Dependable Computing Depends on Structured Fault Tolerance

Algirdas Avižienis
University of California, Los Angeles, CA, U.S.A., and
Vytautas Magnus University, Kaunas, Lithuania
Email:aviz@cs.ucla.edu

Abstract

Fault tolerance is a fundamental technique for the attainment of dependable computing. This paper discusses a general paradigm for the design of fault-tolerant systems and illustrates it by a design paradigm for fault-tolerant software.

1 Introduction

Today we are witnessing an explosive growth in the complexity of contemporary computing and communication systems. Concurrently, one fundamental principle is becoming more and more evident: the more good our sophisticated computing and communication systems can contribute to the wellbeing and the quality of life of the human race, the more harm they can cause when they fail to perform their functions, or perform them incorrectly. Let us consider the control of air, rail, and subway traffic, the emergency response systems of our cities, the flight controls of airliners, the safety systems of nuclear power plants, and most of all, the rapidly growing dependence of health care delivery on high-performance computing and communications.

At the same time, the challenges to dependable operation are also growing in scope and severity. Design faults cause system crashes at the most inopportune times. Complex systems suffer stability problems due to unforeseen interactions of overlapping fault events and mismatches of defense mechanisms. "Hackers" and individuals with criminal intent invade systems and cause disruptions, misuse, and damage. Accidents lead to the severing of communications links that serve entire regions. Finally, it may be foreseen that "info-terrorists" will attempt to cause similar damage with malicious intent.

In the presence of all these threats, fault tolerance is the essential guarantee that our vitally important systems will not, figuratively speaking, turn against their builders and users by failing to serve as expected because of physical, design, or human-machine interaction faults, or even because of malicious attempts to disrupt their essential services.

However, introducing fault tolerance into a complex system is a difficult task. Past experience has shown that fault tolerance is most effective when it is an integral function of every subsystem as well as a hierarchically organized function of the entire system.

This paper describes such a structured approach to the design of fault-tolerant systems, with emphasis on the tolerance of software design faults by means of design diversity.

2 Systematic Design of Fault-Tolerant Systems

The concept of fault tolerance originally appeared in technical literature in 1967 as follows [1]

"We say that a system is fault-tolerant if its programs can be properly executed despite the occurrence of logic faults."

The prime motivation for the creation of the concept was the new challenge of building unmanned spacecraft for interplanetary exploration that was assigned by the U.S. National Aeronautics and Space Administration (NASA) to Caltech's Jet Propulsion Laboratory (JPL) in late 1958. Mission lengths of up to ten years and more were being considered, and onboard computing was a prerequisite for their success. Design of computers that would survive a journey of several years and then deliver their peak performance at a distant planet was an entirely unexplored discipline.

Existing theoretical studies of the long-life problem indicated that large numbers of spare subsystems offered a promise of longevity, given that all spares could be successfully employed in sequence. The JPL problem was to translate the idealized "spare replacement" system model into a flightworthy implementa-

1071-9458/95 $4.00 ©1995 IEEE

tion of a spacecraft guidance and control computer. A proposal to design such a computer, called "A Self-Testing-And-Repairing System for Spacecraft Guidance and Control," and designated by the acronym "STAR" was presented in October, 1961 [2], and the research effort continued for more than ten years, culminating with the construction and demonstration of the laboratory model of the JPL- STAR computer [3]. A flight model of the JPL-STAR was designed for a 10-15 year space mission, but its building was halted when NASA discontinued the Grand Tour mission for which it was intended.

The longevity requirement led to the study of all accessible engineering solutions and theoretical investigations of reliability enhancement. The variety of existing theories and techniques motivated the definition of the unifying concept of fault tolerance that merged diverse approaches into a cohesive view of all system survival attributes, and greatly facilitated the design of the JPL-STAR computer.

During the next two decades - the 70's and the 80's - we have seen a continuing expansion of the universe of faults that are to be tolerated by fault- tolerant systems. The original concept dealt with transient and permanent logic faults of physical origin. Faults due to human mistakes in design were added when the growing complexity of software and of logic on VLSI chips made the removal of all design faults prior to operational use not certain. Experience also led to the addition of interaction faults, inadvertently introduced by humans during the operation or maintenance of a computer.

Finally, consequences of malicious actions intended to alter or to stop the service being delivered by a system were recognized as being deliberate design faults. This concept establishes a common ground for the unified treatment of security and fault tolerance concerns in system design. The assurance of full compatibility and integration of security and fault tolerance techniques is a major challenge for contemporary designers.

In retrospect, it may be said that the concept of fault tolerance has served well during the past quarter of a century in facilitating the appearance of successively more dependable systems for the control and support of various essential functions of contemporary society: computing and communications, transportation, nuclear power, financial transactions, health care delivery, etc.

Thirty years of experience have shown that the building of dependable systems requires the balanced use of both fault avoidance and fault tolerance techniques. An imbalance in either direction leads to an ineffective use of resources and severely limits the attainable dependability. The definition of the concept of fault tolerance initiated the evolution of principles for the systematic design of fault-tolerant systems. The specification and design of the STAR computer at JPL involved much improvisation and experimentation with design alternatives. It became apparent that the lessons learned during this process could serve as the foundation for a more orderly approach that would utilize a set of guidelines for the choice of fault masking, error detection, fault diagnosis, and system recovery techniques.

The first effort to devise such guidelines was presented at the 1967 Fall Joint Computer Conference in the paper "Design of Fault-Tolerant Computers" [1]. That paper first introduced the term "fault-tolerant computer" and the concept of "fault tolerance" into technical literature. It also presented a classification of faults and outlined the alternate forms of masking, diagnosis, and recovery techniques along with some criteria for choices between "massive" (i.e., masking) and "selective" application of redundancy. The design of the JPL- STAR computer was used to illustrate the application of these criteria in choosing the fault tolerance techniques for a spacecraft computer that had long life and autonomy requirements with strict weight and power constraints.

3 A Design Paradigm

The 1967 paper was the first of a sequence of publications that formulated an evolving view of how to attain dependable computing by the structured introduction of fault tolerance during system design. Two different classes of faults - those due to physical causes and those due to human mistakes, oversights, and deliberate actions are considered. This evolving view was presented in a series of papers on guidelines for fault-tolerant system design and implementation, supported by specific discussions of the techniques, scope, and aims of fault tolerance and fault avoidance in hardware, software, communication, and man/machine interfaces. Milestones of this series have been the papers: [4, 5, 6, 7, 8, 9]. Strong motivation for the effort came from the increasing number of successful fault-tolerant systems that offered new design insights and more operational experience.

The unifying theme of the above referenced work over the past three decades has been the evolution of a design paradigm for fault-tolerant systems that guides the designer to consider fault tolerance as a fundamental issue throughout the design process. The word "paradigm" is used here in the dictionary sense of "pattern, example, model" in place of the word "methodology" that implies a study of methods, rather than a set of guidelines with illustrations that is discussed here.

Taken in order of appearance, the papers show a progressive refinement of concepts and an expansion of the scope to include the tolerance of "human made" design and interaction faults. Other recently introduced themes are the balancing of performance and fault tolerance objectives during system partitioning, and the integration of subsystem recovery procedures into a multi-level recovery hierarchy. Strong emphasis has been directed to the application of design diversity in a multichannel system in order to attain tolerance of design faults, [6, 8], including the tolerance of deliberate design faults [10].

Fault tolerance has now been recognized as the key prerequisite of dependability for very large systems, such as the FAA's Advanced Automation System for air traffic control [11]. Because of their great functional complexity, such systems pose the most severe challenge yet in the design of fault-tolerant systems. The introduction of fault tolerance into very complex, distributed systems is most likely to succeed if a methodical approach is employed. This approach begins with the initial design concepts and requires the collaboration of performance and fault tolerance architects during the critical tasks of system partitioning, function allocation, and definition of inter- subsystem communication and control. Such a highly structured design approach is presented here as the design paradigm for fault-tolerant systems.

The design paradigm is an abstraction and refinement of observed design processes, in which the various steps often overlap. Its objective is to minimize the probability of oversights, mistakes, and inconsistencies in the process of meeting the specified goals of dependable service with respect to defined classes of faults by means of the chosen implementation of fault tolerance. The paradigm is stated for the implementation of a new design. If the goal is the improvement of an existing design, then each step is a reexamination, possibly leading to changes of previously made decisions.

The paradigm partitions the system building process into three activities: specification, design, and evaluation. Design consists of system partitioning, subsystem design, and system integration steps. Evaluation takes place during and after each design step. The principal steps of the paradigm are summarized

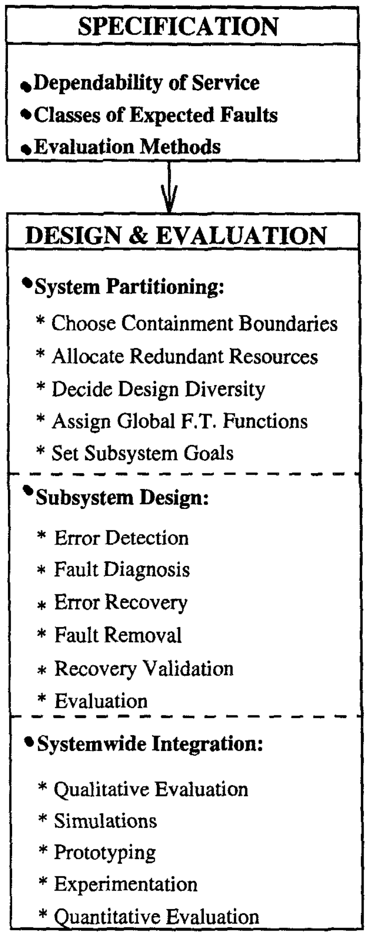

Figure 1: A Design Paradigm

in Figure 1, and a detailed discussion is presented in [9].

4 Software Design Diversity: A Structured Approach

At the present time it is becoming quite apparent that *design faults* are and will continue to be the most costly class of faults. Recent examples are the $475 million design fault in the Pentium microprocessor, the drastic reductions of the $4.8 billion AAS system [11], and major computer systems (for example, the $51 million state of California DMV database system and the $74 million Los Angeles County medical information system) that never became operational because of an excessive number of software design faults.

Design diversity is a fundamental solution to assure fault tolerance of design faults [6]. This section summarizes the principal concepts of design diversity as it is employed to attain software fault tolerance.

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 is 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. 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* [6].

Design diversity is applicable to tolerate design faults in hardware as well. Some multichannel systems with diverse hardware and software have been built; they include the flight control computers for the Boeing 777 [12], and the Airbus [13] 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)[14], shown in Figure 2 and *recovery blocks* (RB)[15] shown in Figure 3. 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 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. The use of two or more diverse 2-version software units leads to the "N self-checking programming" approach[16].

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 *recovery cache* [17] that is part of the EE. In NVS, forward recovery is done by the use of the *community error recovery* algorithm [18] that is supported by the specification of *recovery points* and by the decision algorithm of the EE. Both recovery

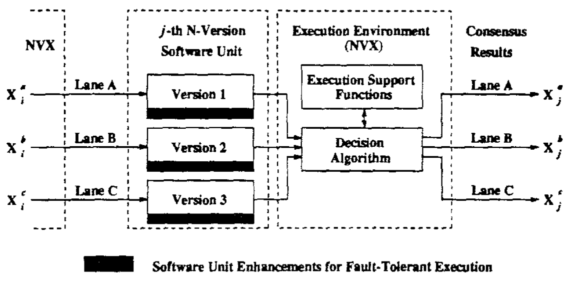

Figure 2: The N-version software (NVS) model with n = 3

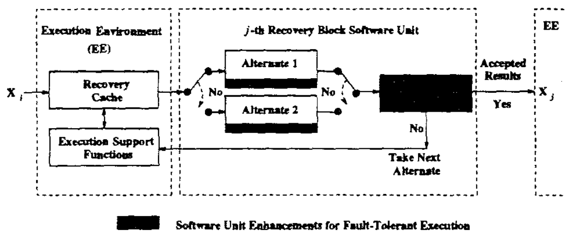
Execution Environment (EE)
j-th Recovery Block Software Unit
EE
X_i
Recovery Cache
Alternate 1
No
Alternate 2
No
Accepted Results
Yes
X_j
Execution Support Functions
No
Take Next Alternate
Software Unit Enhancements for Fault-Tolerant Execution

Figure 3: The recovery block (RB) model

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.

5 A Design Paradigm for N-Version Software

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 was initiated at UCLA in early 1975 as a part of research in reliable computing that was started in 1961 [2]. Table 1 summarizes the investigations conducted at UCLA since 1975.

The experience that had been accumulated during the first four investigations at UCLA [6, 19, 20] led to the rigorous definition and application of a set of guidelines, called the *NVS Design Paradigm* [21] during the subsequent Six-Language NVP project [22]. The paradigm as it was further refined during this project, is summarized in Figure 4 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

Table 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

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 4, 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.

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.

(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 [23].

(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.

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

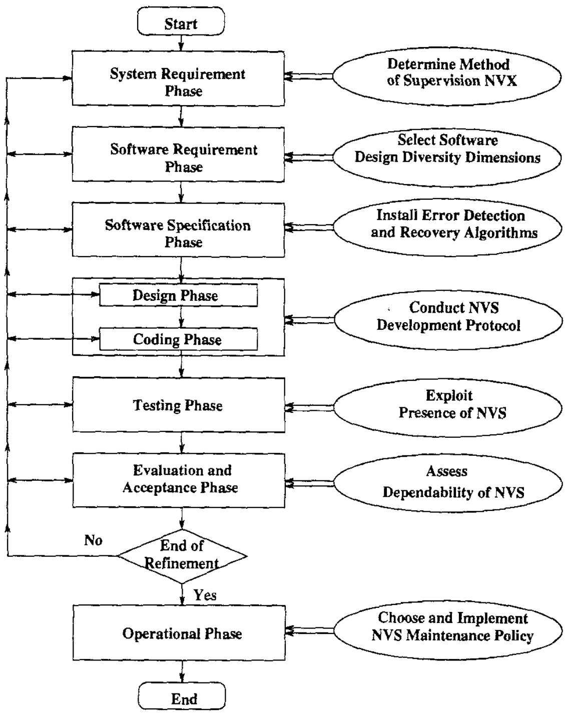
Start
System Requirement Phase
Software Requirement Phase
Software Specification Phase
Design Phase
Coding Phase
Testing Phase
Evaluation and Acceptance Phase
End of Refinement
No
Yes
Operational Phase
End
Determine Method of Supervision NVX
Select Software Design Diversity Dimensions
Install Error Detection and Recovery Algorithms
Conduct NVS Development Protocol
Exploit Presence of NVS
Assess Dependability of NVS
Choose and Implement NVS Maintenance Policy

Figure 4: A design paradigm for N-version programming (NVP)

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.

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* [24].

(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 [6], and a set of two V-specs that were derived by two independent efforts [25]. These approaches provide additional means for the verification of the V-specs, and offer diverse starting points for version implementors.

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.

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.

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 [26].

(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.

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.

6 Conclusion

Fault tolerance is the essential technique to assure dependability of systems that are far too complex for proofs of design correctness or for exhaustive testing. However, imperfections in the implementation of fault tolerance pose a lethal threat, and the utmost rigor needs to be applied when fault tolerance is being introduced during system design. The design paradigms described above are a first step in that direction, and further steps remain a challenge to the builders of future systems that we can justifiably trust to deliver the required services when they are needed.

Acknowledgements

The year 1995 marks 40 years since the author's first professional work on system dependability at JPL. It has been my good fortune to work with and learn from many colleagues at the University of Illinois, Caltech's Jet Propulsion Laboratory and the UCLA Computer Science Department. The work with my friends in the IEEE CS Technical Committee on Fault-Tolerant Computing and IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance has been a rewarding experience and a source of inspiration for this presentation.

References

[1] A. Avižienis, "Design of Fault-Tolerant Computers," *AFIPS Conference Proceedings*, 1967 Fall Joint Computer Conference, Vol. 31, pp. 733–743, 1967.

[2] A. Avižienis and D. A. Rennels, "The Evolution of Fault Tolerant Computing at the Jet Propulsion Laboratory and at UCLA: 1960-1986," in *The Evolution of Fault-Tolerant Computing.* Vienna and New York: Springer-Verlag, 1987.

[3] A. Avižienis, G. C. Gilley, F. P. Mathur, D. A. Rennels, J. A. Rohr, and D. K. Rubin, "The STAR (Self-Testing-and-Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," *IEEE Trans. Computers*, Vol. C-20, pp. 1312-1321, November 1971.

[4] A. Avižienis, "Architecture of Fault-Tolerant Computing Systems," *Digest of FTCS-5*, Paris, pp. 3–16, June 1975.

[5] A. Avižienis, "Fault-Tolerance: The Survival Attribute of Digital Systems," *Proceedings of the IEEE*, Vol. 66, pp. 1109–1125, October 1978.

[6] A. Avižienis and J. P. J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," *Computer*, Vol. 17, pp. 67–80, Aug. 1984.

[7] A. Avižienis and J. C. Laprie, "Dependable Computing: From Concepts to Design Diversity," *Proceedings of the IEEE*, Vol. 74, pp. 629-638, May 1986.

[8] A. Avižienis, "Software Fault Tolerance," in *Information Processing 89, Proceedings of the IFIP 11th World Computer Congress* , San Francisco, CA, G.X. Ritter (Ed.), B.V. North Holland: Elsevier Science Publishers, pp. 491–498. 1989,

[9] A. Avižienis, "Building Dependable Systems: How to Keep Up with Complexity," *25th International Symposium on Fault-Tolerant Computing, Special Isssue.*, Pasadena, CA, pp. 4–14, June 1995.

[10] M. K. Joseph and A. Avižienis, "A Fault Tolerance Approach to Computer Viruses," *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, Oakland, CA, pp. 52–58, April 18–21, 1988.

[11] A. Avižienis and D. E. Ball, "On the Development of a Highly Dependable and Fault Tolerant Air Traffic Control System," *Computer*, Vol. 20, pp. 84–90, February 1987.

[12] R. Riter, "Modeling and Testing a Critical Fault-Tolerant Multi-Process System," *Digest of Papers of FTCS-25*, Pasadena, CA, pp. 516–521, June, 1995.

[13] D. Briere and P. Traverse, "AIRBUS A320/A330/A340 Electrical Flight Controls, A Family of Fault-Tolerant Systems", *Digest of Papers of FTCS-23*, Toulouse, France, pp. 616–623, June, 1993.

[14] A. Avižienis and L. Chen, "On the implementation of N-version programming for software fault tolerance during execution," *Proc. IEEE COMPSAC 77*, pp. 149–155, November 1977.

[15] B. Randell. "System structure for software fault-tolerance," *IEEE Trans. Software Engineering*, vol SE-1, pp. 220–232, June 1975.

[16] J. C.Laprie, J. Arlat, C. Beounes, K. Kanoun, and C. Hourtolle, "Hardware and software fault-tolerance: definition and analysis of architectural solutions", *Digest of Papers of FTCS-17*, Pittsburgh, PA, pp. 116–121, June, 1987.

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

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

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

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

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

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

[23] A. Avižienis, "The N-version approach to fault-tolerant software, *IEEE Trans. Software Engineering*, vol SE-11, pp. 1491–1501, December 1985.

[24] S. S. Brilliant, J. C. Knight, and N. G. Leveson, "The consistent comparison problem in N-version software," *IEEE Trans. Software Engineering*, vol 15, pp. 1481–1485, November 1989.

[25] C. V. Ramamoorthy et al, "Application of a methodology for the development and validation of reliable process control software," *IEEE Trans. Software Engineering*, vol SE-7, pp. 537–555, November 1981.

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