INFORMATICA, 1990, Vol.1, No.2, 003–034

SOFTWARE FAULT TOLERANCE

Algirdas AVIŽIENIS

Vytautas Magnus University, Kaunas, Lithuania, and
UCLA Dependable Computing and Fault–Tolerant
Systems Laboratory Computer Science Department,
University of California, Los Angeles, CA 90024, USA

Abstract. A fault-tolerant software unit is composed of $N \geqslant 2$ diverse member units, usually developed by N separate teams, and an execution environment. The development process employs diversity requirements, communication protocols, and inter–team isolation rules to promote the greatest possible independence of team efforts and diversity among their products. The principal models, specification, building, evaluation, and system integration of fault–tolerant software are discussed, and goals for future work are suggested.

Key words: fault tolerance software, dependable computing, dependable softvare, fault–tolerant softvare.

1. Fault tolerance and design diversity. *Fault tolerance* is a function of computing systems that serves to assure the continued delivery of required services in the presence of faults which cause errors within the system [1]. Defenses against *physical faults* that affect computing hardware have been employed since the first computers were built in the 1940's [2], and the unifying concept of fault tolerance was formulated in 1967 [3,4]. The abbreviations "FT" for "fault tolerance" and "f–t" for "fault–tolerant" will be used throughout this paper.

IFIP Working Group 10.4, "Dependable Computing and Fault Tolerance," was established in 1980 and has taken a leading role in promoting this field of computer science and engineering. A comprehensive book, containing 18 contributions on the evolution of fault–tolerant computing throughout the world was the result of a symposium that was organized by WG 10.4 and held in Baden, Austria in 1986 [2].

Software faults, or "bugs," are not of physical nature; they are *design faults*, due to the mistakes and oversights of humans that occur while they specify, design, build, operate, modify, and maintain the software of computing systems. Fault avoidance and fault removal after failures occur are the usual means to cope with software faults. Research efforts to devise fault tolerance techniques for software faults have been active since the early 1970's, and systems that can tolerate software faults have been built for railway switching, aircraft flight control, and nuclear reactor monitoring [5]. This paper addresses the main issues of software fault tolerance: the models, specification, building, evaluation, and system integration of fault–tolerant software. Two sources are recommended for further insights: the discussion of design fault tolerance by Brian Randell [6], and the 1988 book on software diversity edited by Udo Voges, which also contains an annotated bibliography of 208 entries [5].

We say that a unit of software (module, CSCI, etc.) is *fault–tolerant* 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 or "bugs", called *software faults* in this paper, 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 *simplex* in this paper.

Multiple, redundant computing channels (or "lanes") have been widely used in sets of $N = 2, 3$, or 4 to build f–t hardware systems [2, 7]. 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 $N \geqslant 2$. The *redundant* units are intended to compensate for, or mask a failed software unit when they are not affected by the same software fault. 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 FT. Copying software will also copy the dormant software faults; therefore each simplex unit in the f–t set of N needs to be built separately and independently of the other members of the set. This is the concept of software *design diversity* [9].

Design diversity is applicable to tolerate design faults in hardware as well [7, 9]. A few multichannel systems with diverse hardware and software have been built; they include the flight control computers for the Boeing 737–300 [10], and the Airbus [11] 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, etc. In human activities we have the pilot–copilot–flight engineer teams in cockpits of airliners, two–or three–surgeon teams at difficult operations, and similar arrangements.

Two techniques that support fault tolerance of software remain outside of the scope of this paper. They are: (1) the minimization of the number of undetected dormant software faults in simplex units by the use of advanced software engineering methods in their building, testing, and integration, as well as by the use of proof techniques, and (2) the incorporation of error detecting and fault handling features, such as assertions, reasonableness checks, exception detectors and handlers, etc., into simplex software units. Both techniques enhance the dependability of each member, and thus reinforce the FT capability of a f–t software unit; however, they are not sufficient to make a simplex unit fault–tolerant. Furthermore, the building of executive software that supports and supervises the functioning of hardware FT with respect to physical faults is an important issue that is beyond the bounds of this review.

2. Models and techniques. A set of $N \geqslant 2$ 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 FT 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 $N \geqslant 2$ member units in a fault–tolerant manner; (2) the specifications of the individual member units must define the FT 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 known as *Recovery Blocks* (RB) [6, 12] and *N-Version Software* (NVS) [13, 14]. 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* (DA) 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* (AT) to the output of an individual alternate; this AT must by necessity be *specific* for every distinct service, i.e., it is customdesigned 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 AT in RB. They can detect disagreements between the versions, or between the alternate and the AT, 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 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* [12], or *recovery cache* [15] that is part of the EE. In NVS, forward recovery is done by the use of the *community error recovery* algorithm [16] 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 AT are passed along and do not trigger recovery; in NVS, recovery will be wrong 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 AT is done by NVS technique, i.e., when the AT 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 AT), and that the decision algorithm DA takes the known failures of member versions into account [17, 18]. Reinforcement of the DA by means of a preceding AT (*a filter*) has been addressed in [19], and the use of an AT when the DA cannot make a decision in [20, 21].

The RB technique evolved as a result of the long–term investigation of system reliability that was initiated by Brian Randell at the University of Newcastle upon Tyne in 1970 [22]. In the RB technique the $N-1$ alternates and the AT are organized in a manner similar to the dynamic redundancy (standby sparing) technique in hardware [23]. RB performs run–time software, as well as hardware, error detection by applying the AT to the results delivered by the first alternate. If the AT is not passed, recovery is implemented by state restoration, followed by the execution of the next alternate. Recovery is considered complete when the AT 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 [6]. The properties of RB software units are discussed in section 4 of this paper.

The effort to develop a systematic process (a *paradigm*) for the building of f–t 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 the author had started in 1961 [4]. The process was first called "redundant programming" [24], and was renamed "N–Version Programming" (NVP) in the course of the next two years [13]. The name "N–Version Software" (NVS) is used to designate the f–t software units that are the products of the NVP process. The research effort has continued until the present, and a summary of the results is presented in the following section 3.

N–version software had remained of little interest to the mainstream researchers and developers of software for a relatively long time. Some suggestions had appeared in the early and mid–1970's [25, 26, 27, 28]; however, 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 [29]:

"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."

Charles Babbage himself had written in 1837 in a manuscript that was only recently published [30]:

"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."

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 ($N \geqslant 2$) *member versions* of the NVS unit. The *process* by which the NVS versions are produced is called *N-Version Programming* (NVP). The EE that embeds the N versions and supervises their f–t execution is called the *N-Version Executive* (NVX). The NVX may be implemented by means of software, hardware, or 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.

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) software development methods; (4) programming languages; (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 furthermore possible to impose differing diversity requirements for separate stages of the software development process, such as design, coding, testing, and even for the process of writing of the 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 [32]. 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* [16] 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. [14, 39]. 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. Two cases have been practically explored: a set of three V–specs (formal algebraic OBJ, semi–formal PDL, and English) that were derived together [17, 18], and a set of two V–specs that were derived by two independent efforts [31]. These approaches provide additional means for the verification of the V–specs, and offer diverse starting points for version implementers.

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 [33, 34], 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 contemporary specification languages, promising candidates for V–specs that have been studied and used at UCLA are OBJ [35] that has been further developed, the Larch family of specification languages [36], PAISLey from AT&T Bell Laboratories [37], and also Prolog as a specification language.

3.2 The N–Version Programming Process: NVP NVP has been defined from the beginning as "the independent generation of $N \geqslant 2$ functionally equivalent programs from the same initial specification" [13]. "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" [13]. The NVP process has been developed since 1975 in an effort that included five consecutive experimental investigations [17, 18, 32, 38]. The following description presents the current NVP paradigm for the development of NVS.

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 sound 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.

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* (sec. 4.1) 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 [14, 39].

4. Building recovery blocks. A Recovery Block (RB) is a f–t software unit originally defined as consisting of two or more simplex software units called *alternates,* and one software unit called an *acceptance test* (AT) that "... is a logical expression without side effects (which) is evaluated on exit from any alternate to determine whether the alternate has performed acceptably" [23]. Only one AT (for all alternates) is to be used with any one RB, and "... it is for the designer to decide upon the appropriate level of rigor of the test" [23]. Explicit requirements for independence between the programming of the alternates and the acceptance test are not stated in this original formulation. A later large–scale experimental study required alternate modules of independent design, and "... a strict discipline precluding cooperation or consultation ...", since full isolation was not practical [40]. A rigorous process, analogous to the NVP paradigm, has not been explicitly prescribed for the building of RB units; however, the NVP process is readily adaptable. A similar observation applies also to the RB specification that should serve as the starting point for the alternates and the acceptance test AT. Once again, the V–spec approach of NVS is adaptable; however, it remains to be established how the AT specification should differ from the specification of the alternates in order to convey the unique properties and constraints of the AT.

The RB concept also poses a requirement to provide a *recovery structure* which is common to a set of interacting processes, since it is not known at the time of the interaction whether a process may be "backed up" upon failing the AT at a later time. Such a recovery structure has been termed a *conversation* [23], later implemented in a restricted from as a *dialogue* [40]. The NVS approach avoids the need for such dialogues because it requires that a cc–point should be located at every output of a version. Other sophisticated RB features that have been devised are RB *nesting,* and *multilevel structuring* of error recovery, using f–t virtual machine interfaces [23]. Their specification as well as the specification of dialogues are added requirements for the specification of RBs. The RB EE is required to provide an automatic method for the resetting of the system to the state it had just before the entry to the primary alternate. The mechanism to accomplish this goal was the *recursive cache* [12], which was subsequently refined and implemented in hardware as the *recovery cache* [15, 41]. The recent experimental RB study also shows that additional EE functions had to be provided in the from of extensions to the existing MASCOT operating system to provide support for dialogues and to utilize the hardware recovery cache [40]. Early implementation studies of the RB approach have been described in [15, 42].

A variation of the RB is the *distributed* RB (DRB) scheme [43, 44, 45]. In DRB, two or more identical RBs are executed concurrently on separate hardware channels, with each RB executing a different alternate and then performing the AT. The computation can proceed as long as at least one alternate passes the AT; however, recovery still needs to be implemented in the RBs with failed alternates. A hardware testbed for DRB has been implemented, and a long–term investigation is in progress. The error detection coverage of the AT is the most critical parameter for the DRB scheme, since the same AT is employed throughout all nodes. DRB comes close to NVS when the concurrently executing nodes interact [45], since a consensus decision can be implemented through such interaction. The remaining difference is that the same AT is used for every alternate in DRB, while the NVP process allows every P–team to implement its own internal error detection and exception handling [17, 18].

Another variation in which the alternates are executed concurrently and employ either individual ATs or pairwise comparisons, has been termed *N self–checking programming* [46]. Combinations of RB and NVS have also been proposed. In the *two–step adjudicator* a "filter" AT precedes the decision [19], while in *consensus RB* the AT is invoked if a decision cannot be made by the DA [20, 21].

5. Dependability modeling of f–t software. The benefits of fault tolerance are predicted by quantitative modeling of the *reliability, availability,* and *safety* (i.e., the *dependability*) 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 FT 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 AT has *less than perfect coverage* (due to either incompleteness, or own faults) with respect to the faults in RB alternates. The *correlation factor* c' is the conditional probability of a majority of versions (in NVS) or one alternate and the AT (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 c' [47, 48]. The criticality of related faults had been recognized quite early for both RB [42] and NVS [47]; later the same problem was investigated, apparently without awareness of the earlier results, in [20, 49], reaching similar conclusions. Recent 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 [50, 51, 52, 53].

6. Experimental investigations. At the time when the RB and NVP approaches were 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 effectiveness of the RB approach has been studied in two long-term efforts. Newcastle studies began in the mid–70s [15, 41] and recently dealt with the use of the RB technique and "conversations" in a medium–scale (8000 lines of source code in Coral language, 14 concurrent activities) real–time command and control system [40]. The ongoing UC Irvine effort has investigated distributed RB implementation of real–time radar–tracking programs [44, 45].

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 first detailed review of NVP and a discussion of two sets of results, using 27 and 16 independently written versions, were published in 1977 and 1978, respectively [13, 32]. The subsequent investigation employed three distinct specifications: algebraic OBJ [35], structured PDL, and English, and resulted in 18 versions of an "airport scheduler" program [17]. This effort was followed by five versions of a program for the NASA/Four University study [54], and then by an investigation 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 [38]. In parallel with the last two efforts, a distributed NVX supervisor called DEDIX (DEsign DIversity eXperimenter) was designed and programmed [39].

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 UCLA NVP paradigm that is described here is now considered sufficiently complete for practical application and quantitative predictions.

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 [38], and research into quantitative measures of diversity is in progress at UCLA.

Three extensive practical investigations of NVS have been performed with real–time software for nuclear reactor safety control [31, 55, 56]. Significant insights into specification, the NVP process, and the nature of software faults have resulted from these efforts.

Two other studies that claimed to investigate N–version programming were conducted in which numerical results were the principal objective. One states the intent "... to validate the authors' fault–tolerant software reliability models ..." [20, 21], one of which is called "NVP", although the authors do not reference NVP research. The other study is entitled "An experimental evaluation of the assumption of independence in multiversion programming" [57], in which "multiversion" is explicitly identified with NVP. These efforts serve to illustrate the pitfalls of premature preoccupation with numerical results. Both studies fail to recognize that NVP is rigorous process of software development. The papers do not document the rules of isolation, and the C & D protocol (sec. 3) that are indicators of NVP quality. The V–specs of [57] do not show the essential NVS attributes. It must be concluded that the authors are assessing their own *ad hoc* processes for writing multiple programs, rather than the NVP process as developed at UCLA, and that their numerical results uniquely take the measure of the quality of their casual programming process and their classroom programmers. The claims that the NVP process was investigated are not supported by documentation of the software development process in either study. The use of the term "experiment" is misleading, since it implies repeatability of the experimental procedure that is taken for granted in science.

7. The system context for f–t software. The *host system* for both RB and NVS interacts with the f–t software units through the EE, which communicates with the FT functions of its operating system or with FT 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. The recent Newcastle RB investigation employed both the hardware recovery cache and extensions to the MASCOT operating system as the implementation of the EE [40], while the distributed RB study employs hardware and a custom distributed operating system [44]. A fully developed EE for NVS is the all–software DEDIX supervisor [39], 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. Other examples of solutions are found in [58, 59, 60].

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 [37]. Subsequently, this specification will be used 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 FT 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 [23]. The AT, which is unique for every RB software unit, also may contain design faults. The obvious solution of 2–version or 3–version ATs is costly, and verification or proof of each AT appear to be the practical solutions.

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 [1]. 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 [10], the ATR.42, Airbus A–310, and A–320 aircraft [11]. New designs for the flight control computer of the planned Boeing 7J7 are the three–version GEC Avionics design [61] and the four–version MAFT system [60]. A different concept of *multi-level systems* with fault–tolerant interfaces was formulated by Randell for the RB approach [23]. Diversity is not explicitly considered for the hardware level, but appears practical when additional hardware channels are employed, either for the AT, or for parallel execution of an alternate in distributed RB [44, 45].

Computer security and software FT have the common goal to provide reliable software for computer systems [62, 63]. 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" [64]. 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 FT techniques to DDF detection and tolerance, such as in the case of computer virus containment by program flow monitors [65]. 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 study of these issues has been recently completed [66].

Modification of already operational f–t software occurs for two different reasons: (1) one of the member units (version, alternate, or AT) 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 AT. 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 AT software unit.

8. In conclusion: what is to be gained? Although at first considered as an impractical competitor of highquality single–version programs, fault–tolerant software has gained significant acceptance in academia and industry in the twelve years since the author's review of the state of fault–tolerant computing at IFIP '77 in Toronto [7]. Two, three, and four version software is switching trains [8], performing flight control computations on modern airliners [10, 11], and more NVS applications are on the way [5, 60, 61]. 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 [6, 40], distributed recovery blocks at UC Irvine [44, 45], and N–version software at UCLA [38, 39].

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 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.

Acknowledgement. This view of software fault tolerance has evolved through the author's collaboration and discussions with the members and visitors of the Dependable Computing and Fault–Tolerant Systems Laboratory at UCLA as well as the ninety–nine programmers who took part in our studies since 1975. Special thanks belong to Liming Chen, Per Gunningberg, Mark Joseph, John P.J. Kelly, Jean–Claude Laprie, Michael Lyu, Werner Schuetz, Lorenzo Strigini, Pascal Traverse, Ann Tai, Kam–Sing Tso, Udo Voges, and John F.Williams. Finally, Jacquelyn Trang prepared this pa-

per with care and concern that are greatly appreciated.

An earlier version of this paper was an invited lecture presented at the XI World Computer Congress, San Francisco, USA, August 1989.

REFERENCES

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

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

[3] Avižienis, A. (1967). Design of fault-tolerant computers. *AFIPS Conf. Proc.,* Vol.31, FJCC 1967. 733–743.

[4] Avižienis, A., and D.Rennels (1987). The evolution of fault tolerant computing at the jet propulsion laboratory and at UCLA: 1955–1986. In [2]. pp. 141–191.

[5] Voges, U. (Ed.), (1988). *Software Diversity in Computerized Control Systems.* Springer, Wien-New York.

[6] Randell, B. (1987). Design fault tolerance. In [2]. pp. 251–270.

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

[8] Hagelin, G. (1988). ERICSSON safety system for railway control. In [5]. pp. 11–21.

[9] Avižienis, A. (1982). Design diversity-the challenge for the fighties. In *Digest of 12th Inter. Symp. on Fault-Tolerant Comp.,* Santa Monica, CA, June 1982. pp. 44–45.

[10] Wiliams, J.F., L.J.Yount and J.B. Flannigan (1983). Advanced autopilot flight director system computer architecture for Boing 737-300 aircraft. In *Proc. 5th Dig. Avionics Sys. Conf.,* Seattle, WA, November 1983.

[11] Traverse, P. (1988). AIRBUS and ATR system architecture and specification. In [5]. pp. 95–104.

[12] Horning, J.J., H.C.Lauer, P.M.Melliar-Smith and B.Randell (1974). Program structure for error detection and recovery. In E.Gelenbe and C.Kaiser (Eds.), *Operating Systems,* Lect. Notes Comp. Sci., Vol.16. Springer. pp. 171–187.

[13] Avižienis, A., and L.Chen (1977). On the inplementation of N-version programming for softvare fault tolerance during execution. In *Proc. IEEE COMPSAC 77,* Nov. 1977. pp. 149–155

[14] Avižienis, A. (1985). The N-version approach to fault-tolerant software. *IEEE Trans. on Soft. Eng.,* **SE-11**(12), 1491–1501.

[15] Anderson, T., and R.Kerr (1976). Recovery blocks in action: a system supporting high reliability. In *Proc. 2nd Intern. Conf. on Soft. Eng,*San Francisco, Ca, Oct. 1976. pp. 447–457.

[16] Tso, K.S., and A.Avižienis (1987). Community error recovery in N-version software: a design study with experimentation. In *Dig. 17th Inter. Symp. Fault-Tolerant Comp.*, July 1987. pp. 127–133.

[17] Kelly, J.P.J, and A.Avižienis (1983). A specification-oriented multi-version software experiment. In *Dig. 13th Int. Symp. Fault-Tolerant Comput.,* Milano, Italy,, June 1983. pp. 120–126.

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

[19] Anderson, T. (1986). A structured decision mechanism for diverse software. In *Proc. 5th Symp. Reliability Dist. Soft. Database Systems,* IEEE, LA, January 1986. pp. 125–129.

[20] Scott, R.K., J.W.Gault, D.F.McAllister and J.Wiggs (1984). Experimental validation of six fault-tolerant software reliability models. In *Proc. 14th Int. Symp. on Fault-Tolerant Computing,* Orlando, FL, June 1984. pp. 102–107.

[21] Scott, R.K., J.W.Gault and D.F.McAllister (1987). Fault-tolerant software reliability modeling. *IEEE Trans. on Software Eng.,* **SE-13**(5), 582–592.

[22] Shrivastava, K.S. (Ed.), (1985). *Reliable Computing Systems: Collected Papers of the Newcastle Reliability Project.* Springer.

[23] Randell, B. (1975). System structure for software fault tolerance. *IEEE Trans. Soft. Eng.,* **SE-1**, 220–232.

[24] Avižienis, A. (1975). Fault tolerance and fault intolerance: complementary approaches to reliable computing. In *Proc. 1975 Int. Conf. Rel. Soft.,* LA, Apr. 1975. pp. 458–464.

[25] Elmendorf, W.R. (1972). Fault-tolerant programming. In *Proc. 1972 Int. Symp. Fault-Tolerant Comput.,* Newton, MA, June 1972. pp. 79–83.

[26] Girard, E., and J.C.Rault (1973). A programming technique for software reliability. In *Proc. 1973 IEEE Symp. Comput. Software Rel.,* New York, Apr. 30-May 2, 1973. pp. 44–50.

[27] Kopetz, H. (1974). Software redundancy in real time systems. In *Inform. Processing 74, Proc. IFIP Congress,* Stockholm, Sweden, Aug. 5-10, 1974. pp. 182–186.

[28] Fischler, M.A., O.Firschein and D.L.Drew (1975). Distinct software: an approach to reliable computing. In *Proc. 2and USA-Japan Comput. Conf.,* Tokyo, Aug. 1975. pp. 573–579.

[29] Lardner, D. (1961). Babbage's calculating engine. Reprinted in P.Morrison and E.Morrison (Eds.), *Charles Babbage and His Calculating engines.* Dover, New York. 177pp.

[30] Babbage, C. (1974). On the mathematical powers of the calculating engine, December 1837 (Unpublished Manuscript) Buxton MS7, Museum of the History of Science. Printed in B. Randell (Ed.), *The Origins of Digital Computers: Selected Papers.* Springer. pp. 17–52.

[31] Ramamoorthy, C.V., *et al.* (1981). Application of a methodology for the development and validation of reliable process control software. *IEEE Trans. Software Eng.,* **SE-7**, 537–555.

[32] Chen, L., and A.Avižienis (1978). N-version programming: a fault- tolerance approach to reliability of software operation. In *Digest of 8th Int. Symp. on F-T Comp.,* Toulouse, France, June 1978. pp. 3–9.

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

[34] Berliner, E.F., and P.Zave (1987). In An experiment in technology transfer: PAISLey specification of requirements for an undersea lightwave cable system. *Proc. 9th Int. Conf. on Software engineering,* Monterey, Ca, April 1987. pp. 42–50.

[35] Goguen, J.A., and J.J.Tardo (1979). An introduction to OBJ: a language for writing and testing formal algebaic program specifications. In *Proceedings of Specific. Rel. Software,* Cambridge, MA, April 3-5, 1979. pp. 170–189.

[36] Guttag, J.V., J.J.Horning and J.M.Wing (1985). Larch in five easy pieces. *Digital Equipment Corporation Systems Research Center,* Report No.5, Palo Alto, California, July 24.

[37] Zave, P., and W.Schell (1986). Salient features of an executable specification language and its environment. *IEEE Trans. on Software Eng.,* **SE-12**(2), 312–325.

[38] Avižienis, A., M.R.Lyu and W.Schuetz In search of effective diversity: a six-language study of fault-tolerant flight control software. In *Digest of the 18th Int. Symp. on Fault-Tolerant Comp.,* Tokyo, June 1988. pp. 15–22.

[39] Avižienis, A., M.R.-T.Lyu, W.Schutz, K-S.Tso and U.Voges (1988). DEDIX 87-a supervisory system for design diversity experiments at UCLA. In[5]. pp. 129–168.

[40] Anderson, T., P.A.Barrett, D.N.Halliwell and M.R.Moulding (1988). Tolerating software design Faults in a command and control system. In[5]. pp. 109–128.

[41] Lee, P.A., N.Ghani and K.Heron (1980). A recovery cache for the PDP-11. *IEEE Trans. Computers,* **C-29**(6), 546–549.

[42] Hecht, H. (1976). Fault-tolerant software for real-time applications. *ACM Comp. Surveys,* **8**(4), 391–407.

[43] Kim, K.H. (1984). Distributed execution of recovery bloks: approach to unoform treatment of hardware and software fauls. In *Proc. IEEE 4th Int. Conf. Dist. Comput. Syst.,* May 1984. pp. 526–532.

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

[45] Kim, K.H., and J.C.Yoon (1988). Approaches to implementation of a repairable distributed recovery block scheme. In *18th Inter. Symp. on Fault-Tolerant Comp.,* June 1988. pp. 50–55.

[46] Laprie, J.-C., *et al.* (1987). Hardware– and software–fault tolerance: definition and analysis of architectural solutions. In *Proc. 17th Intern. Symp. on F-T Comp.,* July 1987. pp. 116–121.

[47] Grnarov, A., J.Arlat and A.Avižienis (1980). On the performance of software fault tolerance strategies. In *Dig. 10th Int. Symp. Fault-Tolerant Comput.,* Kyoto, Oct.1980. pp. 251–253.

[48] Grnarov, A., J.Arlat and A. Avižienis (1982). Modeling and performance evaluation of software fault–tolerance strategies. *Technical Report No. CSD-820608,* UCLA Comp. Dept., June 1982.

[49] Eckhardt, D.E., and L.D.Lee (1985). A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Trans. Software Eng.,* **SE-11**, 1511–1517.

[50] Laprie, J.-C. (1984). Dependability evaluation of software systems in operation. *IEEE Trans. on Software Engineering,* **SE-10**(6), 701–714.

[51] Tso, K.S., A.Avižienis and J.P.J.Kelly (1986). Error recovery in multi-version software. In *Proc. IFAC Workshop SAFE-COMP'86,* Sarlat, France, October 1986. pp. 35–41.

[52] Littlewood, B., and D.R.Miller (1987). A conceptual model of multi-version software. In *Proc. 17th Intern. Symp. on Fault-Tolerant Comp.,* PA, July 1987. pp. 150–155.

[53] Arlat, J., K.Kanoun and J.-C.Laprie (1988). Dependability

evaluation of software fault-tolerance. In *The 18th Int. Symp. on Fault-Tolerant Comp.,* June 1988, Tokyo. pp. 142–147.

[54] Kelly, J.P.J., *et al.* (1988). A large scale second Generation experiment in multi-version software: Description and early results. In *The 18th Intern. Symp. on Fault–Tolerant Comput.,* June, 1988, Tokyo, Japan. pp. 9–14.

[55] Bishop, P.G. (1988). The PODS diversity experiment. In [5]. pp. 51–84.

[56] Voges, U. Use of diversity in experimental reactor safety systems. In [5]. pp. 29–49.

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

[58] Makam, S.V., and A.Avižienis (1984). An event-synchronized system architecture for integrated hardware and software fault-tolerance. In *Proceedings of the 4th Int. Conf. Distributed Comp. Systems,*San Francisco, CA, May 1984.

[59] Lala, J.H., and L.S.Alger (1988). Hardware and software Fault tolerance: A unified architectural approach. In *18th Int. Symp. Fault-Tolerant Comp.,*Tokyo, June 1988. pp. 240–245.

[60] Walter, C.J. (1988). MAFT: An architecture for reliable fly-by-wire flight control. In *AIAA/EEE 8th Digital Avionics Systems Conf.,* October 1988, San Jose, CA. pp. 415–421.

[61] Hills, A.D., and N.A.Mirza (1988). Fault tolerant avionics. *AIAA/EEE 8th Digital Avionics Systems Conference,* October 17-20, 1988, San Jose, California. pp. 407–414.

[62] Turn, R., and J.Habibi (1988). On the interactions of security and fault tolerance. In *9th Nat Comp. Security Conf.,* Sept. 1986. pp. 138–142.

[63] Dobson, J.E., and B.Randell (1986). Building reliable secure computing systems out of unreliable insecure components. In *IEEE Symp. Security and Privacy,* April 1986. pp. 187–193.

[64] U.S. Department of Defense (1985). *Trusted Computer System Evaluation Criteria* , DoD Doc. 5200.28-STD, Dec. 1985.

[65] Joseph, M.K., and A.Avižienis (1988). A fault tolerance approach to conputer viruses. In *Proc. 1988 IEEE Symp. Security and Privacy,* Oakland, CA, April 18-20, 1988. pp. 52–58.

[66] Joseph, M.K. (1988). *Architectural Issues in Fault-Tolerant, Secure Computing Systems, Ph.D. Dissertation.* Computer Science Dept., University of California, Los Angeles, CA., June 1988.

Received June 1990

A. Avižienis received the D.S., M.S. and Ph. D. degrees all in electrical engineering at the University of Illinois, Urbana–Champaign, in 1954, 1955; 1960, respectively. Currently he is Professor and Director of the Dependable Computing and Fault–Tolerant Systems Laboratory in the Computer Science Department of University of California, where since 1972 he has been the principle investigator of a continuing research project on fault–tolerant computing and system architectures. Recently he has also been elected Rector of Vytautas Magnus University, Kaunas.