Asian Journal of Computer Science and Technology
ISSN: 2249-0701 (P) Vol.8 No.S3, 2019, pp.16-18

DOI: https://doi.org/10.51983/ajcst-2019.8.S3.2112

The Methodology of N-Version Programming

Phalguna Rao Kuna
Research Scholar, Department of Computer Science,
Sri Satya Sai University of Technology and Medical Sciences, Bhopal, Madhya Pradesh, India
E-Mail: kprao21@gmail.com
(Received 23 March 2019; Revised 1 April 2019; Accepted 15 April 2019; Available online 22 April 2019)

***Abstract* - Software Fault Tolerance is evolved as a technique to increase the dependability of computing systems. Because of limitations with producing of error free software, Majority of software errors are design faults. The root cause for software design errors is the complexity of the problem domain. Software Fault Tolerance (SFT) has become an important concern A number of Fault Tolerance techniques designed at minimizing the effect of software faults are being investigated. An N-version software (NVS) unit is a fault tolerant software unit that depends on a generic decision algorithm to determine a consensus result from the results delivered by two or more member versions of the NVS unit. Results of five consecutive experimental investigations are summarized, and a design paradigm for NVS is presented. In this paper, a critical review of NVP is presented. The advantages, current challenges, and further research areas of NVP are discussed.**
***Keywords:* Design Diversity, Software Complexity, Software Fault Tolerance, N-Version Programming**

I. INTRODUCTION

Services in today's computation based society must be highly dependable. Unplanned service downtime causes revenue loss and, in some cases, contractual penalties. Hence design of fault tolerant systems has gained significant attention. Ensuring shared resources are available despite the failure of certain hardware or a software component is a tremendous challenge for IT specialists. The concept of Fault Tolerance techniques through redundant hardware components was conceived in the early 1950s [5] [6]. Software Fault Tolerance is the ability of software to detect and recover from a fault that is happening or has already happened in either the software or hardware in the system. As a result of research efforts to apply Fault Tolerance to software design faults, a number of techniques have evolved. The following sections give a brief introduction to various techniques and a critical review of N-Version Programming approach.

II. SOFTWARE FAULT TOLERANCE TECHNIQUES

Software Fault Tolerance can be broadly classified into two groups. Single version software and Multiversion software techniques. Single version techniques focus on improving the Fault Tolerance of a single piece of software by adding mechanism into the design, targeting the detection, containment, and handling of errors caused by the design faults. Some of the key attributes of single version techniques are modularity, system closure, atomicity of actions and exception handling.

Multi version Fault Tolerance is based on the use of two or more versions of a piece of software executed either in sequence or in parallel. The modularity, system closure, atomicity of actions and exception handling attributes are desirable and advantageous in each version of the multiversion techniques too. Some of the classical techniques of multiversion Software Fault Tolerance are Recovery blocks (RB) and N-Version programming. Both of these techniques are based on design diversity.

A. Recovery Block Technique

This technique was evolved as a result of first long term systematic investigation of multiversion technique initiated by Brian Randell in early 1970s [4]. In this technique, alternate software versions are organized in a manner similar to the dynamic redundancy (standby) technique in hardware. Its objective is to perform runtime Software Fault Tolerance detection by an acceptance test performed on the results delivered by the first version. Its objective is to perform runtime Software Fault Tolerance detection by an acceptance test performed on the results delivered by the first version. If the acceptance test is not passed, state is restored to what existed prior to the execution of that algorithm and execution of an alternate version on the same hardware is followed. Recovery is considered complete when acceptance test is passed. Checkpoint memory is needed to recover the state after a version fails, to provide a valid starting operational point for the next version (Fig.1).

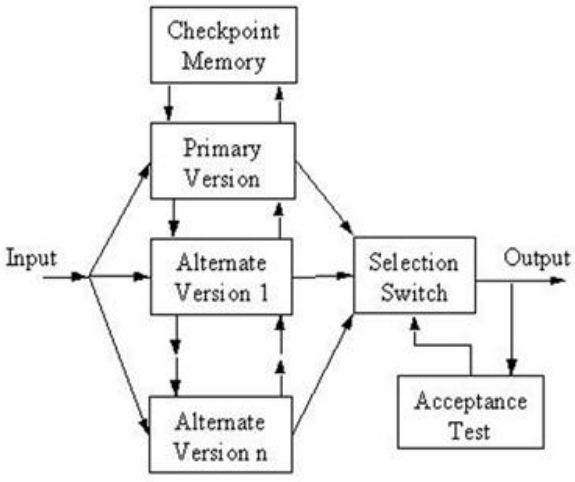

Fig. 1 Recovery Block Model

B. N-Version Programming

The NVP investigation project was started by A. Avizienis in 1975[2]. In this method, N-fold computation is carried out by using N independently designed software modules or "versions" and their results are sent to a decision algorithm that determines a single decision result [10]. In the NVP approach, a decision algorithm that delivers an agreement/disagreement decision is implemented. The N-Version programming is defined as the independent generation of N>=2 software modules, called "versions", from the same initial requirements [10]. "Independent generation" refers to the programming effort by individual or groups that do not interact with each other with respect to programming process. As the goal of NVP is to minimize the probability of similar errors at decision points, different algorithms, programming languages, environments and tools are used wherever possible.

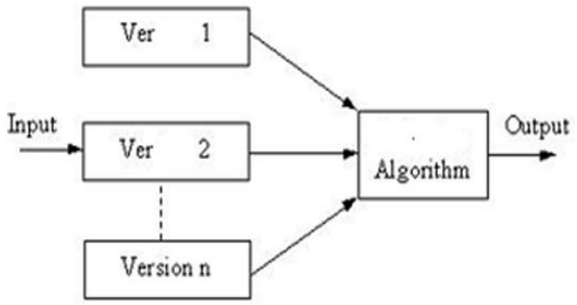

Fig. 2 N-Version Model

In NVP, since all the versions are built to satisfy the same requirements, Comparison of outputs and declaration of single result is carried out by output selection algorithm or voting algorithm (Fig.2). The output selection algorithms should be capable of detecting erroneous version outputs and prevent the propagation of bad values to main output. The output selection algorithm should be developed considering the application attributes like safety and reliability..

For applications where safety is a main concern, algorithm should be capable of detecting erroneous outputs and prevent the propagation of bad values to the main output. Also, the algorithm should be capable of declaring an error condition or initiate an acceptable safe output sequence, when it can not achieve a high confidence of selecting a correct output. For increased reliability, algorithm should be developed such that output is correct with a very high probability.

Some of the generalized selection algorithms are Formalized majority voter, generalized median voter, formalized plurality voter and weighted averaging techniques [11]. Other voting techniques that are being investigated are based on neural network and Genetic algorithm techniques [12]. They are implemented such that their performance is related to the application and the particular characteristic of the software versions.

As an example to demonstrate the NVP, consider a simple program that counts the number of digits in an input text. The program reads strings, calls the procedure count digit for each input string, adds up all the counts and prints the result. The module specifications [15] are as shown.

```
module main{
uses:count_digit(string) returning integer
Implementation: main.o
    }
module string_function_package{
NVP module
Interface:count_digit(string) returning integer
Implementation: "C_string_function.o@system1"
Implementation: "P_string_function.o@system2"
Implementation: "F_string_function.o@system3"
Voter: "vote.o"
Error_handler: "handler.o"
    }
application example {
import main
import string_function_package
bind main.count_digit string_function_package.count_digit
}
```

The specification defines a main module which calls the procedure count_digit. The main module does not have interface definition as this module is not used by any other module. The module string_function_package has an interface definition of count_digit used by module main. Specification for the module string_function_package also defines a voter and different versions of count_digit written in C, Pascal and Fortran. Object code for different versions of count_digit are C_string_function.o, P_string_function.o and F_string_function.o for C, Pascal and Fortran respectively. The module also specifies the target machine on which the version has to execute. Different language versions of function count_digit are as shown

```
int digit(s)
{
--C function--
}
function digit(s:str):integer;
begin
--Pascal function--
End
INTEGER digit(string)
--Fortran function--
END FUNCTION digit
```

III. ADVANTAGES OF NVP

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. Especially in VLSI circuits which is growing complex due to advancements in chip technologies, probability of design

fault is more since a complete verification of the design is very difficult to achieve. Use of N-versions of VLSI circuits allow the continued use of chips with design faults as long as their errors at decision points are not similar. Software verification and validation time is reduced by executing two independent versions in an operating environment thereby completing verification and validation with production operation concurrently.

Given a formal and an effective specification, different versions of software can be written by programmers working at their time and location using their own personal computing equipment. This "mail-order" approach [3] will drastically bring down the cost of programming that accrues in highly controlled professional programming environments.

IV. CHALLENGES OF NVP

The important condition for success of NVP is accurate specification of requirements. A series of experiments have been conducted and significant progress has occurred in the development of specification languages. Current goal of research is to compare and assess the ease of use of these methods by application programmers.

NVP is based on the conjecture that software designed differently will cause very few similar errors at decision points. Though some researchers have developed guidelines and methodologies to achieve design diversity, implementation has remained as a complex issue and evaluation is based on qualitative arguments. Large-scale experiments need to be carried out to statistically evaluate the usefulness of these methods. Cost of using NVP is another important issue. Generation of N versions of a given program instead of a single one increases the cost of software owing to escalated cost of development and supporting environment to complete the implementation. Peter Bishop [13] has argued that development and production cost can be reduced by applying design diversity only to critical paths. Effectiveness of this method however still needs to be quantitatively verified.

V. CONCLUSION

With amazing advancements with hardware technology, the focus of Fault Tolerance is shifting from hardware to software. Research on Software Fault Tolerance is gaining momentum. Hardware reliability theory can not be directly applied to Software, owing to the complexity of Software. N-Version programming approach of Software Fault Tolerance is based on design diversity conjecture. Independence of design and implementation effort with diverse programming languages, algorithms and environment will result in very low probability of similar errors at decision points, thereby increasing the Fault Tolerance capability of software.

REFERENCES

[1] Ken S. Lew, Tharam Dillon, and Kevin Forward "Software Complexity and Its Impact on software Reliability", *IEEE –Software Eng.*, Vol. 14, No. 11, pp. 1645-1655, Nov. 1988.

[2] "Fault Tolerance and fault intolerance. Complimentary approaches to reliable computing", *A. Avizienis, Proc. 1975 Int. Conf. Reliable Software, LosAngels, CA*, pp. 458-464, Apr. 21- 27, 1975.

[3] A. Avizienis, "N-Version Approach to fault tolerant Software", *IEEE Software e.g.*, Vol. SE.11, No.12, pp. 1491 -1501, Dec. 1985

[4] B. Randell, "System structure for Software Fault Tolerance", *IEEE Software Eng.*, Vol. SE.1, pp. 220-232, June 1975.

[5] "Information processing systems-Reliability and requirements", *Proc. East. Joint Comput. Conf., Washington, DC*, pp. 8-10, December 1953.

[6] J. Oblonsky, "A self correcting computer", *Digital Information processors, W. Hoffman, Ed. New York: Inter science*, pp. 533-542, 1962.

[7] J.F. Barlett, "A Non Stop operating system", *Proc. Hawaii Int. Conf. Syst. Sci, Honolulu, HI*, pp 103-119. *Reprinted in Theory and Practice of reliable System Design. Bedford, MA: Digital press*, pp. 453-460, January 5-6, 1978.

[8] Timothy C.K. Chou, "Beyond Fault Tolerance", *IEEE Computer*, pp. 47-49, April 1997.

[9] S. N. Wood field, "An experiment on unit increase in program complexity", *IEEE-Software Eng*., Vol-SE. 5, No. 2, pp. 76-79, 1979.

[10] A. Avizienis and L. Chen, "On the implementation of NVP for Fault Tolerance", *Proc. COMPSAC 77, 1st IEEE-CS Int. Compute. Software. Appl. Conf., Chicago, IL*, pp. 149-155, Nov. 8-11, 1977

[11] "A Theoretical Investigation of Generalized Voters for Redundant Systems", *Lorczak, Digest of Papers FTCS-19:The Nineteenth International Symposium on Fault-Tolerant Computing*, pp. 444-451, 1989.

[12] "Dependable, Intelligent Voting for Real-Time Control Software", *Engineering Applications of Artificial Intelligence*, Vol. 8, No. 6, pp. 615-623, Dec. 1995.

[13] Peter Bishop, "Software Fault Tolerance by Design Diversity", *Software Fault Tolerance*, John Wiley & Sons, 1995.

[14] "Software Fault Tolerance: A Tutorial", *Wilfredo Torres-Pomales, NASA Technical Memorandum*, Oct. 2000.

[15] James M. Purtilo and Pankaj Jalote, "An Environment for Developing Fault-Tolerant Software", *IEEE-Software Eng.*, Vol. 17, No. 2, Feb 1991.