Pergamon

0045-7906(93)E0006-N

Computers Elect. Engng Vol. 20, No. 4, pp. 347–368, 1994

0045-7906/94 $7.00 + 0.00

A MULTI-LEVEL VIEW OF DEPENDABLE COMPUTING

BEHROOZ PARHAMI
Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106-9560, U.S.A.

(Received for publication 5 November 1993)

Abstract—This paper serves a dual purpose. It presents a unified framework and terminology for the study of computer system dependability. It also surveys the field of dependable computing in light of the proposed framework. Specifically, impairments to dependability are viewed from six levels, each being more abstract than the previous one. It is argued that all of these levels are useful, in the sense that proven dependability assurance techniques can be applied at each level, and that it is beneficial to have distinct, precisely defined terminology for describing impairments to, and procurement strategies for, computer system dependability at these levels. The six levels are:

(1) Defect level or component level, dealing with deviant atomic parts.
(2) Fault level or logic level, dealing with deviant signal values or path selections.
(3) Error level or information level, dealing with deviant data or internal states.
(4) Malfunction level or system level, dealing with deviant functional behavior.
(5) Degradation level or service level, dealing with deviant performance.
(6) Failure level or result level, dealing with deviant outputs or actions.

Briefly, a hardware or software component may be defective (hardware may also become defective due to wear and aging). Certain system states will expose the defect, resulting in the development of faults defined as incorrect signal values or decisions within the system. If a fault is actually exercised, it may contaminate the data flowing within the system, causing errors. Erroneous information or states may or may not cause the affected subsystem to malfunction, depending on the subsystem's design and error tolerance. A subsystem malfunction does not necessarily have a catastrophic, unsafe, or even perceivable service-level effect. Finally, degradation of service could eventually lead to system failure. At each of these six levels, the complementary approaches of prevention (avoidance or removal) and tolerance are discussed in relation to inter-level transitions.

Key words: Error handling, fail-safe systems, fail-soft systems, fault tolerance, graceful degradation, highly available systems, long-life systems, redundancy techniques, safety-critical systems, self-repair.

1. INTRODUCTION

Dependable computing is an outgrowth of fault-tolerant computing, a focus area born in the mid 1960s [8,73] following a decade of concern with digital system reliability issues [32,60,93]. Fault tolerance is actually an age-old design technique. Avizienis [14] quotes a 1.5-Century old paper advocating, in connection with Charles Babbage's inventions, the use of different computers and algorithms for checking computation errors. In the related area of fail-safe design, Babbage himself "boasts that it is impossible to wilfully derange or corrupt his machines ... and that his engines will always either produce the correct result or jam, but never deceive" [87]. E. M. Forster [34] fancies a civilization controlled by self-repairing machines that eventually self-destructs because designers ignored the possibility of faults in the 'Mending Apparatus' itself.

Following the initial emphasis on fault tolerance through fault-masking and self-repair, it was realized that tolerating faults is but one way of achieving high reliability. Hence, testing, testability, verification, maintainability, design methodologies, and other fault avoidance/removal techniques were integrated into the programs of the International Fault-Tolerant Computing Symposia [36], held annually since 1971. This broadening of scope shifted the focus to reliable computing, a variant of which had been used as far back as 1963 to complement reliable communication [98]. But the symposia continued with the original restrictive title. To avoid confusion between reliability as a precisely defined statistical measure and reliability as a qualitative attribute of systems and computations, use of dependability was suggested for conveying the second meaning [52]. Dependable computing deals with impairments to dependability (defects, faults, errors, malfunc-

tions, degradations, failures, and crashes), means for coping with them (fault avoidance, fault tolerance, design validation, failure confinement, etc.), and measures of success in the pursuit of dependability (reliability, availability, safety, etc.).

As computers are used for more critical applications by less sophisticated users, the dependability of computer hardware and software assumes even greater importance. Highly dependable computer systems have been in widespread use for almost three decades [28] and have been commercially marketed for about half as long [47]. The trend towards 'intelligent' computers and the expectation of higher safety levels with advanced technology [95] will make computer system dependability an integral part of the design and implementation process for future generation systems.

2. DEPENDABLE SYSTEMS: IDEAL VS REAL

2.1. Defining dependability

Dependability has been defined briefly as "the probability that a system will be able to operate when needed" [43]. This simplistic view, which subsumes both of the well-known notions of reliability and availability, is only useful for systems that are either completely operational or totally failed. Since what we are really interested in is task accomplishment rather than system operation, the following is more appropriate: "... the ability of a system to accomplish the tasks (or equivalently, to provide the service[s]) which are expected from it" [52]. A weakness of this definition is that it is based on expected rather than specified behavior, in order to accommodate possible specification slips. However, if expectations are realistic and precise, they can be viewed as simply another form of specification (perhaps a higher-level one). But if we expect too much, then this definition is an invitation to blame our misguided expectations on the system's undependability. Carter's "... trustworthiness and continuity of computer system service such that reliance can justifiably be placed on this service" [21] has two positive aspects: It takes the time element into account explicitly (continuity) and stresses the need for dependability validation (justifiably). Laprie's version of this definition [53] can be considered a step backwards in that it substitutes quality for trustworthiness and continuity. The use of quality and quality assurance, as in other engineering disciplines, is a welcome trend [16,30]. But precision need not be sacrificed for compatibility.

To formulate a more useful definition of dependable computing, one must examine the various aspects of undependability. To a user, undependability shows up in the form of late, incomplete, inaccurate, or incorrect results or actions [64]. The two notions of trustworthiness (correctness, accuracy) and timeliness can be abstracted from the above; completeness need not be dealt with separately since any missing result or action can be considered to be infinitely late. Actually, dependability should not be considered an intrinsic property of a computer system. A physically unreliable system might become quite dependable by virtue of algorithmic dependability procurement mechanisms, applied in different ways to various computations and/or interactions, given their specified correctness and timeliness requirements [66,67]. However, the ideal of associating dependability with individual data objects or actions is presently impractical.

Hence, dependability of a computer system may be defined as justifiable confidence that it will perform specified actions or deliver specified results in a trustworthy and timely manner. Note that the above definition does not preclude the possibility of having various levels of importance for different (classes of) user interactions or varying levels of criticality for situations in which the computer is required to react. Such variations simply correspond to different levels of confidence and dependability. This new definition retains the positive elements of previous definitions, while presenting a result-level view of the time dimension by replacing the notion of service continuity by timeliness of actions/results.

2.2. Impairments to dependability

Impairments to dependability are often described as hazards, defects, faults, errors, malfunctions, failures, and crashes. There are various definitions for these terms, causing different and sometimes conflicting usages. In this subsection, I review two major proposals on how to view and describe impairments to dependability and present my own model.

Members of the Newcastle Reliability Project [83], led by Professor Brian Randell, advocate a

hierarchic view [4]: a (computer) system is a set of components (themselves systems) that interact according to a design (another system). System failure is defined as deviation of its behavior from that predicted/required by the system's authoritative specification. Such a behavioral deviation results from an erroneous system state. An error is a part of an erroneous state which constitutes a difference from a valid state. The cause of the invalid state transition which first establishes an erroneous state, is a faulty component or design. Similarly, component or design failure can be attributed to an erroneous state within the corresponding (sub)system resulting from a component or design fault, and so on. At each level of the hierarchy, "the manifestation of a fault will produce errors in the state of the system, which could lead to a failure". Hence, failure and fault are simply different views of the same phenomenon. This is quite elegant and enlightening but gives rise to a need for continual establishment of frames of reference when talking about the causes (faults) and effects (failures) of deviant system behavior at various levels of abstraction.

While it is true that a computer system may be viewed at many different abstraction levels, it is also true that some of these levels have proved more useful in practice. Aviziens [11] takes four such levels and proposes distinct terminology for impairments to dependability (undesired events, in his words) at each level. His proposal is summarized in the following cause-effect diagram:

Abstraction level	*Undesired event*
Physical	Failure
⇓	⇓
Logical	Fault
⇓	⇓
Informational	Error
⇓	⇓
External	Crash

There are some problems with the above choices of names for undesired events. The term 'failure' has traditionally been used both at the lowest and the highest abstraction levels; viz. failure rate, failure mode, and failure mechanism used by engineers/physicists alongside system failure, fail-soft operation, and fail-safe system due to computer architects. To comply with the philosophy of distinct naming for different levels, Avizienis retains failure at the physical level and uses crash for the other end. However, minor inaccuracies or delays can hardly be considered 'crashes' in the ordinary sense of the term. Again, this usage emphasizes system operation rather than task accomplishment and ignores the fact that an 'operational' system can yield erroneous results.

Another problem is that there are at least three external views. The maintainer's external view consists of interacting subsystems that must be monitored for detecting possible malfunctions in order to reconfigure the system or, alternatively, to guard against consequences such as total system crash. The operations manager's external view consists of a more abstract black box capable of providing certain services. In this view, isolated malfunctions are acceptable as long as they do not lead to serious degradation of service availability and/or quality. Finally, the end user is mainly concerned with triggered actions and computation results and thus his/her external view is shaped by the system's reaction to particular situations or commands. The following six-level view of impairments to dependability rectifies the problems [65]:

Abstraction level	*Impairment*
Component	Defect
⇓	⇓
Logic	Fault
⇓	⇓
Information	Error
⇓	⇓
System	Malfunction
⇓	⇓
Service	Degradation
⇓	⇓
Result	Failure

Taking into account the fact that a non-atomic component is itself a system, usage of the term 'failure' in failure rate, failure mode, and failure mechanism can be justified by noting that a component is its designer's end product (system). Therefore, we can be consistent by associating the term 'failure' with the highest and the term 'defect' with the lowest level of abstraction. The component designer's failed system is the systems architect's defective component.

2.3. Dependable computing

The field of dependable computing deals with the procurement, forecasting, and validation of computer system dependability. As discussed in Subsection 2.2, impairments to dependability can be viewed from six levels. Thus, subfields of dependable computing can be thought of as dealing with some aspects of one or more of these levels. Specifically, I take the view that a system can be in one of seven states: Ideal, Defective, Faulty, Erroneous, Malfunctioning, Degraded, or Failed. Initially, a system may start up in any of the seven states, depending on the appropriateness and thoroughness of validation efforts. Once in the initial state, the system moves from one state to another as a result of deviations and remedies. Deviations are events that take the system to a lower (less desirable) state, while remedies are measures that enable a system to make the transition to a higher state. As shown in Fig. 1, each state can be entered through the sideways transitions initially, from above due to a deviation, or from below as a result of a remedy. Associated with each transition, are five attributes that can be regarded as transition labels or tags:

c: (natural) cause of the transition
i: (natural) impediment to the transition
f: techniques for facilitating the transition
a: techniques for avoiding the transition
m: tools for modeling the transition.

Recognized subfields of dependable computing deal with one or more of these attributes and some attributes can be the basis of new studies and subfields. In addition, such transition attributes can be used for classifying or indexing of techniques and research studies in the field of dependable computing. For example, a research project or proposed redundancy technique may be described as dealing with a[$\rightarrow$ faulty] and i[faulty $\rightarrow$ erroneous].

Detailed discussions of the non-ideal states and their related transitions appear in Sections 3–8 of the paper. Here, I present some general observations on the various system states. First, note

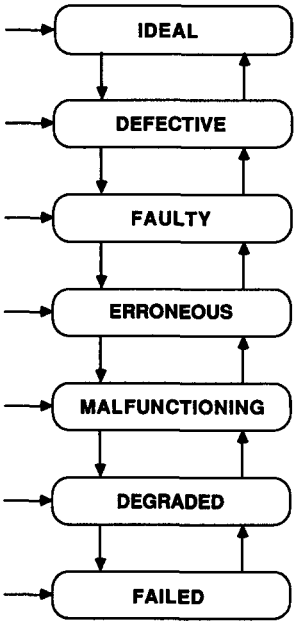


Fig. 1. System states and state transitions in the multi-level model of dependable computing.

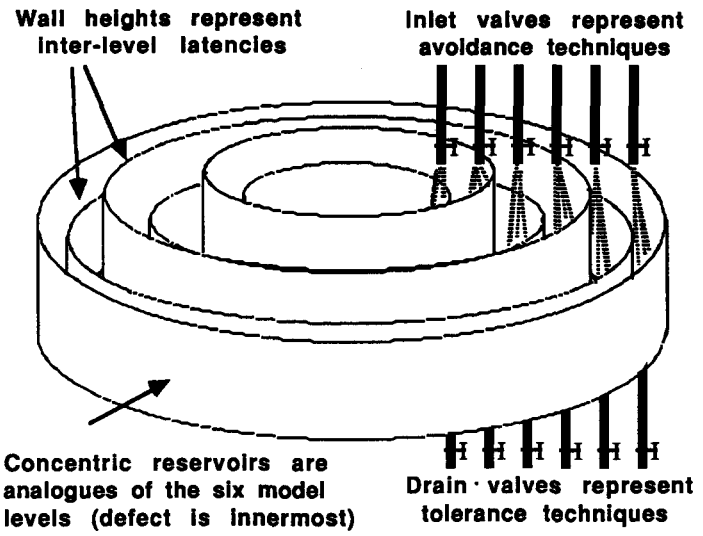


Fig. 2. A simple analogy for the multi-level model of dependable computing. Pouring water represents defects, faults, errors, malfunctions, degradations, and failures.

that the observability of the system state (ease of external recognition that the system is in a particular state) increases as we move downward in Fig. 1. For example, the inference that a system is 'ideal' can only be made through formal proof techniques; a proposition that is impractical for modern computer systems in view of their complexity. At the other extreme, a failed system can usually be recognized with little or no effort. As examples of intermediate states, the faulty state is recognizable by extensive off-line testing, while the malfunctioning state is observable by on-line monitoring with moderate effort. It is therefore common practice to force a system into a lower state (e.g. from defective to faulty, under torture testing) in order to deduce its initial state.

2.4. An analogy

Figure 2 provides an interesting analogy for clarifying the states and state transitions in my six-level hierarchical model, using a system of six concentric water reservoirs. Pouring water from above corresponds to defects, faults, and other impairments, depending on the layer(s) being affected. These impairments can be avoided by controlling the flow of water through valves or tolerated by the provision of drains of acceptable capacities for the reservoirs. The system fails if water ever gets to the outermost reservoir. This may happen, for example, by a broken valve at some layer combined with inadequate drainage at the same and all outer layers. Wall heights between adjacent reservoirs correspond to the natural inter-level latencies in my model. Water overflowing from the outermost reservoir into the surrounding area corresponds to a computer failure adversely affecting the larger physical, corporate, or societal system.

3. DEFECT-LEVEL OR COMPONENT-LEVEL VIEW

3.1. Causes and consequences of defects

Defects are caused in two ways, corresponding to the sideways and downward transitions into the defective state in Fig. 1: (1) physical design slip which results in defective system components, by improper design or inadequate screening; (2) development of defects due to component wear and aging or operating conditions that are harsher than those originally envisaged. An inadequately shielded device, for example, becomes defective when used in a high-noise environment and a real-time system assumes the defective state when the volume of input data exceeds its capacity. Wear and aging are only meaningful for hardware components, although it has been argued that software also 'ages' in a certain sense (due to changes in the environment or user expectations).

The causes of defects in computer hardware are quite varied and technology-dependent. Experimental data on the relative frequencies of defect types for different technologies are gathered by component manufacturers and some governmental organizations concerned with reliability

evaluation and validation [99]. The ultimate goal of these efforts is to facilitate the development and validation of defect models that enable us to predict the occurrence of defects without resorting to extensive independent experimentation. Defects in software components can also be experimentally observed and analyzed. The field of software science [42] deals with proposing plausible defect models for computer software (actually, error models in the prevailing terminology) and validating the models through the use of empirical data.

The frequency of defect occurrence is usually modeled by a defect rate parameter $\lambda(t)$, defined as the expected number of defect occurrences or, equivalently, as the probability of having some defect over a short, unit-length interval of time. For hardware components, the change in defect rate with time follows the so-called bathtub curve (there are also arguments against this characterization). During the burn-in period, high defect rates are common. This is followed by a long interval of relatively constant defect rate, constituting the component's useful life. Finally, in the wear-out period, rapid deterioration causes a corresponding surge in component defect rates. Assuming a constant component defect rate λ, the probability of no defect from time 0 to t in a system with n identical parts is $g(t) = e^{-n\lambda t}$. This is known as the exponential reliability law [7,29]. Equating the probability of no defect with reliability (probability of no failure) is a simple, but pessimistic, reliability estimation method that is widely used.

Three reliability enhancement techniques are suggested by $e^{-n\lambda t}$. Design simplicity, or reducing n, is an important paradigm for dependability. However, for a given overall functionality, a penalty will often be paid at some higher level where the design becomes more complex. Defect avoidance, or reducing λ, implies the use of low-defect-rate components that are quite expensive due to higher manufacturing and/or screening costs. Hence, the third approach of defect removal, or reducing t, is the most common component-level dependability procurement technique. Defect removal strategies include careful initial screening and frequent test/replacement of system components. This is further elaborated in Subsection 3.2. A more radical technique would be to attempt to change the probability expression altogether so that its value is less affected by λ and/or t. This is the aim of defect tolerance methods to be discussed in Subsection 3.3.

3.2. Defect avoidance and removal

A defect may be dormant or ineffective long after its occurrence. During this dormancy period, external detection of the defect is impossible or, at the very least, extremely difficult. If, despite efforts to avoid or remove them, defects are nevertheless present in a product, nothing is normally done about them until they develop into faults. Periodic replacement of sensitive parts is one way of removing defects before they develop into faults. Similarly, burn-in of components [63] and scheduled preventive maintenance of computer hardware tends to remove most dormant defects. Component modifications and improvements, motivated by the analyses of data pertaining to degraded or failed systems, are other major ways of hardware and software defect removal.

Avizienis has suggested the term 'fault intolerance' for the spectrum of techniques dealing with defect avoidance and removal [10]. In my terminology, defect intolerance would be the complementary approach to tolerance methods (discussed in Subsection 3.3). Fault intolerance, error intolerance, malfunction intolerance, degradation intolerance, and failure intolerance are actually implied by defect intolerance but may be considered separately if tolerance techniques are used at the lower levels of abstraction. Frequently, a mixture of intolerance and tolerance may provide the most cost-effective design for given dependability requirements.

3.3. Defect tolerance

Defect tolerance is achieved through component-level redundancy techniques in one of three forms:

(1) redundancy in the form of stronger components (safety factor)
(2) redundancy in the form of alternate components (defect bypassing)
(3) redundancy in the form of load-sharing components (defect masking).

The first strategy actually reduces the component's defect rate and may thus be viewed as a defect avoidance technique. The second alternative can be used along with selection or reconfiguration logic for bypassing defective components recognized under extensive testing. Yield enhancement

for semiconductor components is the most important application of this method [50,61]. The third approach dampens the effect of deviant components by providing additional components that share the load during normal operation and completely take over when a defect develops. The classic hardware example of this technique, first used in connection with the Orbiting Astronomical Observatory satellite [33], is the replacement of a diode by four diodes connected as two parallel branches each having two diodes in series in order to tolerate diode short- and open-circuits. Another example is the provision of the electrical resistance R by using three parallel resistors each having the resistance $2.4R$, thus tolerating one open resistor with moderate change in circuit characteristics (variation from $0.8R$ to $1.2R$).

Defect tolerance and removal are conflicting strategies in the sense that the provision of mechanisms to tolerate defects tends to make defect removal more difficult. If defect tolerance is employed as a yield enhancement technique, with the resulting components adequately tested prior to being used, then defect tolerance has no external manifestation except that it makes the testing more difficult. However, if defect tolerance is incorporated into component designs as a reliability improvement measure, serious problems may be encountered in validating reliability models and predictions. Experience shows that defect (or fault) masking methods can produce catastrophic effects when the masking capability of the mechanism is exceeded. Also, modeling becomes very difficult due to complex interactions between masked and non-masked defects (or faults).

3.4. The defect-to-fault transition

A system makes the transition from the defective state to the faulty state when a dormant defect is awakened and gives rise to fault. Designers try to impede this transition by providing adequate safety margins for the components and/or by using defect tolerance methods. Ironically, one may occasionally try to facilitate this transition for the purpose of exposing defects, since faults are more readily observable than defects. To do this, the components are usually subjected to loads and stresses that are much higher than those encountered during normal operation. This burning in or torture testing of components results in the development of faults in marginal components which are then identified by fault testing methods (see Subsection 4.2.) To be able to deduce the underlying defect from an observed fault, we need to establish a correspondence between various defect and fault classes. This is referred to as fault modeling and is discussed in Subsection 4.1. Alternatively, if the criticality of various fault classes has been established, fault modeling allows us to deduce the criticality of various defect classes and to establish priorities for dealing with them.

4. FAULT-LEVEL OR LOGIC-LEVEL VIEW

4.1. Causes and symptoms of faults

A hardware fault may be defined as any anomalous behavior of logic structures or substructures that can compromise the correct signal values within a logic circuit. As usual, the reference behavior is provided by some form of specification. If the anomalous behavior results from implementing the logic function g rather than the intended function f, then the fault is due to a logical design or implementation slip. The alternative cause of faults is the implementation of the correct logic functions with defective components. Defect-based faults can be classified according to duration (permanent, intermittent/recurring, or transient), extent (local or distributed/catastrophic), and effect (dormant or active). Only active faults produce incorrect logic signals. An example of a dormant fault is a line stuck on logic-value 1 that happens to carry a 1. If incorrect signals are produced as output or stored in memory elements, they cause errors in the system state.

Over the years, logic designers have developed fault models [1] that accurately reflect the logic-level consequences of defects and/or lend themselves easily to mathematical analysis. The single stuck-at fault model is the most popular. It assumes that a single line within the logic circuit has taken on a constant logic value independent of the inputs applied. A stuck-at-1 or stuck-at-0 fault may result from several classes of physical defects (e.g. open-connection or short-to-ground), depending on technology. The multiple stuck-at fault model, covering multiple unidirectional faults as a submodel, assumes that an arbitrary number of lines within the circuit may have become stuck. As a final example, the bridging fault model takes into account the possibility of short circuits in

the logic circuit. For certain technologies, switch-level fault models have been developed to overcome the problems resulting from the unavailability of accurate gate-level models.

4.2. Fault avoidance and removal

The two basic fault avoidance strategies relate directly to the two types of causes for faults. Logic design and implementation slips can be avoided by following rigorous design methodologies and by automating some of the more error-prone implementation steps. Faults that are caused by defects can be avoided by defect avoidance and tolerance methods discussed in Section 3. At present, complete fault avoidance is an unattainable goal. Thus, fault detection and removal techniques are integral parts of all logic design and implementation methodologies.

Fault detection through fault testing [37] is used for the validation of engineering prototypes, screening of manufactured devices, and maintenance (corrective or preventive) of operational systems. The fault testing effort is a combination of test generation, test validation, and test application (Fig. 3). Tests are preset or adaptive depending on the strategy for deciding which tests and in what sequence are to be applied. They can also be classified as functional or structural. Good fault models combined with exhaustive or analytic test generation methods usually allow for theoretical test validation, but most common validation methods are experimental, using simulation as the primary tool. Test application is categorized according to the type of control mechanism used for test initiation and analysis (external or internal to the entity being tested) and based on whether or not testing can proceed with normal circuit operation (on-line/concurrent or off-line). To reduce the cost of testing, which grows exponentially with circuit complexity, much attention has been given to design for testability [37,97] and built-in self-test (BIST) techniques.

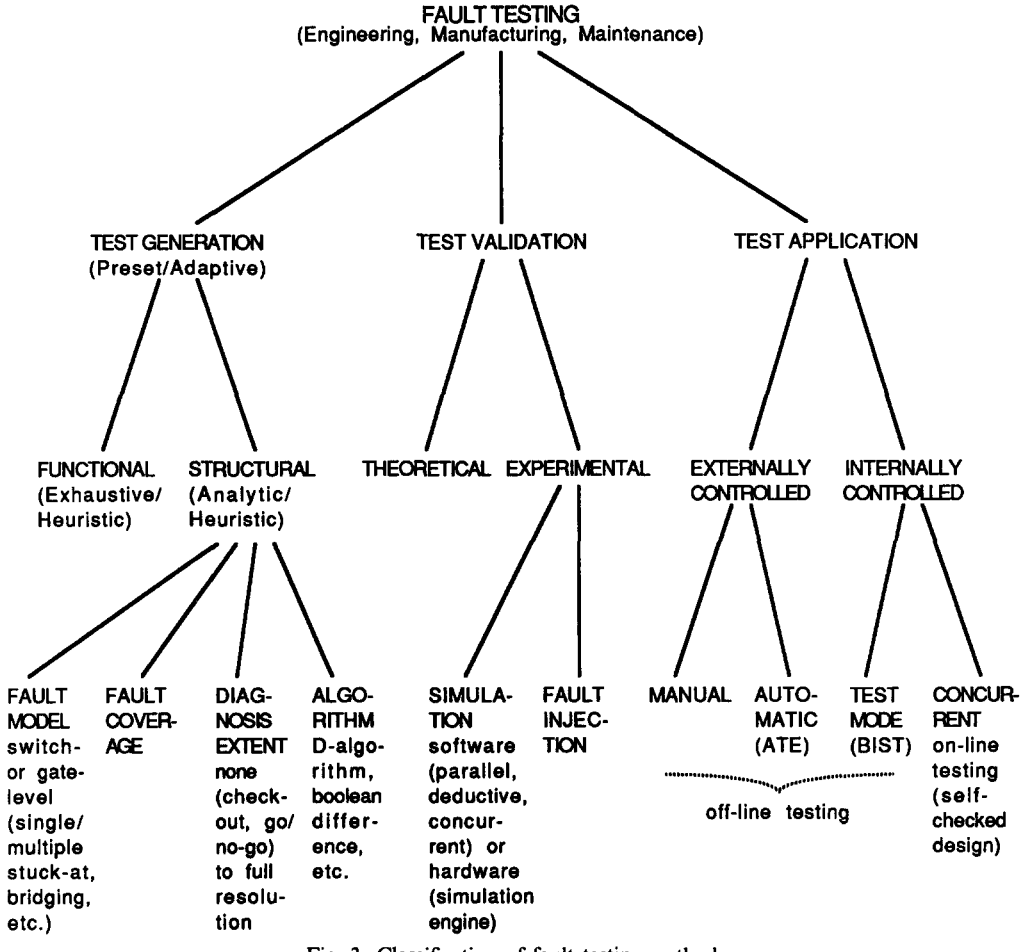


Fig. 3. Classification of fault testing methods.

4.3. Fault tolerance

My use of the term 'fault tolerance' is quite restricted in scope; this contrasts with some uses of the term to denote the entire field of dependable computing [62,74]. Fault tolerance is usually achieved through redundancy applied in hardware, software, or time domains. Hardware and software redundancy are of two types:

(1) static or masking redundancy to prevent faulty components from producing errors
(2) dynamic or standby redundancy to detect and circumvent faults before they lead to errors.

It is also possible to combine the two approaches in hybrid methods.

Masking redundancy techniques date back to the mid 1950s when Moore and Shannon [61] showed how to synthesize reliable contact networks using 'crummy' relays and von Neumann [93] proposed voting on multiple signal versions. Later, Tryon [92] introduced quadded logic, the simplest example of interwoven redundant logic, which takes advantage of the natural masking capability of AND and OR gates; and AND (OR) gate masks an incorrect 0 or 1 if at least one of its other inputs is 0(1). Practical applications of these methods remained quite limited [51] due to the high cost of redundancy with discrete components. The integrated-circuit technology brought with it the possibility of cost-effective redundant implementations. However, the proposed techniques were still not widely applicable for a different reason: The fact that independence of faults in the multiple copies of a component or circuit could no longer be guaranteed.

Redundancy techniques based on voting have found some applications [68,69]. In the simplest case, each circuit is triplicated with the three output signals reduced to the final output by a majority voter. Ignoring faults in the voter, the correct output is produced if at least two of the circuits function properly. If r_c is the probability of fault-free operation of a circuit independent of others, the reliability of the voted output is $R = r_c^3 + 3r_c^2(1 - r_c)$. Denoting the voter's reliability by r_v, the overall reliability is no lower than $r_v(3r_c^2 - 2r_c^3)$. Thus, reliability improvement is achieved iff:

$$\frac{3 - \sqrt{9 - 8/r_v}}{4} < r_c < \frac{3 + \sqrt{9 - 8/r_v}}{4}.$$

For example, if $r_v = 0.95$, we must have $0.56 < r_c < 0.94$ in order for the triplicated voting scheme to be more reliable than a single non-redundant circuit.

Extensions to the basic replicated voting arrangement with replication factor $N = 2n + 1$ include TMR or triple modular redundancy [57] and its generalization to NMR, which replicates the voters in an attempt to tolerate voter as well as circuit failures, use of adaptive voters [72] with a capability to adjust the weights assigned to input signals that have proved unreliable in the past, and hybrid redundancy [58], which combines the advantages of masking and standby redundancy techniques. Majority-voted redundancy has also been proposed for the tolerance of software imperfections. This technique will be dealt with in Subsection 6.3.

4.4. The fault-to-error transition

Transition from a faulty to erroneous state occurs when a fault affects the state of some storage element or output. Designers try to impede this transition by using fault tolerance methods. Another approach is to control this transition so that it leads to an incorrect but safe state. An example is the provision of internal fault detection mechanisms (e.g. comparators, activity monitors, or consistency checkers) that can disable a given module or system, assuming of course that the disabled state is safe.

Ironically, one may also try to facilitate this transition for the purpose of exposing system faults, since errors are more readily observable than faults. This is precisely the objective of all fault-testing schemes. With off-line test application methods (see Fig. 3), special input patterns are applied to the circuit or system under test, while observing possible errors in its outputs or internal state. To deduce underlying faults from observed errors, we need to establish a correspondence between various fault and error classes. This is referred to as error modeling and is discussed in Subsection 5.1. With on-line or concurrent test application, faults must be exposed during normal system operation and without disrupting its service. As such a self-checked mode of operation relies heavily on informational coding techniques, it is treated in Subsection 5.2.

5. ERROR-LEVEL OR INFORMATION-LEVEL VIEW

5.1. Causes and symptoms of errors

An error is any deviation of a system's state from the reference state as defined by its specification. Errors are either built into a system by improper initialization (e.g. incorrect ROM contents) or develop as a result of fault-induced deviations. Assuming that the system's state is encoded as a binary vector, an error consists of a set of $0 \rightarrow 1$ (read 0-to-1) and/or $1 \rightarrow 0$ inversions. With this view, errors can be classified according to the multiplicity of inversions (single vs multiple), their directions (symmetric if both $0 \rightarrow 1$ and $1 \rightarrow 0$ inversions are considered at the same time, asymmetric if, for example, the inversions can only be of the $1 \rightarrow 0$ type, and unidirectional if multiple inversions are of the same type), and their dispersion (random vs correlated). There are finer subdivisions in each category. For example byte errors and bursts confined to a number of adjacent bits are frequently studied instances of correlated multiple errors.

Actually, inversion is not the only type of error that one can consider. When there is *a proiri* or external information that a bit or group of bits may be erroneous or when an invalid symbol or signal is observed, the situation can be dealt with as an erasure error.

Error models reflect the information-level consequences of logic faults. They enable us to analyze error probabilities for given fault classes and distributions. In addition, error models help in the establishment of design techniques that force fault classes of interest to produce detectable and/or correctable errors. Whereas some faults may be transient or intermittent, I view errors as being permanent. Errors seldom disappear on their own and must be corrected before they result in a malfunction. Thus, in my model and terminology, there is no such thing as a transient or intermittent error. Rather, we have transient faults leading to errors in some state variables or outputs which are then either corrected by explicit invocation of the system's error correction and recovery mechanism or propagate to higher levels and are treated there.

5.2. Error detection and location

All error detection schemes use informational redundancy. The differences appear in the level at which redundancy is applied (bit-string/word level, data-structure level, algorithm/application level) and in the classes of errors dealt with.

Classical coding theory deals almost exclusively with the bit-string or word level. This is because coding theory was developed primarily for communication systems that are characterized by randomness and locality of errors (usually induced by random noise) and lack of reliance on structural properties and semantic contents of the encoded data (again reasonable because of generality requirements and secrecy concerns). Coding theory has provided a theoretical foundation and a vast repertory of error codes having particular detection/correction properties. Unfortunately, there is no unified theory to help a designer in selecting the best code for a given set of requirements. Thus, one must rely on detailed knowledge of various important code classes and subclasses (linear, cyclic, BCH, Reed-Solomon, Berger, etc.) and make the final selection on the basis of error detection capability, redundancy level, and encoding/decoding cost and delay.

Historically, the application of informational redundancy to computing systems has followed the path of classical coding theory [38,80]. Several developments in coding theory, such as byte-oriented codes, unidirectional error codes [55], and arithmetic error codes [9,79], can be directly attributed to the requirements of computing applications. Even though coding theory was born at about the same time as electronic digital computers [82], systematic application of informational redundancy at the higher levels of data structures and algorithms did not start until the 1980s.

If data manipulation circuits can be assumed to be much more reliable than storage and transmission facilities, the strategy shown in Fig. 4 can be used. This is in fact the approach taken in most current commercially available computers. For certain data manipulation functions, it may be possible to design circuits that accept encoded data and produce encoded results directly. Where this has not been possible with known error codes, computer designers have devised new ones (e.g. arithmetic error codes mentioned above). A significant innovation in this area is the concept of a totally self-checking (TSC) functional unit designed in such a way that its internal faults either do not affect the correct output or manifest themselves as invalid outputs [94]. Use of such circuits, with self-checking checkers and suitable recovery mechanisms, can lead to ultrareliable and resilient

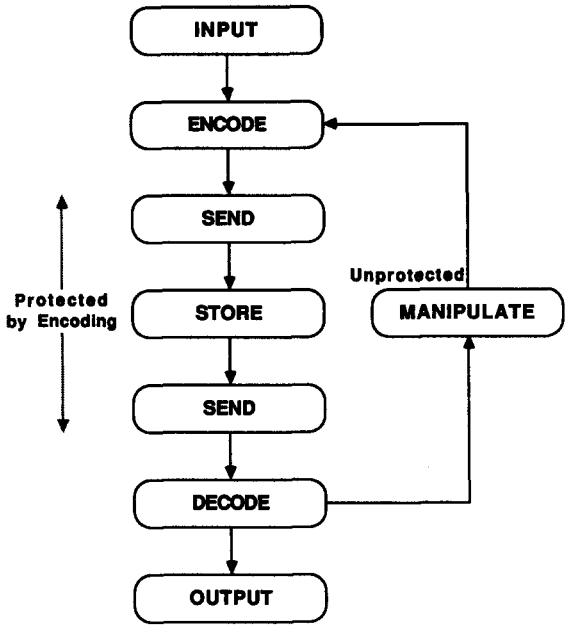


Fig. 4. A common way of using information coding techniques for dependable computing.

computing systems. An important application of error codes is in the design of malfunction-safe (or fail-safe in the prevailing terminology) sequential circuits (see Subsection 5.4).

At the data structure level, multiple words are encoded collectively to provide a higher degree of error protection at lower cost. Two simple examples are the provision of a checksum word at the end of a vector and the addition of row and column checksums to a matrix. Semantic information if available, can also be exploited for error detection. For example, the sorted order of a vector or symmetry of a matrix provide additional clues for error detection/correction. Concern for protecting structure data such as pointers has resulted in the investigation of robust data structures [88].

There are two approaches to error detection at the algorithm level. A fairly general language-based approach is the intermixing of annotations and assertions with program statements [56] for specifying expected data properties at certain execution points, so that any deviation can be detected. A second approach is the use of consistency checks guided by special properties of the algorithms being implemented. Techniques in this category are highly application-dependent. Published results include matrix operations [44] and hardware FFT processors [22,46].

5.3. Error tolerance

Error tolerance is achieved by the application of error-correcting codes (ECCs) or through an error detection/recovery procedure. ECCs are used extensively in data transmission and storage systems. They usually involve a high degree of informational redundancy if applied to short words. Thus, except for single-error-correcting codes used for semiconductor random-access memories, the use of such codes usually involves encoding of large blocks of data. An impressive application area for such ECCs is in digital recording on compact audio discs (detection/recovery is inapplicable due to the real-time nature of sound production). The technique employed relies on sophisticated, highly redundant ECCs leading to the tolerance of extensive information loss from scratches on the disc surface. Similar schemes have been proposed for optical computer disks [41].

The best know ECCs belong to the classes of linear separable codes (also referred to as systematic or separable parity check codes) and polynomial cyclic codes. A codeword in a linear separable code consists of a number of information bits and several separate check bits, each obtainable as a certain linear combination of the information bits. Linear separable codes contain the Hamming single-error-correcting codes as a special subclass. Polynomial cyclic codes, which overlap with

linear separable codes and also contain Hamming single-error-correcting codes as special cases, derive their name from the fact that they are based on polynomial algebra and that any cyclic (end-around) shift of their codewords are also codewords. Polynomial cyclic codes have relatively simple encoders and decoders that are based on linear feedback shift registers. Important subclasses of these codes are the BCH and Reed–Solomon codes.

The theory of ECCs is well developed [6,71] and a wide variety of such codes are known. ECCs can be classified in the same way as error-detecting codes. Here also, there is no unified theory to help the computer system designer with the selection process. The more a designer knows about various subclasses of error-correcting codes and their properties, the closer his or her choice is likely to be to the optimal design.

Recovery from errors detected at the bit-string/word level can take several forms. The simplest form is retransmission or recomputation of the erroneous data. The instruction retry facility of some computers is an example that is effective against errors caused by transient logic faults. Automatic hardward-controlled switchover to a standby circuit in the event of a detected error is a technique that has been suggested but not widely used. Instead, the switching normally takes place at the higher level of functional units (modules) if and when the error leads to a detectable malfunction. Because structural and semantic information are application-dependent, recovery from errors detected at the data structure or algorithm level is usually software-controlled. A recent development in this respect is that of data diversity [2] which in its simplest form is based on retrying an erroneous computation by providing it with a logically equivalent set of input data.

5.4. The error-to-malfunction transition

A system moves from erroneous to malfunctioning state when an error affects the functional behavior of some component subsystem. This transition can be avoided by using error tolerance techniques. An alternative is to control the transition so that it leads to a safe malfunction. This latter technique has been extensively applied to the design of malfunction-safe (fail-safe) sequential circuits. The idea is to encode the states of the sequential circuit in some error code, specify the transitions between valid states (represented by codewords) in the normal way, and define the transitions for erroneous states in such a way that they never lead to a valid state. Thus, an invalid state persists and is eventually detected by an external observer or monitor. In the meantime, the output logic produces safe values when the circuit is in an invalid state.

One may also try to facilitate this transition for the purpose of exposing latent system errors. For example, a memory unit containing incorrect data is in the erroneous state. It will operate correctly as long as the erroneous words are not read out. Systematic testing of memory can result in a memory malfunction that exposes the errors.

6. MALFUNCTION-LEVEL OR SYSTEM-LEVEL VIEW

6.1. Causes and symptoms of malfunctions

A malfunction is any deviation of a system's operation from its expected behavior according to the design specifications. For example, an arithmetic/logic unit computing $2 + 2 = 5$ can be said to be malfunctioning, as is a processor executing an unconditional branch instead of a conditional one. Malfunctions (like defects, faults, and errors) may have no external symptoms, but they can be made externally observable with moderate effort. In fact, malfunction detection (complemented by a recovery mechanism) constitutes the main strategy in the design of today's general-purpose dependable computer systems. Even though such systems are called fault-tolerant in the prevailing terminology [25], I will use the adjective 'malfunction-tolerant' for consistency. Many such systems are built from standard off-the-shelf building blocks with little or no fault and error handling capabilities and use higher-level hardware and software techniques to achieve malfunction tolerance at the module or substystem level.

Malfunctions can be classified as being related to timeliness or correctness of the subsystem responses. Malfunctions of the first type are easier to deal with (e.g. using a time-out mechanism and rescheduling the computation on a different processor in the event of unacceptable delays). In fact, it has been suggested that dependability assurance would become much simpler if the processors were of the malfunction-stop (fail-stop) type [81]; i.e. they either produce correct

responses or produce no response at all. It is worth noting that the malfunction-stop property is actually a special case of the malfunction-safe mode of operation where the lack of output is the only safe output state. In contrast to this highly controlled malfunction mode, arbitrary or Byzantine malfunctions [27] are extremely difficult to handle, even for fairly trivial computations.

Like faults, malfunctions can be classified as being transient, intermittent/recurring, or permanent. Permanent malfunctions (e.g. a module producing incorrect results for every input) are very rare and a recurring malfunction is of course easier to detect than a transient one.

6.2. Malfunction detection and diagnosis

There are two methods for detecting malfunctions: internal monitoring and external observation. Internal monitors are built-in checkers that detect various irregularities and produce an external 'malfunction' indication. To implement this scheme with standard off-the-shelf components and a small amount of specialized hardware, one can use straight (non-reconfigurable) duplication. For example, the Univac 1100/60 has duplicated subprocessors communicating over a pair of buses. At the end of each microcycle, values on the buses are compared to identify a possible processor malfunction [18]. In this scheme, the two subprocessors and the added monitoring hardware can be considered as parts of a single processor with internal malfunction detection capability.

As for external observation, we distinguish two malfunction detection strategies: concurrent checking and periodic diagnosis. With concurrent checking, external monitors check the outputs of various subsystems for reasonableness and consistency. In a bus-based system, monitors may be attached to system buses in order to check all inter-module data transfers. In software systems, external monitors may take the form of acceptance tests performed on various results. The design of reasonableness checks and acceptance tests is problem-dependent, with time and cost overheads varying greatly from one application to another. For example, it is relatively easy to validate the computed inverse of a matrix by performing matrix multiplication. It is much harder to authenticate the result of a scheduling or simulation routine with the same level of confidence.

The periodic diagnosis version of external monitoring has received much attention, but practical applications lag far behind the strong theoretical developments. In this approach, each module is periodically subjected to special tests. The test, which is usually a built-in self-test procedure, is initiated by an external module and its results are evaluated by the same or a different external module. The theory of external malfunction diagnosis (or system-level fault diagnosis [48]) is based on a directed graph model of module testing capabilities (i.e. there is an arc from module i to module j if module i can test module j). Practical applications are much more modest and do not require sophisticated design and analysis theories. For example, the Japanese COMTRAC railroad traffic computer [45] runs identical diagnostic programs on two CPUs. Each program exercises most parts of the CPU in the course of computing a single value. The two values obtained are then compared to a stored constant for determining the malfunctioning CPU.

The COMTRAC approach, which is similar to that of Bell System's local ESS processors [90], is known as reconfigurable duplication, with its pessimistic reliability equation being

$$R = r_s[r_m^2 + 2cr_m(1 - r_m)]$$

where r_s is the reliability of the reconfiguration (switching) mechanism and c is the detection coverage factor (the probability of detecting a malfunction). This reliability is better than r_m iff:

$$r_m < \frac{2c - 1/r_s}{2c - 1}.$$

For $r_s = c = 0.9$, we need $r_m < 0.86$ for reconfigurable duplication to be more reliable.

6.3. Malfunction tolerance

The two approaches to malfunction tolerance are similar to the static and dynamic fault tolerance methods discussed in Subsection 4.3. The static approach is usually based on some form of voting. For example, three processors may be used and the result taken to be the one provided by at least two processors. A technique known as N-version programming applies this same principle to the tolerance of software malfunctions due to residual design slips (more on this later). Three problems

arise when applying replication at this level. First, it is quite difficult (some would even say impossible) to assure malfunction independence [31,49]. Second, synchronization is more difficult compared to voting at the logic circuit level. Third, it is hard to decide on meaningful comparisons without wasting time on high data volumes; e.g. one unit may be declared the master and if it agrees on some critical values with another unit, its results are assumed correct, thus avoiding time-consuming votes on all computed values.

The dynamic approach to malfunction tolerance is based on malfunction detection and recovery. As a simple hardware example, memory module malfunctions in a bit-slice or byte-slice memory can be tolerated if a shadow module whose words are the XOR of the contents of all other modules is provided. When a module malfunctions, it is isolated and its contents reconstructed from the contents of the other modules [84, p. 157]. Software-oriented examples abound in the field of distributed computing [25] where data consistency and atomicity of actions must be guaranteed in the face of interleaved execution of transaction steps and the possibility of malfunctions in participating sites. Useful tools and techniques such as atomic broadcasting, agreement and commit protocols, synchronization, global state determination, and various other distributed algorithms have been devised over the past few years for particular classes of malfunctions and system topologies. This is still a very active research area.

It has been suggested that by using replicated modules with diverse designs, the protection against random malfunctions will extend to related or common-cause malfunctions and design slips. Systematic use of design diversity in computing systems started with the software-related concepts of recovery blocks and N-version programming [14]. A recovery block [77,78] has the structure;

ensure	*acceptance test,*	e.g. sorted list?
by	*primary module,*	e.g. quicksort
else by	*first alternate,*	e.g. bubblesort
⋮		
else by	*last alternate,*	e.g. insertion sort
else error		

Problems with implementing recovery blocks include the coverage and trustworthiness of the acceptance test as well as a suitable checkpointing scheme to enable state restoration for error recovery. For N-version software, assuring independence of failures, consistency of multiple results, and proper functioning of the required inexact voting scheme are major challenges [12,20]. Several modern safety-critical systems use a combination of replicated diverse hardware and software modules (e.g. the Airbus A-310 and the Boeing 737/300 flight control systems).

6.4. The malfunction-to-degradation transition

The recovery block example in Subsection 6.3 shows how a software malfunction can lead to a degradation in performance. In the case of detected hardware malfunctions, the bad unit can be isolated from the rest of the system. This reduces the amount of hardware resources (memory, processing power, I/O bandwidth, etc.) available to computations, thus leading to a corresponding performance degradation. In fact in the absence of special monitoring facilities, a degraded performance (e.g. delayed responses) may be the first indication of the underlying impairments. Provision of backup resources can postpone this transition, as can the ability to replace or repair the malfunctioning modules without having to shut down or otherwise disturb the system. Whereas overall system performance may be degraded, individual users or processes need not experience the same level of service degradation (e.g. due to reassessment of priorities). In fact when degradation is severe, only critical computations may be allowed to run.

7. DEGRADATION-LEVEL OR SERVICE-LEVEL VIEW

7.1. Causes and effects of degradations

I have defined a dependable computer system as one producing trustworthy and timely results. Actually, neither trustworthiness nor timeliness is a binary (all or none) attribute. I have already mentioned that results may be incomplete or inaccurate rather than totally missing or completely

wrong and they may be tardy enough to cause some inconvenience, without being totally useless or obsolete. Thus, various levels of inaccuracy, incompleteness, and tardiness can be distinguished and those that fall below particular thresholds might be viewed as degradations rather than failures. A computer system that is organized such that module malfunctions usually lead to degradations rather than to failures is gracefully degrading or fail-soft and is usually a multiprocessor system.

Degradations occur in many different ways. A byte-sliced arithmetic/logic unit might lose precision if a malfunctioning slice is bypassed through recofiguration logic (inaccuracy). A dual-processor real-time system with one malfunctioning unit might choose to ignore less critical computations in order to keep up with the demands of time-critical events (incompleteness). A malfunctioning disk drive in a transaction processing system can effectively slow down the system's response (tardiness). These are all instances of degraded performance. Performance, of course, is relatively hard to define and even harder to quantify. Meyer [59] summarizes the arguments eloquently (remember that faults/failures in the following quotation are malfunctions in my terminology):

> "Evaluations of computer performance and computer reliability are each concerned, in part, with the important question of computer system 'effectiveness', ... performance evaluations (of the fault-free system) will generally not suffice since structural changes, due to faults, may be the cause of degraded performance. By the same token, traditional views of reliability (probability of success, mean time to failure, etc.) no longer suffice since 'success' can take on various meanings and, in particular, it need not be identified with 'absence of system failure' ".

Meyer then proceeds to suggest performability as a measure that encompasses performance and reliability and that constitutes a proper generalization of both.

7.2. Degradation management

Assuming that a malfunction is correctly identified, the malfunctioning subsystem is effectively isolated, and the reconfiguration and recovery processes are successful, several other steps are still necessary for the system to be able to function in a degraded mode. Figure 5 provides a basis for the following discussion of performance variations with time in three types of systems:

S_1: A fail-hard system with performance P_{max} up to the failure time t_1 and after repair at t'_1.
S_2: A fail-soft system with degrading performance and off-line repair upon failure at time t_2.
S_3: A fail-soft system with on-line repair which postpones its failure time to t_3.

In a fail-hard system, the first malfunction leads to a failure. However, this malfunction may occur

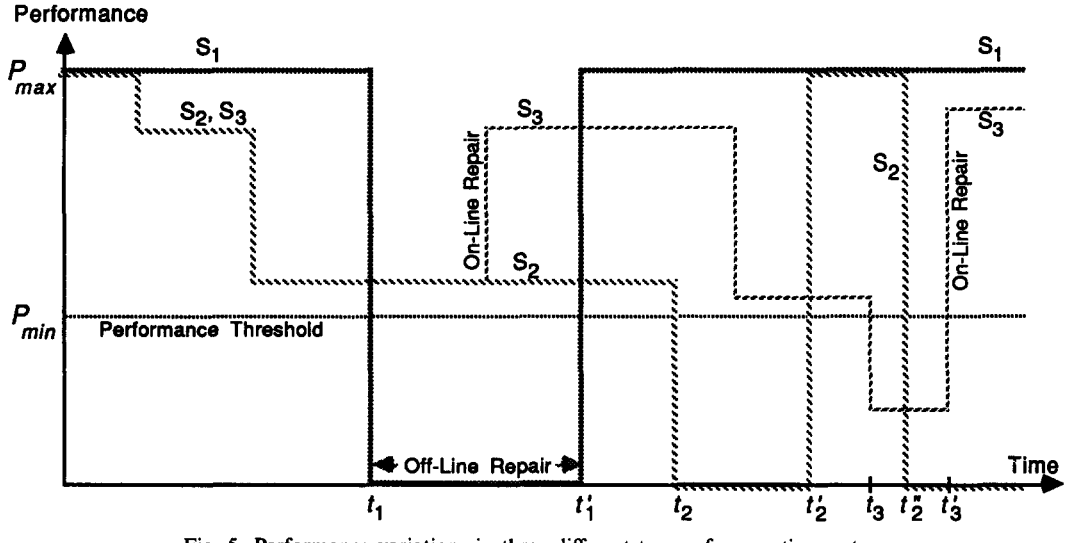


Fig. 5. Performance variations in three different types of computing systems.

at a later time compared to an equivalent fail-soft system due to lower complexity. In a fail-soft system, some malfunctions lead directly to system failure (as shown for S_2 at time t''_2). But many malfunctions only lead to performance degradations. If the system's design is such that malfunctioning modules can be isolated and removed for replacement or repair without disrupting the operation of other modules, then system failure may be postponed and its down time greatly reduced. Such an on-line repair capability is essential for high-availability systems.

By degradation management, I mean the capacity of a system to go smoothly from one performance level to a lower one when a malfunction is detected and properly isolated or to a higher one when a malfunctioning subsystem is replaced. The two transitions are similar as they both relate to the ability to function with a fluctuating pool of resources, when individual resources go in and out of service at essentially unpredictable times. Due to the complex decision and recovery mechanisms, degradation management is usually a software task that is built into the operating system.

The malfunction detection phase was discussed in Subsection 6.2. Assuming that the detected malfunction is not tolerable, the degradation management routine takes over and:

(1) isolates the malfunctioning unit and updates the resource availability information
(2) determines the affected processes and assesses the damage for each
(3) to the extent possible, recovers data and state information from the dead module
(4) initiates appropriate recovery algorithms for each of the affected processes.

Because degradation management can be applied with relatively low hardware and software overheads, it has found its way into many modern computer systems. Examples include [85]:

(1) deactivating malfunctioning processors, channels, and I/O devices
(2) avoiding bad tracks on disk, contaminated files, and noisy communication lines
(3) removing part of main memory through virtual address mapping facilities
(4) bypassing a malfunctioning cache memory (VAX-11/750)
(5) blocking half of cache via restricted mapping in the good half (VAX-11/780, Univac 1100/60).

Occasionally, in the course of recovery from a malfunction, global reconfiguration may be required. This can be accomplished by starting from a minimal working configuration that is known to be trustworthy and systematically expanding to the optimal configuration with the available resources.

7.3. Degradation tolerance

Degradation tolerance has a system component and an application component. Clearly, a gracefully degrading system must be structured so that it can function with a variable pool of resources. This system component is usually built into the operating system functions which must be configuration-independent and table-driven. A degradation is tolerated by the system if Step 4 in the degradation management procedure (discussed in Subsection 7.2) results in complete recovery for all of the affected processes, with performance remaining at an acceptable level after the recovery. In a real-time environment, the speed of recovery actions is critical and extensive hardware support for such actions becomes mandatory. In most practical cases, partial degradation tolerance is more cost-effective. Only the more critical processes may be scheduled for recovery and continued execution, while other processes may receive reduced or no support (with appropriate warnings). In such a case, the system operates in a mixed degradation-tolerant/malfunction-safe mode.

The application component of degradation tolerance is equally important. Application programs may be written such that they tolerate temporary reductions in computation accuracy and/or speed. For example, less important subcomputations may be bypassed, certain periodic computations performed less often, or double-precision computations replaced by single-precision ones. In many cases, it may be possible to write application programs in such a way that their successful completion does not depend on the availability of any particular resource. This is best illustrated by means of an example [3]. Consider an aircraft collision detection computation that is to be performed periodically based on radar data. The application software may be written in such a

way that if in a given cycle radar data is not available by a certain time, then an approximate computation based on the last available position and speed data is performed. In a similar way, the recovery-block scheme can become a degradation tolerance technique if the alternate modules are written in such a way that they require fewer resources for their successful execution.

7.4. The degradation-to-failure transition

A failure occurs when the system's degradation tolerance capacity is exhausted and, as a result, its performance falls below an acceptable threshold. As degradations are themselves consequences of malfunctions, it is interesting to skip a level and relate system failures directly to malfunctions. It has been noted that failures in a gracefully degrading system can be attributed to:

(1) isolated malfunction of a critical subsystem
(2) occurrence of catastrophic (multiple space-domain) malfunctions
(3) accumulation of (multiple time-domain) malfunctions beyond detectability/tolerance
(4) resource exhaustion causing inadequate performance or total shutdown.

Analysis of the PRIME gracefully degrading timesharing system (developed at Berkeley in the early 1970s [19] showed that the first two items, i.e. intolerable malfunctions, are the most common causes of system failures; this conclusion has since been reinforced by other studies. In this context, a degradation is almost good news in that the mere fact of its occurrence means that the highest danger of failure has passed! Thus, minimizing the number and size of critical subsystems (the hard core) and providing strong protection against catastrophic common-cause malfunctions are important requirements for recovery and continued operation in a degraded mode.

8. FAILURE-LEVEL OR RESULT-LEVEL VIEW

8.1. Causes and effects of failures

The causes and effects of computer system failures can be analyzed by maintaining failure or crash logs. One system for which extensive crash data has been reported is the C.mmp multiprocessor of Carnegie-Mellon University [84]. It is generally acknowledged that human errors account for a significant fraction of failures and that a majority of system-related crashes are attributable to software. Studies of human reliability [15,70] are thus quite relevant to the discipline of dependable computing and dependability assurance for software must be emphasized. A form of design diversity is occasionally used for avoiding 'infant mortality' failures that result from serious design slips: several contractors may be asked to build prototypes of a critical system, with the best implementation then selected by rigorous testing under realistic conditions.

Following Leveson [54], I note that failures (or hazards in her safety-related terminology) are characterized by severity and probability. The expected loss or risk associated with a particular failure is a function of both its severity and its probability. Dependability procurement techniques used in designing a computer system depend on the acceptable risk level which varies from one application area to another. Factors causing such variations are numerous and can be classified as:

(1) economic; expected losses versus cost of reducing the risk
(2) functional; required capabilities that cannot be compromised and are in conflict with safety
(3) psychological; e.g. people willing to accept higher risks on highways than in the air
(4) political; e.g. acceptable risk level lowered immediately after a major disaster.

Given that complete failure avoidance or tolerance is not feasible, the next best thing is to be prepared for the inevitable, as failures tend to cause much greater damage when they take us by surprise. Therefore, a realistic dependability evaluation scheme, that accurately predicts the failure probability or that at least establishes an upper bound for it, is particularly important.

8.2. Failure detection and confinement

A prerequisite for containing the damage caused by a failure is the ability to detect it promptly. In this sense, a catastrophic failure with some obvious external indication (e.g. smoke or explosion)

may be preferable to a subtle one that is not immediately noticeable. Failure detection is generally achieved by some form of acceptance test applied by the user. In a way, we can view a computer system as simply a module in a high-level recovery-block architecture. When the outputs or actions resulting from a computer system are unacceptable, recovery is initiated and alternates (manual operation, warning system, etc.) are tried. In fact, the larger system, of which the computer is a module, may similarly experience degraded performance and possibly failure.

Because recovery from computer failures necessarily involves human intervention, and human reaction tends to be slow and at times unreliable, an additional requirement for the detectable failure condition in safety-critical systems is that it be safe. On the hardware side, much of the work reported pertains to fail-safe sequential logic circuits (e.g. [23]) and extension of these ideas to higher system levels needs further work. Safety aspects of computer software are surveyed by Leveson [54]. Goldberg [40] argues for the application of safety engineering to computer systems in general and to software in particular. Here is his compilation of safety engineering principles:

(1) using barriers and interlocks to constrain the access to critical resources or states
(2) performing critical actions incrementally rather than in a single step
(3) dynamically modifying system goals so as to avoid or mitigate the damages of a hazard
(4) managing resources needed to deal with crises so that enough will be available in an emergency
(5) exercising all the critical functions and safety features to determine their viability
(6) designing the operator interface to provide the information and power to deal with exceptions
(7) defending the system against malicious attacks.

The last item above presents particularly challenging design problems and is the main source of concern in certain weapons and defense systems. The operator (or more generally, the human) interface is quite important, particularly where the credibility and informativeness of warning messages are concerned. Experience with conventional disaster warning systems (storms, floods, fires, etc.) indicates that human beings tend to disbelieve and ignore warning messages. Clearly:

> "No warning system will function effectively if its messages, however logically arrived at, are ignored, disbelieved, or lead to inappropriate action" [35].

The paper cited above presents guidelines for designers of disaster warning information systems. These guidelines are equally applicable to failure warnings in a safety-critical computer system.

I have defined a failure as an impairment for individual results/or computations. It is therefore important to protect other computations that may be in progress in a computer system when a failure is detected. This is referred to as failure confinement. Various approaches are possible depending on the application. In a safety-critical system, one may decide to halt all computations in the event of a detected failure in order to initiate a complete system checkout. At the other extreme, computations may be allowed to continue as if nothing had happened, hoping that the failure was an isolated event with no side effects. In the case of communicating processes, a reasonable middle ground is to be suspicious of all processes that have sent or received data to/from the failed process. This strategy sometimes causes a chain reaction in that the suspicious processes may have communicated with other processes, which in turn may have communicated with others, etc. It is thus important to coordinate the checkpointing strategy with the process intercommunication pattern to avoid having to restart everything.

8.3. Failure tolerance

Many malfunction tolerance techniques can be applied at this higher level to achieve failure tolerance. In fact the distinction between malfunctions and failures is sometimes blurred in that, for example, abnormal behavior of a computer in a multicomputer or distributed system can be labeled a computer failure of a subsystem malfunction, depending on our point of view.

The oldest method of failure tolerance is failure detection followed by rollback or restart of the failed computation (possibly on a backup system). This is clearly not easy to implement. First, failure detection is never perfect and second, even with perfect failure detection, not all processes

are reversible so that they can be rolled back and their effects undone. This method is essentially a manually-controlled recovery block scheme. Process outputs or actions are subjected to fuzzy acceptance tests by human observers. When tests indicate a failure, expert judgement is used to select the best course of recovery actions (e.g. rollback to a checkpoint). Many computations or transactions may be declared suspect, with manual or automatic verification requested for them.

The effectiveness of the above failure tolerance method is application-dependent. For example, a writer will have little trouble resorting to a cut-and-paste style of editing based on the last hard-copy of a manuscript in the event of a failed word processor. A pilot may be more inconvenienced by a failed instrumentation computer but can usually bring the aircraft to a safe landing. A spacecraft may be permanently lost if the control computer for its propulsion system fails. The ability to detect the failure may not be of much value in this last case. In most practical cases, partial failure tolerance may be provided, with the higher-level physical or organizational system operating in a mixed failure-tolerant/fail-safe mode. Thus, failures are detected as usual, but not all processes are treated identically afterwards. Some processes are transferred to a backup system while others may be inactivated until the system is repaired or is otherwise judged fit to resume its operation.

8.4. The failure-to-disaster transition

A tolerated failure, by definition, has negligible negative side effects. Also a detected failure does not lead to a disaster unless there is something fundamentally wrong with the larger (physical, societal, or corporate) system into which the computer system is incorporated. Disasters, therefore, are primarily caused by undetected failures. The best protection against such disasters is to acknowledge the fact that a computer failure could go undetected and to have an accurate estimate, or at least a safe upper bound, for its probability. Thus, dependability measures are needed not only because they allow us to compare competing systems for particular dependability and safety goals, but also because they prevent unjustified or misguided confidence in computation results.

However, knowing the failure probability is not enough by itself. Humans are generally weak in properly interpreting small probabilities. It is thus necessary for standards to be set and computer system quality to be monitored in safety-critical systems in much the same way as other safety-critical system components in construction, transportation, and nuclear power industries. It is also necessary to give due attention to the training of operators for safety-critical computer systems so that they react properly to emergency conditions and do not underestimate the seriousness of observed deviations from the normal mode of operation.

Two sources of computer system failures, namely malicious attacks and misoperation, are notoriously difficult to deal with. Systems that have special safeguards against these sources of failures are sometimes described as tamper-proof and foolproof, respectively. In the first arena, considerable progress has been made over the past two decades but somehow the attackers always seem to be a step ahead (e.g. computer viruses). In the second arena, 'fools' appear to be quite inventive in devising new ways to throw off supposedly foolproof systems. The inability of our most ingenious designers to make even the simplest systems completely foolproof is very alarming.

9. CONCLUSIONS

I have provided a unified framework for the study of dependable computing and have surveyed the field in light of the proposed framework. My multi-level model clearly exposes the fact that dependability assurance techniques can be applied at several levels to supplement and strengthen one another. Such a multi-level view provides a better understanding of both the impairments to dependability and the consequences of undependability. Over the years, I have observed a shift of emphasis from dependability procurement techniques at the lower levels of my model (defects, faults, errors) to the higher levels (malfunctions, degradations, failures). Therefore, I expect dependable computers of the future to use avoidance and removal techniques at the lower levels in combination with tolerance techniques at the higher levels to achieve their goals. This, however, does not mean that researchers will become disinterested in the lower levels altogether. Much work needs to be done in the areas of defect and fault modeling and interest will probably remain high in the use of defect and fault tolerance techniques as methods for VLSI yield enhancement.

In spite of significant progress in the field of dependable computing over the past three decades, the field is still full of interesting and challenging open problems and much work remains to be done. Dependability enhancement features will be routinely incorporated in the design of future computer systems. Thus, an important question facing the designers of such systems will be how to use a multitude of seemingly unrelated techniques in an optimal and coherent fashion to achieve given dependability goals. Methodologies for guiding the designers of hardware and software systems in their search for useful tools and strategies in a vast space of dependability procurement techniques are badly needed. I hope that the framework proposed in this paper will contribute to clear formulation of such medthodologies and their areas of applicability.

It is my sincere belief that dependable computing can no longer be treated as a separate discipline. Rather, the useful techniques of the field must be systematically incorporated into the other areas of computer science and engineering (e.g. testability into logic design, reconfiguration strategies into computer architecture, performability analysis into system evaluation methodologies, etc.). Clearly, this change of view must start with educational programs. Taking an isolated advanced course on dependable computing encourages our future computing professionals to think of dependability as an add-on feature. Industrial quality control suffered from a similar problem until it was realized that quality control and the rest of manufacturing processes are inseparable.

Acknowledgement—The research reported in this paper was initiated in 1987 at the University of Waterloo, where the author was a Visiting Professor supported in part by the Natural Sciences and Engineering Research Council of Canada under Grant Nos G1140 and A5515.

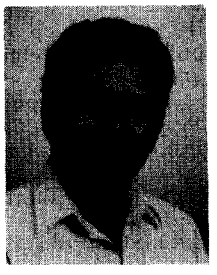
REFERENCES

1. J. A. Abraham and W. K. Fuchs, *Fault and Error Models for VLSI* pp. 639–654, in [76].
2. P. E. Ammann and J. C. Knight, *Data Diversity: An Approach to Software Fault Tolerance*, pp. 418–425, in [91] (1988).
3. T. Anderson and P. A. Lee, *Fault Tolerance: Principles and Practice*. Prentice–Hall, Englewood Cliffs, N.J. (1981).
4. T. Anderson and P. A. Lee, *Fault Tolerance Terminology Proposals*. pp. 29–33, in [36] (1982). Also in [83], pp. 6–13.
5. T. A. Anderson, *Resilient Computing Systems*. Wiley, New York (1986).
6. B. Arazi, *A Commonsense Approach to the Theory of Error-Correcting Codes*. The MIT Press, Cambridge (1987).
7. J. E. Arsenault and J. A. Roberts (Eds), *Reliability and Maintainability of Electronic Systems*. Computer Science Press (1980).
8. A. Avizienis, Design and fault-tolerant computers. *AFIPS Conf. Proc.* **31,** 733–743 (1967).
9. A. Avizienis, *Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design*, pp. 1322–1331 in [91], 1971. Reprinted in [84], pp. 671–686.
10. A. Avizienis, Fault tolerance and fault intolerance: complementary approaches to reliable computing. *Proc. Int. Conf. Reliable Softw.*, pp. 458–464. Los Angeles (1975).
11. A. Avizienis, *The Four-Universe Information System Model for the Study of Fault Tolerance*, pp. 6–13 in [36] (1982). Reprinted in [89], pp. 27–33.
12. A. Avizienis, The N-version approach to fault-tolerant software. *IEEE Trans. Softw. Engng* **11,** 1491–1501 (1985).
13. A. Avizienis, H. Kopetz and J.-C. Laprie (Eds), *The Evolution of Fault-Tolerant Computing (Dependable Computing and Fault-Tolerant Systems)*, Vol. 1. Springer, New York (1987).
14. A. Avizienis, Software fault tolerance. *Proc. IFIP Congr.* San Francisco (1989).
15. R. W. Bailey, *Human Error in Computer Systems*. Prentice–Hall, New York (1983).
16. B. Beizer, *Software System Testing and Quality Assurance*. Van Nostrand, Princeton (1984).
17. B. K. Bhargava (Ed.), *Concurrency Control and Reliability in Distributed Systems*. Van Nostrand, Princeton (1987).
18. L. A. Boone, H. L. Liebergot and R. M. Sedmack, *Availability, Reliability, and Maintainability Aspects of the Sperry Univac* 1100/60, pp. 3–9, in [36] (1980). Reprinted in [84], pp. 423–433.
19. B. R. Borgerson, *A Fail-Softly System for Time-Sharing Use*, pp. 89–93, in [36] (1972).
20. S. S. Brilliant, J. C. Knight and N. G. Leveson, The consistent comparison problem in N-version software. *ACM Softw. Engng Notes* **12,** No. 1, 29–34 January (1987).
21. W. C. Carter, *A Time for Reflection*, p. 41, in [36] (1982).
22. Y.-H. Choi and M. Malek, A fault-tolerant FFT processor. *IEEE Trans. Computers* **37,** 617–621 (1988).
23. H. Chuang and S. Das, Design of fail-safe sequential machines using separable codes. *IEEE Trans. Computers* **27,** 249–251 (1978).
24. *Computer*, Special Issues on Fault-Tolerant Computing, Vol. 4, No. 1 (1971), Vol. 13, No. 3 (1980), Vol. 17, No. 8 (1984), Vol. 23, No. 7 (1990).
25. F. Cristian, Understanding fault-tolerant distributed systems. *Commun. ACM* **34,** 56–78 (1991).
26. DCCA, *Proc IFIP Work. Conf. Depend. Comput. Critic. Applic.* Santa Barbara, Calif., August (1989); Tuscon, Arizona, February (1991); Italy, August (1992); San Diego, Calif., January (1994).
27. D. Dolev, L. Lamport, M. Pease and R. Shostak, *The Byzantine Generals*, pp. 348–369, in [17].
28. R. W. Downing, J. S. Nowak and L. S. Tuomenoska, No. 1 ESS maintenance plan. *Bell Syst. Tech. J.* **43,** 1961–2019 (1964).
29. R. F. Drenick, The failure law of complex equipment. *J. Soc. Indust. Appl. Math.* **8,** 125–149 (1960).
30. R. Dunn and R. Ullman, *Quality Assurance for Computer Software*. McGraw–Hill, New York (1982).
31. D. E. Eckhardt and L. D. Lee, A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Trans. Soft. Engng* **11,** 1511–1517 (1985).

32. EJCC, *Proc. Eastern Joint Computer Conf. (Information Processing Systems–Reliability and Requirements)*. Washington, D. C. (1953).
33. R. M. Fasano and A. G. Lemack, A quad configuration–reliability and design aspects. *Proc. Nat. Symp. Reliab. Qual. Cntrl*, pp. 394–407. Washington, D.C. (1962).
34. E. M. Forster, The machine stops. In *The eternal moment* (collection of short stories). Harcourt, New York (1928).
35. H. D. Foster, Disaster warning systems: learning from failure. *Information Systems: Failure Analysis* (Edited by J. A. Wise and A. Debons), pp. 3–13. Springer, New York (1987).
36. FTCS, *Proc. Int. Symp. Fault-Tolerant Comput.*, held since 1971 (in June, unless noted). Pasadena, Calif., March (1971); Newton, Mass. (1972); Palo Alto, Calif. (1973); Champaign, Ill. (1974); Paris (1975); Pittsburgh (1976); Los Angeles (1977); Tolouse, France (1978); Madison, Wis. (1979); Kyoto, Japan, October (1980); Portland, Me (1981); Santa Monica, Calif. (1982); Milano, Italy (1983); Kissimmee, Fla (1984); Ann Arbor, Mich. (1985); Vienna, Austria, July (1986); Pittsburgh, July (1987); Tokyo (1988); Chicago (1989); Newcastle upon Tyne (1990); Montreal (1991); Boston, July (1992); Tolouse, France (1993); Austin, Tex. (1994).
37. H. Fujiwara, *Logic Testing and Design for Testability*. MIT Press, Cambridge (1985).
38. H. Fujiwara and D. K. Pradhan, *Error-Control Coding in Computers*, pp. 63–72 in [24] (1990).
39. L. H. Goldstein, Controllability/observability analysis of digital circuits. *IEEE Trans. Circ. Syst.* **26,** 685–693 (1979).
40. J. Goldberg, Some principles and techniques for designing safe systems. *ACM Softw. Engng Notes* **12,** No. 3, 17–19 July (1987).
41. S. W. Golomb, Optical disk error correction. *Byte* No. 5, May (1986).
42. M. H. Halstead, *Elements of Software Science*. Elsevier, Amsterdam (1977).
43. J. E. Hosford, Measures of dependability. *Ops. Res.* **8,** 53–64 (1960).
44. K. H. Huang and J. A. Abraham, *Algorithm-Based Fault Tolerance for Matrix Operations*, pp. 518–528, in [91] (1984).
45. H. Ihara, K. Fukuoka, Y. Kubo and S. Yokota, *Fault-Tolerant Computer System with Three Symmetric Computers*, pp. 1160–1177, in [75].
46. J.-Y. Jou and J. A. Abraham, Fault-tolerant FFT networks. *IEEE Trans. Computers* **37,** 548–561 (1988).
47. J. A. Katzman, *A Fault-Tolerant Computing System*. Tandem Computers, Cupertino, Calif. Reprinted in [84], pp. 435–452 (1977).
48. C. R. Kime, *Systems Diagnosis*, pp. 577–632, Chap. 8 in [74].
49. J. C. Knight and N. G. Leveson, An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Trans. Softw. Engng* **12,** 96–109 (1986).
50. I. Koren and A. D. Singh, *Fault Tolerance in VLSI Circuits*, pp. 73–83, in [24] (1990).
51. R. E. Kuehn, Computer redundancy: design, performance, and future. *IEEE Trans. Reliabil.* **18,** 3–11 (1969).
52. J.-C. Laprie, *Dependability: A Unifying Concept for Reliable Computing*, pp. 18–21, in [36] (1982).
53. J.-C. Laprie, *Dependable Computing and Fault Tolerance: Concepts and Terminology*, pp. 2–11, in [36] (1985).
54. N. G. Leveson, Software safety in embedded computer systems. *Commun. ACM* **34,** 34–46 (1991).
55. D. J. Lin and B. Bose, *Theory and Design of t-Error Correcting and d(d > t)-Unidirectional Error Detecting (t-EC d-UED) Codes*, pp. 433–439, in [91] (1988).
56. D. C. Luckham and F. W. von Henke, An overview of ANNA, a specification language for Ada. *IEEE Softw.* **2,** 9–22 (1985).
57. R. E. Lyons and W. Vanderkulk, The use of triple modular redundancy to improve computer reliability. *IBM J. Res. Dev.* **6,** 200–209 (1962).
58. F. P. Mathur and A. Avizienis, Reliability analysis and architecture of a hybrid-redundant digital system: generalized triple modular redundancy with self-repair. *AFIPS Conf. Proc.* **36,** 375–383 (1970).
59. J. F. Meyer, On evaluating the performability of degradable computing systems. *IEEE Trans. Computers* **29,** 720–731 (1980).
60. E. F. Moore and C. E. Shannon, Reliable circuits using less reliable relays. *J. Frankl. Inst.* **262,** No. 3, pp. 191–208 (1956) and No. 4, pp.281–297 (1956).
61. W. R. Moore, *A Review of Fault-Tolerant Techniques for the Enhancement of Integrated Circuit Yield*, pp. 684–698, in [76].
62. V. P. Nelson and B. D. Carroll (Eds), *Tutorial: Fault-Tolerant Computing*. IEEE Computer Society Press (1987).
63. D. Pantic, Benefits of integrated circuit burn-in to obtain high reliability parts. *IEEE Trans. Reliabil.* **35,** 3–6 (1986).
64. B. Parhami, Errors in digital computers: causes and cures. *Austral. Computer Bull.* **2,** No. 2, 7–12 March (1978).
65. B. Parhami, From defects to failures: a view of dependable computing. *ACM Computer Architect. News*, **16,** No. 4, 157–168 September (1988).
66. B. Parhami, *A Data-Driven Dependability Assurance Scheme with Applications to Data and Design Diversity*, pp. 105–112, in [26] (1989). Also in *Dependable Computing for Critical Applications*, pp. 257–282. Springer, New York (1991).
67. B. Parhami, A unified approach to correctness and timeliness requirements for ultrareliable concurrent systems. *Proc. Int. Parallel Processing Symp.*, pp. 733–747. Fullerton, Calif. (1990).
68. B. Parhami, Voting networks. *IEEE Trans. Reliabil.* **40,** 380–394 (1991).
69. B. Parhami, *Optimal Algorithms for Exact, Inexact, and Approval Voting*, pp. 404–411, in [36] (1992).
70. K. S. Park, *Human Reliability: Analysis, Prediction, and Prevention of Human Errors* Elsevier, Amsterdam (1987).
71. W. W. Peterson and E. J. Weldon Jr, *Error-Correcting Codes*, 2nd Edn. MIT Press, Cambridge (1972).
72. W. H. Pierce, *Adaptive Vote-Takers Improve the Use of Redundancy*, pp. 229–250, in [96].
73. W. H. Pierce, *Failure-Tolerant Computer Design*. Academic Press, New York, (1965).
74. D. K. Pradhan (Ed.), *Fault-Tolerant Computing: Theory and Techniques*, 2 Vols, Prentice–Hall, New York (1986).
75. *Proc. of the IEEE*, Special Issue on Fault-Tolerant Computing, Vol. 66, No. 10 (1978).
76. *Proc. of the IEEE*, Special Issue on Fault Tolerance in VLSI, Vol. 74, No. 5 (1986).
77. B. Randell, System structure for software fault tolerance. *IEEE Trans. Soft. Engng* **1,** 220–232 (1975).
78. B. Randell, *Design Fault Tolerance*, pp. 251–270, in [13].
79. T. R. N. Rao, *Error Codes for Arithmetic Processors*, Academic Press, New York (1974).
80. T. R. N. Rao and E. Fujiwara, *Error-Control Coding for Computer Systems*. Prentice–Hall, New York (1989).
81. F. B. Schneider, *The Fail-Stop Processor Approach*, Chap. 13, pp. 370–394, in [17].

82. C. E. Shannon, A mathematical theory of communication. *Bell Syst. Tech. J.* **27**, 379–423 and 623–656 (1948).
83. S. K. Shrivastava (Ed.), *Reliable Computer Systems: Collected Papers of the Newcastle Reliability Project*. Springer, New York (1985).
84. D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable System Design*. Digital Press (1982).
85. D. P. Siewiorek, *Fault Tolerance in Commercial Computers*, pp. 26–37, in [24] (1990).
86. D. P. Siewiorek and R. S. Swarz, *Reliable Computer Systems: Design and Evaluation*, 2nd Edn. Digital Press (1992).
87. D. Swade, Charles Babbage's engines: the genius of failure. *Computer Bull.* **3**, Pt. 5, 12–14 June (1991).
88. D. J. Taylor and J. P. Black, *Principles of Data Structure Error Correction*, pp. 602–608, in [91] (1982).
89. C. C. Timoc (Ed.), *Selected Reprints on Logic Design for Testability*. IEEE Computer Society Press (1984).
90. W. N. Toy, *Fault-Tolerant Design of Local ESS Processors*, pp. 1126–1145, in [75], Reprinted in [84], pp. 461–496.
91. *IEEE Trans. Computers*, Special Issues or Sections on Fault-Tolerant Computing Vol. 20, No. 11, November (1971); Vol. 22, No. 3, March (1973); Vol. 23, No. 7, July (1974); Vol. 24, No. 5, May (1975); Vol. 25, No. 6, June (1976); Vol. 27, No. 6, June (1978); Vol. 29, No. 6, June (1980); Vol. 31, No. 7, July (1982); Vol. 33, No. 6, June (1984); Vol. 35, No. 4, April (1986); Vol. 37, No. 4, April (1988); Vol. 39, No. 4, April (1990); Vol. 41, No. 5, May (1992).
92. J. G. Tryon, *Quadded Logic*, pp. 205–228, in [96].
93. J. von Neumann, Probabilistic logics and the synthesis of reliable organisms from unreliable components. In *Automata Studies* (*Annals of Mathematics Studies*, No. 34) (Edited by C. E. Shannon and J. McCarthy), pp. 43–98. Princeton University Press (1956).
94. J. F. Wakerly, *Error Detecting Codes, Self-Checking Circuits and Applications*, North-Holland, Amsterdam (1978).
95. G. F. Watson (Coordinator), Faults & failures: when speed enforcement didn't work. *IEEE Spectrum*. **28**, No. 11, 76 November (1991).
96. R. H. Wilcox and W. C. Mann (Eds), *Redundancy Techniques for Computing Systems*. Spartan, New York (1962).
97. T. W. Williams, Design for testability, in *VLSI Testing*, Chapter 4, pp. 95–160. North-Holland, Amsterdam (1986).
98. S. Winograd and J. D. Cowan, *Reliable Computation in the Presence of Noise*. MIT Press, Cambridge (1963).
99. M. H. Woods, MOS VLSI reliability and yield trends. *Proc. IEEE* **74**, 1715–1729 (1986).

AUTHOR'S BIOGRAPHY



Behrooz Parhami—B. Parhami (Ph.D. UCLA, 1973) was affiliated with Sharif (Arya-Mehr) University of Technology, Tehran, Iran. Since 1988, he has been Professor of Computer Engineering at University of California, Santa Barbara, with research interests in dependable computing, computer arithmetic, and parallel processing. He has also published on Farsi-language computing and informatics education. He was the Founding President of Informatics Society of Iran (1978–1983), Chair of IEEE Iran Section (1977–1986), and a recipient of the IEEE Centennial Medal (1984).