ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/3188553

The N-Version Approach to Fault-Tolerant Software

Article *in* IEEE Transactions on Software Engineering · January 1986

DOI: 10.1109/TSE.1985.231893 · Source: IEEE Xplore

CITATIONS
1,023

READS
1,257

1 author:

Algirdas Avizienis
University of California, Los Angeles
175 PUBLICATIONS **15,661** CITATIONS

SEE PROFILE

All content following this page was uploaded by Algirdas Avizienis on 02 July 2014.

The user has requested enhancement of the downloaded file.

The *N*-Version Approach to Fault-Tolerant Software

ALGIRDAS AVIŽIENIS, FELLOW, IEEE

Abstract—**Evolution of the *N*-version software approach to the tolerance of design faults is reviewed. Principal requirements for the implementation of *N*-version software are summarized and the DEDIX distributed supervisor and testbed for the execution of *N*-version software is described. Goals of current research are presented and some potential benefits of the *N*-version approach are identified.**

Index Terms—**Design diversity, fault tolerance, multiple computation, *N*-version programming, *N*-version software, software reliability, tolerance of design faults.**

I. Introduction

THE transfer of the concepts of fault tolerance to computer software, that is discussed in this paper, began about 20 years after the first systematic discussion of fault-tolerant logic structures [49], [41], and the first introduction of some fault tolerance by duplication or triplication of hardware subsystems in first generation computers [43], [16], [18]. The 1950–1970 period was a time of evolution in both the theoretical development and the practical application of fault tolerance techniques in digital systems, well illustrated by the presentations at the first International Symposium on Fault-Tolerant Computing in March of 1971 [20]. In the subsequent decade, the field matured with the emergence of several successful system designs [6] followed by the appearance of the product line of Tandem computers [9].

All the efforts discussed above were aimed at the tolerance of physical faults, either of a permanent or transient nature. *Physical faults* are undesirable changes in the hardware components of the system, e.g., short circuits between two leads, open circuited transistor junctions, alpha particle impacts on dynamic MOSFET memory cells, etc. Such faults cause *errors* to appear in the information that is processed by the computing system. The errors most often appear as changes in the patterns of zeros and ones that represent the information. Other errors are time based: the information fails to arrive or arrives at the wrong time. Errors that are not detected and eliminated within the system are likely to lead to the *failure* of the computing system to deliver the expected service to other systems or to human users.

The function of fault tolerance is to preserve the delivery of expected services despite the presence of fault-caused errors within the system itself. Errors are detected and corrected, and permanent faults are located and removed while the system continues to deliver acceptable service. This goal is accomplished by the use of error detection algorithms, fault diagnosis, recovery algorithms, and spare resources. They are all integral parts of the fault-tolerant system, and may be implemented in hardware, firmware, or software.

In addition to physical faults, system failures are also caused by *design faults* and *interaction faults* [6]. These classes of faults originate due to the mistakes and oversights of humans that occur while they specify, design, operate, update, and maintain the hardware and software of computing systems. Fault avoidance and fault removal (after failure) have been generally employed to cope with design faults. Some research efforts to apply fault tolerance to software design faults have been active since the early 1970's; this paper reports the results and current work on the "*N-version*" approach that has been investigated by the Dependable Computing and Fault-Tolerant System research group at UCLA.

The *N*-version approach to fault-tolerant software depends on a generalization of the *multiple computation* method that has been successfully applied to the tolerance of physical faults. The method is reviewed and its application to the tolerance of software faults is discussed in the following section.

II. The Multiple Computation Approach and Its Extension to Design Diversity

Multiple computation is a fundamental method employed to attain fault tolerance. Multiple computations are implemented by *N*-fold ($N \geq 2$) replications in three domains: *time* (repetition), *space* (hardware), and *information* (software). In this section, a shorthand notation is employed to describe the various possible approaches to multiple computation. The nonfault-tolerant, or *simplex*, system is characterized by one execution (simplex time $1T$) of one program (simplex software $1S$) on one hardware channel (simplex hardware $1H$), and is described by the notation: $1T/1H/1S$.

One major distinction in fault-tolerant systems is between single-channel and multiple-channel architectures. The former provide only one physical (hardware) set of components needed for the computation. Error detection invokes recovery by a retry from a "rollback point" system state. This program rollback can succeed when the fault is transient, but will fail for permanent faults, leav-

Manuscript received April 22, 1985; revised July 17, 1985. This work was supported in part by the National Science Foundation under Grants MCS-72-03633, MCS-78-18918, and MCS-81-21696, by the Office of Naval Research under Contract N0014-79-C-0866, by the Battelle Memorial Institute, and by the Advanced Computer Science Program of the Federal Aviation Administration.

The author is with the Department of Computer Science, University of California, Los Angeles, CA 90024.

0098-5589/85/1200-1491$01.00 © 1985 IEEE

ing a safe shutdown of this $2T/1H/1S$ system as the only acceptable outcome. External repair service is then required to remove the fault and to return the system to an operational state. More than one retry is possible, and there can be multiple computations that use the same hardware and software, but replicate the execution N-fold in time: $NT/1H/1S$.

In the N-fold time cases (NT), the consecutive executions of a computation can employ new copies (identical replicas) of the program and associated data. A common practice in coping with system crashes is the loading from stable backup storage and running of a new copy of the program and data on the same hardware: $2T/1H/2S$. A more sophisticated fault tolerance approach that is, for example, used in Tandem computers [9], is the execution of another copy of the program that resides in a standby duplicate hardware channel: $2T/2H/2S$. It must be noted that the use of "new copies" of a program implies the existence of a *reference program* in stable storage from which the copies are made.

In a multiple-channel architecture that is limited to simplex time ($1T$), multiple computations take place as concurrent executions of N copies of the reference program on N hardware channels: $1T/NH/NS$. Examples of such systems are NASA's Space Shuttle with $N = 4$ [48], SRI's SIFT system with $N \geq 3$ [24], C. S. Draper Lab's Fault-Tolerant Multiprocessor with $N = 3$ [31], and AT&T Bell Laboratories' 1ESS central control with $N = 2$ [14]. It is evident that N-fold time may be introduced into multiple-channel architectures to produce additional variations of multiple computations. The general case is $NT/YH/ZS$, with Y channels and Z copies of the program, constrained by $NY \geq Z$.

The systems discussed above attain fault tolerance by the execution of multiple (N-fold, with $N \geq 2$) computations that have the same objective: to deliver a set of N results derived from a given set of initial conditions and inputs. The following two fundamental requirements apply to such multiple computations:

1) the *consistency* of initial conditions and inputs for all N computations must be assured; and

2) a reliable *decision algorithm* that determines a single *decision result* from the multiple results must be provided.

The decision algorithm may utilize only a subset of all N results for a decision; e.g., the first result that passes an acceptance test may be chosen. It is also possible that an acceptable decision result cannot be determined, and a higher level recovery procedure must be invoked. The decision algorithm is often implemented N times—once for each computation in which the decision result is used. In this case, only one computation is affected by the failure of any one implementation, such as a majority voter in triple-modular redundancy (TMR).

The usual partitioning of faults into "single fault" and "multiple fault" classes needs to be refined when we consider multiple computations. Faults that affect only one in a set of N computations are called *simplex* faults. A simplex fault does not affect other computations, although it may be either a single or a multiple fault within one channel. Simplex faults are very effectively tolerated by the multiple computation approach, as long as input consistency and a reliable decision algorithm are provided.

Faults that affect M out of the N separate computations are *M-plex* faults ($2 \leq M \leq N$); they affect M separate results from the set that is used to obtain the decision result. Typically, multiple occurrences of faults are divided into two classes: *independent* and *related*. We say that related *M-plex* faults are those for which a common cause that affects M computations exists or is hypothesized. Examples of common physical causes are: interchannel shorts or sneak paths, common power supply fluctuations, bursts of radiation, etc. The effects of related physical faults, i.e., the errors caused in the individual computations, are much more likely to be *distinct* than *identical* at the points at which the decision algorithm is applied.

Another class of related M-plex faults is quite different: they are *design faults* due to human mistakes committed during the design or the subsequent design modifications of a hardware or software element. All N computations that employ identical copies of that hardware or software element are affected by the design fault in the same manner when a given state of the element is reached, and the resulting errors are all identical at the decision algorithm, causing an erroneous decision result. Such total susceptibility to design faults is the most serious limitation of the multiple computation approach as it is currently applied in fault-tolerant system design.

A potentially effective method to avoid the identical errors that are caused by design faults in multiple computation systems is to use N independently designed software or/and hardware elements instead of identical copies that were generated from one design. This "design diversity" approach directly applies to the parallel (simplex time) systems $1T/NH/NS$, which can be altered to: 1) $1T/NH/NdS$, where NdS stands for *N-fold diverse software*, as used in N-version programming [5]; 2) $1T/NdH/NS$, where NdH stands for *N-fold diverse hardware*; and 3) $1T/NdH/NdS$. The N-fold time (NT) systems have been implemented as recovery blocks [47], with N sequentially applicable alternate programs that use the same hardware channel: $NT/1H/NdS$. An acceptance test is performed for fault detection, and the decision algorithm selects the first set of results that pass the test.

The use of N-version software introduces new "similarity" considerations for multiple results and for errors caused by M-plex faults. The results of individual versions often differ within a certain range when different algorithms are used in the diverse designs. Therefore, the decision algorithm may need to determine the decision result from a set of similar, but not identical, results. *Similar results* are defined to be two or more results (good or erroneous) that are within the range of variation that is allowed by the decision algorithm; consequently, a set of similar results is used to determine the decision result. When two or more similar results are erroneous, they are called *similar errors*. If the set of similar errors outnum-

bers the set of good (similar) results at a decision point, then the decision algorithm will arrrive at an erroneous decision result. For example, two similar errors will outweigh one good result in the three-version case, and a set of three similar errors will prevail over a set of two similar good results when $N = 5$. All possible errors caused by M-plex faults are now classified as either *distinct* or *similar*, and *identical* errors form a subset of *similar* errors.

When design faults occur in $M \geq 2$ diverse versions, the design faults may be either independent or related. An obvious criterion for discrimination is to designate those faults that cause similar errors at a decision point as related, and all others as independent. However, this naive criterion disregards the nature and origin of the design faults themselves. For example, it has been observed that two entirely different design faults have produced a pair of similar errors [7].

Our choice is to follow the preceding treatment of physical faults and to consider two (or $M > 2$) design faults as *potentially related* if they cause similar errors at a decision point. Potentially related faults are considered *related* if they are attributed to some common cause, such as a common link between the separate design efforts. Examples of a "common link" are: an ambiguous specification, a conversation between designers from two efforts, use of the same faulty compiler, or use of the same erroneous programmer's manual. If a common cause cannot be identified or hypothesized, the design faults are considered independent, although the errors they have caused are similar.

We anticipate situations in which two (or more) programmers will apparently quite accidentally make the same mistake. In that case, the preceding definition of related design faults allows the choice of either continuing the search for a nonapparent common cause, or considering the two faults to be independent and only coincidentally identical. This option appears to be a necessary condition at the present stage of our understanding of the causes of design faults.

III. Evolution of Fault Tolerance in Software

The evolution of fault tolerance in software has followed two distinct directions. The first direction consisted of the introduction of special fault detection and recovery features into single-version software. The second direction was the development of multiple-version diverse software for the purpose of attaining fault tolerance.

Considering single-version software, it is known that the first generation machine and assembly language programs already contained *ad hoc* provisions to detect abnormal behavior and to signal its presence to the operator. These fault detection provisions served to detect physical faults as well as software (design) faults introduced by programmers. A programmed time-out counter is a good example of such a fault detector. The sophistication of software fault detection increased with the evolution of operating systems, which took over the monitoring of the behavior of application programs. A large body of literature has evolved on the subjects of detection, confinement, and recovery from abnormal behavior of single-version software. Modularity, system closure, atomicity of actions, decision verification, and exception handling are among the key attributes of reliable single-version applications and system software. It is quite evident that these attributes remain desirable and advantageous in each version of the fault-tolerant multiversion software that will be discussed in the remainder of this paper.

Multiple-version software has remained of little interest to the mainstream researchers and developers of software for a relatively long time. Some suggestions of its potential usefulness had appeared in the early and mid-1970's [17], [21], [36], [19]. However, the first suggestion on record was made by Dr. Dionysius Lardner who, in 1834, wrote in"Babbage's calculating engine" as follows [42]:

> *"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."*

The first long-term systematic investigation of multiple-version software for fault tolerance was initiated by Brian Randell at the University of Newcastle upon Tyne in the early 1970's [47], [1]. From this research evolved the *recovery block* (RB) technique, in which alternate software versions are organized in a manner similar to the dynamic redundancy (standby sparing) technique in hardware. Its objective is to perform run-time software (as well as hardware) fault detection by an acceptance test performed on the results delivered by the first version. If the acceptance test is not passed, recovery is implemented by state restoration followed by the execution of an alternate version on the same hardware (*NT/1H/NdS*). Recovery is considered complete when the acceptance test is passed. Several variations of the RB approach have been recently investigated [3], [11], [25], [30], [35], [50].

Another continuing investigation of multiversion software is the N-version programming (NVP) project started by the author at UCLA in 1975 [4]. N-fold computation (*1T/NH/NdS*, or *1T/NdH/NdS*) is carried out by using N independently designed software modules, or "versions," and their results are sent to a decision algorithm that determines a single decision result [5]. Other pioneering investigations of the multiversion software concept have been carried out at the KFK Karlsruhe, FRG [22], at the Halden Reactor Project, Norway [13], and at UC Berkeley and other organizations in the United States [46].

The fundamental difference between the RB and NVP approaches is the decision algorithm. In the RB approach, an acceptance test that is specific to the application program being implemented must be provided. In the NVP approach, the decision algorithm is a generic consensus algorithm that delivers an agreement/disagreement decision. Otherwise, both RB and NVP can be implemented for concurrent or sequential execution depending on the

number of hardware channels available at once. Analytic models using queuing and Markov modeling techniques have been developed for the comparison of RB and NVP techniques with respect to time requirements and reliability improvement, allowing for the existence of similar errors in NVP and imperfect acceptance tests in RB [26], [37].

A number of variations and combinations of the RB and NVP approaches can be readily identified [2]. First, acceptance tests on individual versions can be employed to support the decision algorithms in NVP [33]. Second, fault detection and exception handling algorithms in individual hardware channels can help to distinguish physical faults from design faults. Third, the RB approach can concurrently use two (or more) hardware channels, either copied: (*N*/2) *T*/2*H*/*NdS*, or diverse: (*N*/2) *T*/2*dH*/*NdS*. A decision algorithm can support the acceptance tests, since two sets of results are made available in parallel, and real-time constraints can be met as long as one channel remains acceptable. The general diverse system is *NT*/*YdH*/*ZdS* with $NY \geq Z$, that includes acceptance tests, exception handling, and the detection of physical faults in each channel to support the decision algorithm. Other diverse systems result when some features are deleted or diversity is reduced.

IV. *N*-Version Programming Experiments at UCLA

N-version programming is defined as the independent generation of $N \geq 2$ software modules, called "versions," from the same initial specification [5]. "Independent generation" means that programming efforts are carried out by individuals or groups that do not interact with respect to the programming process. Wherever possible, different algorithms, programming languages, environments, and tools are used in each separate effort. The goal of NVP is to minimize the probability of similar errors at decision points in an *N*-fold computation.

The purpose of the initial specification is to state the functional requirements completely and unambiguously, while leaving the widest possible choice of implementations to the *N* programming efforts. The specification also prescribes the special features that are needed to execute the set of *N* versions as an "*N*-version software unit" in a fault-tolerant manner. An initial specification defines: 1) the function to be implemented by the *N*-version software unit; 2) the cross-check points ("cc-points") at which the decision algorithm will be applied to the results of each version; 3) content and format of the cross-check vectors ("cc-vectors") to be generated at each cc-point; 4) the decision algorithm to be used at each cc-point; and 5) the response to the possible outcomes of decisions. The decision algorithm explicitly states the allowable range of variation in numerical results, if such a range exists, as well as any other acceptable differences in the version results such as extra spaces in text output or other "cosmetic" variations.

It is the fundamental conjecture of the NVP approach that the independence of programming efforts will assure a low probability that residual software design faults will lead to an erroneous decision by causing similar errors to occur at the same cc-point in two or more versions. Given a reasonable choice of cc-points and cc-vectors, the low probability of similar errors is expected to make *N*-version programming an effective method for achieving software fault tolerance. The effectiveness of the NVP approach depends on the validity of the conjecture, and an experimental investigation was deemed to be the essential next step. The initial NVP research effort at UCLA (1975–1978) addressed two questions: 1) which requirements (e.g., good specifications, choice of suitable types of problems, constraints on the nature of algorithms, timing constraints, decision algorithms, etc.) have to be met to make *N*-version programming feasible at all regardless of the cost; and 2) what methods should be used to compare the cost and the effectiveness of the NVP approach to the two alternatives: single-version programming and the recovery block approach.

The scarcity of previous results and an absence of formal theories on *N*-version programming led to the choice of an experimental approach: to choose some conveniently accessible programming problems, to assess the applicability of *N*-version programming, and then to proceed to generate a set of programs. Once generated, the programs were executed as *N*-version software units in a simulated multiple-hardware system, and the resulting observations were applied to refine the methodology and to build up the concepts of *N*-version programming. The first detailed review of the research approach and a discussion of two sets of experimental results, using 27 and 16 independently written programs, was published in 1978 [12].

The preceding exploratory research demonstrated the practicality of experimental investigation and confirmed the need for high quality software specifications. As a consequence, the first aim of the next phase of UCLA research (1979–1982) was the investigation of the relative applicability of various software specification techniques. Other aims were to investigate the types and causes of software design faults, to propose improvements to software specification techniques and their use, and to propose future experiments for the investigation of design fault tolerance in software and in hardware [33].

To examine the effect of specification techniques on multiversion software, an experiment was designed in which three different specifications were used. The first was written in the formal specification language OBJ [23]. The second specification language chosen was the nonformal PDL that was characteristic of current industry practice. English was employed as the third, or "control," specification language since English had been used in the previous studies [12]. Thirty programmers started the programming effort, and eighteen programs were delivered (seven from OBJ, five from PDL, and six from the English specification). The length of the programs varied from 217 to 689 PL/1 statements, averaging 392 statements per program.

The problem chosen for the experiment was an "airport

scheduler" program. This database problem concerns the operation of an airport in which flights are scheduled to depart for other airports and seats are reserved on those flights. The problem was originally an example of database system specification [15] and was later used to illustrate the use of OBJ [23]. Because the problem is transaction oriented, the natural choice of cross-check points was at the end of each transaction. With the OBJ specification as reference, a specification was written in PDL and another one in English. The OBJ specification was 13 pages long, the English specification took 10 pages, but PDL required 74 pages to contain the same specification [32]. The detailed description of the experiment has been reported in [32], and the main results have been presented in [33] and [7].

A major second generation experiment began in June of 1985. UCLA is cooperating with the University of Illinois, the University of Virginia, and North Carolina State University in a large-scale experiment sponsored by the NASA Langley Research Center. Twenty versions of a program to manage sensor redundancy in an aircraft and spacecraft navigation system are to be written by September of 1985. Hypotheses on similar errors have been formulated and will be validated, and the cost effectiveness and reliability increase of the *N*-version approach will be assessed. To establish a long-term research facility for these second generation experimental investigations, the DEsign DIversity eXperiment system (DEDIX), a distributed software supervisor and testbed system at the UCLA Center for Experimental Computer Science, has been designed and implemented [8]. A subsequent section of this paper describes the requirements, design, and implementation of DEDIX.

V. Principal Issues of *N*-Version Programming

The series of *N*-version programming experiments that were conducted at UCLA allowed us to identify several issues that require resolution in order to attain successful *N*-version fault tolerance in software. The key problems of implementing the multiversion software solution are discussed below.

Initial Specification

The most critical condition for the independence of design faults is the existence of a complete and accurate specification of the requirements that are to be met by the diverse designs. This is the "hard core" of this fault tolerance approach. Latent defects, such as inconsistencies, ambiguities, and omissions, in the specification are likely to bias otherwise entirely independent programming or logic design efforts toward related design faults. The most promising approach to the production of the initial specification is the use of formal, very-high-level specification languages [38], [44], [39]. When such specifications are executable, they can be automatically tested for latent defects and serve as prototypes of the programs suitable for assessing the overall design. With this approach, perfection is required only at the highest level of specification; the rest of the design and implementation process as well as its tools are not required to be perfect, but only as good as possible within existing resource constraints and time limits. The independent writing and subsequent comparison of two specifications, using two formal languages, is the next step that is expected to increase the dependability of specifications beyond the present limits. Our current investigation of specification methods is discussed in a subsequent section.

Independence of Design Efforts

The approach that is employed to attain independence of design faults in a set of *N* programs is maximal independence of design and implementation efforts. It calls for the use of diverse algorithms, programming languages, compilers, design tools, implementation techniques, test methods, etc. The second condition for independence is the employment of independent (noninteracting) programmers or designers, with diversity in their training and experience. Wide geographical dispersion and diverse ethnic backgrounds may also be desirable.

N-Version Execution Support

Implementation of *N*-version fault-tolerant software requires special support mechanisms that need to be specified, implemented, and protected against failures due to physical or design faults. These mechanisms fall into two categories: those *specific* for the application program being implemented, and those that are *generic* for the *N*-version approach. The *application-specific* class contains the specifications of: 1) the initial state of the program; 2) the inputs to be received; 3) the location of cross-check points (partitioning into modules); 4) the content and format of the cross-check vector at each cc-point (outputs are included here); 5) the algorithms for internal checking and exception handling within each version; and 6) the time constraints to be observed by each program module.

The *generic* class of support mechanisms forms the *N*-version execution support environment that includes: 1) the decision algorithm; 2) assurance of input consistency; 3) interversion communication; 4) version synchronization and enforcement of timing constraints; 5) local supervision for each version; 6) the global executive and decision function for the treatment of faulty versions; and 7) the user interface for observation, debugging, injection of stimuli, and data collection during *N*-version execution of application programs. The nature of the generic support mechanisms is illustrated in the discussion of the DEDIX *N*-version supervisor and testbed system that is described in the next section.

Protection of the Support Environment

The success of design fault tolerance by means of *N*-version software depends on uninterrupted and fault-free service by the *N*-version support environment. Protection against physical faults is provided by the physical distribution of *N* versions on separate machines and by the implementation of fault-tolerant communication linkages.

The SIFT system [24] and the DEDIX system [8] are suitable examples in which the global executive is also protected by N-fold replication. The remaining problem is the protection against design faults that may exist in the support environment itself. This may be accomplished by N-fold diverse implementation of the support environment. To explore the feasibility of this approach, the prototype DEDIX environment is currently undergoing formal specification. Subsequently, this specification will be used to generate diverse multiple versions of the DEDIX software to reside on separate physical nodes of the system. The practicality and efficiency of the approach remain to be determined.

Architectural Support

Current hardware architectures were not conceived with the goal of N-version execution; therefore, they lack supporting instructions and other features that would make N-version software execution efficient. For example, the special instructions "take majority vote" and "check input consistency" would be very useful. The practical applicability on N-version software in safety-critical real-time applications hinges on the evolution of custom-tailored instruction sets and supporting architectures.

Recovery of Failed Versions

A problem area that needs to be addressed is the recovery of a failed version at a cc-point in order to allow its continued participation in N-version execution. Since all versions are likely to contain design faults, it is critically important to recover versions as they fail rather than merely degrade to N-1 versions, then N-2 versions, and so on to shutdown. Recovery of a given version is difficult because the other (good) versions are not likely to have identical internal states; they may differ drastically in internal structure while satisfying the specification.

Modification of N-Version Software

It is evident that the modification of software that exists in multiple versions is more difficult. The specification is expected to be sufficiently modular so that a given modification will affect only a few modules. The extent to which each module is affected can then be used to determine whether the existing versions should be modified according to a specification of change, or the existing versions should be discarded and new versions generated from the appropriately modified specification. Experiments need to be conducted to gain insights into the criteria to be used for a choice.

Assessment of Effectiveness

The usefulness of the N-version approach depends 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. Large-scale experiments need to be carried out in order to gain statistically valid evidence, and the "mail order software" approach offers significant promise. In an "international mail order" experiment, the members of fault tolerance research groups from several countries will use a formal specification to write software versions. It is expected that the software versions produced at widely separated locations, by programmers with different training and experience who use different programming languages, will contain substantial design diversity. In further experiments, it may be possible to utilize the rapidly growing population of free-lance programmers on a contractual basis to provide module versions at their own locations. This approach would avoid the need to concentrate programming specialists, have a low overhead cost, and readily allow for the withdrawal of individual programmers.

Cost Investigations

The generation of N versions of a given program instead of a single one shows an immediate increase in the cost of software prior to the verification and validation phase. The question is whether the subsequent cost will be reduced because of the ability to employ two (or more) versions to attain mutual validation under operational conditions. Cost advantages may accrue because of 1) the faster operational deployment of new software; and 2) replacement of costly verification and validation tools and operations by a generic N-version environment in which the versions validate each other.

VI. The DEDIX System: An N-Version Research Tool

In the course of previous experiments at UCLA, it became evident that the usual general-purpose computing services were poorly suited to support the systematic execution, instrumentation, and observation of N-version fault-tolerant software. In order to provide a long-term research facility for experimental investigations of design diversity as a means of achieving fault-tolerant systems, the UCLA Dependable Computing and Fault-Tolerant System research group has designed and implemented the prototype DEDIX (DEsign DIversity eXperiment) system [8], a distributed supervisor and testbed for multiple-version software, at the UCLA Center for Experimental Computer Science. DEDIX is supported by the Center's Olympus Net local network, which utilizes the UNIX-based LOCUS distributed operating system [45] to operate a set of 20 VAX 11/750 computers. The prototype DEDIX system is discussed in this section. DEDIX illustrates the solutions that were chosen for the N-version software implementation problems identified in the preceding section. The three following general requirements were established for DEDIX.

- *Transparency:* The application programmers are not required to take care of the multiplicity, and a version must be able to run in a system with any allowed value of N without modifications.
- *Distribution:* The versions should be able to run on separate physical sites of a network in order to take advantage of physical isolation between sites, to benefit from parallel execution, and to tolerate the crash of a site.

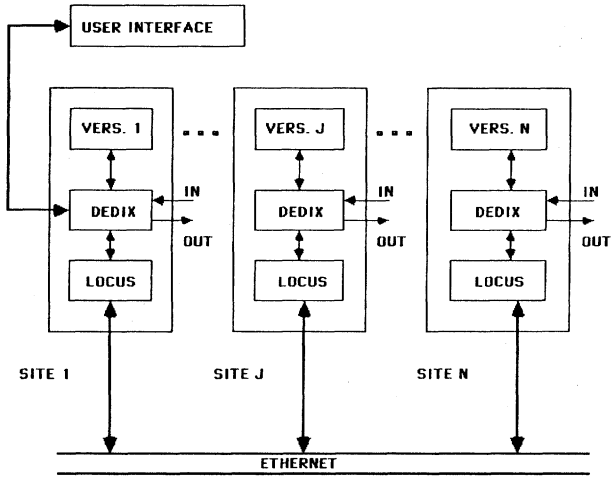

Fig. 1. DEDIX at N sites of Olympus Net.

• *Environment:* DEDIX is designed to run on the distributed LOCUS environment at UCLA and must be portable to other UNIX environments. DEDIX must be able to run concurrently with all other normal activities of the local network.

The DEDIX structure can be considered as a network-based generalization of SIFT [24] that is able to tolerate both hardware and software faults. Both have similar partitioning, with a local executive and a decision algorithm at each site that processes broadcast results, and a copy of the global executive at each site that takes consistent reconfiguration decisions by majority vote. DEDIX is extended to allow some diversity in results and in version execution times. SIFT is a frame synchronous system that uses periodically synchronized clocks to predict when results should be available for a decision. This technique does not allow the diversity in execution times and unpredictable delays in communication that can be found in a distributed N-version environment. Instead, a synchronization protocol is used in DEDIX which does not make reference to global time within the system.

The purpose of DEDIX is to supervise and to observe the execution of N diverse versions of an application program functioning as a fault-tolerant N-version software unit. DEDIX also provides a transparent interface to the users, versions, and the input/output system so that they need not be aware of the existence of multiple versions and recovery algorithms. An abstract view of DEDIX in an N-site environment is given in Fig. 1. In summary, DEDIX provides the following services:

• it handles communications from the user and distributes them to all active versions;

• it handles requests from the versions to have their results (cc-vectors) processed, and returns decision results to the versions and to the user;

• it executes decision algorithms and determines decision results;

• it manages the input and output operations for the versions; and

• it makes reconfiguration and recovery decisions about the handling of faulty versions.

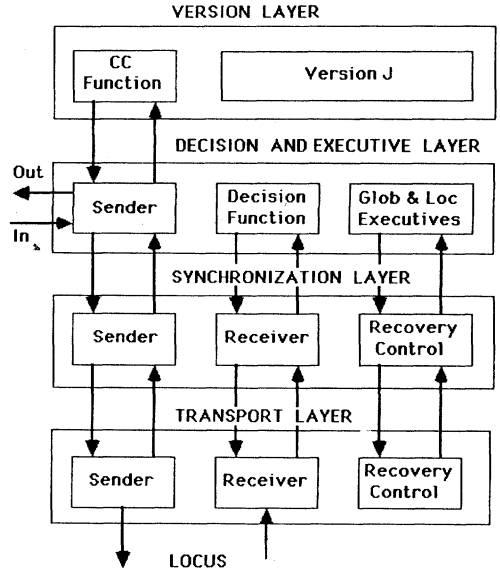

Fig. 2. Layers and entities at one site of DEDIX.

The DEDIX system can be located either in a single computer that executes all versions sequentially, or in a multicomputer system running one or more versions at each site. If DEDIX is supported by a single computer, it is vulnerable to hardware and software faults that affect the host computer, and the execution of N-version software units is relatively slow. In a computer network environment, the system is partitioned to protect it against most hardware faults. This has been done by providing each site with its own local DEDIX software, while an intersite communication facility is common to all computers. The DEDIX design is suitable for any specified number $N \geq 2$ of sites and versions, and currently accommodates the range $2 \leq N \leq 20$, with provision to reduce the number of sites and to adjust the decision algorithm upon the failure of a version or a site.

The prototpye DEDIX has been designed as a hierarchy of four layers to reduce complexity and to facilitate the inevitable modifications. Each site has an identical set of layers and entities as shown in Fig. 2. The purpose of each layer is to offer services to the higher layers, shielding them from details on how those services actually are implemented. The layers, discussed below, are the: *transport layer*, *synchronization layer*, *decision and executive layer*, and *version layer*.

The transport layer handles the communication of messages between the sites. It serves as the interface of DEDIX and the host LOCUS distributed operating system that manages intersite communication. The layer is intended to assure that messages are not lost, duplicated, damaged, or misaddressed, and it preserves the ordering of sent messages. The loss of communication with another site is reported to the layer above.

The synchronization layer at each site has a *sender* that broadcasts local results, and a *receiver* that collects messages with the results (cc-vectors) from other sites, using the services of the transport layer. The collected results are delivered to the decision function. The execution of

versions at all sites needs to be event-synchronized to ensure that only results from corresponding cc-points are matched. A synchronization protocol has been designed to provide the service [27]. It ensures that the results that are supplied to the decision function are from the same cross-check (cc) point in each version. The versions are halted until all of them have reached the same cc-point, and they are not started again until the results are exchanged and a decision is made. To be able to detect versions that are "hung up" and to allow slow versions to catch up, a time-out technique is used by the protocol. Acknowledgments are employed to verify that each site has received enough results to make a decision.

The decision and executive (D&E) layer receives the cc-vectors from its local version directly and from all other versions via the synchronization layer, determines the decision result, establishes whether the local version is faulty or not, and makes recovery decisions. The D&E layer controls the input/output of the local version, and all exceptions that are not handled elsewhere are directed to this layer. This layer has four entities: a *sender*, a *decision function*, and two entities that control the recovery process—a *local executive* and a *global executive*. The *sender* receives the cc-vector result from the local version and returns the decision result to it after a decision has been taken. It also executes the disconnection of the local version, controls its input and output operations, and handles communications with both executives.

The decision function determines a single decision result from the N results of individual versions. DEDIX provides a generic decision algorithm which may be replaced by a user's algorithm provided that the interfaces are preserved. This allows application-specific decision algorithms to be incorporated in those cases where the standard mechanisms are inappropriate. The generic decision algorithm determines a decision by applying one of three agreement tests: 1) *bit by bit*— identical match only; 2) *cosmetic*—detecting character string differences caused by spacing, misspelling, or character substitution; and 3) *numeric*—integer and real number decisions. Numeric decisions use a median value such that, as long as the majority of versions are not faulty, the median of all responses is acceptably close to the desired value. Numeric values are allowed to differ within some "skew interval," thus allowing version results to be nonidentical but still useable in the numeric decision process.

The local executive takes action when the decision function indicates that the decision result is not unanimous, or when any other exception is signaled from the local version or another layer. The local executive will first try to recover locally before it either reports the problem to the global executive or, if it is considered as fatal to the site, closes down the site. Three classes of exceptions can occur. *Functional exceptions* are specified in the functional description of DEDIX and are independent of the implementation. Among them are deviations from a unanimous result, the case when a communication link is disconnected, and the case when a cc-vector is completely missing. For these exceptions, the local executive will attempt to keep the site active, possibly terminating the local version, while keeping the input/output operating. *Implementation exceptions* are dependent on the specific computer system, language, and implementation technique chosen. All UNIX signals, such as segmentation faults, process termination, invalid system call, etc. belong to this class. Other examples are all the exceptions defined in DEDIX, such as signaling when a function is called with an invalid parameter or when an inconsistent state exists. Most of those exceptions will force an orderly closedown of a site in order to be able to provide data for analysis. Finally, there are *exceptions generated by the local version.* The local version program may include provisions for exception handling and some of the exceptions may not be recoverable within the version. These exceptions are sent to the local executive which will terminate the local version while keeping the site alive.

The global executive collects error reports from the decision function and the local executive, exchanges error reports with every other active global executive, and decides on a new configuration based on all error reports. The global executive is invoked after a preset number of exchanges of results (i.e., number of decisions) has taken place. The number of exchanges is maintained as a consistent value at all sites. Thus, by referring to this number, it is possible to ensure that all correctly working sites will exchange error reports and decide on a reconfiguration at the same state of computation. This number is kept consistent by the synchronization protocol.

The version layer interfaces the Jth (local) version with the DEDIX system and alters the variables in the local cc-vector that disagree with the decision result produced by the decision function. The function doing the interfacing is called the cross-check, or CC, function since it is called at each cc-point. Pointers to the results to be corrected are sent as parameters to this function. The CC function transfers the version representation of results into the DEDIX cc-vector format, so that the internal representation of a cc-vector in DEDIX is hidden to the version program. The CC function also returns the decision result to the version.

The user interface of DEDIX allows users to debug the system as well as the versions, to monitor the operations of the system, to apply stimuli to the system, and to collect data during experiments. Several commands are provided. The *break* command enables the user to set breakpoints. At a breakpoint, DEDIX stops executing the versions and allows entering of commands. The *remove* command deletes breakpoints set by the break command. The *continue* command resumes execution of the versions at a breakpoint. The user may terminate execution using the *quit* command. The user can examine the current contents of the message passing through the transport layer by using the *display* command. Since every message is logged, the user may also specify conditions in the *display* command to examine any message logged previously. The user can also examine the internal system states by using the *show*

command, e.g., to examine the breakpoints which have been set, the decision results, etc. The user can inject faults to the system by changing the system states, e.g., the cc-vector, by using the *modify* command. The user interface gathers data and collects statistics of the experiments. Every message passing the transport layer is logged into a file with a time stamp. This enables the user to do postexecution analysis or even replay the experiment. Statistics such as elapsed time, system time, number of cc-points executed, and their decision outcomes are also collected.

The prototype DEDIX system has been operational since early 1985. Several modifications have been introduced since then—most of them intended to improve the speed of the execution of *N*-version software. The first major test of DEDIX will be experimentation with the set of about 20 programs produced by the NASA-sponsored four-university project discussed in Section IV. At the same time, a formal specification effort for DEDIX is being initiated as described in the following section.

VII. Directions of Current Research

The *N*-version software research at UCLA has the following two major long-term objectives:

1) to develop the principles of implementation and experimental evaluation of fault-tolerant *N*-version software units; and

2) to devise and evaluate supervisory systems for the execution of *N*-version software in various environments.

The *N*-version implementation studies address the problems of: 1) methods of specification; 2) the verification of specifications; 3) the assurance of independence in designs; 4) partitioning and matching, i.e., good choices of cc-points and cc-vectors for a given problem; 5) the means to recover a failed version; 6) efficient methods of modification for *N*-version units; 7) evaluation of effectiveness and of cost; and 8) the design of experiments.

The research concerned with the supervisory systems deals with: 1) the functional structure of supervisors; 2) fault-tolerant supervisor implementation, including tolerance of design faults; 3) instrumentation to support *N*-version software experiments; 4) efficient implementation, including custom hardware architectures to support real-time execution; and 5) methods of supervisor evaluation.

Our past experience has pinpointed an effective specification as the keystone of success for *N*-version software implementation [7]. Significant progress has occurred in the development of formal specification languages since our previous experiments. Our current goal is to compare and assess the ease of use by application programmers of several formal program specification methods, including the OBJ specification language developed at UCLA [23] and used in our previous experiment; the Clear specification language developed at the University of Edinburgh and SRI International [10]; the Larch family of specification languages developed at M.I.T. and the Xerox Research Center [28]; the Ina Jo™ specification language developed at SDC [34]; the Z specification language developed at Oxford University [29], and also Prolog and Concurrent Prolog as methods of formal specification.

The comparison focuses on several aspects of applicability: 1) the purpose and scope (problem domain); 2) completeness of development; 3) quality and extent of documentation; 4) existence of support environments; 5) executability and suitability for rapid prototyping; 6) provisions to express timing constraints and concurrency; 7) methods of specification for exception handling; 8) extensibility to specify the special attributes of fault-tolerant multiversion software. The goal is to choose two or more specification languages for an experiment in the design of fault-tolerant multiversion software. The experiment has two major elements: first, it is an attempt to attain concurrent verification of two specifications by symbolic execution with mutual interplay, that is, a "two-version" specification; and second, it provides an assessment of the ease of use of the specifications by application programmers in an *N*-version software experiment. It is conjectured that the application of two diverse formal specifications will help to eliminate residual specification faults and further increase the dependability of the specifications.

The next step in the development of DEDIX is the formal specification of parts of the current DEDIX prototype that are implemented in C, beginning with the synchronization layer, the decision function, and the local and global executives. The specification is intended to provide an executable version of the DEDIX supervisory operating system. This functional specification is expected to provide a starting point for a implementation for real-time systems. Furthermore, the specification can be independently reimplemented in C for the use of multiversion software techniques in the implementation of different sites of DEDIX. The goal is a DEDIX system that supports *N*-version application programs and which is itself diverse in its design at each site.

In the experimental evaluation of the *N*-version approach, the immediate goal is extensive experimentation with about 20 programs to be generated during the summer of 1985 by the four-university effort discussed in Section IV. For the next step, plans are being prepared for the "international mail order" experiment as discussed in Section V of this paper.

VIII. Conclusion: Some Potential Long-Range Advantages of Design Diversity

The most immediate and direct opportunity to apply *N*-version software is offerred by multiple-channel systems that incorporate very complete tolerance of physical faults such as SIFT [24]. The hardware resources and some architectural features that can support *N*-version software are already present, and the implementation requires an extension of the existing physical fault tolerance mechanisms. Furthermore, hardware diversity can be introduced by choosing functionally compatible hardware building blocks from different suppliers for each channel.

A more speculative, and also more general, application of *N*-version software is its partial substitution for current

software verification and validation (V&V) procedures. Instead of extensive preoperational V&V of a single program, two independent versions can be executed in an operational environment, completing V&V concurrently with productive operation. The increased cost of producing the software is compensated for by a reduction of the V&V time and an earlier deployment. Another advantage may be a decrease in the amount of human effort and complexity of software tools needed for the very thorough V&V effort. Design faults in the V&V software are also less critical. The user can take the less efficient ("backup") version off line when adequate reliability of operation is reached, and then bring it back for special operating conditions that require greater reliability assurance, especially after modifications or after maintenance. A potential system lifetime cost reduction exists because a duplex diverse system (possibly operating in the recovery block mode) can support continued operation after latent design faults are uncovered, providing very high availability. The cost of fault analysis and elimination also might be reduced due to the lesser urgency of the repair actions.

The possible use of a "mail-order" approach to the production of two or more versions of software modules suggests an intriguing long-range benefit of the *N*-version approach in software. Given a formal specification that includes a set of single-version acceptance tests, the versions of software can be written by programmers working at their own preferred times and locations, and using their own personal computing equipment. Two potential advantages are as follows.

- The overhead cost of programming that accrues in highly controlled professional programming environments could be drastically reduced through this approach that allows free play to individual initiative and utilizes low-cost home facilities.
- The potential of the rapidly growing number of freelance programmers to serve as "mail-order" programmers would be tapped through this approach. For various reasons, many individuals with programming talents cannot fill the role of a professional programmer as defined by today's rigorous approaches to quality control and the use of centralized sites during the programming process, but they may well succeed as independent programming contractors for *N*-version implementations.

Finally, an important reliability and availability advantage through design diversity may be expected for systems that use VLSI circuits. The growing complexity of VLSI circuits, with 500 000 gates/chip available today and 1 million gates/chip predicted for the near future, raises the probability of design faults since a complete verification of the design is very difficult to attain. Furthermore, the design automation and verification tools themselves are subject to undiscovered design faults. Even with multichannel fault-tolerant system designs, a single design fault may require the replacement of all chips of that type since on-chip modifications are impractical. Such a replacement would be a costly and time-consuming process. On the other hand, use of *N* versions of VLSI circuits in a multiple-channel design does allow the continued use of chips with design faults, as long as their errors are not similar at the circuit boundaries where the decision algorithm is applied.

Design diversity may enable dependable operation throughout the lifetime of a multiple-channel system without a single chip having a perfect design, and without any single perfect program executing on those chips. The building of the first system of this kind will be a milestone in the evolution of fault-tolerant systems, and current results support the prediction that such systems should be in service by the year 2000.

Acknowledgment

Over the past several years, this research has been supported by the National Science Foundation under Grants MCS-72-03633, MCS-78-18918, and MCS 81-21696, by the Office of Naval Research under Contract N0014-79-C-0866, and by a research grant from the Battelle Memorial Institute. Current support is provided by a grant from the Advanced Computer Science Program of the Federal Aviation Administration. The major contributions to experimental research have been made by L. Chen and J. P. J. Kelly. The principal designers of the DEDIX system are P. Gunningberg, L. Strigini, P. Traverse, K.-S. Tso, and U. Voges. Thanks are due to T. Anderson and J.-C. Laprie for their thoughtful comments on this paper.

References

[1] T. Anderson and P. A. Lee, *Fault Tolerance, Principles and Practice*. Englewood Cliffs, NJ: Prentice-Hall, 1981, pp. 249-291.

[2] T. Anderson, "Can design faults be tolerated?" *Fehlertolerierende Rechensysteme* in *Proc. 2nd GI/NTG/GMR Conf.* Bonn, West Germany, Sept. 1984, IFB vol. 84, Springer-Verlag, pp. 426-433.

[3] T. Anderson, D. N. Halliwell, P. A. Barrett, and M. R. Moulding, "An evaluation of software fault tolerance in a practical system," in *Dig. 15th Ann. Int. Symp. Fault-Tolerant Comput.*, Ann Arbor, MI, June 19-21, 1985, pp. 140-145.

[4] A. Avižienis, "Fault tolerance and fault intolerance: Complementary approaches to reliable computing," in *Proc. 1975 Int. Conf. Rel. Software*, Los Angeles, CA, Apr. 21-23, 1975, pp. 458-464.

[5] A. Avižienis and L. Chen, "On the implementation of *N*-version programming for software fault tolerance during execution," in *Proc. COMPSAC 77, 1st IEEE-CS Int. Comput. Software Appl. Conf.*, Chicago, IL, Nov. 8-11, 1977, pp. 149-155.

[6] A. Avižienis, "Fault tolerance: The survival attribute of digital systems," *Proc. IEEE*, vol. 66, pp. 1109-1125, Oct. 1978.

[7] A. Avižienis and J. Kelly, "Fault tolerance by design diversity: Concepts and experiments," *Computer*, vol. 17, no. 8, pp. 67-80, Aug. 1984.

[8] A. Avizienis, P. Gunningberg, J. P. J. Kelly, L. Strigini, P. J. Traverse, K. S. Tso, and U. Voges, "The UCLA DEDIX system: A distributed testbed for multiple-version software," in *Dig. 15th Annu. Int. Symp. Fault-Tolerant Comput.*, Ann Arbor, MI, June 19-21, 1985, pp. 126-134.

[9] J. F. Bartlett, "A 'NonStop' operating system," in *Proc. Hawaii Int. Conf. Syst. Sci.*, Honolulu, HI, Jan. 5-6, 1978, pp. 103-119; reprinted in D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable System Design*. Bedford, MA: Digital Press, 1982, pp. 453-460.

[10] R. M. Burstall and J. A. Goguen, "An informal introduction to specifications using CLEAR," in *The Correctness Problem in Computer Science*, R. Boyer and H. Moore, Eds. New York: Academic, 1981, pp. 185-213.

[11] R. H. Campbell, K. H. Horton, and G. G. Belford, "Simulations of a fault-tolerant deadline mechanisms," in *Dig. 9th Annu. Int. Symp. Fault-Tolerant Comput.*, Madison, WI, June 1979, pp. 95-101.

[12] L. Chen and A. Avižienis, "*N*-version programming: A fault tolerance approach to reliability of software operation," in *Dig. 8th Annu. Int. Conf. Fault-Tolerant Comput.* (FTCS-8), Toulouse, France, June 21–23, 1978, pp. 3–9.

[13] G. Dahll and J. Lahti, "Investigation of methods for production and verification of highly reliable software," in *Proc. IFAC Workshop SAFECOMP 1979*, Stuttgart, Germany, May 16–18, 1979.

[14] R. W. Downing, J. S. Nowak, and L. S. Tuomenoksa, "No. 1 ESS maintenance plan," *Bell Syst. Tech. J.*, vol. 43, no. 5, pt. 1, pp. 1961–2019, Sept. 1964.

[15] H. Ehrig, H. Kreowski, and H. Weber, "Algebraic specification schemes for data base systems," in *Proc. 4th Int. Conf. Very Large Data Bases*, West Berlin, Germany, Sept. 13–15, 1978, pp. 427–440.

[16] "Information processing systems—Reliability and requirements," in *Proc. East. Joint Comput. Conf.*, Washington, DC, Dec. 8–10, 1953.

[17] W. R. Elmendorf, "Fault-tolerant programming," in *Proc. 1972 Int. Symp. Fault-Tolerant Comput.*, Newton, MA, June 19–21, 1972, pp. 79–83.

[18] R. R. Everett, C. A. Zraket, and H. D. Benington, "SAGE-a data-processing system for air defense," in *Proc. East. Joint Comput. Conf.*, Washington, DC, Dec. 1957, pp. 148–155.

[19] M. A. Fischler, O. Firshein, and D. L. Drew, "Distinct software: An approach to reliable computing," in *Proc. 2nd USA-Japan Comput. Conf.*, Tokyo, Japan, Aug. 26–28, 1975, pp. 573–579.

[20] G. C. Gilley, Ed., *Dig. 1971 Int. Symp. Fault-Tolerant Comput.*, Pasadena, CA, Mar. 1–3, 1971.

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

[22] L. Gmeiner and U. Voges, "Software diversity in reactor protection systems: An experiment," in *Proc. IFAC Workshop SAFECOMP 1979*, Stuttgart, Germany, May 16–18, 1979, pp. 75–79.

[23] J. A. Goguen and J. J. Tardo, "An introduction to OBJ: A language for writing and testing formal algebraic program specifications," in *Proc. Specific. Rel. Software*, Cambridge, MA, Apr. 3–5, 1979, pp. 170–189.

[24] J. Goldberg, "SIFT: A provable fault-tolerant computer for aircraft flight control," in *Inform. Processing 80 Proc. IFIP Congr.*, Tokyo, Japan, Oct. 6–9, 1980, pp. 151–156.

[25] S. T. Gregory and J. C. Knight, "A new linguistic approach to backward error recovery," in *Dig. 15th Annu. Int. Symp. Fault-Tolerant Comput.*, Ann Arbor, MI, June 19–21, 1985, pp. 404–409.

[26] A. Grnarov, J. Arlat, and A. Avižienis, "On the performance of software fault tolerance strategies," in *Dig. 10th Int. Symp. Fault-Tolerant Comput.*, FTCS-10, Kyoto, Japan, Oct. 1–3, 1980, pp. 251–253.

[27] P. Gunningberg and B. Pehrson, "Protocol and verification of a synchronization protocol for comparison of results," in *Dig. 15th Annu. Int. Symp. Fault-Tolerant Comput.*, Ann Arbor, MI, June 19–21, 1985, pp. 172–177.

[28] J. V. Guttag and J. J. Horning, "An introduction to the Larch shared language," in *Inform. Processing '83 Proc. IFIP Congr.*, Paris, France, Sept. 19–23, 1983, pp. 809–814.

[29] I. J. Hayes, "Applying formal specification to software development in industry," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 169–178, Feb. 1985.

[30] H. Hecht, "Fault-tolerant software," *IEEE Trans. Rel.*, vol. R-28, pp. 227–232, Aug. 1979.

[31] A. L. Hopkins, Jr., T. B. Smith, III, and J. H. Lala, "FTMP—A highly reliable fault-tolerant multiprocessor for aircraft," in *Proc. IEEE*, vol. 66, pp. 1221–1239, Oct. 1978.

[32] J. P. J. Kelly, "Specification of fault-tolerant multi-version software: Experimental studies of a design diversity approach, Dep. Comput. Sci., Univ. California, Los Angeles, Tech. Rep. CSD-820927, Sept. 1982.

[33] J. P. J. Kelly and A. Avižienis, "A specification-oriented multi-version software experiment," in *Dig. 13th Annu. Int. Symp. Fault-Tolerant Comput.* (FTCS-13), Milano, Italy, June 28–30, 1983, pp. 120–126.

[34] R. A. Kemmerer, "Testing formal specifications to detect design errors," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 32–43, Jan. 1985.

[35] K. H. Kim, "Distributed execution of recovery blocks: Approach to uniform treatment of hardware and software faults," in *Proc. IEEE 4th Int. Conf. Distributed Comput. Syst.*, San Francisco, CA, May 14–18, 1984, pp. 526–532.

[36] H. Kopetz, "Software redundancy in real time systems," in *Inform. Processing 74 Proc. IFIP Congr.*, Stockholm, Sweden, Aug. 5–10, 1974, pp. 182–186.

[37] J.-C. Laprie, "Dependability evaluation of software systems in operation," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 701–714, Nov. 1984.

[38] B. H. Liskov and V. Berzins, "An appraisal of program specifications," in *Research Directions in Software Technology*, P. Wegner, Ed. Cambridge, MA: M.I.T. Press, 1979, pp. 170–189.

[39] P. M. Melliar-Smith, "System specifications," in *Computing Systems Reliability*, T. Anderson and B. Randell, Eds. New York: Cambridge University Press, 1979, pp. 19–65.

[40] B. Meyer, "A system description method," in *Int. Workshop Models and Languages for Software Specification and Design*, B. G. Babb, II, and A. Mili, Eds. Orlando, FL, Mar. 1984, pp. 42–46.

[41] E. F. Moore and C. E. Shannon, "Reliable circuits using less reliable relays," *J. Franklin Inst.*, vol. 262, no. 9 and 10, pp. 191–208 and 281–297, Sept.–Oct. 1956.

[42] P. Morrison and E. Morrison, Eds., *Charles Babbage and His Calculating Engines.* New York: Dover, 1961, p. 177.

[43] J. Oblonsky, "A self-correcting computer," in *Digital Information Processors*, W. Hoffman, Ed. New York: Interscience, 1962, pp. 533–542.

[44] D. L. Parnas, "The role of program specification," in *Research Directions in Software Technology*, P. Wegner, Ed. Cambridge, MA: M.I.T. Press, 1979, pp. 364–370.

[45] G. Popek *et al.*, "LOCUS—A network transparent, high reliability distributed system," in *Proc. 8th Symp. Operating Syst. Principles*, Dec. 1981, pp. 169–177.

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

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

[48] J. R. Sklaroff, "Redundancy management technique for space shuttle computers," *IBM J. Res. Develop.*, vol. 20, pp. 20–28, Jan. 1976.

[49] J. Von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," in *Automata Studies*, C. E. Shannon and J. McCarthy, Eds. Princeton, NJ: Princeton University Press, Ann. Math. Studies, 1956, no. 34, pp. 43–98.

[50] H. O. Welch, "Distributed recovery block performance in a real-time control loop," in *Proc. Real-Time Syst. Symp.*, Arlington, VA, 1983, pp. 268–276.

Algirdas Avižienis (S'55–M'56–F'73) was born in Kaunas, Lithuania, in 1932. He received the B.S., M.S., and Ph.D. degrees, all in electrical engineering, from the University of Illinois, Urbana-Champaign, in 1954, 1955, and 1960, respectively.

He has recently completed three years of service as Chairman of the Department of Computer Science, University of California, Los Angeles, where he has served on the faculty since 1962. He also directs the UCLA Dependable Computing and Fault-Tolerant System research group that he established in 1972. Current projects of the group include design diversity for the tolerance of design faults, faults tolerance in distributed systems, and fault-tolerant supercomputer architectures. From 1956 to 1960 he was associated with the Digital Computer Laboratory, University of Illinois, as a Research Assistant and Fellow participating in the design of the Illiac II computer. From 1960 to 1973 he directed research on fault-tolerant spacecraft computers at the Jet Propulsion Laboratory, California Institute of Technology, Pasadena. This effort resulted in the construction and evaluation of the experimental JPL STAR (self-testing-and-repairing) computer, for which he received a U.S. Patent in 1970. A paper that described the JPL STAR computer won the Best Paper selection of the IEEE TRANSACTIONS ON COMPUTERS in 1971.

Dr. Avižienis was elected Fellow of IEEE for his pioneering work in fault-tolerant computing. and also received the NASA Apollo Achievement Award, the Honor Roll of the IEEE Computer Group, the AIAA Information Systems Award, the NASA Exceptional Service Medal, and the IEEE Computer Society Technical Achievement Award. As a member of the IEEE Computer Society, he founded and was the first Chairman of the Technical Committee on Fault-Tolerant Computing (1969–1973) as well as the General Chairman of the First International Symposium on Fault-Tolerant Computing in 1971. He also served for four years as a member of the Computer Society Governing Board. Since 1980 he has been the first Chairman of Working Group 10.4 on "Reliable Computing and Fault Tolerance" of the International Federation for Information Processing.