Dependable Computing: From Concepts to Design Diversity

ALGIRDAS AVIŽIENIS, FELLOW, IEEE, AND JEAN-CLAUDE LAPRIE, MEMBER, IEEE

Invited Paper

This paper is composed of two sections. The first provides a conceptual framework for expressing the attributes of what constitutes dependable and reliable computing: a) the impairments to dependability (faults, errors, and failures), b) the means for dependability (fault avoidance, tolerance, removal, and forecasting), and c) the measures of dependability (reliability, availability, safety). The second section focuses on one of the most challenging problems for dependable computing: coping with design faults.

INTRODUCTION

Achieving dependable computer systems is essentially a complexity problem: mastering the assemblage and interaction of larger and larger numbers of more and more complex components. "Design" is a central word in such a problem; on one hand, this design involves large teams which need to interact, exchange information, and thus need a set of common reference terms enabling to express the concepts they are dealing with; on the other, the more complex the system, the more design faults (in a broad sense, from specification to implementation) are likely to be introduced (and remain) in the system. Those design faults defeat the approaches intended to tolerate physical faults.

What precedes is a rationale for the two sections of this paper:

- basic concepts of dependable computing,
- design fault tolerance.

Those sections result from the elaboration on [41] and [12], respectively.

I. THE DEPENDABILITY CONCEPT

A. Motivation

This section is a contribution to the work undertaken within the "Reliable and Fault Tolerant Computing" scientific and technical community [1], [8], [17], [18], [38], [52], [56], in order to propose clear and widely acceptable definitions for some basic concepts.

Going backward in time, a milestone was the special session organized at the 12th International Symposium on Fault Tolerant Computing, devoted to the presentation of viewpoints elaborated in several places and institutions [2], [10], [37], [39], [42], [54].

The guidelines which have governed the presentation of this section can be summed up as follows:

- search for the minimum number of concepts enabling the dependability attributes to be expressed;
- use of terms which are identical to (whenever possible) or as close as possible to those generally used;
- emphasis on integration (as opposed to specialization) [30].

B. Basic Definitions and Associated Terminology

Dependability is that property of a computer system that allows *reliance to be justifiably placed on the service it delivers*. The **service** delivered by a system is its behavior *as it is perceived* by its user(s); a **user** is another system that interacts with the former.

A system **failure** occurs when the delivered service deviates from the specified service, where the service **specification** is an *agreed* description of the expected service. The failure occurred because the system was erroneous: an **error** is that part of the system state which is liable to lead to failure. The cause—in its phenomenological sense—of an error is a **fault**. An error is thus the manifestation of a fault *in the system*, and a failure is the effect of an error *on the service*.

Achieving a dependable computing system calls for the *combined* utilization of a set of methods which can be classed into:

- **fault avoidance**: how to prevent, *by construction*, fault occurrence;
- **fault tolerance**: how to provide, *by redundancy*, service complying with the specification in spite of faults having occurred or occurring;

Manuscript received January 29, 1986; revised March 10, 1986.

A. Avižienis is with the Computer Science Department, UCLA, Los Angeles, CA 90024, USA.

J-C. Laprie is with LAAS-CNRS, 31077 Toulouse-Cedex, France.

0018-9219/86/0500-0629$01.00 ©1986 IEEE

Authorized licensed use limited to: University of Massachusetts Amherst. Downloaded on June 03,2023 at 22:27:44 UTC from IEEE Xplore. Restrictions apply.

- **fault removal**: how to minimize, *by verification*, the presence of faults;
- **fault forecasting**: how to estimate, *by evaluation*, the presence, the occurrence, and the consequences of faults.

Fault avoidance and fault tolerance may be seen as constituting **dependability procurement**: how to *provide* the system with the ability to deliver the specified service; fault removal and fault forecasting may be seen as constituting **dependability validation**: how to *reach confidence* in the system's ability to deliver the specified service.

The life of a system is perceived by its users as an alternation between two states of the delivered service with respect to the specified service:

- **proper service**, where the service is delivered as specified;
- **improper service**, where the delivered service is different from the specified service.

The events which constitute the transitions between these two states are the **failure** and the **restoration** of service. Quantifying the alternation between delivery of proper and improper service leads to the two main measures of dependability:

- **reliability**: a measure of the *continuous* delivery of proper service (or, equivalently, of the *time to* failure) from a reference initial instant;
- **availability**: a measure of the delivery of the proper service *with respect to the alternation* of delivery of proper and improper service.

C. Comments

1) On the Introduction of Dependability as a Generic Concept: Why should another term be added to an already long list: reliability, availability, safety, security, etc.? The reasons are basically two-fold:

- to remedy the existing confusion between reliability in its general meaning (reliable system) and reliability as a mathematical quantity (system reliability);
- to show that reliability, maintainability, availability, safety, etc., are quantitative measures corresponding to distinct perceptions of the same attribute of a system: its dependability.

In regard to the term "dependability," it is noteworthy that from an etymological point of view, the term "reliability" would be more appropriate: ability to rely upon. Although dependability is synonymous with reliability, it brings in the notion of dependence at a second level. This may be felt as a negative connotation at first sight, when compared to the positive notion of trust as expressed by reliability, but it does highlight our society's ever increasing dependence upon sophisticated systems in general and especially upon computing systems. Moreover, "to rely" comes from the French "relier," itself from the Latin "religare," to bind back: re-, back and ligare, to fasten, to tie. The French word for reliability, "fiabilité," traces back to the 12th century, to the word "fiableté" whose meaning was "character of being trustworthy"; the Latin origin is "fidare," a popular verb meaning "to trust." Simultaneous consideration of the English and French origins for reliability thus leads to the (unsurprising) association of two key concepts: ties and trust.

Finally, it is interesting to note that viewing dependability as a more general concept than reliability, availability, etc., and embodying the latter terms, has already been attempted in the past (see e.g. [33]), although with less generality than here, since the goal was then to define a measure.

2) On the Notions of Service and its Specification, and of System: From its very definition, the service delivered by a system is clearly an abstraction of its behavior. It is noteworthy that this abstraction is highly dependent on the application that the computer system supports. An example of this dependence is the important role played in this abstraction by time: the time granularities of the system and of its user(s) are generally different.

Concerning specification, what is essential within the present context is that it is a description of the expected service which is *agreed upon* by two persons or corporate bodies: the system supplier (in a broad sense of the term: designer, builder, vendor, etc.) and its human user(s). This agreement means that the service specification can (at least) serve as a basis for adjudicating whether the delivered service is acceptable or not. What precedes does not mean that a service specification will not change once established. This would be simple ignorance of the facts of life, which imply *change*. The changes may be motivated by modifying the service expectation: modification of functionality, or correction of some undesired features such as deficiencies in the agreed specification. Once more, what is important is that the specification is (again) agreed upon. It must be stressed that such matters as environmental conditions, exposure time duration, observability, readiness, etc., can (and should) be captured by an appropriately stated specification.

Up to now, a system has been considered as a whole, emphasizing its externally perceived behavior; a definition of a system complying with this "black box" view is: an entity having interacted, interacting, or likely to interact with other entities, i.e., with other systems. The **behavior** is then simply what the system *does* [62]. What *enables it to do what it does* is the **structure** of the system or its **organization**. Adopting the spirit of [1], a system, from a structural ("glass box") viewpoint, is a set of components bound together in order to interact; a **component** is another system, etc. The recursion stops when a **system** is considered as being atomic: any further internal structure cannot be discerned, or is not of interest and can be ignored. The term "component" has to be understood in a broad sense: layers of a computing system as well as intralayer components; in addition, a component being itself a system, it embodies the interrelation(s) of the components of which it is composed.

Based on the preceding view of system structure, the notions of service and specification apply equally naturally to the components. This is especially interesting in the design process, when off-the-shelf components, either hardware or software ("reusable" software) are used: what is of actual interest is the service they are able to provide, not their detailed (internal) behavior.

3) On the Notions of Fault, Error, and Failure: The faults

Authorized licensed use limited to: University of Massachusetts Amherst. Downloaded on June 03,2023 at 22:27:44 UTC from IEEE Xplore. Restrictions apply.

fall classically [8] into two classes: physical faults and human-made faults, which may be defined as follows:

- **physical faults**: adverse physical phenomena, either internal (physico-chemical disorders: threshold changes, short circuits, open circuits,...) or external (environmental perturbations: electromagnetic perturbations, temperature, vibrations,...);
- **human-made faults**: imperfections which may be:
  design faults, committed either a) during the initial design of the system (broadly speaking, from requirement specification to implementation) or during subsequent modifications, or b) during the establishment of operating or maintenance procedures,
  interaction faults: inadvertent or deliberate violations of operating or maintenance procedures.

The adjective "deliberate" in the definition of human-made interaction faults is intended to include "undesired accesses" or "intrusions" in the sense of computer security and privacy; however, the corresponding methods and techniques will not be specifically addressed in this paper.

It could be argued that viewing a fault as the phenomenological cause of an error may lead us recursively "a long way back." This definition simply emphasizes the fact that the very notion of fault is arbitrary: a fault is *the adjudged or hypothesized* cause of an error, and this cause may vary depending upon the adopted viewpoint: fault-tolerance mechanisms, maintenance engineer, repair shop, semiconductor physicist, etc. In our view, *recursion stops at the cause which is intended to be avoided or tolerated.* Furthermore, this view is consistent with the distinction between human-made and physical faults, since a computing system is a human-made artifact and as such any fault in it is ultimately human-made since it represents human inability to master all the phenomena which govern the behavior of a system. In an absolute sense, a distinction between physical and human-made faults (especially design faults) may be considered unnecessary; however, this distinction is of importance when considering (current) methods and techniques for procuring and validating dependability. If the recursion mentioned above is *not* stopped, then *a fault is nothing other than the consequence of a failure of a system that has delivered or is now delivering a service to the considered system.*

An error was defined as being liable to lead to a failure. Whether or not an error will actually lead to a failure depends on two major factors:

1) The **system composition**, and especially the nature of existing redundancy:

- **intentional redundancy** (introduced to provide fault tolerance) which is directly intended to prevent an error from leading to a failure,
- **unintentional redundancy** (it is practically difficult to build a system without any form of redundancy) which may have the same (unexpected) result as intentional redundancy.

2) The **definition of a failure** from the user's viewpoint: what is a failure for a given user may be a bearable nuisance for another one. Examples are a) accounting for the user's time granularity: an error which "passes through" the interface between the system and its user(s) may or may not be viewed as a failure depending on the user's time granularity, b) the notion of "acceptable error rate" (implicitly before considering that a failure has occurred) in data transmission.

Based on the given definition of a system, the discussion of whether "failure" applies to a system or to a component is irrelevant, since a component is itself a system. When atomic systems are dealt with, the notion of an "elementary" failure comes naturally.

The structural view of a system enables fault pathology to be made more precise. The creation and manifestation mechanisms of faults, errors, and failures may be summarized as follows:

1) A fault may be **dormant** or **active**; a fault is active when it produces an error. A fault may cycle between its dormant and active states. Physical faults can directly affect the physical components only, whereas human-made faults may affect any component.

2) An error may be **latent** or **detected**, either a) by a detection algorithm or mechanism within the component, or b) when it is noted by the user. An error may, and in general does, propagate from one component to another; by propagating, an error creates other (new) errors. An error within a component may thus originate from:

- activation of a dormant fault within the same component,
- propagation of an error within the same component or from another component.

3) A **component failure** occurs when an error affects the service delivered by the component as a response to request(s). These requests have to comply with the specification conditions: a component that is requested to deliver service(s) outside of its specification cannot be regarded as failed.

The discussion exposed in the above paragraph enables the "fundamental chain" to be completed:

$$\cdots \rightarrow \textbf{failure} \rightarrow \textbf{fault} \rightarrow \textbf{error} \rightarrow \textbf{failure} \rightarrow \cdots$$

In conclusion, we observe that faults, errors, and failures are all *undesired* circumstances. Assignment of the terms fault, error, and failure simply takes into account current usage: fault avoidance, tolerance, and diagnosis, error detection and correction, failure rate. It has to be noted that the labels "fault" and "error" are sometimes interchanged, as in [35]. The assignment adopted here acknowledges the long-time usage of coding theory.

4) On Fault Avoidance, Tolerance, Removal, and Forecasting: All the "how to's" which appear in the basic definitions given in Section I-A are in fact goals which cannot be fully reached, as all the corresponding activities are human activities, and thus imperfect. These imperfections bring in dependencies which explain why it is only the *combined* utilization of the above methods (preferably *at each step* of the design and implementation process) which can lead to a dependable computing system. These dependencies can be sketched as follows: in spite of fault avoidance by means of design methodologies and construction rules (imperfect in order to be workable), faults occur; hence the need for fault removal; fault removal is itself imperfect, as are the off-the-shelf components of the system, hence the need for fault forecasting; our increasing dependence on computing systems brings in fault tolerance, which in turn necessitates

Authorized licensed use limited to: University of Massachusetts Amherst. Downloaded on June 03,2023 at 22:27:44 UTC from IEEE Xplore. Restrictions apply.

further construction rules, and thus fault removal, fault forecasting, etc. It must be noted that the process is even more recursive than it appears from the above: current computer systems are so complex that their design and implementation need computerized tools in order to be cost-effective (in a broad sense, including the capability of succeeding within an acceptable time scale). These tools have themselves to be dependable, and so on.

The preceding reasoning explains why in the given definitions fault removal and fault forecasting are gathered into validation. Validation is often limited to what has been termed as verification; in that case these two terms are often associated, e.g. "V and V" [15], the distinction being related to the difference between "building the system right" (related to verification) and "building the right system" (related to validation). What is proposed is simply an extension of this concept: the answer to the question "am I building the right system?" being complemented by "for how long will it be right?": validation stems from "validity," which encapsulates two notions:

- validity *at a given moment*, which relates to fault removal;
- validity *for a given duration*, which relates to fault forecasting.

Besides highlighting the need for validating the procedures and mechanisms of fault tolerance, considering fault removal and fault forecasting as two constituents of the same activity—validation—is of great interest because it enables a better understanding of the notion of *coverage*, and thus of an important problem introduced by the above recursion(s): *the validation of the validation*, or how to reach confidence in the methods and tools used in building confidence in the system. **Coverage** refers here to a measure of the representativity of the situations to which the system is submitted during its validation with respect to the actual situations it will be confronted with during its operational life.

5) On the Measures of Dependability: The term "probability" has intentionally not been employed in the given definitions, so as to keep the discussion informal, and to reinforce the physical significance of the defined measures. However, as the considered circumstances are nondeterministic, random variables are associated with them, and the measures which are dealt with are probabilities.

Only two basic measures (or "metrics") have been considered, reliability and availability. Usually a third one, **maintainability**, is also considered, which may be defined as a measure of the continuous delivery of improper service, or equivalently, of the time to restoration. This measure is no less important than those previously defined; it was not introduced earlier because it may, at least conceptually, be deduced from the other two. It is noteworthy that availability encapsulates both the frequency of failure and the duration of proper service delivery at each alternation of proper–improper service.

A system may not, and generally does not, always fail in the same way. This immediately brings in the notion of the **consequences of a failure** upon the other systems with which the considered system is interacting, i.e. its **environment**. Several failure modes can generally be distinguished (in the specification), ordered according to the increasing severity of their consequences. A special case of great interest is that of systems which exhibit two failure modes whose severities differ considerably:

- **benign** failures, where the consequences are of the same order of magnitude (generally in terms of cost) as those of the service delivered in the absence of failure;
- **malign** or **catastrophic** failures, where the consequences are not comparable with those of the service delivered in the absence of failure.

By grouping the states of proper service and improper service subsequent to benign failures into a *safe* state (in the sense of being free from damage, not from danger), the generalization of reliability leads to an additional measure: a measure of *continuous safeness*, or equivalently, a measure of the *time to catastrophic failure*, i.e., **safety**. It is worth noting that a direct generalization of availability, thus providing a measure of safeness with respect to the alternation of safeness and improper service after catastrophic failure, would not provide a significant measure. When a catastrophic failure has occurred, the consequences are generally so important that system restoration is not of prime importance for at least the two following reasons:

- it comes second to repairing (in the broadest sense of the term, including legal aspects) the consequences of the catastrophe;
- the lengthy period prior to being allowed to operate the system again (investigations, hearings) would lead to meaningless numerical values.

However, a "hybrid" reliability–availability measure can be defined: a measure of proper service with respect to the alternation of proper and improper service after benign failure. This measure is of interest in that it provides a quantification of the system availability *before* occurrence of a catastrophic failure, and as such enables quantification of the so-called "reliability- (or availability-) safety tradeoff."

D. Summary

What has been presented in this section actually constitutes an attempt to build a taxonomy of dependable computing. The concepts introduced may be gathered into three main classes of attributes:

- the **impairments** to dependability, which are undesired (not unexpected) circumstances causing or resulting from un-dependability, whose definition is very simply derived from the definition of dependability: reliance cannot, or will not, be any longer justifiably placed on the service;
- the **means** for dependability, which are the methods, tools, and solutions enabling a) the system to be provided with the ability to deliver a service on which reliance can be placed, and b) the user to reach confidence in this ability;
- the **measures** of dependability, which enable the service quality resulting from the impairments and the means opposing them to be appraised.

This (beginning of a) taxonomy is represented in the form of a tree in Fig. 1.

The independence of the basic definitions with respect

Authorized licensed use limited to: University of Massachusetts Amherst. Downloaded on June 03,2023 at 22:27:44 UTC from IEEE Xplore. Restrictions apply.

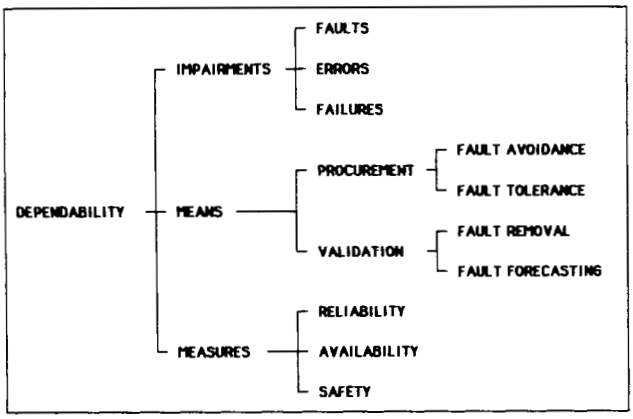

Fig. 1. Taxonomy of dependable computing.

to any fault class should facilitate the bringing together of activities which are often considered as separate, such as:

- VLSI testing and software testing,
- hardware reliability (with respect to physical faults) and software reliability (with respect to design faults), to say nothing of hardware reliability with respect to design faults,
- tolerance of physical faults and tolerance of design faults,
- computer system security, safety, and reliability.

II. Fault Tolerance in VLSI and Design Diversity

A. Design Faults: The Real Problem?

The first section of this paper has clearly shown that achieving a dependable computing system is a *global* problem. This problem is dominated by the struggle against the impairments. How have the impairments evolved and what is today's situation? Hardware components have seen a dramatic increase in both complexity and reliability: in the last 20 years, the failure rates have decreased by an order of magnitude every five years, while the number of components per chip has been multiplied by more than three orders of magnitude [34]. This improvement is all the more remarkable since it has been accompanied by an evolution in the fault classes: if a better understanding of the underlying phenomena has led to mastering of phenomena which were thought some years ago to be "random" faults, the technological evolution has been accompanied by the emergence of fault sources and types which were previously unknown. Mastering those new fault types may call for fault tolerance, such as error-correcting codes for storage errors caused by alpha particles.

The current trend of computer systems towards tolerance of physical faults (as examplified in transaction-oriented systems) makes the other classes of faults more apparent to the user as cause of system failure. Design faults are especially visible:

- about one third of the failures experienced by Tandem machines were "infant mortality" problems, both in hardware and software [31];
- design faults caused about 50 percent of system outage in the early AT & T ESSs [60]; they were either software faults or recovery deficiencies, not even including procedural faults, a large proportion of which can be identified as design faults.

More specifically considering hardware design faults, the data about the AT & T ESS-4 [20] show that hardware design faults range from 5 to 20 percent of all identified faults.

Within the hardware design faults, the results reported in [24] exhibit the following breakdown:

- specification faults: 19 percent,
- environment faults (resulting from the physical constraints or environment in which the actual design is implemented: design rules broken, noise, design automation faults): 31 percent,
- implementation faults: 50 percent.

We see that design faults in the hardware still significantly contribute to system failure. By this, we do not claim that those faults that *are considered as* "random" are not any more a problem: the race situation mentioned above between the mastering of the underlying phenomena and the emergence of new fault sources will still continue, and the generalization of the incorporation of functional sections together with fault-tolerance mechanisms in the circuits has still to come, in spite of early work such as [16], without speaking of fault tolerance for yield improvement. We simply want to stress the fact that both the steady component reliability improvement and the system fault tolerance will make design faults more prominent. Additional arguments are:

a) the fact that hardware design faults as well as software faults are still being discovered a long time after systems are being put into operation; see, e.g., [47] for the IBM 3081 series or [14] for the AT & T ESS-5,
b) the extremely high cost of field fixes to hardware design faults, even compared to fixes of software faults.

Safety-related systems constitute a domain which is extremely sensitive to design faults, since they are a source of the so-called "common-mode failures." The pre-computer realizations of safety-related systems systematically used (hardware) design diversity; typical examples are a hardwired electronic channel backed up by an electromechanic or a pneumatic channel.

In critical computer applications, design fault tolerance is usually achieved through two nonexclusive approaches: functional partitioning or/and design diversity.

In **functional partitioning**, the function of the system is partitioned into subfunctions in such a way that the failure of a component implementing a subfunction will be confined and will not prevent the execution of the total function, although perhaps in a degraded mode. Each subfunction is then implemented by a "complete" computer: hardware, executive software, and application software. Examples of this approach may be found in airplane flight control systems (e.g., the Boeing 757/767 [63]) or nuclear plant monitoring (e.g., the SPIN system [53]). This approach generally leads to a very large number of processing elements, larger than what would be necessary in terms of computing power; for instance, the Boeing 757/767 flight management control system is composed of 80 distinct functional microprocessors (300 when redundancy is accounted for). Although such large numbers of processing elements are affordable due to the low cost of today's

Authorized licensed use limited to: University of Massachusetts Amherst. Downloaded on June 03,2023 at 22:27:44 UTC from IEEE Xplore. Restrictions apply.

hardware, the question may be raised whether this approach when employed alone will not limit the future growth of the application, since the isolation of individual functions a) inhibits the cooperative use of total computer resources, and b) limits the improvement of dependability.

The other approach, **design diversity**, consists of delivering the expected service through multiple, independently designed and implemented computation channels, as discussed below.

B. Multiple-Channel Systems for Fault Tolerance

As introduced in Section I-B, the function of fault tolerance is to preserve the delivery of expected service 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 all are an integral part of the fault-tolerant system, and may be implemented in hardware, firmware, or software.

Multiple computation is a fundamental method employed to attain fault tolerance. Multiple computations are implemented by N-fold ($N \geqslant 2$) replications in three domains: time (repetition), space (hardware), and information (software and data). In a multiple-channel architecture, computations take place as concurrent executions of N copies of the reference program on N hardware channels. Examples of such systems are NASA's Space Shuttle with $N = 4$ [57], SRI's SIFT system with $N \geqslant 3$ [29], C. S. Draper Lab's Fault-Tolerant Multiprocessor with $N = 3$ [32], AT & T Bell Laboratories' no.1 ESS central control with $N = 2$ [21], and the DEDIX system at UCLA with $2 \leqslant N \leqslant 20$ [12], [13].

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

1) the **consistency** of initial conditions and inputs for all N computations must be assured;
2) a reliable **decision algorithm** that determines a single decision result from the multiple results must be provided.

It is 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 decision element, 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 channels. Faults that affect only one in a set of N computations are called **simplex faults**. A simplex fault does not affect other channels, 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 channels are **M-plex faults** ($2 \leqslant M \leqslant N$); 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. Classical examples of common physical causes are: interchannel shorts or sneak paths, common power supply fluctuations, bursts of radiation, etc. A more subtle cause may arise from fault dormancy: a "random" fault may actually be due to some weakness in the design or implementation. Thus the concept of fault independence has a time dimension: two successive independent faults appearing at different time instants in different channels may turn to be related if the first one is still dormant when the second one occurs [5].

Another class of related M-plex faults are *design* faults that are 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.

Design diversity is a potentially effective method to avoid the identical errors that are caused by design faults in multiple computation systems. Here we use N independently designed software or/and hardware elements instead of identical copies that were generated from one design.

The use of N diverse channels for N-fold computation introduces new "similarity" considerations for multiple results and for errors caused by M-plex faults. The results of individual channels may 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 outnumbers the set of good (similar) results at a decision point, then the decision algorithm will arrive 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 $M \geqslant 2$ diverse channels, 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

Authorized licensed use limited to: University of Massachusetts Amherst. Downloaded on June 03,2023 at 22:27:44 UTC from IEEE Xplore. Restrictions apply.

different design faults have produced a pair of similar errors [11].

Our choice is to follow the preceding treatment of physical faults and to consider two (or $M \geq 2$) 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 design tool, or use of the same erroneous designer's manual. If a common cause cannot be identified or hypothesized, the design faults are considered **independent**, even though the errors they have caused are similar.

We anticipate situations in which two (or more) designers 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.

C. Evolution of Design Fault Tolerance

As already mentioned, the incidence of design faults increases with the growth of system complexity. Thus it is not surprising that design fault tolerance has first been studied for software, and has reached hardware later and to a lesser extent.

1) Tolerance of Software Design Faults: 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. 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.

Multiple-version software had 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-1970s [23], [25], [26], [36]. However, the first suggestion on record was made by Dr. Dionysius Lardner, who wrote in his article "Babbage's Calculating Engine," published in the *Edinburgh Review*, no. CXX, July 1834, as follows [48]:

> *"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 at the University of Newcastle upon Tyne in the early 1970s [1], [51]. 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) error 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. Recovery is considered complete when the acceptance test is passed.

Another continuing investigation of multiversion software is the N-version Programming (NVP) project started at UCLA in 1975 [6]. *N*-fold computation 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 [7]. Other pioneering investigations of the multiversion software concept have been carried out at the KFK Karlsruhe, FRG [27], at the Halden Reactor Project, Norway [19], and at UC Berkeley and other organizations, USA [50].

The fundamental difference between the RB and NVP approaches is the decision algorithm. In the RB approach, the acceptance test checks the validity of the alternate results with respect to the specification of the application program being implemented. In the NVP approach, the decision algorithm is a generic consensus algorithm that delivers an agreement/disagreement decision.

2) Tolerance of Related Faults in VLSI Circuits: Two major choices exist for multiple-channel hardware implementations that employ VLSI circuitry. First, it is possible to place two (or more) channels on the same chip. Second, separate chips can be employed to implement each channel of computation.

The single-chip implementation of two or more channels is very attractive because it offers a self-checking chip that can be employed as a building block for fault-tolerant systems. However, it is also more susceptible to the occurrence of related faults that can negate the advantages of multiple computation.

Related faults in the logic circuitry of VLSI circuits that are employed in multiple-channel systems fall into two categories:

1) *Physical faults* due to external influences (e.g., power supply fluctuations, electromagnetic interference, radiation) and common weaknesses in the manufacturing process.

2) *Design faults* introduced by human mistakes and faulty design tools during the design of the circuit, as well as by ambiguities and errors in the initial specification.

The occurrence of related physical faults is more likely when the channels are contained on the same chip because of the closer electrical and physical coupling. Fabrication weaknesses specific to a given wafer are also a potential source of related faults. Similar errors are likely to be caused by such related faults, and the benefit of multiple channels is partially negated.

Methods to detect the errors caused by related physical faults on one chip have been investigated by Sedmak and Liebergot [55], and by Tamir and Sequin [59]. The former study employs the method of "complementary logic" to provide two diverse (complementary) implementations of functional logic on the same chip. The later study by Tamir and Sequin identifies severe performance penalties that can

Authorized licensed use limited to: University of Massachusetts Amherst. Downloaded on June 03,2023 at 22:27:44 UTC from IEEE Xplore. Restrictions apply.

be incurred in creating such dual implementations in MOS technology. They consider the properties of CMOS and NMOS gates and develop guidelines for diversifying the second implementation by selectively converting positive logic modules to negative logic.

Both studies discussed above represent pioneering efforts to introduce diversity in two-channel VLSI designs on a single chip. However, the diversification begins at the level of logic design, and therefore all design faults that are introduced prior to this stage will be reproduced in both channels.

A more general method of diversification of multiple channels is the use of two or more independently designed and manufactured but functionally compatible VLSI circuits. This approach is directly compatible with the multiple-version software discussed previously and provides tolerance of design faults up to the level of the initial functional specification of the circuit. An early investigation of this approach is described in [4], which was aimed at detecting similar errors and tolerating externally induced transient faults. The corresponding system was based on two diverse microprocessors: a monolithic microprocessor (TMS 9900), and its emulation at the instruction set level by a bit-slice microprocessor (AMD 2900 series).

Not surprisingly, real systems using this approach are found in safety-related applications, especially commercial flight control systems. Examples are the Airbus A-310 and the Boeing B 737/300.

The slat and flap control system of the Airbus A-310 [44] uses as a building block a computer system made up of two different and independently programmed microprocessors; it thus employs both hardware and software design diversity. The building block has the capacity of detecting errors generated by hardware or software design faults. Duplication of the building block is used to assure high availability with respect to independent physical faults.

The flight control of the Boeing 737/300 [61] uses hardware design diversity. A building block is composed of two computation channels, supported by three different CPUs: one CPU (CPU-1) implements the whole flight control system software, the other two (CPU-2 and CPU-3) implement the critical functions only. This building block is replicated and design fault tolerance is provided through comparison monitors. There are two monitors in each building block, comparing the outputs of a) CPU-1 and -3, -2 and -3 in one building block, and of b) CPU-1 and -2, -2 and -3 in the other one.

D. Research Directions for Design Diversity

Design diversity is still in a preliminary stage. Many problems are to be solved. We address in this section problems which we consider to be of foremost importance in attaining complete systems that incorporate software as well as hardware design diversity for the tolerance of design faults.

1) Initial Specification: The most critical condition to avoid related design faults is the existence of a complete and accurate specification of the service to be delivered by the diverse designs. This is the "hard core" of design diversity: latent defects in the specification such as inconsistencies, ambiguities, and omissions are likely to bias otherwise entirely independent 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 [43], [45], [49]. When such specifications are executable, they can be automatically tested for latent defects and serve as prototypes of the designs, 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.

When design diversity is applied to application software, the common specification must incorporate the specification of the decision points. This has to be paid a very careful attention, in order not to limit the subsequent diversity.

2) Independence of Design Efforts: The approach that is employed to avoid related design faults in a set of N designs is maximal independence of design and implementation efforts. It calls for the use of diverse algorithms, design description languages, design tools, implementation techniques, test methods, etc. The second condition for independence is the employment of independent (noninteracting) designers, with diversity in their training and experience. Wide geographical dispersion and diverse ethnic backgrounds may also be desirable.

3) Maintenance of N-Channel Diverse Designs: A first problem is the corrective maintenance of each channel. In order to preserve diversity, it is likely that the independence of the design efforts has to be pursued in maintenance. The "hard core" is then constituted by the removal of related faults in two or more channels. Again, a scheme similar to the design scheme is likely to prevail: a common "specification" of the maintenance action, "implemented" by independent maintenance teams.

Another problem is the modification of N-channel diverse designs due to a modification or improvement of functionality or performance. 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 channels 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.

4) Multilayer Design Diversity: The existence of a relatively small number of systems that have adopted hardware design diversity may be explained through the following argument: execution of diversified software versions leads to *diversified states* of the underlying hardware interpreter(s); design faults of the interpreters are thus unlikely to lead to similar errors in the interpreted software. This argument may be generalized saying that design diversity of the layer which delivers service to the user (the application software) precludes the need of design diversity in underlying layers (executive software, hardware). This is, however, an "end-to-end" argument which needs a more thorough investigation. For instance, the design diversity of a layer delivering service to another diversified layer may be desired in order to enforce the design diversity of the latter.

A multilayer design diversity experiment is currently being

Authorized licensed use limited to: University of Massachusetts Amherst. Downloaded on June 03,2023 at 22:27:44 UTC from IEEE Xplore. Restrictions apply.

conducted at UCLA, where the DEDIX system (a prototype supporting *N*-version programming [12], [13]) is undergoing formal specification. Subsequently, this specification will be used to generate 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.

5) Assessment of Effectiveness: Assessing the effectiveness of design diversity from the dependability viewpoint through experimentation [3], [11], [27] and modeling [22], [28], [40] has to be pursued and developed. Interaction between experimentation and modeling is needed. Large-scale experiments need to be carried out in order to gain statistically valid evidence, and the "mail-order design" approach offers significant promise. In "international mail-order" experiments, the members of fault tolerance research groups from several countries will perform design and implementation from a common specification. This may concern both software and hardware. It is expected that the several versions produced at widely separated locations, by different designers with different training and experience, will contain substantial design diversity.

Cost effectiveness of design diversity is a crucial question [46]. The generation of *N* versions of a given design instead of a single one shows an immediate increase in the cost 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 designs, and 2) reduction in size and complexity of validation tools and operations through the use of generic *N*-version environments in which the versions validate each together.

Conclusion

From its origins to only a few decades in the past, the technical activities of our technological society were based on mechanics; we have now reached the age of informatics. Very early in their history, humans became interested in constructing reliable systems (as already mentioned the French word for reliability, "fiabilité," dates back to the 12th century). For a long time, the construction of reliable mechanical systems has cherished the notion of "safety factor" which compensates for our imperfect mastery of the underlying failure phenomena. Owing to the abstract nature of information (and to our limited current state of knowledge) no similar concept is applicable to a computing system as a whole: designing a dependable computing system calls for the intentional and explicit incorporation of redundancies, as opposed to the "implicit" redundancy brought about by a safety factor.

The highly desirable nature of the incorporation of explicit redundancies into data processing has long been acknowledged, as witnessed by the 1834 quotation mentioned in Section II-C1. However, until a few years ago, a fraction of the theoreticians and practitioners of the abstract side of computing systems held the firm belief that humans were sufficiently clever to master the complexity of their own creations! Such a situation certainly contributed greatly to the fairly late emergence of Computing System Dependability as a *unified discipline* [9], the purpose of which is [58]:

"designing, implementing and using computer systems where faults are natural, foreseeable and tolerable."

In order to make progress, a discipline needs at least a minimum consensus on fundamental concepts and on the terminology to express them. The purpose of the first section of this paper was to present informally, but accurately, the basic concepts of computer systems dependability, emphasizing the unifying aspect of the concepts.

The current, practical, realizations of design fault tolerance that are addressed in the second section are essentially devoted to safety-related applications. Historically, fault tolerance evolved from such systems to commercial general-purpose systems. The question is: how long will it take for design diversity to evolve in the same way?

Acknowledgment

The topics addressed in this paper have greatly benefited from the contributions of many individuals at LAAS and UCLA, as well as from our colleagues in the IFIP WG 10.4 "Reliable Computing and Fault-Tolerance." Special thanks go to T. Anderson, W. C. Carter, A. Costes, and J. P. J. Kelly.

References

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

[2] ——, "Fault tolerance terminology proposals" in *Proc. 12th Int. Symp. on Fault-Tolerant Computing* (Los Angeles, CA, June 1982), pp. 29–33.

[3] T. Anderson, P. A. Barrett, D. N. Halliwell, and M. R. Moulding, "Software fault tolerance: An evaluation," *IEEE Trans. Software Eng.*, vol. SE-11, no. 12, pp. 1502–1510, Dec. 1985.

[4] J. Arlat, "Design of a microcomputer tolerating faults through functional diversity," Dr. Eng. dissertation, National Polytechnic Institute, Toulouse, France, Apr. 1979 (in French).

[5] J. Arlat and J. C. Laprie, "On the dependability evaluation of high safety systems," in *Proc. 15th Int. Symp. on Fault-Tolerant Computing* (Ann Arbor, MI, June 1985), pp. 318–323.

[6] 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.

[7] 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.

[8] A. Avižienis, "Fault tolerance, the survival attribute of digital systems," *Proc. IEEE*, vol. 66, no. 10, pp. 1109–1125, Oct. 1978.

[9] ——, "Towards a discipline of reliable computing," presented at IFIP Working Conf. on Reliable Computing and Fault-Tolerance in the 1980's, in *Proc. EURO IFIP 79* (London, England, Sept. 1979), pp. 701–705.

[10] ——, "The four-universe information system model for the study of fault tolerance," in *Proc. 12th Int. Symp. on Fault-Tolerant Computing* (Los Angeles, CA, June 1982), pp. 6–13.

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

[12] A. Avižienis, "The N-version approach to fault tolerant software," *IEEE Trans. Software Eng.*, vol. SE-11, no. 12, pp. 1491–1501, Dec. 1985.

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

[14] H. A. Bauer, L. M. Croxall, and E. A. Davis, "The 5 ESS switching system: System test, first-office application, and early field experience," *AT&T Tech. J.*, vol. 64, no. 6, pp. 1503–1522, Jul.–Aug. 1985.

[15] B. W. Boehm, "Guidelines for verifying and validating soft-

Authorized licensed use limited to: University of Massachusetts Amherst. Downloaded on June 03,2023 at 22:27:44 UTC from IEEE Xplore. Restrictions apply.

ware requirements and design specifications," in *Proc. EURO IFIP 79* (London, England, Sept. 1979), pp. 711–719.

[16] W. C. Carter, G. R. Putzolv, A. B. Wadia, W. G. Bourricius, D. C. Jessept, E. P. Hsieh and C. J. Tan, "Cost effectiveness of self checking computer design," in *Proc. 7th Int. Symp. on Fault-Tolerant Computing* (Los Angeles, CA, June 1977), pp. 117–123.

[17] W. C. Carter, "Fault detection and recovery algorithms for fault-tolerant systems," in *Proc. EUROIFIP 79* (London, England, Sept. 1979), pp. 725–734.

[18] F. Cristian, "Exceptions, failures and errors," *Tech. et Sci. Informat.*, vol. 4, no. 4, pp. 385–390, July–Aug. 1985, (in French). English version: IBM Res. Rep. RJ 4130.

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

[20] E. A. Davis and P. K. Giloth, "No. 4 ESS: Performance objectives and service experience," *Bell Syst. Tech. J.*, vol. 60, no. 6, pp. 1203–1224, July–Aug. 1981.

[21] 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.

[22] D. E. Eckhardt and L. D. Lee, "A theoretical basis for the analysis of multiversion software subject to coincident errors," *IEEE Trans. Software Eng.*, vol. SE-11, no. 12, pp. 1511–1517, Dec. 1985.

[23] W. R. Elmendorf "Fault-tolerant programming," in *Proc. 1972 Int. Symp. Fault Tolerant Computing* (Newton, MA, June 19–21, 1982), pp. 79–83.

[24] T. L. Faulkner, C. W. Bartlett, and M. Small, "Hardware logic design faults: A classification and some measurements," in *Proc. 12th Int. Symp. Fault-Tolerant Computing* (Santa Monica, CA, June 1982), pp. 377–380.

[25] M. A. Fischler, P. 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.

[26] 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.

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

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

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

[30] ——, "A time for integration," in *Proc. 12th Int. Symp. on Fault-Tolerant Computing* (Los Angeles, CA, June 1982), p. 82.

[31] J. Gray, "Why do computers stop and what can be done about it?" in *Proc. 5th Symp. on Reliability in Distributed Software and Database Systems* (Los Angeles, CA, Jan. 1986), pp. 3–12.

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

[33] J. E. Hosford, "Measures of dependability," *Operations Res.*, vol. 8, no. 1, pp. 204–206, 1960.

[34] M. V. Hsiao, W. C. Carter, J. W. Thomas, and W. R. Stringfellow "Reliability, availability and serviceability of IBM computer systems: A quarter century of progress," *IBM J. Res. Develop.*, vol. 25, no. 5, pp. 453–465, Sept. 1981.

[35] IEEE Std. 729, *IEEE Standard Glossary of Software Engineering Terminology.* New York: IEEE, 1982.

[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] ——, "The failure-fault (FF) model," in *Proc. 12th Int. Symp. on Fault-Tolerant Computing* (Los Angeles, CA, June 1982), pp. 14–17.

[38] J. C. Laprie, A. Costes, and R. Troy, "Dependability: Requirements and solutions," in *Proc. SEE Congr. on Electrical and Electronical System Dependability* (Toulouse, France, Oct. 1979, in French).

[39] J. C. Laprie and A. Costes, "Dependability: A unifying concept for reliable computing," in *Proc. 12th Int. Symp. on Fault-Tolerant Computing* (Los Angeles, CA, June 1982), pp. 18–21.

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

[41] J. C. Laprie, "Dependable computing and fault tolerance: concepts and terminology," in *Proc. 15th Int. Symp. on Fault-Tolerant Computing*, Ann Arbor, June 1985, pp. 2–11.

[42] P. A. Lee and D. E. Morgan, Eds., "Fundamental concepts of fault-Tolerant computing, progress report," in *Proc. 12th Int. Symp. on Fault-Tolerant Computing* (Los Angeles, CA, June 1982), pp. 34–38.

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

[44] D. J. Martin, "Dissimilar software in high integrity applications in flight controls," in *Proc. AGARD Symp. on Software for Avionics* (The Hague, The Netherlands, 1982), p. 36:1.

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

[46] G. E. Migneault, "The cost of software fault tolerance," NASA Tech. Memo. 84546, Sept. 1986.

[47] M. Monachino, "Design verification system for large-scale LSI designs," *IBM J. Res. Develop.*, vol. 26, no. 1, pp. 89–99, Jan. 1982.

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

[49] J. V. Guttag, J. J. Horning, and J. M. Wing, "The Larch family of specification languages," *IEEE Software*, vol. 2, no. 5, pp. 24–36, Sept. 1985.

[50] 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.

[51] B. Randell, "System structure for software fault-tolerance," *IEEE Trans. Software Eng.*, vol. SE-1, no. 2, pp. 220–232, June 1975.

[52] B. Randell, P. A. Lee, and P. C. Treleaven, "Reliability issues in computing system design," *Comput. Surv.*, vol. 10, no. 2, pp. 123–165, June 1978.

[53] L. Remus, "Methodology for software development of a digital integrated protection system," presented at the EWICS-TC7 Meet., Jan. 1982.

[54] A. S. Robinson, "A user oriented perspective of fault tolerant systems models and terminologies," in *Proc. 12th Int. Symp. on Fault-Tolerant Computing* (Los Angeles, CA, June 1982), pp. 22–28.

[55] R. M. Sedmak and H. L. Liebergot, "Fault-tolerance of a general purpose computer implemented by very large scale integration," in *Proc. 8th Int. Symp. on Fault-Tolerant Computing* (Toulouse, France, June 1978), pp. 137–143.

[56] D. P. Siewiorek and R. S. Swarz, *The theory and Practice of Reliable System Design.* Billenca, MA: Digital Press, 1982.

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

[58] "Definition of the pilot-project on computer system dependability," Joint Rep. UPS-LSI/ONERA-CERT/CNRS-LAAS, Toulouse, France, Jan. 1976 (in French).

[59] Y. Tamir and C. H. Sequin, "Reducing common mode failures in duplicate modules," in *Proc. Int. Conf. on Computer Design: VLSI in Computers (ICCD'84)* (Port Chester, NY, Oct. 1984), pp. 302–307.

[60] W. N. Toy, "Fault-tolerant design of local ESS processors," *Proc. IEEE*, vol. 66, no. 10, pp. 1126–1145, Oct. 1978.

[61] L. J. Yount, "Architectural solutions to safety problems of digital flight-critical systems for commercial transports," in *Proc. 6th Digital Avionics Systems Conf.* (Baltimore, MD, Dec. 1984), pp. 28–35.

[62] B. P. Ziegler, *Theory of Modeling and Simulation.* New York: Wiley, 1976.

[63] R. E. Spradlin, "Boeing 757 and 767 flight management system," in *Proc. Radio Technical Commission for Aeronautics Tech. Symp.* (Washington, DC, Nov. 20–21, 1980), pp. 107–118.

[64] B. Courtois, "Failure mechanisms, fault hypotheses and analytical testing of LSI-NMOS (HMOS) circuits," in *Proc. Int. Conf.* (Edinburgh, Aug. 1981), pp. 341–350.

Authorized licensed use limited to: University of Massachusetts Amherst. Downloaded on June 03,2023 at 22:27:44 UTC from IEEE Xplore. Restrictions apply.