Techniques of Software Fault Tolerance

Dr. K.C. Joshi
Computer Center, IASE,
M.J.P. Rohilkhand University, Bareilly
India
Email: kc-joshi@indiatimes.com

ABSTRACT
Fault tolerance is the ability of a system to perform its function correctly even in the presence of internal faults. We should accept that, relying on software techniques for obtaining dependability means accepting some overhead in terms of increased size of code and reduced performance (or slower execution). N-version programming achieves redundancy through the use of multiple versions. Failures are detected by comparing the results of the different versions. In this paper we will discuss the techniques of software fault tolerance such as recovery blocks, N-version programming, single version programming, multi-version programming, Comparison of N-Version with recovery block .

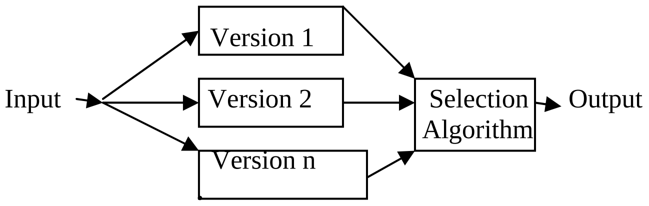
Key Words: Software Fault, Recovery blocks, multi-version programming.

I. Introduction:

A *system* is made up of *components*, which can be seen as 'systems' (i.e., further subdivided in components) in their own right (in which case they are often referred to as *subsystems*). A system is meant to provide services to other systems (which may include human users), and *fails* when its behavior departs from what was intended. When it is useful to identify a cause of failures, this is called a *fault*. The terms 'design fault', 'design defect', 'bug' are used as synonyms. Faults may lead to erroneous states, or *error*s, inside a system, which do not become system failures until they manifest themselves as deviations of the system's externally visible behavior from the intended service.

II. N-version Programming

N-Version Programming (NVP) has been defined as "*the independent generation of N* ≥ 2 *functionally equivalent programs from the same initial specification*" (Avizienis 77). "Independent generation" meant that the programming efforts were to be carried out by individuals or groups that did not interact with respect to the programming process. Wherever practical, different algorithms, programming languages, environments, and tools were to be used in each separate effort. In the NVP technique, the N versions of a program (a module) are developed independently by different programmers to be run concurrently. Their results are compared by an adjudicator. The simplest way is to use the majority voting here: the results produced by the majority of versions are assumed to be correct, the rest of the versions are assumed to have errors, their faults having been triggered in the execution due to one or more reasons. Essentially it is the N-Modular Redundancy (NMR) equivalent of the Hardware Fault Tolerant models.
The N-version programming concept is the same as the n-way redundant hardware i.e. N-Modular Redundancy (NMR). This was proposed by Avizienis . As NMR is designed to cope with the physical failures of components, the same component is replicated N times. In N-version programming, since the goal is to mask software design faults, the N-versions have different designs.



N-version programming Model.

III. Main issues in N-version programming:

There is some evidence that faults in software design are sometimes due not to the design itself but to the problem being solved: the hard parts of a problem often lead to design errors no matter which design approach is selected (Bishop 1986 ; Knight 1986). In other experiments, however, these phenomena did not occur (Avizienis 1986). Although NVP does not claim to achieve complete fault elimination, the existence of evidence showing that there may be some types of errors for which it is not effective makes it a matter of cost/benefit analysis whether or not to use it.

The NVP approach is also questionable from the traditional software engineering perspective. The NVP technique proposes that diverse designs be generated by designers who are isolated from each other. Although it is possible that isolated designers will by chance come up with different designs, isolation certainly does not guarantee that they will.

IV. Recovery Blocks:

The recovery blocks scheme provides a unifying frame work for that implementation of fault tolerance software incorporating strategies for error detection by the acceptance test and interface checks provided by the underlying system; backward error recovery provided automatically by a mechanism; and fault treatment which simply uses an alternate module as a temporary stand-by spare. The concept of recovery blocks was developed by Randell . Recovery blocks utilize temporal redundancy for fault-tolerance by partitioning a system into a number of self contained recoverable modules. Each recoverable block is responsible for validating correct operation within its boundaries. A module in software is written to implement some specification. This module is called primary module. It is assumed that the primary module may have some design faults . If the design fault in the module causes the failure of the module, it implies that the state of the module is erroneous. Error detection in a recovery block is done by an acceptance test . The alternate module is utilized to perform the requirement task, namely to implement the given specification when the primary module fails :

```
ensure     <acceptance test>
    by       <primary module>
else by     <alternate module1>
else by     <alternate module2>
            .
            .
            .
else by <alternate module n>
else error.
```

i.e. the alternate module i is executed only if the primary module and all the alternate modules before i have failed . A recovery block works by providing an acceptance test to validate the output of the primary processing routine. Failure of the test will result in the automatic invocation of an alternative routine. This alternative routine might not be as efficient or as complete as the primary routine but should be robust enough to satisfactory meet the conditions of recovery blocks. A recovery block is ultimately responsible for the correctness of its own output; therefore, if the acceptance test exhausts all of the alternative processing routines, a mechanism should be provided to signal that a failure condition has detected and contained.

V. Multi-version programming

Multi-version fault tolerance is based on the use of two or more versions (or “variants”) of a piece of software, executed either in sequence or in parallel. The versions are used as alternatives (with a separate means of error detection), in pairs (to implement detection by replication checks) or in larger groups (to enable masking through voting). The rationale for the use of multiple versions is the expectation that components built differently (i.e., different designers, different algorithms, different design tools, etc) should fail differently [Avizienis 77]. Therefore, if one version fails on a particular input, at least one of the alternate versions should be able to provide an appropriate output.

VI. Future prospects:

The N-version programming approaches employ design diversity in an effort to increase the reliability of software. If the failures of the different versions are co-related, then the design diversity approach will not offer as great an increase in reliability as is anticipated. The actual benefit to reliability will depend on how closely to failures of versions are co-related. N-version programming has the potential of increasing the reliability of some aspects of a program, provided that development and testing time and funding are increased to cover the extra costs. However, the probability of common-mode failures must be factored into any calculation of increased reliability, and one should show that the extra time and money couldn't be better spent improving a single version . The research is done in this area to improve the reliability approaches in a better way.

VII. Comparison of N-Version with recovery block Approaches:

In comparison to N-version programming the recovery block approach has one apparent advantage. In some situations, a software system evolves by replacement of some of its modules with newly developed ones. The replaced modules can be used as supplementary alternates to the new modules. The production cost is lower in this case. However, there are also certain disadvantages associated with the recovery block approach.
1. The system state before entry into a recovery block must be saved until sane reasonable results are obtained from the block. Considerable storage overhead may then be involved for nested recovery block structures.
2. Special precautions are needed to coordinate parallel processes within a nested recovery block structure. Otherwise the interdependencies among these processes may require that a long chain of process effects should be undone after a process has failed.
3. Some intermediate output from a recovery block may not be reversible in a real-time environment. Therefore, no recovery action can be performed before the incorrect output causes its damage.
4. Special system support is necessary to alleviate the above weaknesses. This limits the generality of applications of recovery block technique.

VIII. Limitations:

The most significant limitations may be:
(1) In a real-time environment a system failure may be caused by performance limitations rather than functional problems. Two typical examples are timing constraints violations and resource contentions. A likely source for these problems is system overload. N-version programming may produce adverse effects in these situations.
(2) In certain other circumstances, there exists no unique path to the solution of a problem. Step-by-step matching or voting of correspondent results cannot be used as a criterion of correctness. Therefore, N-version programming is not applicable for situations in which distinct multiple solutions (or intermediate solutions) exist.
(3) In still other cases a long sequence of outputs may not lend itself to be specified in a specific order. In these cases, the outputs from the component versions cannot be readily compared.
(4) In some situations the sequence of outputs from a version is context-dependent. Any error that pushes the rest of output off its proper position makes the subsequent comparison of results s meaningless.
(5) In the event that an allowable range of discrepancy cannot be easily determined, the problem of inexact voting is difficult to handle. Acceptable results are difficult to reach in this case.

IX. Conclusion:

The main objective of this paper has been to introduce some of the techniques of fault tolerance. In fulfilling this objective, we have introduced the techniques as N-version programming, recovery blocks, the main issues related to N-version programming of fault-tolerant systems, comparison of recovery blocks with N-version programming and limitations of N-version programming . We have also discussed multi-version programming.

References

[1] J.C. Laprie, et al, ‘Architectural Issues in Software Fault Tolerance’, in Software Fault Tolerance, Michael R. Lyu, editor, Wiley, 1995, pp. 47 – 80.

[2] Jie Xu and Brian Randell, ‘Software Fault Tolerance: t/(n-1)-Variant Programming’, IEEE Transactions on Reliability, Vol. 46, No. 1, March 1997, pp. 60 - 68.

[3] P. G. Bishop, ed, ‘Dependability of Critical Computer Systems 3: Techniques Directory’, Elsevier (1990). Guidelines produced by the European Workshop on Industrial Computer Systems, Technical Committee 7 (EWICS TC7).

[4] D. J. Pradhan, ed, ‘Fault Tolerant Computing: Theory and Practice’, Prentice-Hall (1986).

[5] A. Avizienis and L. Chen, ‘On the Implementation of N-Version Programming for Software Fault Tolerance During Execution’, Proceedings of the IEEE COMPSAC’77, November 1977, pp. 149 – 155.

[6] Algirdas Avizienis and Jean-Claude Laprie, ‘Dependable Computing: From Concepts to Design Diversity’, Proceedings of the IEEE, Vol. 74, No. 5, May 1986, pp. 629 – 638.

[7] K. H. Kim and Howard O. Welch, ‘Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications’, *IEEE Trans. Comp.* 38, 5 (May 1989), 626–636.

[8] John C. Knight and Nancy G. Leveson, ‘An Experimental Evaluation of the Assumption of Independence in Multiversion Programming’, *IEEE Trans. Soft. Eng.* **12**, 1 (January 1986), 96–109.

[9] Nancy G. Leveson and Peter R. Harvey, ‘Analyzing Software Safety’, *IEEE Trans. Soft. Eng.* 9, 5 (September 1983), 569–579.

[10] Bishop, P.G. et al. ‘PODS: A project on diverse software’, IEEE Trans. Soft. Eng. SE-12, 9(Sept.), 1986, pp.929-940.

[11] R. M. Smith, Kishor S. Trivedi and A. V. Ramesh, 'Performability Analysis: Measures, an Algorithm, and a Case Study', *IEEE Trans. Comp.* **37**, 4 (April 1988), 406–417.
[12] Charles H. Sauer and K. Mani Chandy, *Computer Systems Performance Modeling*, Prentice-Hall (1981).
[13] Robert M. Geist, Mark Smotherman, Kishor Trivedi, and Joanne B. Dugan, 'The Reliability of Life-Critical Computer Systems', *Acta Informatica* **23**, 6 (1986), 621–642 .
[14] Mark D. Hansen, 'Survey of Available Software-Safety Analysis Techniques', *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1989), 46–49.
[15] Jalote, P., Fault Tolerance in Distributed Systems, Prentice-Hall PTR, New Jersey, 1994.
[16] 16.Jorg Kienzle, 'Software Fault Tolerance Implementing N-Version Programming', McGill, COMP-667 - Implementing N-Version Programming, © 2009.
[17] K.C. Joshi, 'Reliability and Software Fault Tolerant Computing'**,** Proceeding of the NCSOFT-2008, 4-5 December, 2008.
[18] K.C. Joshi & Durgesh Pant, 'Software Fault Tolerant Computing: Needs and Prospects' , ACM Ubiquity, Vol 8 issues 16 , April 24-April 30 , 2007 .
[19] Andrea Bondavalli, 'Design of Fault Tolerant Software', CNUCE/CNR, via S.Maria 36, 56126 Pisa, Italy.
[20] Liming Chen, Algirdas Avizienis**, '**N-version programming : a fault-tolerance approach to reliability of software operation', Reprinted from FTCS-8 1978, pp. 3-9,@ IEEE 0-8186-7150-5/96 $5 @ 1996 ,IEEE Proceedings of FTCS-25, Volume III.