Contents

2

The Methodology of N-Version Programming

ALGIRDAS A. AVIŽIENIS
University of California, Los Angeles and Vytautas Magnus University, Kaunas, Lithuania

ABSTRACT

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. The *process* by which the NVS versions are produced is called *N-version programming* (NVP). 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. This chapter describes the methodology of N-version programming. First, the concepts, goals, and basic techniques of N-version programming are introduced and two major fault-tolerant software models, N-version software and recovery blocks are reviewed. Next, the process of building N-version software is discussed in detail, including the specification, programming and execution support of NVS units. Results of five consecutive experimental investigations are summarized, and a design paradigm for NVS is presented. A discussion of several novel system analysis and design issues that are specific to the use of NVS and an assessment of the unique advantages of fault-tolerant software conclude the chapter.

2.1 INTRODUCTION

The concept of N-version programming (NVP) was first introduced in 1977 [Avi77b] as follows:

> "*N-version programming* is defined as the independent generation of $N \geq 2$ functionally equivalent programs from the same initial specification. The N programs possess all the necessary attributes for concurrent execution, during which *comparison vectors* ("c-vectors") are

Software Fault Tolerance, Edited by Lyu

generated by the programs at certain points. The program state variables that are to be included in each c-vector and the *cross-check points* ("cc-points") at which the c-vectors are to be generated are specified along with the initial specification.

"Independent generation of programs" here means that the programming efforts are carried out by N individuals or groups that do not interact with respect to the programming process. Wherever possible, different algorithms and programming languages (or translators) are used in each effort. The *initial specification* is a formal specification in a specification language. The goal of the initial specification is to state the functional requirements completely and unambiguously, while leaving the widest possible choice of implementations to the N programming efforts. The actions to be taken at the cc-points after the exchange of c-vectors are also specified along with the initial specification. "

The following seventeen years have seen numerous investigations and some practical applications of the above stated concept, which identifies three elements of the NVP approach to software fault tolerance:

1. The *process* of initial specification and N-version programming which is intended to assure the independence and the functional equivalence of the N individual programming efforts;
2. The *product* (N-version software, or NVS) of the NVP process, which has the attributes for concurrent execution with specified cross-check points and comparison vectors for decisions;
3. The *environment* (N-version executive, or NVX) that supports the execution of the N-version software and provides decision algorithms at the specified cross-check points.

All three elements of the NVP approach are unique to the effort of independently and concurrently generating $N \geq 2$ functionally equivalent programs that are to be executed with decision-making by consensus. They are unnecessary in the case of a single program; thus the definition of NVP as an approach to software fault tolerance introduced new research topics designated as NVP, NVS, and NVX above.

This chapter presents the principles of the NVP approach to fault-tolerant software as it has evolved through a series of investigations in the 1977-1994 time period.

The justification for studying and applying NVP was stated in 1977 and has remained since then as follows [Avi77b]:

"The second 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*. Distinguishable software faults are faults that will cause a disagreement between c-vectors at the specified cc-points during the execution of the N-version set of programs that have been generated from the initial specification. Distinguishability is affected by the choice of c-vectors and cc-points, as well as by the nature of the faults themselves.

It is a fundamental conjecture of the N-version approach that the independence of programming efforts will greatly reduce the probability of identical software faults occurring in two or more versions of the program. In turn, the distinctness of faults and a reasonable choice of c-vectors and cc-points is expected to turn N-version programming into an effective method to achieve tolerance of software faults. The effectiveness of the entire N-version approach depends on the validity of this conjecture, therefore it is of critical importance that the initial specification should be free of any flaws that would bias the independent programmers toward introducing the same software faults."

It is essential to recognize that the independence of faults is an *objective* and *not* an *assumption* of the NVP approach, contrary to what was stated in [Sco84, Sco87, Kni86].

It is interesting that while the "NVP" concept did not appear in technical literature until 1977, searches through the earliest writings on the problem of errors in computing have led to two relevant observations dating back to the 1830's.

The first suggestion of multi-version computing was published in the *Edinburgh Review* of July 1834 by Dionysius Lardner, who wrote in his article "Babbage's calculating engine" as follows [Lar34]:

> "*The most certain and effectual check upon errors which arise in the process of computation, is to cause the same computations to be made by separate and independent computers; and this check is rendered still more decisive if they make their computations by different methods.*"

It must be noted that the word "computer" above refers to a person who performs the computation, and not to the calculating engine. Charles Babbage himself had written in 1837 in a manuscript that was only recently published [Bab37]:

> "*When the formula to be computed is very complicated, it may be algebraically arranged for computation in two or more totally distinct ways, and two or more sets of cards 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.*"

The evolution of the software of modern computers relied on the principle of finding and eliminating software design faults either before or during the operational use of the software. Suggestions on the use of multiple versions of software for fault tolerance began appearing in the early and mid-1970's [Elm72, Gir73, Kop74, Fis75].

The effort to develop a systematic process (a *paradigm*) for the building of multiple-version software units that tolerate software faults, and function analogously to majority-voted multichannel hardware units, such as TMR, was initiated at UCLA in early 1975 as a part of research in reliable computing that was started in 1961 [Avi87b]. The process was first called "redundant programming" [Avi75], and was renamed "N-version programming" (NVP) in the course of the next two years [Avi77b].

Another major direction of evolution of fault-tolerant software has been the recovery block (RB) approach, which evolved as a result of the long-term investigation of reliable computing systems that was initiated by Brian Randell at the University of Newcastle upon Tyne in 1970 [Shr85]. In the RB technique $M \geq 2$ *alternates* and an *acceptance test* are organized in a manner similar to the dynamic redundancy (standby sparing) technique in hardware [Ran75]. RB performs run-time software, as well as hardware, error detection by applying the acceptance test to the results delivered by the first alternate. If the acceptance test is not passed, recovery is implemented by state restoration, followed by the execution of the next alternate. Recovery is considered complete when the acceptance test is passed. A concise view of the evolution of the RB concept and its place in the general context of dependable computing is presented in [Ran87]. The properties of RB software are discussed in other chapters of this book.

2.2 FAULT-TOLERANT SOFTWARE: MODELS AND TECHNIQUES

We say that a unit of software (module, CSCI, etc.) is *fault-tolerant* (abbreviated "f-t")if it can continue delivering the required service, i.e., supply the expected outputs with the expected timeliness, after *dormant* (previously undiscovered, or not removed) imperfections, called *software faults*, have become active by producing *errors* in program flow, internal state, or results generated within the software unit. When the errors disrupt (alter, halt, or delay) the service expected from the software unit, we say that it has *failed* for the duration of service disruption. A non-fault-tolerant software unit will be called a *simplex* unit.

Multiple, redundant computing channels (or "lanes") have been widely used in sets of $N = 2$, 3, or 4 to build f-t hardware systems [Avi87a]. To make a simplex software unit fault-tolerant, the corresponding solution is to add one, two, or more simplex units to form a set of $N \geq 2$ units. The redundant units are intended to compensate for, or mask a failed software unit when they are not affected by software faults that cause similar errors at cross-check points. The critical difference between multiple-channel hardware systems and f-t software units is that the simple replication of one design that is effective against random physical faults in hardware is not sufficient for software fault tolerance. Copying software will also copy the dormant software faults; therefore each simplex unit in the f-t set of N units needs to be built separately and independently of the other members of the set. This is the concept of software *design diversity* [Avi82].

Design diversity is applicable to tolerate design faults in hardware as well [Avi77a, Avi82]. Some multichannel systems with diverse hardware and software have been built; they include the flight control computers for the Boeing 737-300 [Wil83], and the Airbus [Tra88] airliners. Variations of the diversity concept have been widely employed in technology and in human affairs. Examples in technology are: a mechanical linkage backing up an electrical system to operate aircraft control surfaces, an analog system standing by for a primary digital system that guides spacecraft launch vehicles, a satellite link backing up a fiber-optic cable, etc. In human activities we have the pilot-copilot-flight engineer teams in cockpits of airliners, two- or three-surgeon teams at difficult surgery, and similar arrangements.

A set of $N \geq 2$ diverse simplex units alone is not fault-tolerant; the simplex units need an *execution environment* (EE) for f-t operation. Each simplex unit also needs fault tolerance features that allows it to serve as a *member* of the f-t software unit with support of the EE. The simplex units and the EE have to meet three requirements: (1) the EE must provide the support functions to execute the $N \geq 2$ member units in a fault-tolerant manner; (2) the specifications of the individual member units must define the fault tolerance features that they need for f-t operation supported by the EE; (3) the best effort must be made to minimize the probability of an undetected or unrecoverable failure of the f-t software unit that would be due to a single cause.

The evolution of techniques for building f-t software out of simplex units has taken two directions. The two basic models of f-t software units are *N-version software* (NVS), shown in Figure 2.1 and *recovery blocks* (RB) shown in Figure 2.2. The common property of both models is that two or more diverse units (called *versions* in NVS, and *alternates* and *acceptance tests* in RB) are employed to form a f-t software unit. The most fundamental difference is the method by which the decision is made that determines the outputs to be produced by the f-t unit. The NVS approach employs a generic *decision algorithm* that is provided by the EE and looks for a *consensus* of two or more outputs among N member versions. The RB model applies the *acceptance test* to the output of an individual alternate; this acceptance test

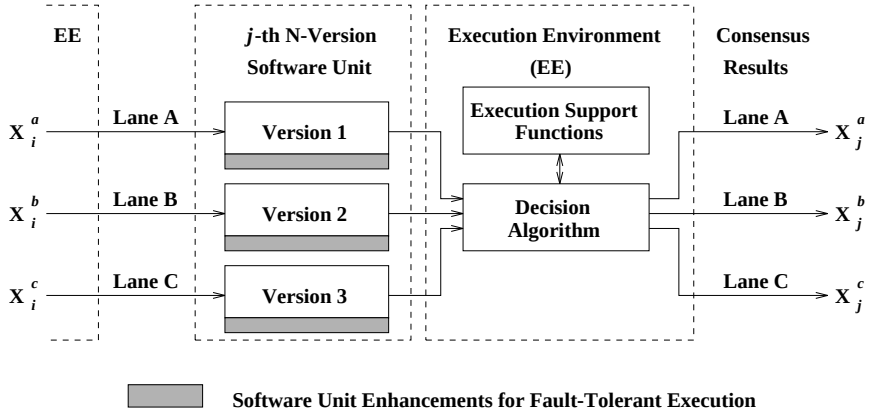

Figure 2.1 The N-version software (NVS) model with n = 3

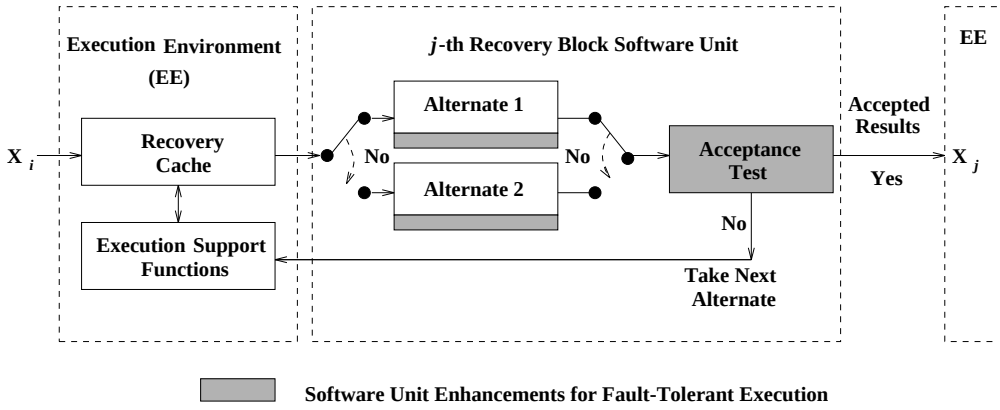

Figure 2.2 The recovery block (RB) model

must by necessity be *specific* for every distinct service, i.e., it is custom-designed for a given application, and is a member of the RB f-t software unit, but not a part of the EE.

$N = 2$ is the special case of *fail-safe* software units with two versions in NVS, and one alternate with one acceptance test in RB. They can detect disagreements between the versions, or between the alternate and the acceptance test, but cannot determine a consensus in NVS, or provide a backup alternate in RB. Either a *safe shutdown* is executed, or a *supplementary recovery process* must be invoked in case of a disagreement.

Both RB and NVS have evolved procedures for error recovery. In RB, backward recovery is achieved in a hierarchical manner through a *nesting* of RBs, supported by a *recursive cache* [Hor74], or *recovery cache* [And76] that is part of the EE. In NVS, forward recovery is done by the use of the *community error recovery* algorithm [Tso87] that is supported by

the specification of *recovery points* and by the decision algorithm of the EE. Both recovery methods have limitations: in RB, errors that are not detected by an acceptance test are passed along and do not trigger recovery; in NVS, recovery will fail if a majority of versions have the same erroneous state at the recovery point.

It is evident that the RB and NVS models converge if the acceptance test is done by NVS technique, i.e., when the acceptance test is specified to be one or more independent computations of the same outputs, followed by a choice of a consensus result. It must be noted that the individual versions of NVS usually contain error detection and exception handling (similar to an acceptance test), and that the NVP decision algorithm takes the known failures of member versions into account [Kel83, Avi84]. Reinforcement of the decision algorithm by means of a preceding acceptance test (*a filter*) has been addressed in [And86], and the use of an acceptance test when the decision algorithm cannot make a decision in [Sco84, Sco87]

The remaining parts of this chapter present the various aspects of specifying, generating, and executing N-version software by the NVP method.

2.3 BUILDING N-VERSION SOFTWARE

An NVS unit is a f-t software unit that depends on a generic *decision algorithm* (part of the EE) to determine a *consensus result* from the results delivered by two or more $(N \geq 2)$ *member versions* of the NVS unit. The *process* by which the NVS versions are produced is called *N-version programming*. The EE that embeds the N versions and supervises their f-t execution is called the *N-version executive*. The NVX may be implemented by means of software, hardware, or a combination of both. The major objective of the NVP process is to minimize the probability that two or more versions will produce similar erroneous results that coincide in time for a decision (consensus) action of NVX.

Building and using NVS requires three major efforts that are discussed below: (1) *to specify* the member versions of the NVS unit, including all features that are needed to embed them into the NVX; (2) *to define and execute* the NVP process in a manner that maximizes the independence of the programming efforts; (3) *to design and build* the NVX system for a very dependable and time-efficient execution of NVS units.

2.3.1 The Specification of Member Versions for NVS

The specification of the member versions, to be called "V-spec", represents the starting point of the NVP process. As such, the V-spec needs to state the functional requirements completely and unambiguously, while leaving the widest possible choice of implementations to the N programming efforts. It is the "hard core" of the NVS fault tolerance approach. Latent defects, such as inconsistencies, ambiguities, and omissions in the V-spec are likely to bias otherwise entirely independent programming or design efforts toward related design faults. The specifications for simplex software tend to contain guidance not only "what" needs to be done, but also "how" the solution ought to be approached. Such specific suggestions of "how" reduce the chances for diversity among the versions and should be systematically eliminated from the V-spec.

The V-spec may explicitly require the versions to differ in the "how" of implementation.

Diversity may be specified in the following elements of the NVP process: (1) training, experience, and location of implementing personnel; (2) application algorithms and data structures; (3) programming languages; (4) software development methods; (5) programming tools and environments; (6) 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 implementation. It is also possible to impose differing diversity requirements for separate software development stages, such as design, coding, testing, and even for the process of writing V-specs themselves, as discussed later.

Each V-spec must prescribe the *matching features* that are needed by the NVX to execute the member versions as an NVS unit in a fault-tolerant manner [Che78]. The V-spec defines: (1) the *functions* to be implemented, the time constraints, the inputs, and the initial state of a member version; (2) requirements for internal *error detection* and *exception handling* (if any) within the version; (3) the *diversity* requirements; (4) the *cross-check points* ("cc-points") at which the NVX decision algorithm will be applied to specified outputs of all versions; (5) the *recovery points* ("r-points") at which the NVX can execute *community error recovery* [Tso87] for a failed version; (6) the choice of the NVX *decision algorithm* and its *parameters* to be used at each cc-point and r-point; and (7) the *response* to each possible outcome of an NVX decision, including absence of consensus.

The NVX decision algorithm applies generic *consensus rules* to determine a consensus result from all valid version outputs. It has separate variants for real numbers, integers, text, etc. [Avi85b, Avi88b]. The *parameters* of this algorithm describe the allowable range of variation between numerical results, if such a range exists, as well as any other acceptable differences in the results from member versions, such as extra spaces in text output or other "cosmetic" variations.

The limiting case of required diversity is the use of two or more distinct V-specs, derived from the same set of user requirements. In early work, two cases have been practically explored: a set of three V-specs (formal algebraic OBJ, semi-formal PDL, and English) that were derived together [Kel83, Avi84], and a set of two V-specs that were derived by two independent efforts [Ram81]. These approaches provide additional means for the verification of the V-specs, and offer diverse starting points for version implementors.

In the long run, the most promising means for the writing of the V-specs are formal specification languages. When such specifications are executable, they can be automatically tested for latent defects [Kem85, Ber87], and they serve as prototypes of the versions that may be used to develop test cases and to estimate the potential for diversity. With this approach, verification is focused at the level of specification; the rest of the design and implementation process as well as its tools need not be perfect, but only as good as possible within existing resource and time constraints. The independent writing and testing by comparison of two specifications, using two formal languages, should increase the dependability of specifications beyond the present limits. Most of the dimensions of required diversity that were discussed above then can also be employed in V-spec writing. Among recently developed specification languages, promising candidates for V-specs that have been studied and used at UCLA are OBJ [Gog79] that has been further developed, the Larch family of specification languages [Gut85], PAISLey from AT&T Bell Laboratories [Zav86], and also Prolog as a specification language. A general assessment of specification languages for NVP is presented in [Avi90] and [Wu90].

2.3.2 The N-Version Programming Process: NVP

NVP has been defined from the beginning as "*the independent generation of $N \geq 2$ functionally equivalent programs from the same initial specification*" [Avi77b]. "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. 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*" [Avi77b]. The NVP process has been developed since 1975 in an effort that included five consecutive experimental investigations [Avi77b, Che78, Kel83, Kel86, Avi88a].

The application of a proven software development method, or of diverse methods for individual versions, remains the core of the NVP process. However, contemporary methods were not devised with the intent to reach the special goal of NVP, which is to minimize the probability that two or more member versions of an NVS unit will produce similar erroneous results that are coincident in time for an NVX decision at a cc-point or r-point.

NVP begins with the choice of a suitable software development process for an individual version. This process is supplemented by procedures that aim: (1) to attain the *maximum isolation* and *independence* (with respect to software faults) of the N concurrent version development efforts, and (2) to encourage the *greatest diversity* among the N versions of an NVS unit. Both procedures serve to minimize the chances of *related software faults* being introduced into two or more versions via potential *"fault leak" links*, such as casual conversations or E-mail exchanges, common flaws in training or in manuals, use of the same faulty compiler, etc.

Diversity requirements support this objective, since they provide more natural isolation against "fault leaks" between the teams of programmers. Furthermore, it is conjectured that the probability of a random, independent occurrence of faults that produce the same erroneous results in two or more versions is less when the versions are more diverse. A second conjecture is that even if related faults are introduced, the diversity of member versions may cause the erroneous results not to be similar at the NVX decision.

In addition to required diversity, two techniques have been developed to maximize the isolation and independence of version development efforts: (1) a set of mandatory rules of isolation, and (2) a rigorous communication and documentation protocol. The *rules of isolation* are intended to identify and eliminate all potential "fault leak" links between the *programming teams* (P-teams). The development of the rules is an ongoing process, and the rules are enhanced when a previously unknown "fault leak" is discovered and its cause is pinpointed. The *communication* and *documentation* (C&D) *protocol* imposes rigorous control on the manner in which all necessary information flow and documentation efforts are conducted. The main goal of the C&D protocol is to avoid opportunities for one P-team to influence another P-team in an uncontrollable and unnoticed manner. In addition, the C&D protocol documents communications in sufficient detail to allow a search for "fault leaks" if potentially related faults are discovered in two or more versions at some later time.

A coordinating team (C-team) is the keystone of the C&D protocol. The major functions of the C-team are: (1) to prepare the final texts of the V-specs and of the test data sets; (2) to set up the implementation of the C&D protocol; (3) to acquaint all P-teams with the NVP process, especially rules of isolation and the C&D protocol; (4) to distribute the V-specs, test

data sets, and all other information needed by the P-teams; (5) to collect all P-team inquiries regarding the V-specs, the test data, and all matters of procedure; (6) to evaluate the inquiries (with help from expert consultants) and to respond promptly either to the inquiring P-team only, or to all P-teams via a broadcast; (7) to conduct formal reviews, to provide feedback when needed, and to maintain synchronization between P-teams; (8) to gather and evaluate all required documentation, and to conduct acceptance tests for every version. All communications between the C-team and the P-teams must be in standard written format only, and are stored for possible post mortems about "fault leaks". Electronic mail has proven to be the most effective medium for this purpose. Direct communications between P-teams are not allowed at all.

2.3.3 Functions of the N-Version Executive NVX

The NVX is an implementation of the set of functions that are needed to support the execution of N member versions as a f-t NVS unit. The functions are *generic*; that is, they can execute any given set of versions generated from a V-spec, as long as the V-spec specifies the proper *matching features* for the NVX. The NVX may be implemented in software, in hardware, or in a combination of both. The principal criteria of choice are very high dependability and fast operation. These objectives favor the migration of NVX functions into VLSI-implemented fault-tolerant hardware: either complete chips, or standard modules for VLSI chip designs.

The basic functions that the NVX must provide for NVS execution are: 1) the decision algorithm, or set of algorithms; 2) assurance of input consistency for all versions; 3) inter-version communication; 4) version synchronization and enforcement of timing constraints; 5) local supervision for each version; 6) the global executive and decision function for version error recovery at r-points, or other treatment of faulty versions; and 7) a user interface for observation, debugging, injection of stimuli, and data collection during N-version execution of application programs. The nature of these functions is extensively illustrated in the description of the DEDIX (DEsign DIversity eXperiment) NVX and testbed system that was developed at UCLA to support NVP research [Avi85a, Avi88b].

2.4 EXPERIMENTAL INVESTIGATIONS

At the time when the NVP approach was first formulated, neither formal theories nor past experience was available about how f-t software units should be specified, built, evaluated and supervised. Experimental, "hands-on" investigations were needed in order to gain the necessary experience and methodological insights.

The NVP research approach at UCLA was to choose some practically sized problems, to assess the applicability of N-version programming, and to generate a set of versions. The versions were executed as NVS units, and the observations were applied to refine the process and to build up the concepts of NVP. The original definition of NVP and a discussion of the first two sets of results, using 27 and 16 independently written versions, were published in 1977 and 1978, respectively [Avi77b, Che78]. The subsequent investigation employed three distinct specifications: algebraic OBJ [Gog79], structured PDL, and English, and resulted in 18 versions of an "airport scheduler" program [Kel83]. This effort was followed by five versions of a program for the NASA/Four University study [Kel86], and then by an investigation

Table 2.1 N-version programming studies at UCLA

Years	Project and Sponsor	No. of Versions	P-Team Size	Required Diversity	Programming Language
1975-76	Text Editor Software engineering class	27	1	personnel	PL/1
1977-78	PDE Solution Software engineering class	16	2	personnel & 3 algorithms	PL/1
1979-83	Airport Scheduler NSF research grant	18	1	personnel & 3 specifications	PL/1
1984-86	Sensor Redundancy Management NASA research grant	5	2	personnel	Pascal
1986-88	Automatic Aircraft Landing Research grant from Sperry Flight Systems, Phoenix, AZ	6	2	personnel & 6 languages	Pascal, Ada, Modula-2, C, Prolog, T

in which six versions of an automatic landing program for an airliner were written, using six programming languages: Pascal, C, Ada, Modula-2, Prolog, and T [Avi88a]. Table 2.1 summarizes the five NVP studies at UCLA. In parallel with the last two efforts, a distributed NVX supervisor called DEDIX (DEsign DIversity eXperimenter) was designed and programmed [Avi85a, Avi88b].

The primary goals of the five consecutive UCLA investigations were: to develop and refine the NVP process and the NVX system (DEDIX), to assess the methods for NVS specification, to investigate the types and causes of software design faults, and to design successively more focused studies. Numerical predictions of reliability gain through the use of NVS were deemphasized, because the results of any one of the NVP exercises are uniquely representative of the quality of the NVP process, the specification, and the capabilities of the programmers at that time. The extrapolation of the results is premature when the NVP process is still being refined. The NVP paradigm that is described in the next section is now considered sufficiently complete for practical application and quantitative predictions. The paradigm has been further refined by an investigation at the University of Iowa, in which 40 programmers formed 15 teams to program the automatic aircraft landing program in the C language [Lyu93].

An important criterion for NVS application is whether sufficient *potential for diversity* is evident in the version specification. Very detailed or obviously simple specifications indicate that the function is poorly suited for f-t implementation, and might be more suitable for extensive single-version V&V, or proof of correctness. The extent of diversity that can be observed between completed versions may indicate the effectiveness of NVP. A qualitative assessment of diversity through a detailed structural study of six versions has been carried out for the Six-Language NVS investigation [Avi88a]. These results have led to further research into quantitative measures of software diversity [Che90, Lyu92b].

Three pioneering practical investigations of NVS have been performed with real-time software for nuclear reactor safety control [Ram81, Bis88, Vog88b]. Significant insights into specification, the NVP process, and the nature of software faults have resulted from these efforts.

Two other early studies investigating N-version programming were conducted in which numerical results were the principal objective and an "assumption of independence" was associated with NVP [Sco84, Sco87, Kni86]. We should stress that the definition of NVP [Avi77b]

has never postulated an "assumption of independence" and that NVP is a rigorous process of software development, which is examined in detail in the next section.

2.5 A DESIGN PARADIGM FOR N-VERSION SOFTWARE

The experience that had been accumulated during the preceding four investigations at UCLA [Avi77b, Che78, Kel83, Kel86] led to the rigorous definition and application of a set of guidelines, called the *NVS Design Paradigm* [Lyu88, Lyu92a] during the subsequent Six-Language NVP project [Avi88a]. The paradigm as it was further refined during this project, is summarized in *Figure 2.3* and described in this section. The purpose of the paradigm is to integrate the unique requirements of NVP with the conventional steps of software development methodology. The word "paradigm," used in the dictionary sense, means "pattern, example, model," presented here as a set of guidelines and rules with illustrations.

The objectives of the design paradigm are: (1) to reduce the possibility of oversights, mistakes, and inconsistencies in the process of software development and testing; (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 coincide in time for a decision (consensus) action of NVX.

The application of a proven software development method, or of diverse methods for individual versions, is the foundation of the NVP paradigm. The chosen method is supplemented by procedures that aim: (1) to attain suitable isolation and independence (with respect to software faults) of the N concurrent version development efforts, (2) to encourage potential diversity among the N versions of an NVS unit, and (3) to elaborate efficient error detection and recovery mechanisms. The first two procedures serve to reduce the chances of related software faults being introduced into two or more versions via potential "fault leak" links, such as casual conversations or mail exchanges, common flaws in training or in manuals, use of the same faulty compiler, etc. The last procedure serves to increase the possibilities of discovering manifested errors before they can cause an incorrect decision and consequent failure.

In Figure 2.3, the NVP paradigm is shown to be composed of two categories of activities. The first category, represented by boxes and single-line arrows at the left, contains standard software development procedures. The second category, represented by ovals and double-line arrows at the right, specifies the concurrent implementation of various fault tolerance techniques unique to N-version programming. The descriptions of the incorporated activities and guidelines are presented next.

2.5.1 System Requirement Phase : Determine Method of NVS Supervision

The NVS Execution Environment has to be determined in the system requirement phase in order to evaluate the overall system impact and to provide required support facilities. There are three aspects of this step:

(1) Choose NVS execution method and allocate resources. The overall system architecture is defined during system requirement phase, and the software configuration items are identified. The number of software versions and their interaction is determined.

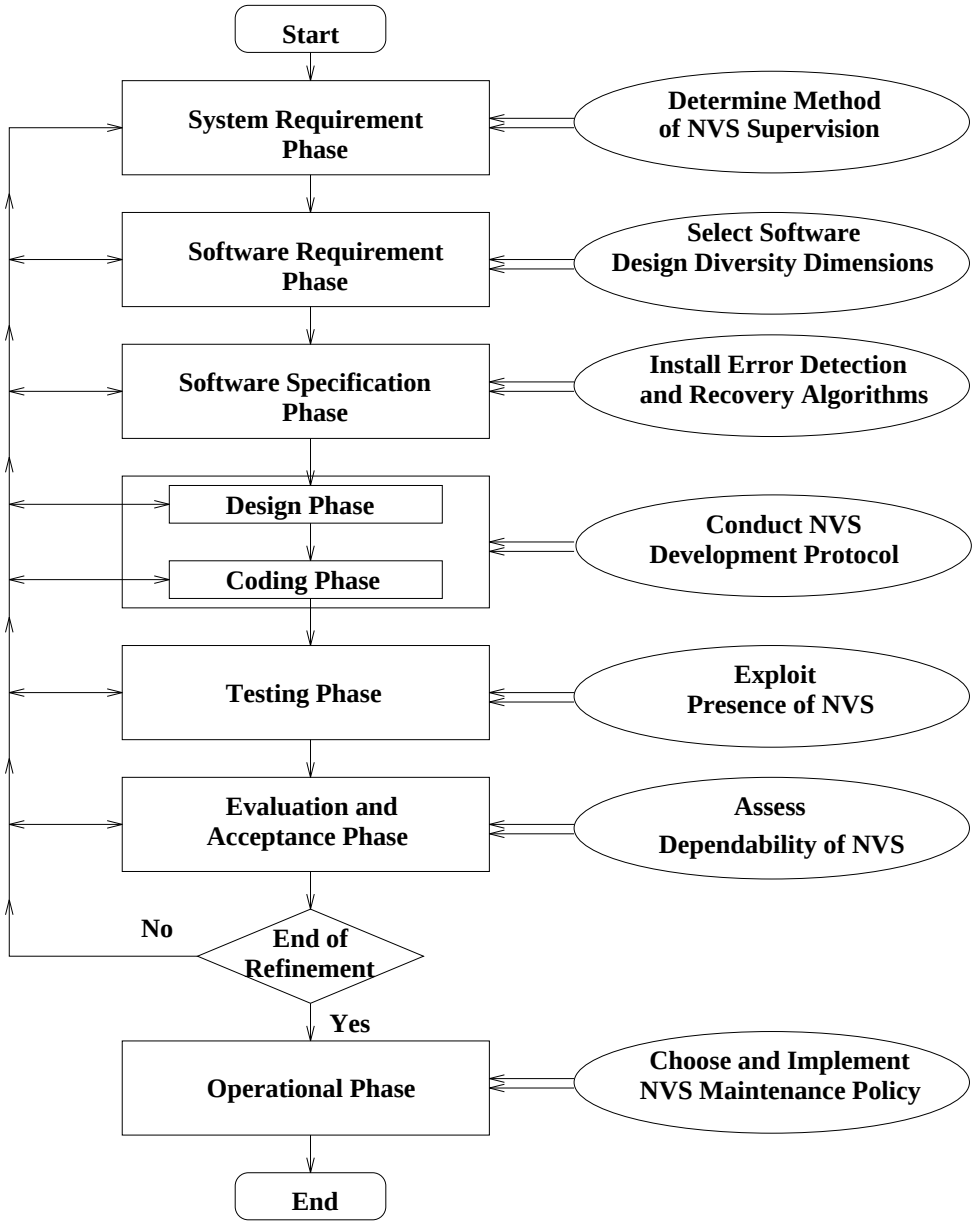

Figure 2.3 A design paradigm for N-version programming (NVP)

(2) Develop support mechanisms and tools. An existing NVX may be adapted, or a new one developed according to the application. The NVX may be implemented in software, in hardware, or in a combination of both. The basic functions that the NVX must provide for NVS execution are: (a) the decision algorithm, or set of algorithms; (b) assurance of input consistency for all versions; (c) interversion communications; (d) version synchronization and enforcement of timing constraints; (e) local supervision for each version; (f) the global executive and decision function for version error recovery; and (g) a user interface for observation, debugging, injection of stimuli, and data collection during N-version execution of applications programs. The nature of these functions was extensively illustrated in the descriptions of the DEDIX testbed system [Avi88b].

(3) Select hardware architecture. Special dedicated hardware processors may be needed for the execution of NVS systems, especially when the NVS supporting environments need to operate under stringent requirements (e.g., accurate supervision, efficient CPUs, etc.). The options of integrating NVX with hardware fault tolerance in a hybrid configuration also must be considered.

2.5.2 Software Requirement Phase : Select Software Diversity Dimensions

The major reason for specifying software diversity is to eliminate the commonalities between the separate programming efforts, as they have the potential to cause related faults among the multiple versions. Three steps of the selection process are identified.

(1) Assess random diversity vs. required diversity. Different dimensions of diversity could be achieved either by randomness or by requirement. The *random diversity*, such as that provided by independent personnel, causes dissimilarity because of an individual's training and thinking process. The diversity is achieved in an uncontrolled manner. The *required diversity*, on the other hand, considers different aspects of diversity, and requires them to be implemented into different program versions. 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 implementation.

(2) Evaluate required design diversity. There are four phases in which design diversity could be applied: specification, design, coding, and testing. Different implementors, different languages, different tools, different algorithms, and different software development methodologies, including phase-by-phase software engineering, prototyping, computer-aided software engineering, or even the "clean room" approach may be chosen for every phase. Since adding more diversity implies higher cost, it is necessary to evaluate cost-effectiveness of the added diversity along each dimension and phase.

(3) Specify diversity under application constraints. After the preceding assessments, the final combination of diversity can be determined under specific project constraints. Typical constraints are: cost, schedule, and required dependability. This decision presently involves substantial qualitative judgment, since quantitative measures for design diversity and its cost impact are not yet developed.

2.5.3 Software Specification Phase : Install Error Detection and Recovery Algorithms

The specification of the member versions, to be called "V-spec," needs to state the functional requirements completely and unambiguously, while leaving the widest possible choice of implementations to the N programming efforts. Sufficient error detection and recovery algorithms have to be selected and specified in order to detect errors that could potentially lead to system failures. Three aspects are considered below.

(1) Specify the matching features needed by NVX. Each V-spec must prescribe the *matching features* that are needed by the NVX to execute the member versions as an NVS unit in a fault-tolerant manner. The V-spec defines: (a) the *functions* to be implemented, the time constraints, the inputs, and the initial state of a member version; (b) requirements for internal *error detection* and *exception handling* (if any) within the version; (c) the *diversity* requirements; (d) the *cross-check points* ("cc-points") at which the NVX decision algorithm will be applied to specified outputs of all versions; (e) the *recovery points* ("r-points") at which the NVX can execute *community error recovery* for a failed version; (f) the choice of the NVX *decision algorithm* and its *parameters* to be used at each cc-point and r-point; (g) the *response* to each possible outcome of an NVX decision, including absence of consensus; and (h) the safeguards against the *Consistent Comparison problem* [Bri89].

(2) Avoid diversity-limiting factors. The specifications for simplex software tend to contain guidance not only "what" needs to be done, but also "how" the solution ought to be approached. Such specific suggestions of "how" reduce the chances for diversity among the versions and should be eliminated from the V-spec. Another potential diversity-limiting factor is the over-specification of cc-points and r-points. The installation of cc-points and r-points enhances error detection and recovery capability, but it imposes common constraints to the programs and may limit design diversity, The choice of the number of these points and their placement depend on the size of the software, the control flow of the application, the number of variables to be checked and recovered, and the time overhead allowed to perform these operations.

(3) Diversify the specification. The use of two or more distinct V-specs, derived from the same set of user requirements, can provide extensive protection against specification errors. Two examples are: a set of three V-specs (formal algebraic OBJ, semi-formal PDL, and English) that were derived together [Kel83, Avi84], and a set of two V-specs that were derived by two independent efforts [Ram81]. These approaches provide additional means for the verification of the V-specs, and offer diverse starting points for version implementors.

2.5.4 Design and Coding Phase : Conduct NVS Development Protocol

In this phase, multiple programming teams (P-teams) start to develop the NVS concurrently according to the V-spec. The main concern here is to maximize the isolation and independence of each version, and to smooth the overall software development. A coordinating team (C-team) is formed to supervise the effort. The steps are :

(1) Impose a set of mandatory rules of isolation. The purpose of imposing such rules on the P-teams is to assure the *independent generation* of programs, which means that programming efforts are carried out by individuals or groups that do not interact with respect

to the programming process. The rules of isolation are intended to identify and avoid potential "fault leak" links between the P-teams. The development of the rules in an ongoing process, and the rules are enhanced when a previously unknown "fault leak" is discovered and its cause pinpointed. Current isolation rules include: prohibition of any discussion of technical work between P-teams, widely separated working areas (offices, computer terminals, etc.) for each P-team, use of different host machines for software development, protection of all on-line computer files, and safe deposit of technical documents.

(2) Define a rigorous communication and documentation (C&D) protocol. The C&D protocol imposes rigorous control on all necessary information flow and documentation efforts. The goal of the C&D protocol is to avoid opportunities for one P-team to influence another P-team in an uncontrollable, and unnoticed manner. In addition, the C&D protocol documents communications in sufficient detail to allow a search for "fault leaks" if potentially related faults are discovered in two or more versions at some later time.

(3) Form a coordinating team (C-team). The C-team is the executor of the C&D protocol. The major functions of this team are: (a) to prepare the final texts of the V-specs and of the test data sets; (b) to set up the implementation of the C&D protocol; (c) to acquaint all P-teams with the NVP process, especially rules of isolation and the C&D protocol; (d) to distribute the V-specs, test data sets, and all other information needed by the P-teams; (e) to collect all P-team inquiries regarding the V-specs, the test data, and all matters of procedure; (f) to evaluate the inquiries (with help from expert consultants) and to respond promptly either to the inquiring P-team only, or to all P-teams via a broadcast; (g) to conduct formal reviews, to provide feedback when needed, and to maintain synchronization between P-teams; (h) to gather and evaluate all required documentation, and to conduct acceptance tests for every version.

2.5.5 Testing Phase : Exploit the Presence of NVS

A promising application of NVS is its use to reinforce current software verification and validation procedures during the testing phase, which is one of the hardest problems of any software development. The uses of multiple versions are:

(1) Support for verification procedures. During software verification, the NVS provides a thorough means for error detection since every discrepancy among versions needs to be resolved. Moreover, it is observed that consensus decision of the existing NVS may be more reliable than that of a "gold" model or version that is usually provided by an application expert.

(2) Opportunities for "back-to-back" testing. It is possible to execute two or three versions "back-to-back" in a testing environment. However, there is a risk that if the versions are brought together prematurely, the independence of programming efforts may be compromised and "fault leaks" might be created among the versions. If this scheme is applied in a project, it must be done by a testing team independent of the P-teams (e.g., the C-team), and the testing results should not be revealed to a P-team, if they contain information from other versions that would influence this P-team.

2.5.6 Evaluation and Acceptance Phase : Assess the Dependability of NVS

Evaluation of the software fault-tolerance attributes of an NVS system is performed by means of analytic modeling, simulation, experiments, or combinations of those techniques. The evaluation issues are:

(1) Define NVS acceptance criteria. The acceptance criteria of the NVS system depend on the validity of the conjecture that residual software faults in separate versions will cause very few, if any, similar errors at the same cc-points. These criteria depend on the applications and must be elaborated case by case.

(2) Assess evidence of diversity. Diversity requirements support the objective of independence, since they provide more natural isolation against "fault leaks" between the teams of programmers. Furthermore, it is conjectured that the probability of random, independent faults that produce the same erroneous results in two or more versions is less when the versions are more diverse. Another conjecture is that even if related faults are introduced, the diversity of member versions may cause the erroneous results not to be similar at the NVX decision. Therefore, evidence and effectiveness of diversity need to be identified and assessed [Che90, Lyu92b].

(3) Make NVS dependability predictions. For dependability prediction of NVS, there are two essential aspects: the choice of suitable software dependability models, and the definition of quantitative measures. Usually, the dependability prediction of the NVS system is compared to that of the single-version baseline system.

2.5.7 Operational Phase : Choose and Implement an NVS Maintenance Policy

The maintenance of NVS during its lifetime offers a special challenge. The two key issues are:

(1) Assure and monitor NVX functionality. The functionality of NVX should be properly assured and monitored during the operational phase. Critical parts of the NVS supervisory system NVX could themselves be protected by the NVP technique. Operational status of the NVX running NVS should be carefully monitored to assure its functionality. Any anomalies are recorded for further investigation.

(2) Follow the NVP paradigm for NVS modification. For the modification of the NVS unit, the same design paradigm is to be followed, i.e., a common specification of the modification should be implemented by independent maintenance teams. The cost of such a policy is higher, but it is hypothesized that the extra cost in maintenance phase, compared with that for single version, is relatively lower than the extra cost during the development phase. This is due to two reasons: (a) the achieved NVS reliability is higher than that of a single version, leaving fewer costly operational failures to be experienced; (b) when adding new features to the operating software, the existence of multiple program versions should make the testing and certification tasks easier and more cost-effective.

2.6 THE SYSTEM CONTEXT FOR FAULT-TOLERANT SOFTWARE

The introduction of fault-tolerant software raises some novel system analysis and design issues that are reviewed below.

2.6.1 Modeling of Fault-Tolerant Software

The benefits of fault tolerance need to be predicted by quantitative modeling of the *reliability, availability,* and *safety* of the system for specified time intervals and operating conditions. The conditions include acceptable *service levels, timing constraints* on service delivery, and *operating environments* that include expected *fault classes* and their *rates* of occurrence. The quality of fault tolerance mechanisms is expressed in terms of *coverage* parameters for error detection, fault location, and system recovery. A different fault tolerance specification is the *minimal tolerance* requirement to tolerate one, two, or more faults from a given set, regardless where in the system they occur. The one-fault requirement in frequently stated as "no single point of failure" for given operating conditions. An analysis of the design is needed to show that this requirement is met.

The similarity of NVS and RB as f-t software units has allowed the construction of a model for the prediction of reliability and average execution time of both RB and NVS. The model employs queuing theory and a state diagram description of the possible outcomes of NVS and RB unit execution. An important question explored in this model is how the gain in reliability due to the fault tolerance mechanisms is affected when *related faults* appear in two or more versions of NVS, and when the acceptance test has less than perfect coverage (due to either incompleteness, or own faults) with respect to the faults in RB alternates. The *correlation factor* is the conditional probability of a majority of versions (in NVS) or one alternate and the acceptance test (in RB) failing in such a way that a faulty result is passed as an output of the f-t software unit. The model shows strong variation of the reliability of f-t software units as a function of the correlation factor [Grn80, Grn82].

The criticality of related faults had been recognized quite early for both RB [Hec76] and NVS [Grn80], and later in [Sco84, Eck85] which reached similar conclusions. Later studies have further explored the modeling of f-t software, including the impact of related faults on the reliability and safety of both NVS and RB [Lap84, Tso86, Lit87, Arl88]. A recent study has assessed the performability of f-t software [Tai93a, Tai93b]. Several extensive studies of the modeling and analysis of fault tolerant software are being presented in other chapters of this book and will not be reviewed here.

2.6.2 Hosting and Protection of the Execution Environment EE

The *host system* for NVS (and RB as well) interacts with the f-t software units through the EE, which communicates with the fault tolerance functions of its operating system or with fault tolerance management hardware, such as a service processor. The EE itself may be integrated with the operating system, or it may be in part, or even fully implemented in hardware. A fully developed EE for NVS is the all-software DEDIX supervisor [Avi88b], which interacts with Unix on a local network. Such a software-to-software linkage between the EE and the operating system accommodates any hardware operating under Unix, but causes delays in inter-version communication through the network. In practical NVS implementations with real-time constraints either implementing the DEDIX functions in custom hardware, or building an operating system that provides EE services along with its other functions is necessary.

The recent Newcastle RB investigation employed both the hardware recovery cache and extensions to the MASCOT operating system as the implementation of the EE [And88], while the distributed RB study employs hardware and a custom distributed operating system [Chu87]. Other examples of solutions are found in [Mak84, Lal88, Wal88].

The remaining question is the protection against design faults that may exist in the EE itself. For NVS this may be accomplished by N-fold diverse implementation of the NVX. To explore the feasibility of this approach, the prototype DEDIX environment has undergone formal specification in PAISLey [Zav86]. This specification is suitable to generate multiple diverse versions of the DEDIX software to reside on separate physical nodes of the system. It is evident that diversity in separate nodes of the NVX will cause a slowdown to the speed of the slowest version. Since the NVX provides generic, reusable support functions of limited complexity, it may be more practical to verify a single-version NVX and to move most of its functionality into custom processor hardware.

In the case of RBs, special fault tolerance attention is needed by the recovery cache and any other custom hardware. The tolerance of design faults in the EE has been addressed through the concept of multilevel structuring [Ran75]. The acceptance test, which is unique for every RB software unit, also may contain design faults. The obvious solution of 2-version or 3-version acceptance tests is costly, and verification or proof of each acceptance test appear to be the most practical solutions.

2.6.3 Multilayer Diversity

Multilayer diversity occurs when diversity is introduced at several *layers* of an N-channel computing system: application software, system software, hardware, system interfaces (e.g., diverse displays), and even specifications [Avi86]. The justification for introducing diversity in hardware, system software, and user interfaces is that tolerance should extend to design faults that may exist in those layers as well. The second argument, especially applicable to hardware, is that diversity among the channels of the hardware layer will naturally lead to greater diversity among the versions of system software and application software.

The use of diverse component technologies and diverse architectures adds more practical dimensions of hardware diversity. The diversity in component technologies is especially valuable against faults in manufacturing processes that lead to deterioration of hardware and subsequent delayed manifestation of related physical faults that could prematurely exhaust the spare supply of long-life f-t systems. The counter-argument that such diversity in hardware is superfluous may be based on the assumption that diversity in software will cause the identical host hardware channels to assume diverse states. The same hardware design fault then would not be likely to produce similar and time-coincident errors in system and application software.

Tolerance of design faults in human-machine interfaces offers an exceptional challenge. When fault avoidance is not deemed sufficient, dual or triplex diverse interfaces need to be designed and implemented independently. For example, dual or triple displays of diverse design and component technology will provide an additional safety margin against design and manufacturing faults for human operators in air traffic control, airliner cockpits, nuclear power plant control rooms, hospital intensive care facilities, etc. Redundant displays often are already employed in these and other similar applications due to the need to tolerate single physical faults in display hardware without service interruption.

The major limitations of layered diversity are the *cost* of implementing multiple independent designs and the *slowdown* of operation that is caused by the need to wait for the slowest version at every system layer at which diversity is employed. The latter is especially critical for real-time applications in which design fault tolerance is an essential safety attribute. Speed considerations strongly favor the migration of f-t EE functions into diverse VLSI circuit implementations. A few two and three version systems that employ diverse hardware

and software have been designed and built. They include the flight control computers for the Boeing 737-300 [Wil83], the ATR.42, Airbus A-310, and A-320 aircraft [Tra88]. Proposed designs for the flight control computer of the planned Boeing 7J7 are the three-version GEC Avionics design [Hil88] and the four-version MAFT system [Wal88].

A different concept of *multilevel systems* with fault-tolerant interfaces was formulated by Randell for the RB approach [Ran75]. Diversity is not explicitly considered for the hardware level, but appears practical when additional hardware channels are employed, either for the acceptance test, or for parallel execution of an alternate in distributed RB [Chu87, Kim88].

2.6.4 NVS Support for System Security

Computer security and software fault tolerance have the common goal to provide reliable software for computer systems [Tur86, Dob86]. A special concern is *malicious logic*, which is defined as: "Hardware, software, or firmware that is intentionally included in a system for the purpose of causing loss or harm" [DOD85]. The loss or harm here is experienced by the user, since either incorrect service, or no service at all is delivered. Examples of malicious logic are *Trojan horses*, *trap doors*, and *computer viruses*. The deliberate nature of these threats leads us to classify malicious logic as *deliberate design faults* (DDFs), and to apply fault tolerance techniques to DDF detection and tolerance, such as in the case of computer virus containment by program flow monitors [Jos88a].

Three properties of NVS make it effective for tolerating DDFs: (1) the independent design, implementation, and maintenance of multiple versions makes a single DDF detectable, while the covert insertion of identical copies of DDFs into a majority of the N versions is difficult; (2) NVS enforces completeness, since several versions ensure (through consensus decision) that all specified actions are performed (i.e., omitting a required function can be a DDF); and (3) time-out mechanisms at all decision points prevent prolonged period without action (i.e., slowing down a computer system is a denial-of-service DDF). A detailed study of these issues appears in [Jos88b].

2.6.5 Modification of Operational NVS

Modification of already operational f-t software occurs for two different reasons: (1) one of the member units (version, alternate, or acceptance test) needs either the removal of a newly discovered fault, or an improvement of a poorly programmed function, while the specification remains intact; (2) all member units of a f-t software unit need to be modified to add functionality or to improve its overall performance.

In the first case, the change affects only one member and should follow the standard fault removal procedure. The testing of the modified unit should be facilitated by the existence of other members of the f-t software. Special attention is needed when a *related fault* is discovered in two or more versions or alternates, or in one alternate and the acceptance test. Here independence remains important, and the NVP process needs to be followed, using a *removal specification*, followed by isolated fault removals by separate maintenance teams.

In the second case, N independent modifications need to be done. First, the specification is modified, re-verified, and tested to assess the impact of the modification. Second, the affected f-t software units are regenerated from the specification, following the standard NVP or RB processes. The same considerations apply to modification of the alternates in RB software, but special treatment is required for modifying the unique acceptance test software unit.

2.7 CONCLUSIONS

Although at first considered to be an impractical competitor of high-quality single-version programs, fault-tolerant software has gained significant acceptance in academia and industry during the past decade. Two, three, and four version software is switching trains [Hag88], performing flight control computations on modern airliners [Wil83, Tra88], and more NVS applications are on the way [Vog88a, Wal88, Hil88]. Publications about f-t software are growing in numbers and in depth of understanding, and at least three long-term academic "hands-on" efforts are in their second decade: recovery blocks at Newcastle [Ran87, And88], distributed recovery blocks at UC Irvine [Chu87, Kim88], and N-version software at UCLA [Avi85b, Avi88a, Avi88b].

Why should we pursue these goals? Every day, humans depend on computers more and more to improve many aspects of their lives. Invariably, we find that those applications of computers that can deliver the greatest improvements in the quality of life or the highest economic benefits also can cause the greatest harm when the computer fails. Applications that offer great benefits at the risk of costly failures are: life support systems in the delivery of health care and in adverse environments; control systems for air traffic and for nuclear power plants; flight control systems for aircraft and manned spacecraft; surveillance and early warning systems for military defense; process control systems for automated factories, and so on.

The loss of service for only a few seconds or, in the worst case, service that looks reasonable but is wrong, is likely to cause injuries, loss of life, or grave economic losses in each one of these applications. As long as the computer is not sufficiently trustworthy, full benefits of the application cannot be realized, since human surveillance and decision making are superimposed, and the computers serve only in a supporting role. At this time it is abundantly clear that the trustworthiness of software is the principal prerequisite for the building of a trustworthy system. While hardware dependability also cannot be taken for granted, tolerance of physical faults is proving to be very effective in contemporary fault-tolerant systems.

At present, fault-tolerant software is the only alternative that can be expected to provide a higher level of trustworthiness and security for critical software units than test or proof techniques without fault tolerance. The ability to guarantee that any software fault, as long as it only affects only one member of an N-version unit, is going to be tolerated without service disruption may by itself be a sufficient reason to adapt fault-tolerant software as a safety assurance technique for life-critical applications.

Another attraction of fault-tolerant software is the possibility of an economic advantage over single-version software in attaining the same level of trustworthiness. The higher initial cost may be balanced by significant gains, such as faster release of trustworthy software, less investment and criticality in verification and validation, and more competition in procurement as versions can be acquired from small, but effective, enterprises in widely scattered locations.

Finally, there is a fundamental shift of emphasis in software development that takes place when N-version software is produced. In single-version software, attention is usually focused on testing and verification, i.e., the programmer-verifier relationship. In NVS, the key to success is the version specification; thus the focus shifts to the user-specifier relationship and the quality of specifications. The benefits of this shift are evident: a dime spent on specification is a dollar saved on verification.

ACKNOWLEDGMENTS

The concept and methodology of N-version programming have evolved through the author's collaboration and discussions with the members and visitors of the Dependable Computing and Fault-Tolerant Systems Laboratory at UCLA and through the efforts of the ninety-nine programmers who took part in our studies since 1975. Liming Chen had the courage to initiate the experimental investigations, John P.J. Kelly brought them to maturity, and Michael R. Lyu formulated the NVS design paradigm as it exists today. Special thanks also belong to Jean Arlat, Jia-Hong (Johnny) Chen, Per Gunninberg, Mark Joseph, Jean-Claude Laprie, Srinivas V. Makam, Werner Schuetz, Lorenzo Strigini, Pascal Traverse, Ann Tai, Kam-Sing Tso, Udo Voges, John F. Williams, and Chi-Sharn Wu for their valuable contributions. Rimas Avižienis prepared this text with care and dedication, and Yutao He put the chapter into its final form.

Financial support for fault-tolerant software research at UCLA has been provided by the following sources: National Science Foundation, Grants No. MCS-72-03633, MCS-78-18918, and MCS-81-21696; Office of Naval Research, Contract No. N0014-79-C-0866; National Aeronautics and Space Administration, Contract No. NAG1-512; Battelle Memorial Institute; Federal Aviation Administration-Advanced Computer Science Program; Sperry Commercial Flight Systems Division of Honeywell, Inc.; State of California "MICRO" Program; and CALSPAN-UB Research Center.

REFERENCES

[And76] T. Anderson and R. Kerr. Recovery blocks in action: a system supporting high reliability. In *Proc. 2nd International Conference on Software Engineering*, pages 447–457, San Francisco, CA, October 1976.

[And86] T. Anderson. A structured decision mechanism for diverse software. In *Proc. 5th Symposium on Reliability in Distributed Software and Database Systems*, pages 125–129, Los Angeles, CA, January 1986.

[And88] T. Anderson, P. A. Barrett, D. N. Halliwell, and M. R. Moulding. Tolerating software design faults in a command and control system, pages 109–128. In [Vog88a].

[Arl88] J. Arlat, K. Kanoun, and J. C. Laprie. Dependability evaluation of software fault-tolerance. In *Digest of 18th FTCS*, pages 142–147, Tokyo, Japan, June 1988.

[Avi75] A. Avižienis. Fault tolerance and fault intolerance: complementary approaches to reliable computing. In *Proc. 1975 International Conference on Reliable Software*, pages 458–464, April 1975.

[Avi77a] A. Avižienis. Fault-tolerant computing — progress, problems, and prospects. In *Information Processing 77, Proc. of IFIP Congress*, pages 405–420, Toronto, Canada, August 1977.

[Avi77b] A. Avižienis 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.

[Avi82] A. Avižienis. Design diversity — the challenge for the eighties. In *Digest of 12th FTCS*, pages 44–45, Santa Monica, CA, June 1982.

[Avi84] A. Avižienis and J. Kelly. Fault tolerance by design diversity: concepts and experiments. In *IEEE Computer*, 17(8):67–80, August 1984.

[Avi85a] A. Avižienis, P. Gunningberg, J. P. J. Kelly, L. Strigini, P. J. Traverse, K. S. Tso and U. Voges. The UCLA DEDIX system: a distributed testbed for multiple-version software. In *Digest of 15th FTCS*, pages 126–134, Ann Arbor, MI, June 1985.

[Avi85b] A. Avižienis. The N-version approach to fault-tolerant software. In *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, December 1985.

[Avi86] A. Avižienis and J. C. Laprie. Dependable computing: from concepts to design diversity. In *Proc. IEEE*, 74(5):629–638, May 1986.

[Avi87a] A. Avižienis, H. Kopetz, and J. C. Laprie, editors. *The Evolution of Fault-Tolerant Computing*. Springer, Wien, New York, 1987.

[Avi87b] A. Avižienis and D. Rennels. The evolution of fault tolerant computing at the Jet Propulsion Laboratory and at UCLA: 1955-1986, pages 141–191. In [Avi87a].

[Avi88a] A. Avižienis, M. R. Lyu, and W. Schuetz. In search of effective diversity: a six-language study of fault-tolerant flight control software. In *Digest of 18th FTCS*, pages 15–22, Tokyo, Japan, June 1988.

[Avi88b] A. Avižienis, M. R-T. Lyu, W. Schuetz, K-S. Tso, and U. Voges. DEDIX 87 — A supervisory system for design diversity experiments at UCLA, pages 129–168. In [Vog88a].

[Avi90] A. Avižienis and C-S. Wu. A comparative assessment of system description methodologies and formal specification languages. *Technical Report CSD-900030, Computer Science Department, UCLA*, September 1990.

[Bab37] C. Babbage. On the mathematical powers of the calculating engine, December 1837 (Unpublished Manuscript) Buxton MS7, Museum of the History of Science, Oxford. In B. Randell, editor. *The Origins of Digital Computers: Selected Papers*. Springer, New York, pages 17-52, 1974.

[Ber87] E. F. Berliner and P. Zave. An experiment in technology transfer: PAISLey specification of requirements for an undersea lightwave cable system. In *Proc. 9th Int. Conference on Software Engineering*, pages 42–50, Monterey, CA, April 1987.

[Bis88] P. G. Bishop. The PODS diversity experiment, pages 51–84. In [Vog88a].

[Bri89] S. S. Brilliant, J. C. Knight, and N. G. Leveson. The consistent comparison problem in N-version software. In *IEEE Transactions on Software Engineering*, SE-15(11):1481–1485, November 1989.

[Che78] L. Chen and A. Avižienis. N-version programming: a fault-tolerance approach to reliability of software operation. In *Digest of 8th FTCS*, pages 3–9, Toulouse, France, June 1978.

[Che90] J. J. Chen. *Software Diversity and Its Implications in the N-Version Software Life Cycle*. PhD dissertation, UCLA, Computer Science Department, 1990.

[Chu87] W. W. Chu, K. H. Kim, and W. C. McDonald. Testbed-based validation of design techniques for reliable distributed real-time systems. In *Proc. IEEE*, 75(5): 649–667, May 1987.

[DOD85] U. S. Department of Defense. *Trusted Computer System Evaluation Criteria*. DoD Doc. 5200.28-STD. December 1985.

[Dob86] J. E. Dobson and B. Randell. Building reliable secure computing systems out of unreliable insecure components. In *IEEE Symposium Security and Privacy*, pages 187–193, Oakland, CA, April 1986.

[Eck85] D. E. Eckhardt and L. D. Lee. A theoretical basis for the analysis of multiversion software subject to coincident errors. In *IEEE Transactions on Software Engineering*, SE-11(12):1511–1517, December 1985.

[Elm72] W. R. Elmendorf. Fault-tolerant programming. In *Digest of 2nd FTCS*, pages 79–83, Newton, MA, June 1972.

[Fis75] M. A. Fischler, O. Firschein, and D. L. Drew. Distinct software: an approach to reliable computing. In *Proc. 2nd USA-Japan Computer Conference*, pages 573–579, Tokyo, Japan, August 1975.

[Gir73] E. Girard and J. C. Rault. A programming technique for software reliability. In *Proc. 1973 IEEE Symposium on Computer Software Reliability*, pages 44–50, New York, May 1973.

[Gog79] J. A. Goguen and J.J. Tardo. An introduction to OBJ: a language for writing and testing formal algebraic program specifications. In *Proc. Specification of Reliable Software Conference*, pages 170–189, Cambridge, MA, April 3-5 1979.

[Grn80] A. Grnarov, J. Arlat, and A. Avižienis. On the performance of software fault tolerance strategies. In *Digest of 10th FTCS*, pages 251–253, Kyoto, Japan, October 1980.

[Grn82] A. Grnarov, J. Arlat, and A. Avižienis. Modeling and performance evaluation of software fault-tolerance strategies. *Technical Report CSD-820608, Computer Science Department, UCLA*, June 1982.

[Gut85] J. V. Guttag, J. J. Horning, and J. M. Wing. *Larch in Five Easy Pieces*, pages 11–21. Report No. 5. DEC Systems Research Center, Palo Alto, CA, July 24 1985.

[Hag88] G. Hagelin. ERICSSON safety system for railway control, pages 11–21. In [Vog88a].

[Hec76] H. Hecht. Fault-tolerant software for real-time applications. In *ACM Computing Surveys*, 8(4):391–407, December 1976.

[Hil88] A. D. Hills and N. A. Mirza. Fault tolerant avionics. In *AIAA/IEEE 8th Digital Avionics Systems Conference*, pages 407–414, San Jose, CA, October 1988.

[Hor74] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell. A program structure for error detection and recovery, E. Gelenbe and C. Kaiser, editors. *Lecture Notes in Computer Science*, 16:171–187. Springer, 1974.

[Jos88a] M. K. Joseph and A. Avižienis. A fault tolerance approach to computer viruses. In *IEEE Symposium Security and Privacy*, pages 52–58, Oakland, CA, April 1988.

[Jos88b] M. K. Joseph. *Architectural Issues in Fault-Tolerant, Secure Computing Systems*. PhD dissertation, UCLA, Computer Science Department, June 1988.

[Kel83] J. P. J. Kelly and A. Avižienis. A specification-oriented multi-version software experiment. In *Digest of 13th FTCS*, pages 120–126, Milano, Italy, June 1983.

[Kel86] J. Kelly, A. Avižienis, B. Ulery, B. Swain, M. Lyu, A. Tai, and K. Tso. Multi-version software development. In *Proc. IFAC Workshop SAFECOMP'86*, pages 35–41, Sarlat, France, October 1986.

[Kem85] R. A. Kemmerer. Testing formal specifications to detect design errors. In *IEEE Transactions on Software Engineering*, SE-11(1):32–43, January 1985.

[Kim88] K. H. Kim and J. C. Yoon. Approaches to implementation of a repairable distributed recovery block scheme. In *Digest of 18th FTCS*, pages 50–55, Tokyo, Japan, June 1988.

[Kni86] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. In *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.

[Kop74] H. Kopetz. Software redundancy in real time systems. In *Information Processing 74, Proc. of IFIP Congress*, pages 182–186, Stockholm, Sweden, August 1974.

[Lal88] J. H. Lala and L. S. Alger. Hardware and software fault tolerance: a unified architectural approach. In *Digest of 18th FTCS*, pages 240–245, Tokyo, Japan, June 1988.

[Lap84] J. C. Laprie. Dependability evaluation of software systems in operation. In *IEEE Transactions on Software Engineering*, SE-10(11):701–714, November 1984.

[Lar34] D. Lardner. Babbage's calculating engine, *Edinburgh Review, July 1834*. Reprinted in P. Morrison and E. Morrison, editors, *Charles Babbage and His Calculating Engines*, page 177, Dover, New York, 1961.

[Lit87] B. Littlewood and D. R. Miller. A conceptual model of multi-version software. In *Digest of 17th FTCS*, pages 150–155, Pittsburgh, PA, July 1987.

[Lyu88] M. R. Lyu. *A Design Paradigm for Multi-Version Software*. PhD dissertation, UCLA, Computer Science Department, May 1988.

[Lyu92a] M. R. Lyu and A. Avižienis. Assuring design diversity in N-version software: a design paradigm for N-version programming, pages 197–218. In J. F. Meyer and R. D. Schlichting, editors, *Dependable Computing for Critical Applications 2*. Springer-Verlag, Wien, New York, 1992.

[Lyu92b] M. R. Lyu, Chen J. H., and A. Avižienis. Software diversity metrics and measurements. In *Proc. IEEE COMPSAC 1992*, pages 69–78, Chicago, Illinois, September 1992.

[Lyu93] M. R. Lyu and Y-T. He. Improving the N-version programming process through the evolution of a design paradigm. In *IEEE Transactions Reliability*, R-42(2):179–189, June 1993.

[Mak84] S. V. Makam and A. Avižienis. An event-synchronized system architecture for integrated hardware and software fault-tolerance. In *Proc. IEEE 4th International Conference Distributed Computing Systems*, pages 526–532, San Francisco, CA, May 1984.

[Ram81] C. V. Ramamoorthy et al. Application of a methodology for the development and validation of reliable process control software. In *IEEE Transactions on Software Engineering*, SE-7(11):537–555, November 1981.

[Ran75] B. Randell. System structure for software fault-tolerance. In *IEEE Transactions on Software Engineering*, SE-1(6):220–232, June 1975.

[Ran87] B. Randell. Design fault tolerance, pages 251–270. In [Avi87a].

[Sco84] R. K. Scott, J. W. Gault, D. F. McAllister, and J. Wiggs. Experimental validation of six fault-tolerant software reliability models. In *Digest of 14th FTCS*, pages 102–107, Orlando, FL, June 1984.

[Sco87] R. K. Scott, J. W. Gault, and D. F. McAllister. Fault-tolerant software reliability modeling. In *IEEE Transactions on Software Engineering*, SE-13(5):582–592, May 1987.

[Shr85] K. S. Shrivastava, editor. *Reliable Computing Systems: Collected Papers of the Newcastle Reliability Project*. Springer, Wien, New York, 1985.

[Tai93a] A. T. Tai, A. Avižienis, and J. F. Meyer. Evaluation of Fault-Tolerant Software: A Performability Modeling Approach, pages 113–135. In C. E. Landwehr, B. Randell and L. Simoncini, editors, *Dependable Computing for Critical Applications 3*. Springer-Verlag, Wien, New York, 1993.

[Tai93b] A. T. Tai, J. F. Meyer, and A. Avižienis. Performability enhancement of fault-tolerant software. In *IEEE Transactions Reliability*, R-42(2):227–237, June 1993.

[Tra88] P. Traverse. AIRBUS and ATR system architecture and specification, pages 95–104. In [Vog88a].

[Tso87] K. S. Tso and A. Avižienis. Community error recovery in N-version software: a design study with experimentation. In *Digest of 17th FTCS*, pages 127–133, Pittsburgh, PA, July 1987.

[Tso86] K. S. Tso, A. Avižienis, and J. P. J. Kelly. Error recovery in multi-version software. In *Proc. IFAC Workshop SAFECOMP'86*, pages 35–41, Sarlat, France, October 1986.

[Tur86] R. Turn and J. Habibi. On the interactions of security and fault tolerance. In *9th National Computer Security Conference*, pages 138–142, September 1986.

[Vog88a] U. Voges, editor. *Software Diversity in Computerized Control Systems*. Springer, Wien, New York, 1988.

[Vog88b] U. Voges. Use of diversity in experimental reactor safety systems, pages 29–49. In [Vog88a].

[Wal88] C. J. Walter. MAFT: An architecture for reliable fly-by-wire flight control. In *AIAA/IEEE 8th Digital Avionics Systems Conference*, pages 415–421, San Jose, CA, October 1988.

[Wil83] J. F. Williams, L. J. Yount, and J. B. Flannigan. Advanced autopilot flight director system computer architecture for Boeing 737-300 aircraft. In *AIAA/IEEE 5th Digital Avionics Systems Conference*, Seattle, WA, November 1983.

[Wu90] C. S. Wu. *Formal Specification Techniques and Their Applications in N-Version Programming*. PhD dissertation, UCLA, Computer Science Department, October 1990.

[Zav86] P. Zave and W. Schell. Salient features of an executable specification language and its environment. In *IEEE Transactions on Software Engineering*, SE-12(2):312–325, February 1986.