Fuzzy logic based N Version Programming for improving Software Fault Tolerance

Y. Manas Kumar[1], Y.Jnapika[2]
1 & 2 Assistant Professor, CSE Department, Pragati Engineering College (Autonomous), Andhra Pradesh, India

Abstract: Software Fault tolerance refers to the in-built ability of a software system to recover from failure. The gravity of the failure must be considered while designing such a system. In this paper we propose a hybrid model to improve software fault tolerance which is inspired by the features of N-Version software and Recovery Block Model. Our proposed model envisions factors to measure the consensus and disagreement between the developer teams and acceptance testing teams. Selective features are extracted and used to develop our proposed model from the existing models. Our model is a step towards enhanced software fault tolerance.
Key Words: Fault tolerance, Consensus, Disagreement, N-Version, Recovery Block.

INTRODUCTION:

The N-version software concept attempts to parallel the traditional hardware fault tolerance concept of N-way redundant hardware. In an N-version software system, each module is made with up to N different implementations. Each variant accomplishes the same task, but hopefully in a different way. Each version then submits its answer to voter or decider which determines the correct answer, (hopefully, all versions were the same and correct,) and returns that as the result of the module. This system can hopefully overcome the design faults present in most software by relying upon the design diversity concept. An important distinction in N-version software is the fact that the system could include multiple types of hardware using multiple versions of software. The goal is to increase the diversity in order to avoid common mode failures. Using N-version software, it is encouraged that each different version be implemented in as diverse a manner as possible, including different tool sets, different programming languages, and possibly different environments. The various development groups must have as little interaction related to the programming between them as possible. N-version software can only be successful and successfully tolerate faults if the required design diversity is met.
The dependence on appropriate specifications in N-version software, (and recovery blocks,) cannot be stressed enough. The delicate balance required by the N-version software method requires that a specification be specific enough so that the various versions are completely inter-operable, so that a software decider may choose equally between them, but cannot be so limiting that the software programmers do not have enough freedom to create diverse designs. The flexibility in the specification to encourage design diversity, yet maintain the compatibility between versions is a difficult task, however, most current software fault tolerance methods rely on this delicate balance in the specification.

MATERIALS:

Fault Tolerance with N-Version Programming:

N-version programming (NVP) is another way to tolerate software faults by directly introducing duplications into the software itself. NVP is generally more suitable than recovery blocks when timely decisions or performance are critical, such as in many real-time control systems, or when software faults**,** instead of environmental disturbances, are more likely to be the primary sources of problems.

METHOD:

The basic technique is illustrated below and briefly described below:

NVP: Basic technique and implementation

Step 1: 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.

Step 2: The system input is distributed to all the N versions.

Step 3: The individual output for each version is fed to a decision unit.
Step 4: The decision unit determines the system output using a specific decision algorithm.

The most commonly used algorithm is a simple majority vote, but other algorithms are also possible. The decision algorithm determines the degree of fault tolerance. For example, when the simple majority rule is used, the system output would be the correct one as long as at least half of the versions are operational and produce correct results. In this case, we say that the overall system is fault tolerant up to N/2 -1.

The determination of overall system reliability is one major topic in fault tolerance studies. System reliability can be determined by the reliability of its individual versions, the decision algorithm, as well as the relationship among these different versions. For example, if all individual versions are highly correlated, then they tend to fail at the same time or under similar operational conditions, thus defeating the whole purpose of multiple versions. Therefore, the most fundamental assumption and the enabling factor in NVP is that faults in different versions are independent .When different versions are independent, even if there is a fault that causes a local failure in version i, the whole system is likely to function correctly because the other (independent) versions are likely to function correctly under the same dynamic environment. In this way, the causal relation between local faults and system failures is broken for most local faults under most situations, thus improving the quality and reliability of the software system. One of the main research topics in NVP is to ensure that the software versions are as independent as possible so that local faults can be tolerated and the resulting local failures can be contained effectively. The other assumption for fault tolerance, the rare and unanticipated event assumption (Al), is not directly used in NVP. However, it does affect NVP implementation to a large degree. NVP costs significantly more to implement than a single version, typically by a factor of N or more if we count also the decision unit and the coordination in addition to the N individual versions. Therefore, in actual implementations, we can only afford it for selected critical components or units in large systems where rare individual failure events that cannot be anticipated ahead of time need to be tolerated. For frequent and anticipated events, other solutions are much more effective and economical.

DISCUSSION:

We propose an aggregation of NVP Model and RB-Model by selecting features efficient to our context. The N versions which are considered will contain recovery blocks as per the specification document. Our hybrid decision algorithm will be fed with these versions. The hybrid decision algorithm works as mentioned below.

Due to the presence of recovery blocks, some version may have consensus while few versions may have disagreements. Measuring consensus, disagreements are difficult due to different viewpoints in specification operations.

If consensus is represented by λ, and disagreements by μ, then

$$P(\lambda) = 1 - P(\mu) \qquad \text{---------- Equation (1)}$$

Consequently, if $P(\lambda)$ is high, it means $P(\mu)$ is low and vice-versa.

ANALYSIS:

Now, consider 3 versions of a program/module, say P_i, P_j, P_k. If $P_{ij}(\lambda)$ is higher than $P_{jk}(\lambda)$ which is in turn higher than $P_{ki}(\lambda)$; we shortlist P_i, P_j and ignore P_k. After this stage only, we apply acceptance test and eliminate P_i or P_j to get the candidate version of the program.

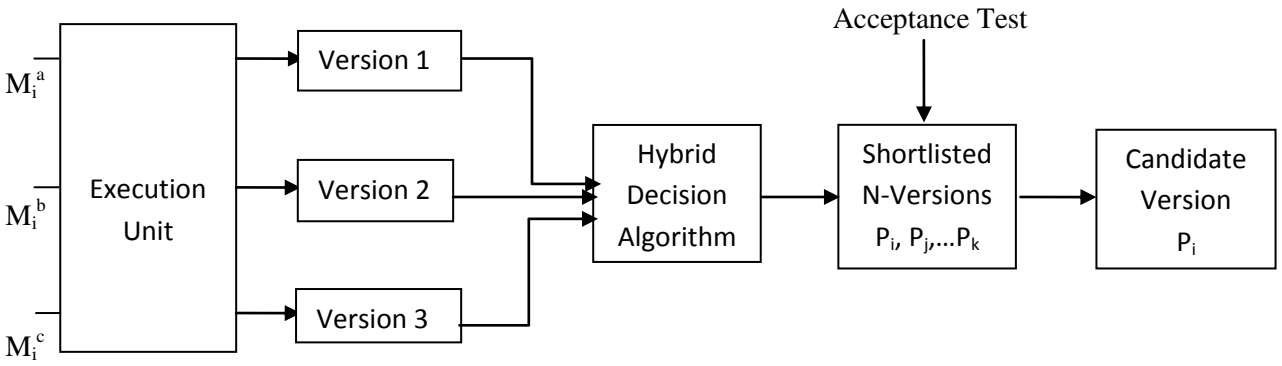

Fig. 1 Using NVP Model in Fault Tolerance

Factors on which consensus depends are manifold. One aspect is the service limitations. Service limitations are derived from user requirements document. The ambiguity in understanding the requirements document may lead to different versions leading up to a high (or) medium amount of disagreement. Yet another factor on which consensus depends is on the development strategy used by the teams irrespective of team-composition and other aspects like dissimilar software and hardware tools employed while developing the versions.

RESULT:

We obtain the candidate version of the program based on the factors like consensus and disagreements from N versions.

RECOMMENDATIONS:

We recommend observing the below threats to validate the proposed model

1. Using fuzzy measure to indicate consensus, disagreements bring a certain level of unsureness while deciding up to what level the fault can be tolerated while the software is operable.

2. The next threat comes from using the probabilistic approach. The degree of belief in service operation of the software depends on various reasons. For instance, false positives can exist in software faults. The user thinks a fault will be tolerated by the software as the condition in acceptance test has been met. But, in reality 100% acceptance testing is not performed and few faults may still exist in the software. They may be hidden (or) ignored while doing the acceptance test. This laxity in user or tester may be a false software tolerance issue. The user may be marginally threatened by this aspect. Nevertheless we should not ignore this threat.

CONCLUSION:

In future, we intend to enhance the degree of certainty while calculating the values of measurement for consensus & disagreements. We plan to set a clear guideline package, using which consensus and disagreements can be confirmed. We also plan to reduce fuzziness while measuring consensus and disagreements. Software Fault tolerance is a practise towards high quality software so in emerging technological markets its significance cannot be ruled out.

REFERENCES:

1. Cai, X, M. R. Lyu and M. A.Vouk. “Experimental Evaluation of Reliability Features of N-Version Programming.” Proc. 16th IEEE Intl. Symp. on Software Reliability Engineering, Nov. 2005, pp 161-170.
2. Daniels, F., K. Kim and M. Vouk. The Reliable Hybrid Pattern: A Generalized Software Fault Tolerant Design Pattern. Presented at PLoP 1997, Monticello, IL. September 1997.
3. Hanmer, R. Patterns for Fault Tolerant Software. Chichester, UK: John Wiley & Sons, 2007.
4. Knight, J. C. and N. G. Leveson, “A reply to the criticisms of the Knight & Leveson experiment,” *SIGSOFT Softw. Eng. Notes* 15, 1 (Jan. 1990), 24-35.
5. Saridakis, T. A System of Patterns for Fault Tolerance, Proceedings of EuroPLoP 2002, Kloster Irsee, Germany, July 2002, pp 535-582.