N-Version Programming

Robert Hanmer
Alcatel-Lucent
1960 Lucent Lane
Naperville, IL 60566 USA
+1 630 979 4786
hanmer@alcatel-lucent.com

With N-Version Programming, NVP, independent development teams use the same specification to generate multiple implementations. During development the design teams are kept separate and do not share their designs nor do they discuss the specification's meaning with each other. The design teams should use different algorithms and different programming languages to produce multiple versions that contain different faults from the other versions.

NVP employs redundancy at all levels from design/development to execution. NVP enables parallel execution of computational blocks. A technique discussed in Chapter Four, Recovery Blocks (4), provides for sequential computation of computational blocks. Serial execution of this parallelism is possible on a uniprocessor as long as there is separation between versions.

The fundamental principle of NVP is that there are multiple implementations done by different development teams. The same developer cannot make multiple implementations with multiple understandings of the same specification. The assumption is that when a latent fault activates in one version then the other versions of the system that do not contain the same fault. Research by Knight and Leveson [KL86][KL90] has shown that the assumption of independence between the faults produced by the different design teams does not hold. Research into NVP continues for example work by Cai, Lyu and Vouk [CLV05].

Because of the need for multiple design teams, NVP is not something that an individual or small team can decide on their own to undertake. For this reason, this book does not discuss NVP. [Hanmer07]

This PLOP paper will provide the pattern for N-Version programming that was not included in Patterns for Fault Tolerant Software. [Hanmer07] As a further note, the originators of N-Version Programming were Drs. Liming Chen and Algirdas Avizienis, who developed the technique in the late 1970's at UCLA. Their paper [CA78] provides one of the earliest explanations of the technique.

***

I'd like to acknowledge the suggestions of Anjali Das who shepherded this paper. Also Eduardo Fernandez and Michael Pont who offered suggestions.

1. N-Version Programming

… The system needs to be both reliable and available. It needs to be reliable and have as few faults as possible; Fault Prevention is of paramount importance. For example, a spacecraft carrying astronauts on a round trip journey to space needs to execute flawlessly to return those astronauts safely to the ground. It also needs to be highly available so that the astronauts have access to the systems continuously.

Resources are unlimited. Reliability is so important that the project has at least an implicit carte blanche to design and build the most reliable system possible with the highest quality possible.

A new system is being started. Requirements specifications are being created afresh for this system. Only the most basic parts of previous systems will be reused in the new system, the new system is not an evolutionary product. The system will include Redundancy (3)[*] to provide for continuous availability.

❖ ❖ ❖

Incorrect understandings of system specifications lead to faults. When architects and designers compare notes or direct each others work incorrect understandings can be either corrected or passed to others. How can faults due to incorrect understandings of specifications be eliminated?

A major component of fault tolerance is Redundancy (3), either spatial or temporal redundancy. Spatial redundancy is the redundancy of different components performing the same task in parallel. Temporal redundancy refers to redundancy in time. The same task is done serially, or at non-serial different times.

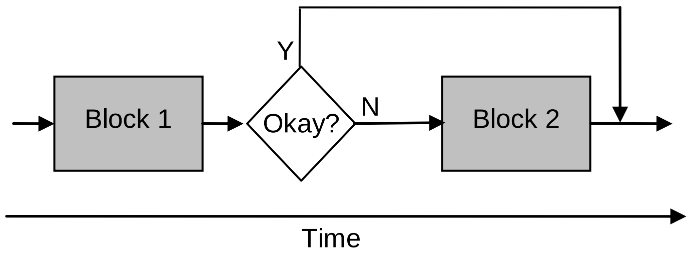
Redundancy can be designed into the system's hardware or software. Hardware redundancy involves providing multiple system components, such as multiple processors, network interfaces or other, specific peripheral devices. Redundant hardware generally implements spatial redundancy. Additional hardware, in the form of specialized circuits or arbiters, is required to combine the results from the redundant hardware into the single result that will be acted upon.

In the design of the current system, hardware redundancy will be used extensively. All kinds of hardware will be redundant, from sensors to processing elements. To make the system even more reliable, software redundancy is also desired.

Redundancy in software is typically done temporally. A common way of implementing this is through a Recovery Block (4) structure. In this structure a version of the code is executed and the results checked. The check determines if the result is correct, or within acceptable parameters. If the check passes the system continues operation. If the result does not pass the check then another bock of code is executed. The results of this block are then also checked. Two or more blocks can be used.

---

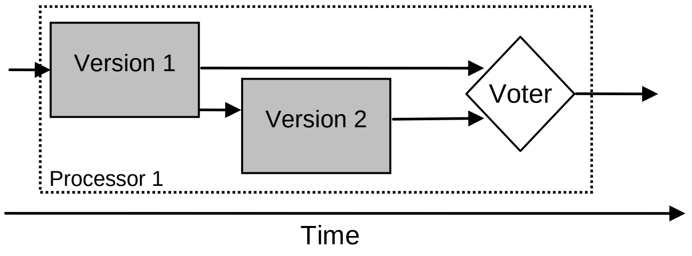
[*] Numbers in parenthesis after the names of patterns refer to the index number of the pattern within Patterns for Fault Tolerant Software [Hanmer2007].



```
ensure      Successful Execution
by          executing primary block
else by     executing secondary block #1
else by     executing secondary block #2
…
else by     executing secondary block #n
else trigger exception ().
```

Two representations of Recovery Blocks (4)

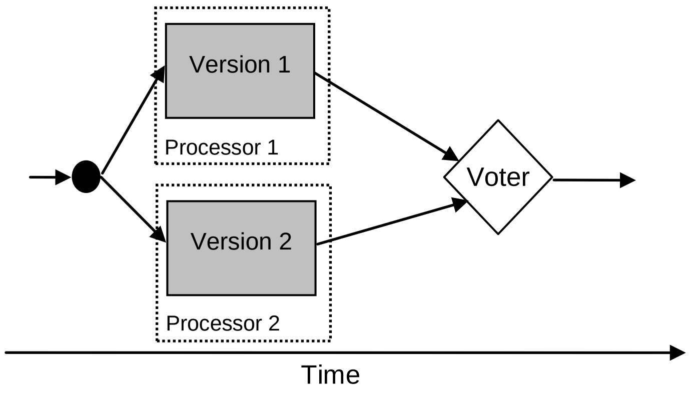
Another way of achieving software redundancy is to execute multiple versions of the same code serially. The results are compared through Voting (21) to select the result to be acted upon. The versions that are executed can be either the same version or different versions. The advantage of using the same version code version is to reduce costs. The disadvantage is that the same software when confronted with the same stimuli will produce the same results.



Multiple serial executions with voting

The serial execution of the versions takes time. All the instances execute to completion. The assumption is that they are all executed on the same processor.

Parallel, redundant processors could be executing the different versions simultaneously to achieve greater processing throughput, turning the temporal redundancy into spatial redundancy. As in the case of hardware spatial redundancy, an external voter or arbiter is required to select the result that should be acted upon. A variation would put the voter into one of the existing processors, as another software process. This will introduce single points of failure in case the voting processor goes unexpectedly out of service.



Multiple parallel executions with voting

Faults in the processor hardware will result in the versions behaving differently. But if the processors are error free, and the inputs are identical, then the same version of the software code will produce the same results. Those results might be either correct or incorrect. Fault prevention in the software development will have resulted in high quality software which is nearly error free. So these two versions should always compute identical and correct results. However, it is widely believed that no software is 100% fault free [CITE]. When a software defect is encountered both versions will make the same errors.

To reduce this risk of all the software containing the same faults, use different versions of the software. This concept of multiple versions was proposed as early as 1837:

> When the formula to be computed is complicated, it may be algebraically arranged for computation in two or more totally distinct ways, and two or more sets of cards [versions of software] may be made. If the same constants are now employed with each set, and if under these circumstances the results agree, we may then be quite secure of the accuracy of them all. [Babbage37]

At a its simplest, the same development team produces two (or more) versions of the software without copying anything from one version to another. But this can still result in the same fault being present in the multiple versions, since the fault might have come from a misunderstanding of the system's specification. Unless the misunderstanding is corrected then all the versions will have the same incorrect design and implementation.

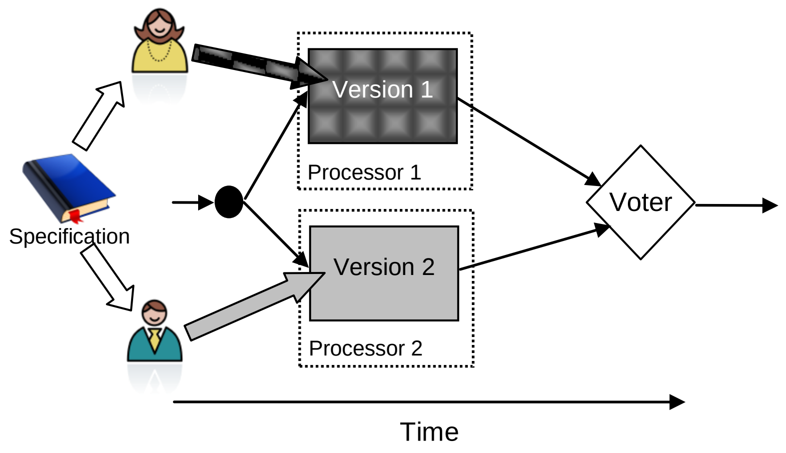
To further reduce the risk of the same fault being introduced into multiple versions, use different development teams to produce the different versions of the software. If they are allowed to collaborate and share their understanding of the specifications however then misunderstandings and errors can still result.

If the same implementation language is used for the multiple versions, certain faults that are in some way language-specific might be introduced. For example, in a language that relies on garbage collection failing to properly account for the frequency of garbage collection may result in performance requirements not being met. If multiple versions are created in this same language the versions might all have similar performance faults.

Therefore,

Build the system using "n" different development teams that interpret the specification and implement it without sharing their notes. Using different languages and different development techniques prevents the teams from introducing language-specific faults into the system.

Add a Voter element that will mediate and decide choose between the different versions at execution time.



N-Version Programming, N=2

❖ ❖ ❖

N-version programming was first invented by Chen and Avizienis in the late 1970's [CA78]. It has been much discussed in the literature; the references contain only a snapshot of the related publications. The pattern Active Replication by Titos Saridakis mentions N-version programming as a way of masking even Byzantine failures. [Saridakis02] The Reliable Hybrid Pattern by Fonda Daniels, et. al, describes a way of designing a system that can take advantage of multiple software redundancy techniques, such as variations of Recovery Blocks (4) and N-version programming. [DKV97]

Once the decision to employ N-version programming has been made there are a number of parameters of the system design process that must be determined. These include: how many versions of the software should be created?   How will the differences between the different versions be measured and quantified? How will the development teams be isolated from each other to reduce the probability of common misunderstandings? Will a separate team be created to monitor the N teams creating the versions? [Torres00]

Research ([KL86] [KL90]) has shown that the reliability of N-version systems might not be as high as the theory predicts for some applications. The experiment described in these papers had the same or similar software faults 50% of the time in more than one program. Even given this criticism the technique is still widely viewed as effective. …

[Babbage37] Babbage, C. "On the mathematical powers of the calculating engine," December 1837 (unpublished manuscript) Buxton MS7, Museum of the History off Science, Oxford. In B. Randell, editor. The Origins of Digital Computers: Selected Papers. Springer, New York, pages 17-52, 1974.

[CA78] Chen, L., and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," Digest of Papers FTCS-8: Eight Annual International Conference on Fault-Tolerant Computing, Toulouse, pp. 3-9 (June 1978).

[CLV05] 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.

[DKV97] 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. [http://hillside.net/plop/plop97/Workshops.html]

[Hanmer07] Hanmer, R. Patterns for Fault Tolerant Software. Chichester, UK: John Wiley & Sons, 2007.

[KL86] Knight, J. C. and N. G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multi-version Programming," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 1 (January 1986), pp 96-109.

[KL90] 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.

[Saridakis02] Saridakis, T. A System of Patterns for Fault Tolerance, Proceedings of EuroPLoP 2002, Kloster Irsee, Germany, July 2002, pp 535-582.

[Torres00] Torres-Pomales, w., Software Fault Tolerance: A Tutorial, Technical Report, Report No. NASA-2000-tm210616, 2000.