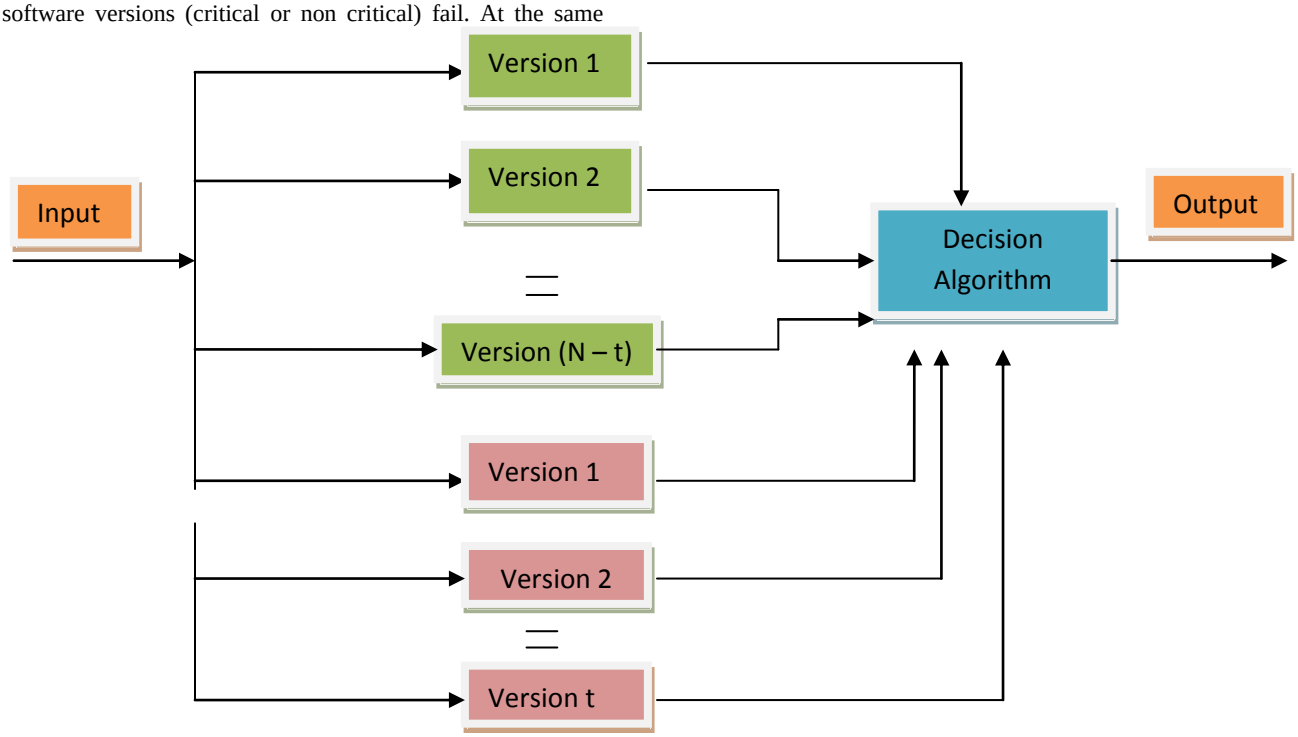

Figure 2: Block Diagram for Proposed System

Begin
Set flag=0
If ((Critical_flag == 1) && (flag==0))
{
Receive output from t critical functionality versions (version 1, 2 ….t)
Wait (a units of time)
If (number of outputs received > 0)
{
Compare the output from one or more t critical functionality versions received within a units of time and forward the majority result as output.
}
Set flag=1
}Else if ((Critical_flag == 1) && (flag!=1))
{
Wait ((b – a) units of time)

Receive critical output from any of the N versions (version 1, 2 ….N)

Compare the output from one or more of the N versions obtained within (b - a) units of time and forward the majority result as output.

Set flag = 1

}

Else if (Critical_flag == 0)

{

Receive output from (N – t) non critical functionality versions (version 1, 2 …. (N - t))

Compare output from (N – t) complete functionality versions and forward the majority result as output.

}

End

The t critical functionality versions are programmed to give the output in *a units* of time. For critical functionality, the decision algorithm waits for a units of time for obtaining the output from the t critical functionality versions. If it obtains the output from one or more t critical functionality versions within a units of time it takes these outputs as input and outputs the result which is obtained from the majority of the t versions. It sets the value of variable flag to 1.

In the extremely rare case if all the t critical functionality versions fail to give the output within a units of time, the decision algorithm waits for another (b - a) units of time. This time the decision algorithm considers the output of all the N versions (critical functionality versions as well as complete functionality versions) and if it obtains the critical output from one or more of the N versions, it takes this output as input and outputs the result which is obtained from the majority of the N versions. For all normal cases output from t critical versions is used for critical functionality because response time is much smaller in t critical versions than (N – t) complete versions (t critical versions take less processing time due to absence of non critical tasks).

For non critical functionality, the decision algorithm takes the output from the (N – t) complete functionality versions as input and outputs the result which is obtained from the majority of the (N - t) versions. Here there are no timing constraints as the functionality is not critical

3.2 Advantages of Proposed System

3.2.1 Cost Saving

Old system consists of only N complete functionality versions of the software. Proposed system consists of (N - t) complete functionality versions and t critical functionality versions of the software.

Table 1. Comparison of cost of old system and proposed system

Requirements	**Old system**	**Proposed system**
Hardware	All high end processors	(N - t) high end processors and t low end processors
Cost	High	Significantly lower as compared to old system

Cost of old system:

Total no. of software versions = N

All N versions of the software have been developed independently by separate development teams and contain complete functionality of the software.

Let cost of Hardware system to be used for running each complete functionality version software be $A

Total Cost of hardware = N* $A---------- (1)

Cost of proposed system

Total no. of software versions = N

No. of versions containing critical functionality only = t

No. of versions containing complete (critical + non critical) functionality = (N - t)

All (N - t) complete functionality versions and t critical functionality versions of the software have been developed independently by separate development teams and contain complete functionality of the software.

Let cost of Hardware system to be used for running each critical functionality version software be $B

$A >> $B, high end processors are much more costly as compared to low end processors**.**

Total Cost of hardware = ((N-t) * $A) + (t * $B)-------- (2)

By using equations (1) and (2),

[N* $A] >> [((N-t) * $A) + (t * $B)]

Hence, the total cost of old system is much higher than the total cost of proposed system.

3.2.2 Higher Reliability

Since the critical t versions contain only a small part of the complete software reliability is greatly enhanced as there are less chances of system failure due to memory leakage, segmentation fault etc which are encountered in large software systems.

Since all the versions work in parallel, the system is considered to be operational if any one of the N versions is available and the system is considered failed when all parts fail. The reliability of parallel systems as given in [9] is:

If there n parallel connected components, with reliability of $R_k(t)$ for kth component.

Then the total failure probability is,

$$Q(t) = \prod_{i=1}^{n} Q_i(t) = \prod_{i=1}^{n} (1 - R_i(t)) \quad \text{-------(3)}$$

Therefore the total reliability is,

$$R(t) = 1 - Q(t) = 1 - \prod_{i=1}^{n} (1 - R_i(t)) \quad \text{------(4)}$$

Reliability increases with increase in the number of parallel components.

The implications of the above equation are that the combined reliability of two components in parallel is always much higher than the highest reliability of its individual components [10].

Therefore, according to equation (4) the reliability of

$$R_{new} = 1 - [(1 - R1)^{t}(1 - R2)^{(N - t)}] \text{----------(5)}$$

Where R1 is the combined reliability of t critical functionality versions working in parallel and R2 is the combined reliability of (N – t) complete functionality versions working in parallel.

According to equation (4) the reliability of old system is,

$R_{old} = 1 – [(1 – R2)^N]$----------(6)

Since, R1 > R2 then according to equations (5) and (6) $R_{new} > R_{old}$, because combined reliability of parallel system is always much higher than the highest reliability of its individual components and the highest reliability of t critical functionality versions is more than the highest reliability of (N – t) complete functionality versions.

3.2.3 Other Advantages

1. Also since all the critical t versions are developed independently by separate teams and are also relatively smaller as compared to (N - t) complete versions the chances of human error in coding, design etc is greatly reduced.

2. Complete and through testing of smaller critical functionality versions can be done in a cost effective and efficient manner. It is also less time consuming as compared to testing the larger complete functionality versions of the software.

4. CONCLUSION AND FURTHER SCOPE

The proposed system of N version programming is found to be highly suitable for mission critical real time systems where some critical output is required within a fixed deadline. As NVP is based on design diversity technique, the built program will fail independently and with low probability of coincidental failures. This ensures that one of the other versions will continue to provide the required functionality. This system provides a high degree of fault tolerance which is required for safety critical systems. Also this system is proved to be highly reliable and cost effective. This system can be used for various defense applications such as missile control system, real time wireless communication and various other army, navy and air force systems. Such a system can be extended for use in medical applications, industrial plant controllers as well as safety critical systems where reliability and fault tolerance are criteria of major concern.

5. ACKNOWLEDGEMENT

The authors wish to thank Prof. D.S. Yadav, Director, Government Engineering College, Banda, U.P., India, and Mr. Dhananjay Pandey, Accreditation Officer, Quality Council of India, New Delhi for their valuable suggestions and unconditional support for the revision of this paper.

6. REFERENCES

[1] L. Chen and A. Avizienis, ‘‘N-version programming: A fault-tolerance approach to reliability of software operation,’’ *Digest of Papers FTCS-8: Eighth Annual International Conference on Fault Tolerant Computing*, Toulouse, France, pp. 3-9, June 1978.

[2] A. A. Aviˇzienis. *The Methodology of N-Version Programming*, chapter 2. John Wiley and Sons, 1995.

[3] Jeff Tian, *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement*, John Wiley and Sons, 2005, pp. 272-275.

[4] A. Aviˇzienis and L. Chen. “On the implementation of N-version programming for software fault tolerance during execution”. In *Proc. IEEE COMPSAC 77*, pages 149–155, November 1977.

[5] K.H. Kim, ‘‘Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults,’’ in *Proceedings IEEE 4th International Conference on Distributed Computing Systems*, pp. 526-532, San Francisco, California, May 1984.

[6] J.-C. Laprie, ‘‘Hardware-and-Software Dependability Evaluation,’’ in *Proceedings 11th World IFIP Congress*, pp. 109-114, San Francisco, California, September 1989.

[7] J. Lala, L. Alger, S. Friend, G. Greeley, S. Sacco, and S. Adams, ‘‘Study of A Unified Hardware and Software Fault Tolerant Architecture,’’ Report No. 181759, NASA Contract No. NAS1-18061, January 1989.

[8] Michael R. Lyu, Algirdas Avižienis, “Assuring Design Diversity in N-Version Software: A Design Paradigm for N-Version Programming”, in *Chapter Dependable Computing for Critical Applications 2 Volume 6 of the series Dependable Computing and Fault-Tolerant Systems* pp 197-218, 1992.

[9] Romeu, J.L. *Reliability Estimations for Exponential Life*, RAC START, Volume 10, Number 7.

[10] Hoyland, A. and M. Rausand, *System Reliability Theory: Models and Statistical Methods*, Wiley, NY, 1994.