Continued intensive discussions led to the 1992 book *Dependability: Basic Concepts and Terminology* [41], which contained a 34-page English text with an eight-page glossary and its translations into French, German, Italian, and Japanese. The principal innovations were the addition of *security* as an attribute and of the class of *intentional malicious faults* in the taxonomy of faults. Many concepts were refined and elaborated.

The next major step was the recognition of security as a composite of the attributes of *confidentiality*, *integrity*, and *availability* and the addition of the class of *intentional nonmalicious faults, together with an analysis of the problems of inadequate system specifications* [42], though this account provided only a summary classification of dependability threats.

The present paper represents the results of a continuous effort since 1995 to expand, refine, and simplify the taxonomy of dependable and secure computing. It is also our goal to make the taxonomy readily available to practitioners and students of the field; therefore, this paper is self-contained and does not require reading of the above mentioned publications. The major new contributions are:

1. *The relationship between dependability and security* is clarified (Section 2.3).
2. *A quantitative definition of dependability* is introduced (Section 2.3).
3. *The criterion of capability* is introduced in the classification of human-made nonmalicious faults (Sections 3.2.1 and 3.2.3), enabling the consideration of *competence*.
4. *The discussion of malicious faults* is extensively updated (Section 3.2.4).
5. *Service failures* (Section 3.3.1) are distinguished from *dependability failures* (Section 3.3.3): The latter are recognized when service failures over a period of time are too frequent or too severe.
6. *Dependability issues of the development process* are explicitly incorporated into the taxonomy, including partial and complete *development failures* (Section 3.3.2).
7. The concept of *dependability is related to dependence and trust* (Section 4.2), and compared with three recently introduced similar concepts, including survivability, trustworthiness, high-confidence systems (Section 4.4).

After the present extensive iteration, what future opportunities and challenges can we foresee that will prompt the evolution of the taxonomy? Certainly, we recognize the desirability of further:

- expanding the discussion of security, for example to cover techniques for protecting confidentiality, establishing authenticity, etc.,
- analyzing issues of trust and the allied topic of risk management, and
- searching for unified measures of dependability and security.

We expect that some challenges will come unexpectedly (perhaps as so-called "emergent properties," such as those of the HAL computer in Arthur C. Clarke's "2001: A Space Odyssey") as the complexity of man-machine systems that we can build exceeds our ability to comprehend them. Other challenges are easier to predict:

1. New technologies (nanosystems, biochips, chemical and quantum computing, etc.) and new concepts of man-machine systems (ambient computing, nomadic computing, grid computing, etc.) will require continued attention to their specific dependability issues.
2. The problems of complex human-machine interactions (including user interfaces) remain a challenge that is becoming very critical—the means to improve their dependability and security need to be identified and incorporated.
3. The dark side of human nature causes us to anticipate new forms of maliciousness that will lead to more forms of malicious faults and, hence, requirements for new defenses as well.

In view of the above challenges and because of the continuing and unnecessarily confusing introduction of purportedly "new" concepts to describe the same means, attributes, and threats, the most urgent goal for the future is to keep the taxonomy complete to the extent that this is possible, but at the same time as simple and well-structured as our abilities allow.

2 The Basic Concepts

In this section, we present a basic set of definitions that will be used throughout the entire discussion of the taxonomy of dependable and secure computing. The definitions are general enough to cover the entire range of computing and communication systems, from individual logic gates to networks of computers with human operators and users. In what follows, we focus mainly on computing and communications systems, but our definitions are also intended in large part to be of relevance to **computer-based systems**, i.e., systems which also encompass the humans and organizations that provide the immediate environment of the computing and communication systems of interest.

2.1 System Function, Behavior, Structure, and Service

A **system** in our taxonomy is an entity that interacts with other entities, i.e., other systems, including hardware, software, humans, and the physical world with its natural phenomena. These other systems are the **environment** of the given system. The **system boundary** is the common frontier between the system and its environment.

Computing and communication systems are characterized by fundamental properties: *functionality*, *performance*, *dependability and security*, and *cost*. Other important system properties that affect dependability and security include *usability*, *manageability*, and *adaptability*—detailed consideration of these issues is beyond the scope of this paper. The **function** of such a system is what the system is intended to do and is described by the **functional specification** in terms of functionality and performance. The **behavior** of a system is what the system does to implement its function and is described by a sequence of states. The **total state** of a given system is the set of the following states: computation, communication, stored information, interconnection, and physical condition.

The **structure** of a system is what enables it to generate the behavior. From a structural viewpoint, a system is composed of a set of components bound together in order to interact, where each **component** is another system, etc. The recursion stops when a component is considered to be **atomic**: Any further internal structure cannot be discerned, or is not of interest and can be ignored. Consequently, the total state of a system is the set of the (external) states of its atomic components.

The **service** delivered by a system (in its role as a **provider**) is its behavior as it is perceived by its user(s); a **user** is another system that receives service from the provider. The part of the provider's system boundary where service delivery takes place is the provider's **service interface**. The part of the provider's total state that is perceivable at the service interface is its **external state**; the remaining part is its **internal state**. The delivered service is a sequence of the provider's external states. We note that a system may sequentially or simultaneously be a provider and a user with respect to another system, i.e., deliver service to and receive service from that other system. The interface of the user at which the user receives service is the **use interface**.

We have up to now used the singular for function and service. A system generally implements more than one function, and delivers more than one service. Function and service can be thus seen as composed of function items and of service items. For the sake of simplicity, we shall simply use the plural—functions, services—when it is necessary to distinguish several function or service items.

2.2 The Threats to Dependability and Security: Failures, Errors, Faults

Correct service is delivered when the service implements the system function. A **service failure**, often abbreviated here to **failure**, is an event that occurs when the delivered service deviates from correct service. A service fails either because it does not comply with the functional specification, or because this specification did not adequately describe the system function. A service failure is a **transition** from correct service to incorrect service, i.e., to not implementing the system function. The period of delivery of incorrect service is a **service outage**. The transition from incorrect service to correct service is a **service restoration**. The deviation from correct service may assume different forms that are called **service failure modes** and are ranked according to **failure severities**. A detailed taxonomy of failure modes is presented in Section 3.

Since a service is a sequence of the system's external states, a service failure means that at least one (or more) external state of the system deviates from the correct service state. The deviation is called an *error*. The adjudged or hypothesized cause of an error is called a **fault**. Faults can be internal or external of a system. The prior presence of a **vulnerability**, i.e., an internal fault that enables an external fault to harm the system, is necessary for an external fault to cause an error and possibly subsequent failure(s). In most cases, a fault first causes an error in the service state of a component that is a part of the internal state of the system and the external state is not immediately affected.

For this reason, the definition of an **error** is the part of the total state of the system that may lead to its subsequent service failure. It is important to note that many errors do not reach the system's external state and cause a failure. A fault is **active** when it causes an error, otherwise it is **dormant**.

When the functional specification of a system includes a set of several functions, the failure of one or more of the services implementing the functions may leave the system in a **degraded mode** that still offers a subset of needed services to the user. The specification may identify several such modes, e.g., slow service, limited service, emergency service, etc. Here, we say that the system has suffered a **partial failure** of its functionality or performance. Development failures and dependability failures that are discussed in Section 3.3 also can be partial failures.

2.3 Dependability, Security, and Their Attributes

The original definition of **dependability** is the ability to deliver service that can justifiably be trusted. This definition stresses the need for justification of trust. The alternate definition that provides the criterion for deciding if the service is dependable is the **dependability** of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable.

It is usual to say that the dependability of a system should suffice for the dependence being placed on that system. The **dependence** of system A on system B, thus, represents the extent to which system A's dependability is (or would be) affected by that of System B. The concept of dependence leads to that of **trust**, which can very conveniently be defined as *accepted dependence*.

As developed over the past three decades, dependability is an integrating concept that encompasses the following attributes:

- **availability**: readiness for correct service.
- **reliability**: continuity of correct service.
- **safety**: absence of catastrophic consequences on the user(s) and the environment.
- **integrity**: absence of improper system alterations.
- **maintainability**: ability to undergo modifications and repairs.

When addressing security, an additional attribute has great prominence, **confidentiality**, i.e., the absence of unauthorized disclosure of information. **Security** is a composite of the attributes of confidentiality, integrity, and availability, requiring the concurrent existence of 1) availability for authorized actions only, 2) confidentiality, and 3) integrity with "improper" meaning "unauthorized."

Fig. 1 summarizes the relationship between dependability and security in terms of their principal attributes. The picture should *not* be interpreted as indicating that, for example, security developers have no interest in maintainability, or that there has been no research at all in the dependability field related to confidentiality—rather it indicates where the main balance of interest and activity lies in each case.

The **dependability and security specification** of a system must include the requirements for the attributes in terms of the acceptable frequency and severity of service failures for specified classes of faults and a given use environment. One or more attributes may not be required at all for a given system.

2.4 The Means to Attain Dependability and Security

Over the course of the past 50 years many means have been developed to attain the various attributes of dependability and security. Those means can be grouped into four major categories:

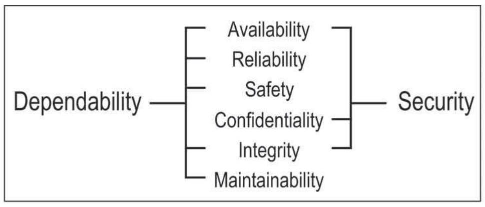


Fig. 1. Dependability and security attributes.

- **Fault prevention** means to prevent the occurrence or introduction of faults.
- **Fault tolerance** means to avoid service failures in the presence of faults.
- **Fault removal** means to reduce the number and severity of faults.
- **Fault forecasting** means to estimate the present number, the future incidence, and the likely consequences of faults.

Fault prevention and fault tolerance aim to provide the ability to deliver a service that can be trusted, while fault removal and fault forecasting aim to reach confidence in that ability by justifying that the functional and the dependability and security specifications are adequate and that the system is likely to meet them.

2.5 Summary: The Dependability and Security Tree

The schema of the complete taxonomy of dependable and secure computing as outlined in this section is shown in Fig. 2.

3 The Threats to Dependability and Security

3.1 System Life Cycle: Phases and Environments

In this section, we present the taxonomy of threats that may affect a system during its entire life. The **life cycle** of a system consists of two phases: *development* and *use*.

The development phase includes all activities from presentation of the user's initial concept to the decision that the system has passed all acceptance tests and is ready to deliver service in its user's environment. During the development phase, the system interacts with the development environment and *development faults* may be introduced into the system by the environment. The **development environment** of a system consists of the following elements:

1. the *physical world* with its natural phenomena,
2. *human developers*, some possibly lacking competence or having malicious objectives,
3. *development tools*: software and hardware used by the developers to assist them in the development process,
4. *production and test facilities*.

The use phase of a system's life begins when the system is accepted for use and starts the delivery of its services to the users. Use consists of alternating periods of correct service delivery (to be called **service delivery**), service outage, and service shutdown. A *service outage* is caused by a service failure. It is the period when incorrect service (including no service at all) is delivered at the service interface. A **service shutdown** is an intentional halt of service by an authorized entity. **Maintenance** actions may take place during all three periods of the use phase.

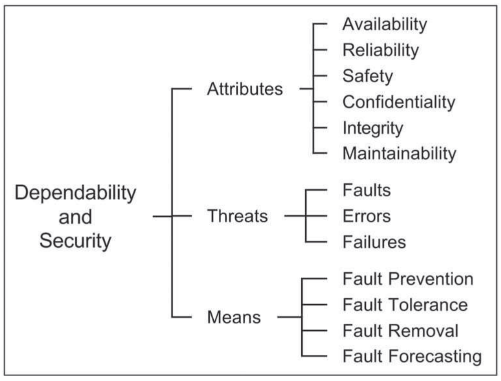


Fig. 2. The dependability and security tree.

During the use phase, the system interacts with its *use environment* and may be adversely affected by faults originating in it. The **use environment** consists of the following elements:

1. *the physical world* with its natural phenomena;
2. *administrators* (including maintainers): entities (humans or other systems) that have the authority to manage, modify, repair and use the system; some authorized humans may lack competence or have malicious objectives;
3. *users*: entities (humans or other systems) that receive service from the system at their use interfaces;
4. *providers*: entities (humans or other systems) that deliver services to the system at its use interfaces;
5. *the infrastructure*: entities that provide specialized services to the system, such as information sources (e.g., time, GPS, etc.), communication links, power sources, cooling airflow, etc.
6. *intruders*: malicious entities (humans and other systems) that attempt to exceed any authority they might have and alter service or halt it, alter the system's functionality or performance, or to access confidential information. Examples include hackers, vandals, corrupt insiders, agents of hostile governments or organizations, and malicious software.

As used here, the term **maintenance**, following common usage, includes not only repairs, but also all modifications of the system that take place during the use phase of system life. Therefore, maintenance is a development process, and the preceding discussion of development applies to maintenance as well. The various forms of maintenance are summarized in Fig. 3.

It is noteworthy that repair and fault tolerance are related concepts; the distinction between fault tolerance and maintenance in this paper is that maintenance involves the participation of an external agent, e.g., a repairman, test equipment, remote reloading of software. Furthermore, repair is part of fault removal (during the use phase), and fault forecasting usually considers repair situations. In fact, repair can be seen as a fault tolerance activity within a larger system that includes the system being repaired and the people and other systems that perform such repairs.

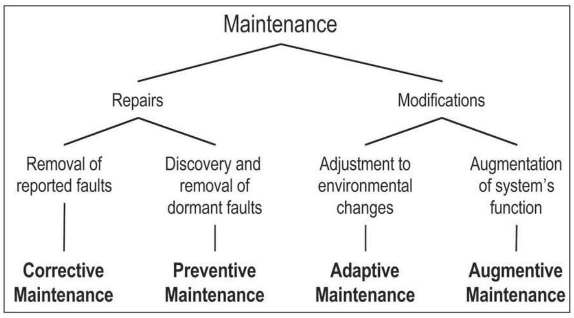

Maintenance
Repairs
Modifications
Removal of reported faults
Discovery and removal of dormant faults
Adjustment to environmental changes
Augmentation of system's function
Corrective Maintenance
Preventive Maintenance
Adaptive Maintenance
Augmentive Maintenance

Fig. 3. The various forms of maintenance.

3.2 Faults

3.2.1 A Taxonomy of Faults

All faults that may affect a system during its life are classified according to eight basic viewpoints, leading to the *elementary fault* classes, as shown in Fig. 4.

If all combinations of the eight elementary fault classes were possible, there would be 256 different *combined fault classes*. However, not all criteria are applicable to all fault classes; for example, natural faults cannot be classified by objective, intent, and capability. *We have identified 31 likely combinations*; they are shown in Fig. 5.

More combinations may be identified in the future. The combined fault classes of Fig. 5 are shown to belong to three major partially overlapping groupings:

- **development faults** that include all fault classes occurring during development,
- **physical faults** that include all fault classes that affect hardware,
- **interaction faults** that include all external faults.

The boxes at the bottom of Fig. 5a identify the names of some illustrative fault classes.

Knowledge of all possible fault classes allows the user to decide which classes should be included in a dependability and security specification. Next, we comment on the fault classes that are shown in Fig. 5. Fault numbers (1 to 31) will be used to relate the discussion to Fig. 5.

3.2.2 On Natural Faults

Natural faults (11-15) are physical (hardware) faults that are caused by natural phenomena without human participation. We note that humans also can cause physical faults (6-10, 16-23); these are discussed below. *Production defects* (11) are natural faults that originate during development. During operation the natural faults are either *internal* (12-13), due to natural processes that cause physical deterioration, or *external* (14-15), due to natural processes that originate outside the system boundaries and cause physical interference by penetrating the hardware boundary of the system (radiation, etc.) or by entering via use interfaces (power transients, noisy input lines, etc.).

3.2.3 On Human-Made Faults

The definition of human-made faults (that result from human actions) includes absence of actions when actions should be performed, i.e., **omission faults**, or simply **omissions**. Performing wrong actions leads to **commission faults**.

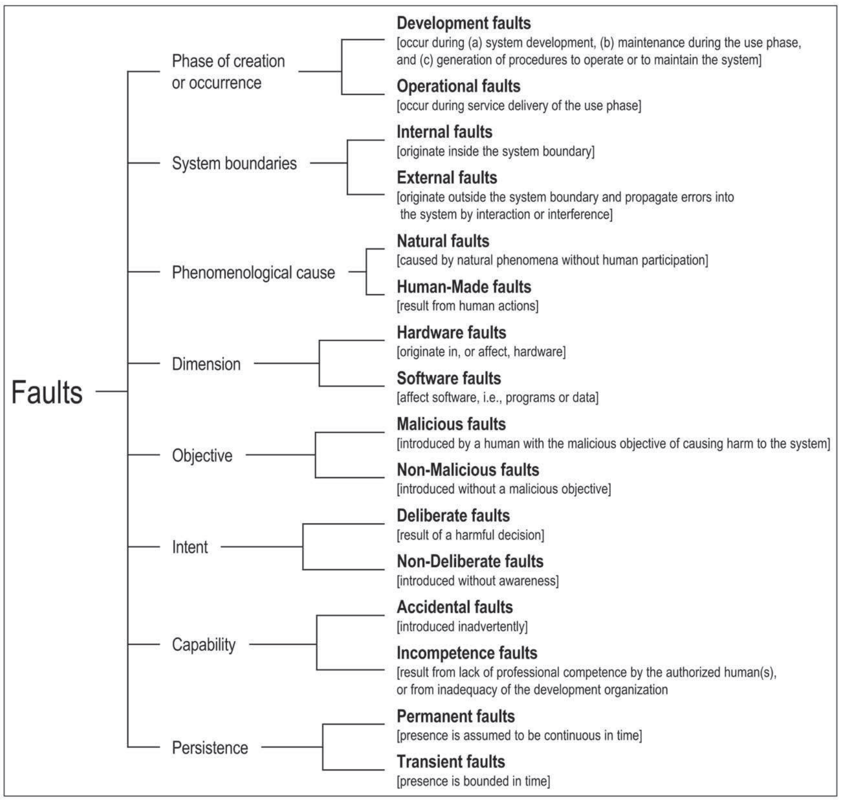
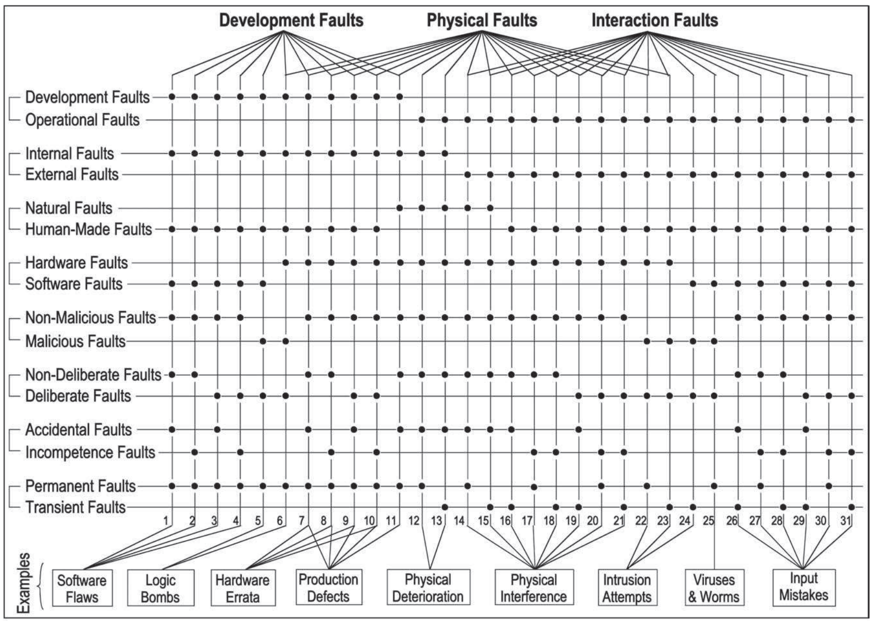
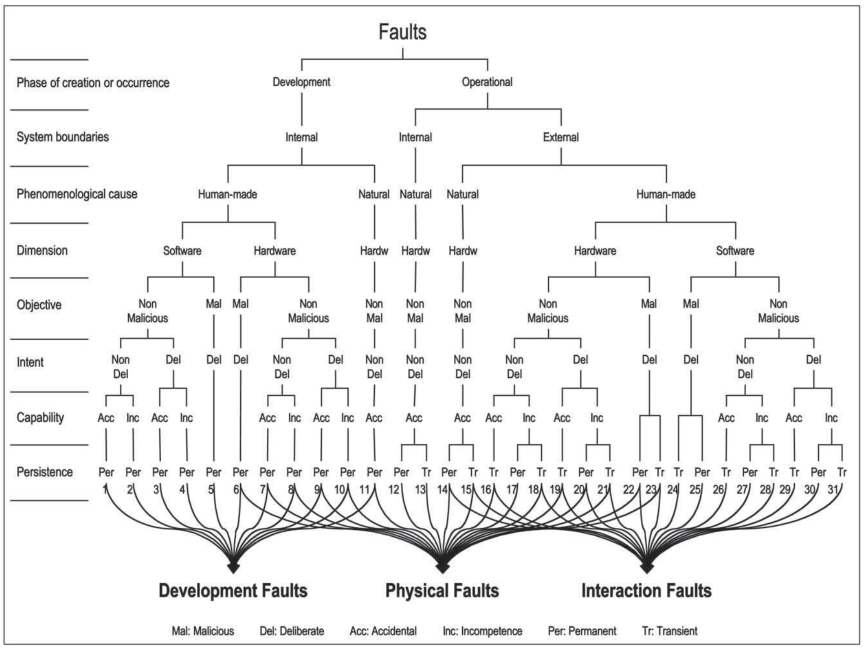

Faults
Phase of creation or occurrence
Development faults
[occur during (a) system development, (b) maintenance during the use phase, and (c) generation of procedures to operate or to maintain the system]
Operational faults
[occur during service delivery of the use phase]
System boundaries
Internal faults
[originate inside the system boundary]
External faults
[originate outside the system boundary and propagate errors into the system by interaction or interference]
Phenomenological cause
Natural faults
[caused by natural phenomena without human participation]
Human-Made faults
[result from human actions]
Dimension
Hardware faults
[originate in, or affect, hardware]
Software faults
[affect software, i.e., programs or data]
Objective
Malicious faults
[introduced by a human with the malicious objective of causing harm to the system]
Non-Malicious faults
[introduced without a malicious objective]
Intent
Deliberate faults
[result of a harmful decision]
Non-Deliberate faults
[introduced without awareness]
Capability
Accidental faults
[introduced inadvertently]
Incompetence faults
[result from lack of professional competence by the authorized human(s), or from inadequacy of the development organization
Persistence
Permanent faults
[presence is assumed to be continuous in time]
Transient faults
[presence is bounded in time]

Fig. 4. The elementary fault classes.



(a)



(b)

Fig. 5. The classes of combined faults (a) Matrix representation. (b) Tree representation.

The two basic classes of human-made faults are distinguished by the *objective* of the developer or of the humans interacting with the system during its use:

- *Malicious faults*, introduced during either system development with the objective to cause harm to the system during its use (5-6), or directly during use (22-25).
- *Nonmalicious faults* (1-4, 7-21, 26-31), introduced without malicious objectives.

We consider nonmalicious faults first. They can be partitioned according to the developer's intent:

- *nondeliberate* faults that are due to *mistakes*, that is, *unintended actions* of which the developer, operator, maintainer, etc. is not aware (1, 2, 7, 8, 16-18, 26-28);
- *deliberate* faults that are due to *bad decisions*, that is, *intended actions* that are wrong and cause faults (3, 4, 9, 10, 19-21, 29-31).

Deliberate, nonmalicious, development faults (3, 4, 9, 10) result generally from trade offs, either 1) aimed at preserving acceptable performance, at facilitating system utilization, or 2) induced by economic considerations. Deliberate, nonmalicious interaction faults (19-21, 29-31) may result from the action of an operator either aimed at overcoming an unforeseen situation, or deliberately violat-

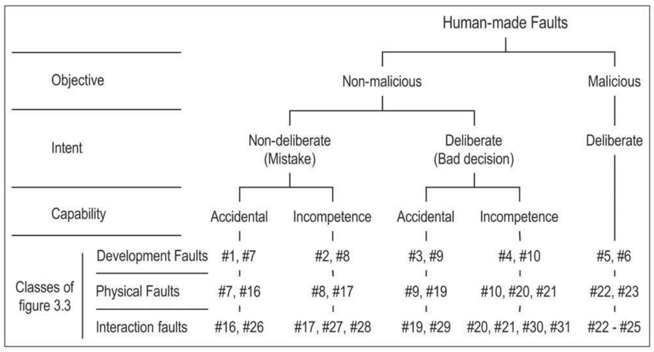


Fig. 6. Classification of human-made faults.

ing an operating procedure without having realized the possibly damaging consequences of this action. Deliberate, nonmalicious faults are often recognized as faults only *after* an unacceptable system behavior; thus, a failure has ensued. The developer(s) or operator(s) did not realize at the time that the consequence of their decision was a fault.

It is usually considered that both mistakes and bad decisions are *accidental*, as long as they are not made with malicious objectives. However, *not all* mistakes and bad decisions by nonmalicious persons are accidents. Some very harmful mistakes and very bad decisions are made by persons who lack professional competence to do the job they have undertaken. A complete fault taxonomy should not conceal this cause of faults; therefore, we introduce a further partitioning of nonmalicious human-made faults into 1) *accidental faults*, and 2) *incompetence faults*. The structure of this taxonomy of human-made faults is shown in Fig. 6.

The question of how to recognize incompetence faults becomes important when a mistake or a bad decision has consequences that lead to economic losses, injuries, or loss of human lives. In such cases, independent professional judgment by a board of inquiry or legal proceedings in a court of law are likely to be needed to decide if professional malpractice was involved.

Thus far, the discussion of incompetence faults has dealt with individuals. However, human-made efforts have failed because a team or an entire organization did not have the organizational competence to do the job. A good example of organizational incompetence is the development failure of the AAS system, that was intended to replace the aging air traffic control systems in the USA [67].

Nonmalicious development faults can exist in hardware and in software. In hardware, especially in microprocessors, some development faults are discovered after production has started [5]. Such faults are called "errata" and are listed in specification updates. The finding of errata typically continues throughout the life of the processors; therefore, new specification updates are issued periodically. Some development faults are introduced because human-made tools are faulty.

Off-the-shelf (OTS) components are inevitably used in system design. The use of OTS components introduces additional problems. They may come with known development faults and may contain unknown faults as well (bugs, vulnerabilities, undiscovered errata, etc.). Their specifications may be incomplete or even incorrect. This problem is especially serious when *legacy* OTS components are used that come from previously designed and used systems, and must be retained in the new system because of the user's needs.

Some development faults affecting software can cause **software aging** [27], i.e., progressively accrued error conditions resulting in performance degradation or complete failure. Examples are memory bloating and leaking, unterminated threads, unreleased file-locks, data corruption, storage space fragmentation, and accumulation of round-off errors [10].

3.2.4 On Malicious Faults

Malicious human-made faults are introduced with the malicious objective to alter the functioning of the system during use. Because of the objective, classification according to intent and capability is not applicable. The goals of such faults are: 1) to disrupt or halt service, causing **denials of service**; 2) to access confidential information; or 3) to improperly modify the system. They are grouped into two classes:

1. **Malicious logic faults** that encompass development faults (5,6) such as *Trojan horses*, logic or timing *bombs*, and *trapdoors*, as well as operational faults (25) such as *viruses*, *worms*, or *zombies*. Definitions for these faults [39], [55] are given in Fig. 7.
2. **Intrusion attempts** that are operational external faults (22-24). The external character of intrusion attempts does not exclude the possibility that they may be performed by system operators or administrators who are exceeding their rights, and intrusion attempts may use physical means to cause faults: power fluctuation, radiation, wire-tapping, heating/cooling, etc.

What is colloquially known as an "exploit" is in essence a software script that will exercise a system vulnerability and allow an intruder to gain access to, and sometimes control of, a system. In the terms defined here, invoking the exploit is an operational, external, human-made, software, malicious interaction fault (24-25). Heating the RAM with a hairdryer to cause memory errors that permit software security violations would be an external, human-made, hardware, malicious interaction fault (22-23). The vulnerability that an exploit takes advantage of is typically a software flaw (e.g., an unchecked buffer) that could be characterized as a developmental, internal, human-made, software, nonmalicious, nondeliberate, permanent fault (1-2).

3.2.5 On Interaction Faults

Interaction faults occur during the use phase, therefore they are all *operational* faults. They are caused by elements of the use environment (see Section 3.1) interacting with the system; therefore, they are all *external*. Most classes originate due to some human action in the use environment; therefore, they are *human-made*. They are fault classes 16-31 in Fig. 5. An exception are external natural faults (14-15) caused by cosmic rays, solar flares, etc. Here, nature interacts with the system without human participation.

A broad class of human-made operational faults are **configuration faults**, i.e., wrong setting of parameters that can affect security, networking, storage, middleware, etc. [24]. Such faults can occur during configuration changes performed during adaptive or augmentative maintenance performed concurrently with system operation (e.g.,

logic bomb: *malicious logic* that remains dormant in the host system till a certain time or an event occurs, or certain conditions are met, and then deletes files, slows down or crashes the host system, etc.

Trojan horse: *malicious logic* performing, or able to perform, an illegitimate action while giving the impression of being legitimate; the illegitimate action can be the disclosure or modification of information (attack against confidentiality or integrity) or a *logic bomb*;

trapdoor: *malicious logic* that provides a means of circumventing access control mechanisms;

virus: *malicious logic* that replicates itself and joins another program when it is executed, thereby turning into a *Trojan horse*; a virus can carry a *logic bomb*;

worm: *malicious logic* that replicates itself and propagates without the users being aware of it; a worm can also carry a *logic bomb*;

zombie: *malicious logic* that can be triggered by an attacker in order to mount a coordinated attack.

Fig. 7. Malicious logic faults.

introduction of a new software version on a network server); they are then called **reconfiguration faults** [70].

As mentioned in Section 2.2, a common feature of interaction faults is that, in order to be "successful," they usually necessitate the prior presence of a *vulnerability*, i.e., an internal fault that enables an external fault to harm the system. Vulnerabilities can be development or operational faults; they can be malicious or nonmalicious, as can be the external faults that exploit them. There are interesting and obvious similarities between an intrusion attempt and a physical external fault that "exploits" a lack of shielding. A vulnerability can result from a deliberate development fault, for economic or for usability reasons, thus resulting in limited protections, or even in their absence.

3.3 Failures

3.3.1 Service Failures

In Section 2.2, a *service failure* is defined as an event that occurs when the delivered service deviates from correct service. The different ways in which the deviation is manifested are a system's *service failure modes*. Each mode can have more than one *service failure severity*.

The occurrence of a failure was defined in Section 2 with respect to the function of a system, not with respect to the description of the function stated in the functional specification: a service delivery complying with the specification may be unacceptable for the system user(s), thus uncovering a specification fault, i.e., revealing the fact that the specification did not adequately describe the system function(s). Such specification faults can be either omission or commission faults (misinterpretations, unwarranted assumptions, inconsistencies, typographical mistakes). In such circumstances, the fact that the event is undesired (and is in fact a failure) may be recognized only after its occurrence, for instance via its consequences. So, failures can be subjective and disputable, i.e., may require judgment to identify and characterize.

The service failure modes characterize incorrect service according to four viewpoints:

1. the failure domain,
2. the detectability of failures,
3. the consistency of failures, and
4. the consequences of failures on the environment.

The **failure domain** viewpoint leads us to distinguish:

- **content failures**. The content of the information delivered at the service interface (i.e., the service content) deviates from implementing the system function.
- **timing failures**. The time of arrival or the duration of the information delivered at the service interface (i.e., the timing of service delivery) deviates from implementing the system function.

These definitions can be specialized: 1) the content can be in numerical or nonnumerical sets (e.g., alphabets, graphics, colors, sounds), and 2) a timing failure may be **early** or **late**, depending on whether the service is delivered too early or too late. Failures when both information and timing are incorrect fall into two classes:

- **halt failure**, or simply **halt**, when the service is *halted* (the external state becomes constant, i.e., system activity, if there is any, is no longer perceptible to the users); a special case of halt is **silent failure**, or simply **silence**, when no service at all is delivered at the service interface (e.g., no messages are sent in a distributed system).
- **erratic failures** otherwise, i.e., when a service is delivered (not halted), but is *erratic* (e.g., babbling).

Fig. 8 summarizes the service failure modes with respect to the failure domain viewpoint.

The **detectability** viewpoint addresses the *signaling* of service failures to the user(s). Signaling at the service interface originates from detecting mechanisms in the system that check the correctness of the delivered service. When the losses are detected and signaled by a warning signal, then **signaled failures** occur. Otherwise, they are **unsignaled failures**. The detecting mechanisms themselves have two failure modes: 1) signaling a loss of function when no failure has actually occurred, that is a **false alarm**, 2) not signaling a function loss, that is an *unsignaled failure*. When the occurrence of service failures result in reduced modes of service, the system signals a degraded mode of service to the user(s). Degraded modes may range from minor reductions to emergency service and safe shutdown.

The **consistency** of failures leads us to distinguish, when a system has two or more users:

- **consistent failures**. The incorrect service is perceived identically by all system users.
- **inconsistent failures**. Some or all system users perceive differently incorrect service (some users may actually perceive correct service); inconsistent failures are usually called, after [38], **Byzantine failures**.

Grading the *consequences of the failures* upon the system environment enables failure *severities* to be defined. The failure modes are ordered into severity levels, to which are generally associated maximum acceptable probabilities of occurrence. The number, the labeling, and the definition of the severity levels, as well as the acceptable probabilities of occurrence, are application-related, and involve the dependability and security attributes for the considered

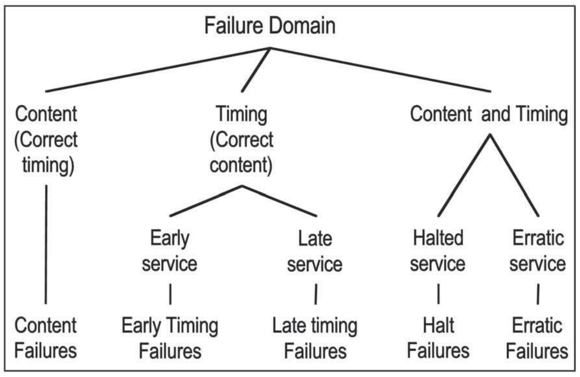


Fig. 8. Service failure modes with respect to the failure domain viewpoint.

application(s). Examples of criteria for determining the classes of failure severities are

1. for availability, the outage duration;
2. for safety, the possibility of human lives being endangered;
3. for confidentiality, the type of information that may be unduly disclosed; and
4. for integrity, the extent of the corruption of data and the ability to recover from these corruptions.

Generally speaking, two limiting levels can be defined according to the relation between the benefit (in the broad sense of the term, not limited to economic considerations) provided by the service delivered in the absence of failure, and the consequences of failures:

- **minor failures**, where the harmful consequences are of similar cost to the benefits provided by correct service delivery;
- **catastrophic failures**, where the cost of harmful consequences is orders of magnitude, or even incommensurably, higher than the benefit provided by correct service delivery.

Fig. 9 summarizes the service failure modes.

Systems that are designed and implemented so that they fail only in specific modes of failure described in the dependability and security specification and only to an acceptable extent are **fail-controlled systems**, e.g., with stuck output as opposed to delivering erratic values, silence as opposed to babbling, consistent as opposed to inconsistent failures. A system whose failures are to an acceptable extent halting failures only is a **fail-halt** (or fail-stop) **system**; the situations of stuck service and of silence lead, respectively, to **fail-passive systems** and **fail-silent systems** [53]. A system whose failures are, to an acceptable extent, all minor ones is a **fail-safe system**.

As defined in Section 2, delivery of incorrect service is an *outage*, which lasts until *service restoration*. The outage duration may vary significantly, depending on the actions involved in service restoration after a failure has occurred: 1) automatic or operator-assisted recovery, restart, or reboot; 2) corrective maintenance. Correction of development faults (by patches or workarounds) is usually performed offline, after service restoration, and the upgraded components resulting from fault correction are then introduced at some appropriate time with or without interruption of system operation. Preemptive interruption of system operation for an upgrade or for preventive maintenance is a *service shutdown*, also called a *planned outage* (as opposed to an outage consecutive to failure, which is then called an *unplanned outage*).

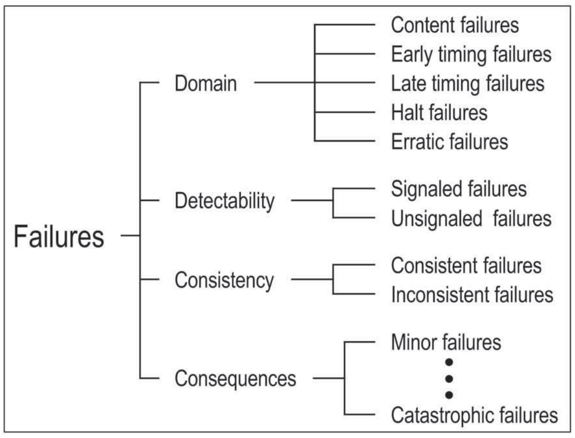


Fig. 9. Service failure modes.

3.3.2 Development Failures

As stated in Section 3.1, development faults may be introduced into the system being developed by its environment, especially by human developers, development tools, and production facilities. Such development faults may contribute to partial or complete development failures, or they may remain undetected until the use phase. A complete **development failure** causes the development process to be terminated before the system is accepted for use and placed into service. There are two aspects of development failures:

1. *Budget failure*. The allocated funds are exhausted before the system passes acceptance testing.
2. *Schedule failure*. The projected delivery schedule slips to a point in the future where the system would be technologically obsolete or functionally inadequate for the user's needs.

The principal causes of development failures are: incomplete or faulty specifications, an excessive number of user-initiated specification changes; inadequate design with respect to functionality and/or performance goals; too many development faults; inadequate fault removal capability; prediction of insufficient dependability or security; and faulty estimates of development costs. All are usually due to an underestimate of the complexity of the system to be developed.

There are two kinds of **partial development failures**, i.e., failures of lesser severity than project termination. Budget or schedule **overruns** occur when the development is completed, but the funds or time needed to complete the effort exceed the original estimates. Another form of partial development failure is **downgrading**: The developed system is delivered with less functionality, lower performance, or is predicted to have lower dependability or security than was required in the original system specification.

Development failures, overruns, and downgrades have a very negative impact on the user community, see, e.g., statistics about large software projects [34], or the analysis of the complete development failure of the AAS system, that resulted in the waste of $1.5 billion [67].

3.3.3 Dependability and Security Failures

It is to be expected that faults of various kinds will affect the system during its use phase. The faults may cause unacceptably degraded performance or total failure to deliver the specified service. For this reason, a *dependability and security specification* is agreed upon that states the goals for each attribute: availability, reliability, safety, confidentiality, integrity, and maintainability.

The specification explicitly identifies the *classes of faults* that are expected and the *use environment* in which the system will operate. The specification may also require safeguards against certain undesirable or dangerous conditions. Furthermore, the inclusion of specific fault prevention or fault tolerance techniques may be required by the user.

A **dependability or security failure** occurs when the given system suffers service failures more frequently or more severely than acceptable.

The dependability and security specification can also contain faults. Omission faults can occur in description of the use environment or in choice of the classes of faults to be prevented or tolerated. Another class of faults is the unjustified choice of very high requirements for one or more attributes that raises the cost of development and may lead to a cost overrun or even a development failure. For example, the initial AAS complete outage limit of 3 seconds per year was changed to 5 minutes per year for the new contract in 1994 [67].

3.4 Errors

An *error* has been defined in Section 2.2 as the part of a system's total state that may lead to a failure—a failure occurs when the error causes the delivered service to deviate from correct service. The cause of the error has been called a fault.

An error is **detected** if its presence is indicated by an *error message* or *error signal*. Errors that are present but not detected are **latent** errors.

Since a system consists of a set of interacting components, the total state is the set of its component states. The definition implies that a fault originally causes an error within the state of one (or more) components, but service failure will not occur as long as the external state of that component is not part of the external state of the system. Whenever the error becomes a part of the external state of the component, a service failure of that component occurs, but the error remains internal to the entire system.

Whether or not an error will actually lead to a service failure depends on two factors:

1. The structure of the system, and especially the nature of any redundancy that exists in it:
   - *protective* redundancy, introduced to provide fault tolerance, that is explicitly intended to prevent an error from leading to service failure.
   - *unintentional* redundancy (it is in practice difficult if not impossible to build a system without any form of redundancy) that may have the same—presumably unexpected—result as intentional redundancy.
2. The behavior of the system: the part of the state that contains an error may never be needed for service, or an error may be eliminated (e.g., when overwritten) before it leads to a failure.

A convenient classification of errors is to describe them in terms of the elementary service failures that they cause, using the terminology of Section 3.3.1: content versus timing errors, detected versus latent errors, consistent versus inconsistent errors when the service goes to two or more users, minor versus catastrophic errors. In the field of error control codes, content errors are further classified according to the damage pattern: single, double, triple, byte, burst, erasure, arithmetic, track, etc., errors.

Some faults (e.g., a burst of electromagnetic radiation) can simultaneously cause errors in more than one component. Such errors are called **multiple related errors**. **Single errors** are errors that affect one component only.

3.5 The Pathology of Failure: Relationship between Faults, Errors, and Failures

The creation and manifestation mechanisms of faults, errors, and failures are illustrated by Fig. 10, and summarized as follows:

1. A fault is *active* when it produces an error; otherwise, it is *dormant*. An active fault is either 1) an internal fault that was previously dormant and that has been activated by the computation process or environmental conditions, or 2) an external fault. **Fault activation** is the application of an input (the activation pattern) to a component that causes a dormant fault to become active. Most internal faults cycle between their dormant and active states.
2. Error propagation within a given component (i.e., *internal* propagation) is caused by the computation process: An error is successively transformed into other errors. Error propagation from component A to component B that receives service from A (i.e., *external* propagation) occurs when, through internal propagation, an error reaches the service interface of component A. At this time, service delivered by A to B becomes incorrect, and the ensuing service failure of A appears as an external fault to B and propagates the error into B via its use interface.
3. A service failure occurs when an error is propagated to the service interface and causes the service delivered by the system to deviate from correct service. The failure of a component causes a permanent or transient fault in the system that contains the component. Service failure of a system causes a permanent or transient external fault for the other system(s) that receive service from the given system.

These mechanisms enable the "chain of threats" to be completed, as indicated by Fig. 11. The arrows in this chain express a causality relationship between faults, errors, and failures. They should be interpreted generically: by propagation, several errors can be generated before a failure occurs. It is worth emphasizing that, from the mechanisms above listed, propagation, and, thus, instantiation(s) of this chain, can occur via interaction between components or systems, composition of components into a system, and the creation or modification of a system.

Some illustrative examples of fault pathology are given in Fig. 12. From those examples, it is easily understood that

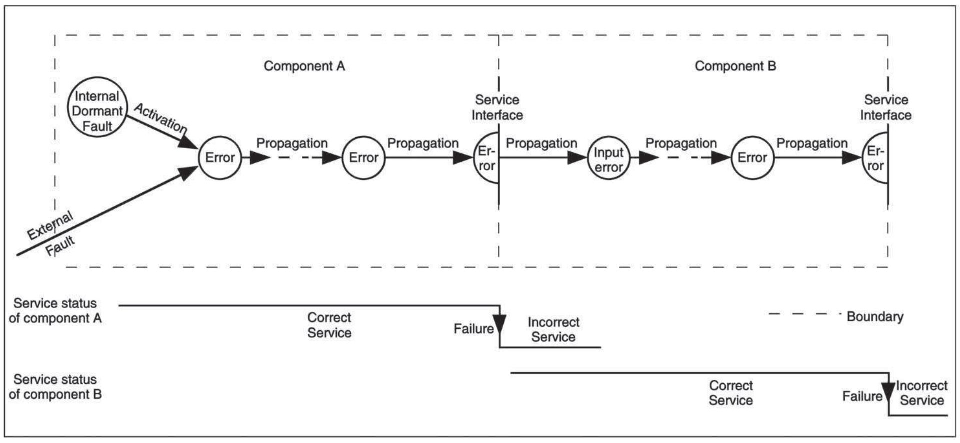


Fig. 10. Error propagation.

$$\begin{tikzcd} \cdots \arrow[r] & \text{fault} \arrow[r, "activation"] & \text{error} \arrow[r, "propagation"] & \text{failure} \arrow[r, "causation"] & \text{fault} \arrow[r] & \cdots \end{tikzcd}$$

Fig. 11. The fundamental chain of dependability and security threats.

- A short circuit occurring in an integrated circuit is a *failure* (with respect to the function of the circuit); the consequence (connection stuck at a Boolean value, modification of the circuit function, etc.) is a *fault* that will remain dormant as long as it is not activated. Upon activation (invoking the faulty component and uncovering the fault by an appropriate input pattern), the fault becomes *active* and produces an *error*, which is likely to propagate and create other errors. If and when the propagated error(s) affect(s) the delivered service (in information content and/or in the timing of delivery), a *failure* occurs.
- The result of an *error* by a programmer leads to a *failure* to write the correct instruction or data, that in turn results in a *(dormant) fault* in the written software (faulty instruction(s) or data); upon activation (invoking the component where the fault resides and triggering the faulty instruction, instruction sequence or data by an appropriate input pattern) the fault becomes *active* and produces an *error*; if and when the error affects the delivered service (in information content and/or in the timing of delivery), a *failure* occurs. This example is not restricted to accidental faults: a l*ogic bomb* is created by a malicious programmer; it will remain *dormant* until activated (e.g. at some predetermined date); it then produces an *error* that may lead to a storage overflow or to slowing down the program execution; as a consequence, service delivery will suffer from a so-called *denial-of-service*.
- The result of an *error* by a specifier' leads to a *failure* to describe a function, that in turn results in a *fault* in the written specification, e.g. incomplete description of the function. The implemented system therefore does not incorporate the missing (sub-)function. When the input data are such that the service corresponding to the missing function should be delivered, the actual service delivered will be different from expected service, i.e., an *error* will be perceived by the user, and a *failure* will thus occur.
- An inappropriate human-system interaction performed by an operator during the operation of the system is an external *fault* (from the system viewpoint); the resulting altered processed data is an *error;* etc.
- An *error* in reasoning leads to a maintenance or operating manual writer's *failure* to write correct directives, that in turn results in a *fault* in the corresponding manual (faulty directives) that will remain *dormant* as long as the directives are not acted upon in order to address a given situation, etc.
- A failure often results from the combined action of several faults; this is especially true when considering security issues: a trap-door (i.e., some way to by-pass access control) that is inserted into a computing system, either accidentally or deliberately, is a development *fault*; this fault may remain *dormant* until some malicious human makes use of it to enter the system; the intruder login is a deliberate interaction *fault*; when the intruder is logged in, he or she may deliberately create an *error*, e.g., modifying some file (integrity attack); when this file is used by an authorized user, the service will be affected, and a *failure* will occur.
- A given fault in a given component may result from various different possible sources; for instance, a permanent *fault* in a physical component — e.g., stuck at ground voltage — may result from:
  - a physical *failure* (e.g., caused by a threshold change),
  - an *error* caused by a development *fault* — e.g., faulty microinstruction decoding circuitry propagating 'down' through the layers and causing an illegal short between two circuit outputs for a duration long enough to provoke a short-circuit having the same consequence as a threshold change.
- Another example of top-down propagation is the exploitation during operation of an inadvertently introduced buffer overflow for gaining root privilege and subsequently re-writing the flash-ROM.

Fig. 12. Examples illustrating fault pathology.

fault dormancy may vary considerably, depending upon the fault, the given system's utilization, etc.

The ability to identify the activation pattern of a fault that had caused one or more errors is the **fault activation reproducibility**. Faults can be categorized according to their activation reproducibility: Faults whose activation is reproducible are called **solid**, or **hard**, faults, whereas faults whose activation is not systematically reproducible are **elusive**, or **soft**, faults. Most residual development faults in large and complex software are elusive faults: They are intricate enough that their activation conditions depend on complex combinations of internal state and external

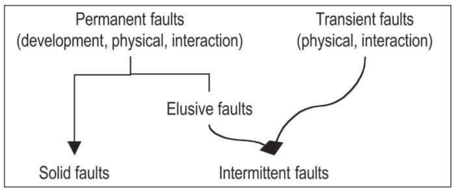


Fig. 13. Solid versus intermittent faults.

requests, that occur rarely and can be very difficult to reproduce [23]. Other examples of elusive faults are:

- "pattern sensitive" faults in semiconductor memories, changes in the parameters of a hardware component (effects of temperature variation, delay in timing due to parasitic capacitance, etc.).
- conditions—affecting either hardware or software—that occur when the system load exceeds a certain level, causing, for example, marginal timing and synchronization.

The similarity of the manifestation of elusive development faults and of transient physical faults leads to both classes being grouped together as **intermittent faults**. Errors produced by intermittent faults are usually termed **soft errors**. Fig. 13. summarizes this discussion.

Situations involving multiple faults and/or failures are frequently encountered. System failures often turn out on later examination to have been caused by errors that are due to a number of different coexisting faults. Given a system with defined boundaries, a **single fault** is a fault caused by *one* adverse physical event or *one* harmful human action. **Multiple faults** are *two or more* concurrent, overlapping, or sequential single faults whose consequences, i.e., errors, overlap in time, that is, the errors due to these faults are concurrently present in the system. Consideration of multiple faults leads one to distinguish 1) **independent faults**, that are attributed to different causes, and 2) **related faults**, that are attributed to a common cause. Related faults generally cause *similar errors*, i.e., errors that cannot be distinguished by whatever detection mechanisms are being employed, whereas independent faults usually cause *distinct errors*. However, it may happen that independent faults (especially omissions) lead to similar errors [6], or that related faults lead to distinct errors. The failures caused by similar errors are **common-mode failures**.

Three additional comments, about the words, or labels, "threats," "fault," "error," and "failure:"

1. The use of *threats*, for generically referring to faults, errors, and failures has a broader meaning than its common use in security, where it essentially retains it usual notion of potentiality. In our terminology, it has both this potentiality aspect (e.g., faults being not yet active, service failures not having impaired dependability), and a realization aspect (e.g., active fault, error that is present, service failure that occurs). In security terms, a malicious external fault is an *attack*.
2. The exclusive use in this paper of faults, errors, and failures does not preclude the use in special situations of words which designate, briefly and unambiguously, a specific class of threat; this is especially applicable to faults (e.g., bug, defect, deficiency, flaw, erratum) and to failures (e.g., breakdown, malfunction, denial-of-service).
3. The assignment made of the particular terms fault, error, and failure simply takes into account common usage: 1) fault prevention, tolerance, and diagnosis, 2) error detection and correction, 3) failure rate.

4 Dependability, Security, and Their Attributes

4.1 The Definitions of Dependability and Security

In Section 2.3, we have presented two alternate definitions of dependability:

- the original definition: the ability to deliver service that can justifiably be trusted.
- an alternate definition: the ability of a system to avoid service failures that are more frequent or more severe than is acceptable.

The original definition is a general definition that aims to generalize the more classical notions of availability, reliability, safety, integrity, maintainability, etc., that then become attributes of dependability. The alternate definition of dependability comes from the following argument. A system can, and usually does, fail. Is it however still dependable? When does it become undependable? The alternate definition thus provides a criterion for deciding whether or not, in spite of service failures, a system is still to be regarded as dependable. In addition, the notion of dependability failure, that is directly deduced from that definition, enables the establishment of a connection with development failures.

The definitions of dependability that exist in current standards differ from our definitions. Two such differing definitions are:

- "The collective term used to describe the availability performance and its influencing factors: reliability performance, maintainability performance and maintenance support performance" [31].
- "The extent to which the system can be relied upon to perform exclusively and correctly the system task(s) under defined operational and environmental conditions over a defined period of time, or at a given instant of time" [29].

The ISO definition is clearly centered upon availability. This is no surprise as this definition can be traced back to the definition given by the international organization for telephony, the CCITT [11], at a time when availability was the main concern to telephone operating companies. However, the willingness to grant dependability a generic character is noteworthy, since it goes beyond availability as it was usually defined, and relates it to reliability and maintainability. In this respect, the ISO/CCITT definition is consistent with the definition given in [26] for dependability: "the probability that a system will operate when needed." The second definition, from [29], introduces the notion of reliance, and as such is much closer to our definitions.

Terminology in the security world has its own rich history. Computer security, communications security, information security, and information assurance are terms that have had a long development and use in the community of security researchers and practitioners, mostly without direct

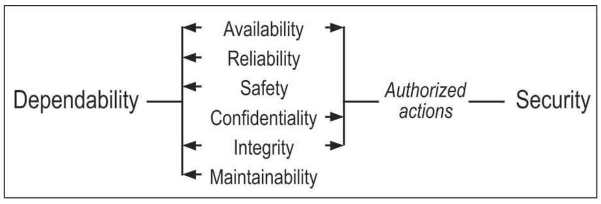


Fig. 14. Relationship between dependability and security.

reference to dependability. Nevertheless, all of these terms can be understood in terms of the three primary security attributes of confidentiality, integrity, and availability.

Security has not been characterized as a single attribute of dependability. This is in agreement with the usual definitions of security, that view it as a *composite* notion, namely, "the combination of confidentiality, the prevention of the unauthorized disclosure of information, integrity, the prevention of the unauthorized amendment or deletion of information, and availability, the prevention of the unauthorized withholding of information" [12], [52]. Our unified definition for **security** is the absence of unauthorized access to, or handling of, system state. The relationship between dependability and security is illustrated by Fig. 14, that is a refinement of Fig. 1.

4.2 Dependence and Trust

We have introduced the notions of dependence and trust in Section 2.3:

- The dependence of system A on system B represents the extent to which System A's dependability is (or would be) affected by that of System B.
- Trust is accepted dependence.

The dependence of a system on another system can vary from total dependence (any failure of B would cause A to fail) to complete independence (B cannot cause A to fail). If there is reason to believe that B's dependability will be insufficient for A's required dependability, the former should be enhanced, A's dependence reduced, or additional means of fault tolerance provided. Our definition of dependence relates to the relation *depends upon* [50], [14], whose definition is a component *a* depends upon a component *b* if the correctness of *b*'s service delivery is necessary for the correctness of *a*'s service delivery. However, this relation is expressed in terms of the narrower concept of correctness, rather than dependability, and, hence, is only binary, whereas our notion of dependence can take values on a measurable space.

By accepted dependence, we mean the dependence (say of A on B) allied to a judgment that this level of dependence is acceptable. Such a judgment (made by or on behalf of A) about B is possibly explicit and even laid down in a contract between A and B, but might be only implicit, even unthinking. Indeed, it might even be unwilling—in that A has no alternative option but to put its trust in B. Thus, to the extent that A trusts B, it need not assume responsibility for, i.e., provide means of tolerating, B's failures (the question of whether it is capable of doing this is another matter). In fact, the extent to which A fails to provide means of tolerating B's failures is a measure of A's (perhaps unthinking or unwilling) trust in B.

4.3 The Attributes of Dependability and Security

The attributes of dependability and security that have been defined in Section 2.3 may be of varying importance depending on the application intended for the given computing system: Availability, integrity, and maintainability are generally required, although to a varying degree depending on the application, whereas reliability, safety, and confidentiality may or may not be required according to the application. The extent to which a system possesses the attributes of dependability and security should be considered in a relative, probabilistic, sense, and not in an absolute, deterministic sense: Due to the unavoidable presence or occurrence of faults, systems are never totally available, reliable, safe, or secure.

The definition given for integrity—absence of improper system state alterations—goes beyond the usual definitions, that 1) relate to the notion of authorized actions only, and 2) focus on information (e.g., prevention of the unauthorized amendment or deletion of information [12], assurance of approved data alterations [33]): 1) naturally, when a system implements an authorization policy, "improper" encompasses "unauthorized," 2) "improper alterations" encompass actions that prevent (correct) upgrades of information, and 3) "system state" includes system modifications or damages.

The definition given for maintainability intentionally goes beyond corrective and preventive maintenance, and encompasses the other forms of maintenance defined in Section 3, i.e., adaptive and augmentative maintenance. The concept of **autonomic computing** [22] has as its major aim the provision of high maintainability for large networked computer systems, though automation of their management.

Besides the attributes defined in Section 2 and discussed above, other, *secondary*, attributes can be defined, which refine or specialize the *primary* attributes as defined in Section 2. An example of a specialized secondary attribute is **robustness**, i.e., dependability with respect to external faults, which characterizes a system reaction to a specific class of faults.

The notion of secondary attributes is especially relevant for security, and is based on distinguishing among various types of information [9]. Examples of such secondary attributes are:

- **accountability**: availability and integrity of the identity of the person who performed an operation;
- **authenticity**: integrity of a message content and origin, and possibly of some other information, such as the time of emission;
- **nonrepudiability**: availability and integrity of the identity of the sender of a message (nonrepudiation of the origin), or of the receiver (nonrepudiation of reception).

The concept of a **security policy** is that of a set of security-motivated constraints, that are to be adhered to by, for example, an organization or a computer system [47]. The enforcement of such constraints may be via technical, management, and/or operational controls, and the policy may lay down how these controls are to be enforced. In effect, therefore, a security policy is a (partial) system specification, lack of adherence to which will be regarded as a security failure. In practice, there may be a hierarchy of

Concept	Dependability	High Confidence	Survivability	Trustworthiness
Goal	1) ability to deliver service that can justifiably be trusted 2) ability of a system to avoid service failures that are more frequent or more severe than is acceptable	consequences of the system behavior are well understood and predictable	capability of a system to fulfill its mission in a timely manner	assurance that a system will perform as expected
Threats present	1) development faults (e.g., software flaws, hardware errata, malicious logic) 2) physical faults (e.g., production defects, physical deterioration) 3) interaction faults (e.g., physical interference, input mistakes, attacks, including viruses, worms, intrusions)	• internal and external threats • naturally occurring hazards and malicious attacks from a sophisticated and well-funded adversary	1) attacks (e.g., intrusions, probes, denials of service) 2) failures (internally generated events due to, e.g., software design errors, hardware degradation, human errors, corrupted data) 3) accidents (externally generated events such as natural disasters)	1) hostile attacks (from hackers or insiders) 2) environmental disruptions (accidental disruptions, either man-made or natural) 3) human and operator errors (e.g., software flaws, mistakes by human operators)
Reference	This paper	"Information Technology Frontiers for a New Millennium (Blue Book 2000)" [48]	"Survivable network systems" [16]	"Trust in cyberspace" [62]

Fig. 15. Dependability, high confidence, survivability, and trustworthiness.

such security policies, relating to a hierarchy of systems —for example, an entire company, its information systems department, and the individuals and computer systems in this department. Separate, albeit related policies, or separate parts of an overall policy document, may be created concerning different security issues, e.g., a policy regarding the controlled public disclosure of company information, one on physical and networked access to the company's computers. Some computer security policies include constraints on how information may flow within a system as well as constraints on system states.

As with any set of dependability and security specifications, issues of completeness, consistency, and accuracy are of great importance. There has thus been extensive research on methods for formally expressing and analyzing security policies. However, if some system activity is found to be in a contravention of a relevant security policy then, as with any system specification, the security failure may either be that of the system, or because the policy does not adequately describe the intended security requirement. A well-known example of an apparently satisfactory security policy that proved to be deficient, by failing to specify some particular behaviour as insecure, is discussed by [44].

Dependability and security classes are generally defined via the analysis of failure frequencies and severities, and of outage durations, for the attributes that are of concern for a given application. This analysis may be conducted directly or indirectly via risk assessment (see, e.g., [25] for availability, [58] for safety, and [32] for security).

The variations in the emphasis placed on the different attributes directly influence the balance of the techniques (fault prevention, tolerance, removal, and forecasting) to be employed in order to make the resulting system dependable and secure. This problem is all the more difficult as some of the attributes are conflicting (e.g., availability and safety, availability and confidentiality), necessitating that trade offs be made.

4.4 Dependability, High Confidence, Survivability, and Trustworthiness

Other concepts similar to dependability exist, such as **high confidence**, **survivability**, and **trustworthiness**. They are presented and compared to dependability in Fig. 15. A side-by-side comparison leads to the conclusion that all four concepts are essentially equivalent in their goals and address similar threats.

5 The Means to Attain Dependability and Security

In this section, we examine in turn fault prevention, fault tolerance, fault removal, and fault forecasting. The section ends with a discussion on the relationship between these various means.

5.1 Fault Prevention

Fault prevention is part of general engineering, and, as such, will not be much emphasized here. However, there are facets of fault prevention that are of direct interest regarding dependability and security, and that can be discussed according to the classes of faults defined in Section 3.2.

Prevention of development faults is an obvious aim for development methodologies, both for software (e.g., information hiding, modularization, use of strongly-typed programming languages) and hardware (e.g., design rules). Improvement of development processes in order to reduce the number of faults introduced in the produced systems is a step further in that it is based on the recording of faults in the products, and the elimination of the causes of the faults via process modifications [13], [51].

5.2 Fault Tolerance

5.2.1 Fault Tolerance Techniques

Fault tolerance [3], which is aimed at failure avoidance, is carried out via error detection and system recovery. Fig. 16 gives the techniques involved in fault tolerance.

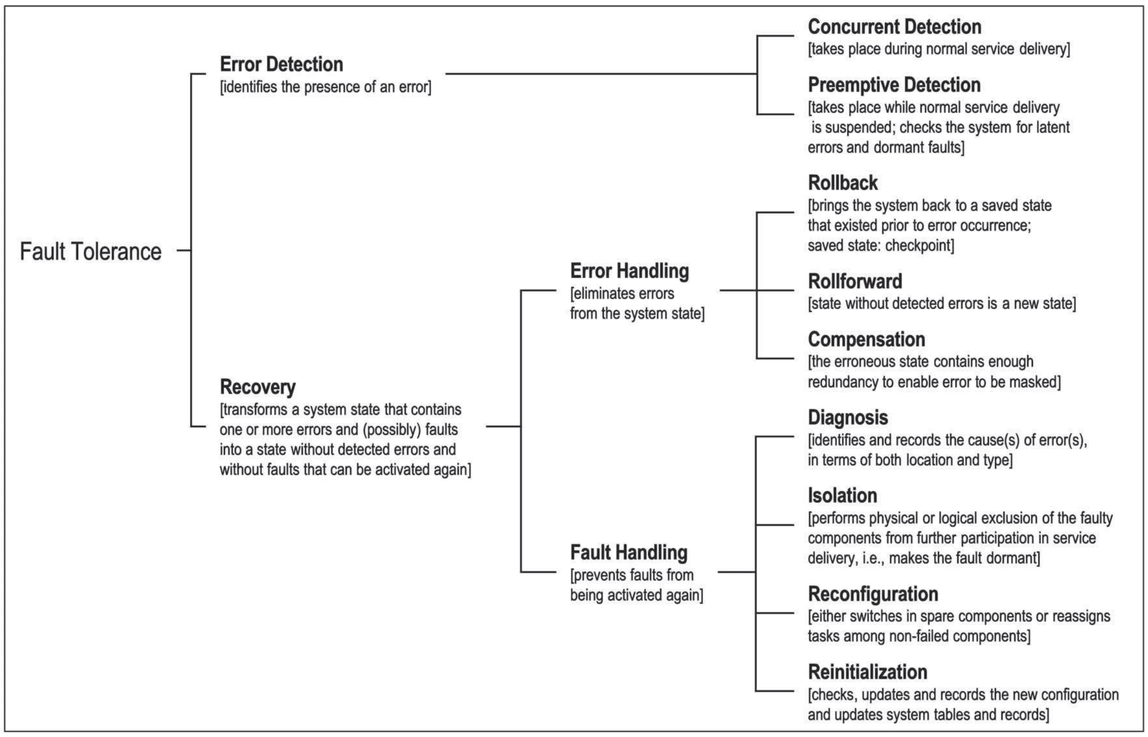

Fault Tolerance
Error Detection
[identifies the presence of an error]
Concurrent Detection
[takes place during normal service delivery]
Preemptive Detection
[takes place while normal service delivery is suspended; checks the system for latent errors and dormant faults]
Recovery
[transforms a system state that contains one or more errors and (possibly) faults into a state without detected errors and without faults that can be activated again]
Error Handling
[eliminates errors from the system state]
Rollback
[brings the system back to a saved state that existed prior to error occurrence; saved state: checkpoint]
Rollforward
[state without detected errors is a new state]
Compensation
[the erroneous state contains enough redundancy to enable error to be masked]
Fault Handling
[prevents faults from being activated again]
Diagnosis
[identifies and records the cause(s) of error(s), in terms of both location and type]
Isolation
[performs physical or logical exclusion of the faulty components from further participation in service delivery, i.e., makes the fault dormant]
Reconfiguration
[either switches in spare components or reassigns tasks among non-failed components]
Reinitialization
[checks, updates and records the new configuration and updates system tables and records]

Fig. 16. Fault tolerance techniques.

Usually, fault handling is followed by corrective maintenance, aimed at removing faults that were isolated by fault handling; in other words, the factor that distinguishes fault tolerance from maintenance is that maintenance requires the participation of an external agent. Closed systems are those systems where fault removal cannot be practically implemented (e.g., the hardware of a deep space probe).

Rollback and rollforward are invoked on demand, after error detection has taken place, whereas compensation can be applied either on demand or systematically, at predetermined times or events, independently of the presence or absence of (detected) error. Error handling on demand followed by fault handling together form **system recovery**; hence, the name of the corresponding strategy for fault tolerance: **error detection and system recovery** or simply **detection and recovery**.

Fault masking, or simply **masking**, results from the systematic usage of compensation. Such masking will conceal a possibly progressive and eventually fatal loss of protective redundancy. So, practical implementations of masking generally involve error detection (and possibly fault handling), leading to **masking and recovery**.

It is noteworthy that:

1. Rollback and rollforward are not mutually exclusive. Rollback may be attempted first; if the error persists, rollforward may then be attempted.
2. Intermittent faults do not necessitate isolation or reconfiguration; identifying whether a fault is intermittent or not can be performed either by error handling (error recurrence indicates that the fault is not intermittent), or via fault diagnosis when rollforward is used.
3. Fault handling may directly follow error detection, without error handling being attempted.

Preemptive error detection and handling, possibly followed by fault handling, is commonly performed at system power up. It also comes into play during operation, under various forms such as spare checking, memory scrubbing, audit programs, or so-called **software rejuvenation** [27], aimed at removing the effects of software aging before they lead to failure.

Fig. 17 gives four typical and schematic examples for the various strategies identified for implementing fault tolerance.

5.2.2 Implementation of Fault Tolerance

The choice of error detection, error handling and fault handling techniques, and of their implementation is directly related to and strongly dependent upon the underlying fault assumption: The class(es) of faults that can actually be tolerated depend(s) on the fault assumption that is being considered in the development process and, thus, relies on the *independence* of redundancies with respect to the process of fault creation and activation. A (widely used) method of achieving fault tolerance is to perform multiple computations through multiple channels, either sequentially or concurrently. When tolerance of physical faults is foreseen, the channels may be of identical design, based on the assumption that hardware components fail independently. Such an approach has proven to be adequate for elusive development faults, via rollback [23], [28]; it is however not suitable for the tolerance of solid development faults, which necessitates that the channels implement the same function via separate designs and implementations [57], [4], i.e., through **design diversity** [6].

The provision within a component of the required functional processing capability together with concurrent error detection mechanisms leads to the notion of **self-checking component**, either in hardware or in software;

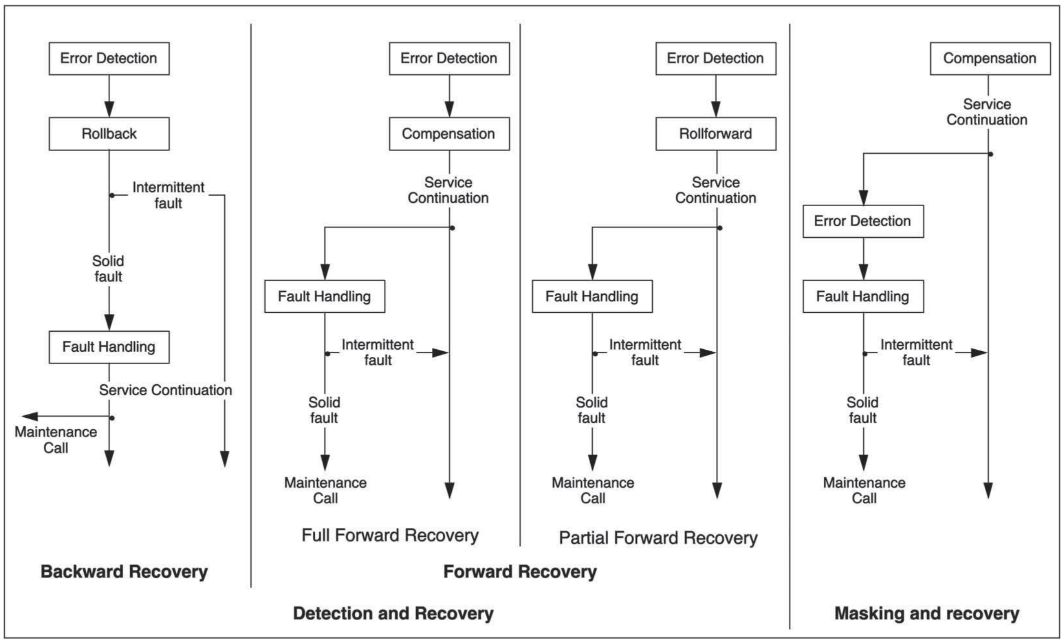


Fig. 17. Examples for the basic strategies for implementing fault tolerance.

one of the important benefits of the self-checking component approach is the ability to give a clear definition of *error confinement areas* [63].

It is evident that not all fault tolerance techniques are equally effective. The measure of effectiveness of any given fault tolerance technique is called its **coverage**. The imperfections of fault tolerance, i.e., the lack of *fault tolerance coverage*, constitute a severe limitation to the increase in dependability that can be obtained. Such imperfections of fault tolerance (Fig. 18) are due either

1. to development faults affecting the fault tolerance mechanisms with respect to the fault assumptions stated during the development, the consequence of which is a lack of *error and fault handling coverage* (defined with respect to a class of errors or faults, e.g., single errors, stuck-at faults, etc., as the conditional probability that the technique is effective, given that the errors or faults have occurred), or
2. to fault assumptions that differ from the faults really occurring in operation, resulting in a lack of *fault assumption coverage*, that can be in turn due to either 1) failed component(s) not behaving as assumed, that is a lack of *failure mode coverage*, or 2) the occurrence of common-mode failures when independent ones are assumed, that is a lack of *failure independence coverage*.

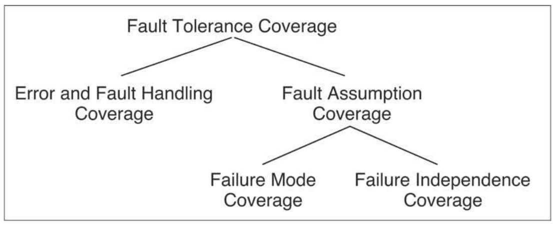

Fault Tolerance Coverage
Error and Fault Handling Coverage
Fault Assumption Coverage
Failure Mode Coverage
Failure Independence Coverage

Fig. 18. Fault tolerance coverage.

The lack of error and fault handling coverage has been shown to be a drastic limit to dependability improvement [8], [1]. Similar effects can result from the lack of failure mode coverage: conservative fault assumptions (e.g., Byzantine faults) will result in a higher failure mode coverage, at the expense of necessitating an increase in the redundancy and more complex fault tolerance mechanisms, which can lead to an overall decrease in system dependability and security [54].

An important issue in coordination of the activities of multiple components is prevention of error propagation from affecting the operation of nonfailed components. This issue becomes particularly important when a given component needs to communicate some information to other components. Typical examples of such *single-source information* are local sensor data, the value of a local clock, the local view of the status of other components, etc. The consequence of this need to communicate single-source information from one component to other components is that nonfailed components must reach an agreement as to how the information they obtain should be employed in a mutually consistent way. This is known as the *consensus* problem [43].

Fault tolerance is (also) a recursive concept: it is essential that the mechanisms that implement fault tolerance should be protected against the faults that might affect them. Examples of such protection are voter replication, self-checking checkers, "stable" memory for recovery programs and data.

Systematic introduction of fault tolerance is often facilitated by the addition of support systems specialized for fault tolerance (e.g., software monitors, service processors, dedicated communication links).

Reflection, a technique for transparently and appropriately augmenting all relevant actions of an object or software component, e.g., in order to ensure that these actions can be undone if necessary, can be used in object-oriented software and through the provision of middleware [17].

Fault tolerance applies to all classes of faults. Protection against intrusions traditionally involves cryptography and firewalls. Some mechanisms of error detection are directed towards both nonmalicious and malicious faults (e.g., memory access protection techniques). Intrusion detection is usually performed via likelihood checks [18], [15]. Approaches and schemes have been proposed for tolerating:

- intrusions and physical faults, via information fragmentation and dispersal [20], [56],
- malicious logic, and more specifically to viruses, either via control flow checking [35], or via design diversity [36],
- intrusions, malicious logic, vulnerabilities due to physical or development faults, via server diversity [68].

Finally, it is worth mentioning that 1) several synonyms exist for fault tolerance: **self-repair**, **self-healing**, **resilience**, and that 2) the term **recovery-oriented computing** [19] has recently been introduced for what is essentially a fault tolerance approach to achieving overall system dependability, i.e., at the level above individual computer systems, in which the failures of these individual systems constitute the faults to be tolerated.

5.3 Fault Removal

In this section, we consider fault removal during system development, and during system use.

5.3.1 Fault Removal During Development

Fault removal *during the development phase* of a system life-cycle consists of three steps: verification, diagnosis, and correction. We focus in what follows on **verification**, that is the process of checking whether the system adheres to given properties, termed the **verification conditions**; if it does not, the other two steps have to be undertaken: diagnosing the fault(s) that prevented the verification conditions from being fulfilled, and then performing the necessary corrections. After correction, the verification process should be repeated in order to check that fault removal had no undesired consequences; the verification performed at this stage is usually termed **nonregression verification**.

Checking the specification is usually referred to as **validation** [7]. Uncovering specification faults can happen at any stage of the development, either during the specification phase itself, or during subsequent phases when evidence is found that the system will not implement its function, or that the implementation cannot be achieved in a cost-effective way.

Verification techniques can be classified according to whether or not they involve exercising the system. Verifying a system without actual execution is **static verification**. Such verification can be conducted:

- on the system itself, in the form of 1) *static analysis* (e.g., inspections or walk-through, data flow analysis, complexity analysis, abstract interpretation, compiler checks, vulnerability search, etc.) or 2) *theorem proving*;
- on a model of the system behavior, where the model is usually a state-transition model (Petri nets, finite or infinite state automata), leading to *model checking*.

Verifying a system through exercising it constitutes **dynamic verification**; the inputs supplied to the system can be either symbolic in the case of **symbolic execution**, or actual in the case of verification testing, usually simply termed **testing**.

Fig. 19 summarizes the verification approaches.

Exhaustive testing of a system with respect to all its possible inputs is generally impractical. The methods for the determination of the test patterns can be classified according to two viewpoints: criteria for selecting the test inputs, and generation of the test inputs.

Fig. 20 summarizes the various testing approaches according to test selection. The upper part of the figure identifies the elementary testing approaches. The lower part of the figure gives the combination of the elementary approaches, where a distinction is made between hardware and software testing since hardware testing is mainly aimed at removing production faults, whereas software testing is concerned only with development faults: hardware testing is usually fault-based, whereas software testing is criteria-based, with the exception of mutation testing, which is fault-based.

The *generation* of the test inputs may be deterministic or probabilistic:

- In **deterministic testing**, test patterns are predetermined by a selective choice.
- In **random**, or **statistical**, **testing**, test patterns are selected according to a defined probability distribution on the input domain; the distribution and the number of input data are determined according to the given fault model or criteria.

Observing the test outputs and deciding whether or not they satisfy the verification conditions is known as the **oracle problem**. The verification conditions may apply to the whole set of outputs or to a compact function of the latter (e.g., a system signature when testing for physical faults in hardware, or to a "partial oracle" when testing for development faults of software [69]). When testing for physical faults, the results—compact or not—anticipated from the system under test for a given input sequence are determined by simulation or from a reference system (**golden unit**). For development faults, the reference is generally the specification; it may also be a prototype, or another implementation of the same specification in the case of design diversity (**back-to-back testing**).

Verification methods can be used in combination. For instance, symbolic execution may be used to facilitate the determination of the testing patterns, theorem proving may be used to check properties of infinite state models [60], and mutation testing may be used to compare various testing strategies [66].

As verification has to be performed throughout a system's development, the above techniques are applicable to the various forms taken by a system during its development: prototype, component, etc.

The above techniques apply also to the verification of fault tolerance mechanisms, especially 1) formal static verification [59], and 2) testing that necessitates faults or errors to be part of the test patterns, that is usually referred to as **fault injection** [2].

Verifying that the system *cannot do more* than what is specified is especially important with respect to what the system should not do, thus with respect to safety and security (e.g., **penetration testing**).

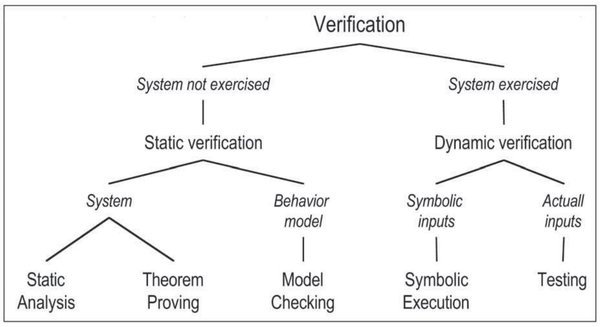


Fig. 19. Verification approaches.

Designing a system in order to facilitate its verification is termed **design for verifiability**. This approach is well-developed for hardware with respect to physical faults, where the corresponding techniques are termed **design for testability**.

5.3.2 Fault Removal During Use

Fault removal *during the use* of a system is corrective or preventive maintenance. Corrective maintenance aims to remove faults that have produced one or more errors and have been reported, while preventive maintenance is aimed at uncovering and removing faults before they might cause errors during normal operation. The latter faults include 1) physical faults that have occurred since the last preventive maintenance actions, and 2) development faults that have led to errors in other similar systems. Corrective maintenance for development faults is usually performed in stages: The fault may be first isolated (e.g., by a workaround or a patch) before the actual removal is completed. These forms of maintenance apply to nonfault-tolerant systems as well as to fault-tolerant systems, that can be maintainable online (without interrupting service delivery) or offline (during service outage).

5.4 Fault Forecasting

Fault forecasting is conducted by performing an *evaluation of the system behavior* with respect to fault occurrence or activation. Evaluation has two aspects:

- **qualitative**, or **ordinal**, **evaluation**, that aims to identify, classify, and rank the failure modes, or the event combinations (component failures or environmental conditions) that would lead to system failures;
- **quantitative**, or **probabilistic**, **evaluation**, that aims to evaluate in terms of probabilities the extent to which some of the attributes are satisfied; those attributes are then viewed as *measures*.

The methods for qualitative and quantitative evaluation are either specific (e.g., failure mode and effect analysis for qualitative evaluation, or Markov chains and stochastic Petri nets for quantitative evaluation), or they can be used to perform both forms of evaluation (e.g., reliability block diagrams, fault-trees).

The two main approaches to probabilistic fault-forecasting, aimed to derive probabilistic estimates, are *modeling* and *(evaluation) testing*. These approaches are complementary since modeling needs data on the basic processes modeled (failure process, maintenance process, system activation process, etc.), that may be obtained either by testing, or by the processing of failure data.

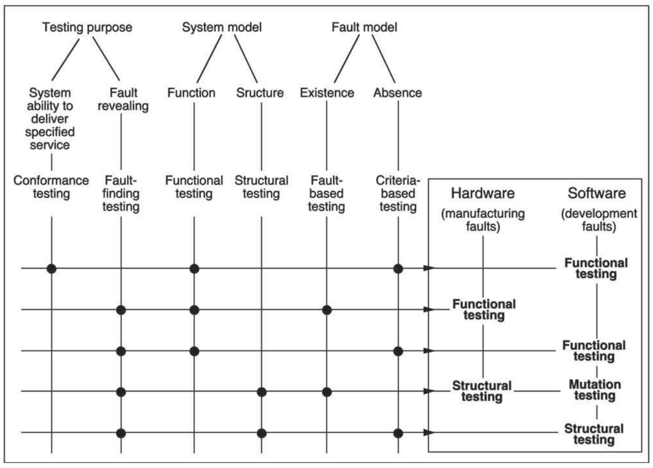


Fig. 20. Testing approaches according to test pattern selection.

Modeling can be conducted with respect to 1) physical faults, 2) development faults, or 3) a combination of both. Although modeling is usually performed with respect to nonmalicious faults, attempts to perform modeling with respect to malicious faults are worth mentioning [49], [61]. Modeling is composed of two phases:

- The *construction* of a model of the system from the elementary stochastic processes that model the behavior of the components of the system and their interactions; these elementary stochastic processes relate to failure, to service restoration including repair, and possibly to system duty cycle or phases of activity.
- *Processing* the model to obtain the expressions and the values of the dependability measures of the system.

Generally, several services can be distinguished, as well as two or more modes of service, e.g., ranging from full capacity to emergency service. These modes distinguish less and less complete service deliveries. Performance-related measures of dependability are usually subsumed into the notion of **performability** [45], [64].

Reliability growth models, either for hardware, for software, or for both, are used to perform reliability predictions from data about past system failures.

Evaluation testing can be characterized using the viewpoints defined in Section 5.3.1, i.e., conformance, functional, non-fault-based, statistical, testing, although it is not—primarily—aimed at verifying a system. A major concern is that the input profile should be representative of the operational profile [46]; hence, the usual name of evaluation testing is **operational testing**.

When evaluating fault-tolerant systems, the coverage provided by error and fault handling mechanisms has a drastic influence [8], [1] on dependability measures. The evaluation of coverage can be performed either through modeling or through testing, i.e., *fault injection*.

The notion of **dependability and security benchmark**, that is a procedure to assess measures of the behavior of a computer system in the presence of faults, enables the integration of the various techniques of fault forecasting in a unified framework. Such a benchmark enables 1) *characterization of* the dependability and security of a system, and 2) *comparison of* alternative or competitive solutions according to one or several attributes [37].

5.5 Relationships between the Means for Dependability and Security

All the "how to's" that appear in the definitions of fault prevention, fault tolerance, fault removal, fault forecasting given in Section 2 are in fact goals that can rarely if ever be fully reached since all the design and analysis activities are human activities, and thus imperfect. These imperfections bring in *relationships* that explain why it is only the *combined* utilization of the above activities—preferably at each step of the design and implementation process—that can best lead to a dependable and secure computing system. These relationships can be sketched as follows: In spite of fault prevention by means of development methodologies and construction rules (themselves imperfect in order to be workable), faults may occur. Hence, there is a need for fault removal. Fault removal is itself imperfect (i.e., all faults cannot be found, and another fault(s) may be introduced when removing a fault), and off-the-shelf components —hardware or software—of the system may, and usually do, contain faults; hence the importance of fault forecasting (besides the analysis of the likely consequences of operational faults). Our increasing dependence on computing systems brings in the requirement for fault tolerance, that is in turn based on construction rules; hence, the need again for applying fault removal and fault forecasting to fault tolerance mechanisms themselves. It must be noted that the process is even more recursive than it appears above: Current computing systems are so complex that their design and implementation need software and hardware tools in order to be cost-effective (in a broad sense, including the capability of succeeding within an acceptable time scale). These tools themselves have to be dependable and secure, and so on.

The preceding reasoning illustrates the close interactions between fault removal and fault forecasting, and motivates their gathering into **dependability and security analysis**, aimed at *reaching confidence* in the ability to deliver a service that can be trusted, whereas the grouping of fault prevention and fault tolerance constitutes **dependability and security provision**, aimed at *providing* the ability to deliver a service that can be trusted. Another grouping of the means is the association of 1) fault prevention and fault removal into **fault avoidance**, i.e., how to *aim for* fault-free systems, and of 2) fault tolerance and fault forecasting into **fault acceptance**, i.e., how to *live with* systems that are subject to faults. Fig. 21 illustrates the groupings of the means for dependability. It is noteworthy that, when focusing on security, such analysis is called *security evaluation* [32].

Besides highlighting the need to assess the procedures and mechanisms of fault tolerance, the consideration of fault removal and fault forecasting as two constituents of the same activity—dependability analysis—leads to a better understanding of the notion of coverage and, thus, of an important problem introduced by the above recursion: *the assessment of the assessment*, or how to reach confidence in the methods and tools used in building confidence in the system. **Coverage** refers here to a measure of the representativeness of the situations to which the system is subjected during its analysis compared to the actual situations that the system will be confronted with during its operational life. The notion of coverage as defined here is very general; it may be made more precise by indicating its range of application, e.g., coverage of a software test with respect to the software text, control graph, etc., coverage of an integrated circuit test with respect to a fault model, coverage of fault tolerance with respect to a class of faults, coverage of a development assumption with respect to reality.

The *assessment* of whether a system is truly dependable and, if appropriate, secure—i.e., the delivered service can justifiably be trusted—goes beyond the analysis techniques as they have been addressed in the previous sections for, at least, the three following reasons and limitations:

- Precise checking of the coverage of the design or validation assumptions with respect to reality (e.g., relevance to actual faults of the criteria used for determining test inputs, fault hypotheses in the design of fault tolerance mechanisms) would imply a knowledge and a mastering of the technology used, of the intended utilization of the system, etc., that exceeds by far what is generally achievable.
- The evaluation of a system for some attributes of dependability, and especially of security with respect to certain classes of faults is currently considered as unfeasible or as yielding nonsignificant results because probability-theoretic bases do not exist or are not yet widely accepted; examples are safety with respect to accidental development faults, security with respect to intentional faults.
- The specifications with respect to which analysis is performed are likely to contain faults—as does any system.

Among the numerous consequences of this state of affairs, let us mention:

- The emphasis placed on the development process when assessing a system, i.e., on the methods and techniques utilized in development and how they are employed; in some cases, a *grade* is assigned and delivered to the system according to 1) the nature of the methods and techniques employed in development, and 2) an assessment of their utilization [51], [58], [32], [65].
- The presence, in the specifications of some fault-tolerant systems (in addition to probabilistic requirements in terms of dependability measures), of a list of types and numbers of faults that are to be tolerated; such a specification would not be necessary if the limitations mentioned above could be overcome (such specifications are classical in aerospace applications, under the form of a concatenation of "fail-operational" (FO) or "fail-safe" (FS) requirements, e.g., FO/FS, or FO/FO/FS, etc.).

6 Conclusion

Increasingly, individuals and organizations are developing or procuring sophisticated computing systems on whose services they need to place great trust—whether to service a set of cash dispensers, control a satellite constellation, an airplane, a nuclear plant, or a radiation therapy device, or to maintain the confidentiality of a sensitive data base. In differing circumstances, the focus will be on differing properties of such services—e.g., on the average real-time response achieved, the likelihood of producing the required results, the ability to avoid failures that could be catastrophic to the system's environment, or the degree to which deliberate intrusions can be prevented. Simultaneous consideration of dependability and security provides a very convenient means of subsuming these various concerns within a single conceptual framework. It includes as special

(3) *Permanent, External, Physical Faults*. These are faults induced on the system by the physical environment.

(4) *Temporary, External, Physical Faults (also known as Transient Faults)* [Bondavalli et al. 1997]. These are faults induced by environmental phenomena, such as EMI.

Design Faults. The next category of fault class consits of those faults introduced at design time.

(1) *Intentional, though Not Malicious, Permanent / Temporary Design Faults*. These are basically tradeoffs introduced at application-layer design time. A typical example is insufficient dimensioning (underestimations of the size of a given field in a communication protocol[2].)

(2) *Accidental, Permanent, Design Faults (also called Systematic Faults, or Bohrbugs)*. These are flawed algorithms that systematically make the same errors in the presence of the same input conditions and initial states; for instance, an unchecked divisor can result in a division-by-zero error.

(3) *Accidental, Temporary Design Faults (known as Heisenbugs, as in "bugs of Heisenberg", for their elusive character).* While systematic faults have an evident, deterministic behavior, these bugs depend on subtle combinations of the system state and environment.

Interaction Faults. The final fault class is comprised of those faults that occur during interactions.

(1) *Temporary, External, Operational, Human-Made Accidental Faults.* These include operator faults in which an operator does not correctly perform his or her role in system operation.

(2) *Temporary, External, Operational, Human-Made Nonmalicious Faults.* These faults arise through "neglect, interaction, or incorrect use problems" [Sibley 1998]. Examples include poorly chosen passwords and bad system parameter setting.

(3) *Temporary, External, Operational, Human-Made Malicious Faults.* This class includes the so-called malicious replication faults, namely faults that occur when the replicated information in a system becomes inconsistent, for example, because processes that are supposed to provide identical results no longer do so.

2.1. A Need for Software Fault-Tolerance

Research in fault tolerance concentrated for many years on hardware fault tolerance, that is, on devising a number of effective and ingenious hardware structures to cope with faults [Johnson 1989]. For some time this approach was considered as the only one needed in order to reach the availability and data integrity demands of modern, complex computer services. Probably the first researcher to realize this was far from assumption true was B. Randell, who in Randell [1975] questioned hardware fault tolerance as the only approach to pursue; in the cited paper he states:

[2]A noteworthy example is given by the bad dimensioning of IP addresses which gave raise to IPv6.

Hardware component failures are only *one* source of unreliability in computing systems, decreasing in significance as component reliability improves, while software faults have become increasingly prevalent with the steadily increasing size and complexity of software systems.

Indeed, most of the complexity supplied by modern computing services lies in their software, rather than hardware, layer [Lyu 1998a, 1998b; Huang and Kintala 1995; Wiener 1993; Randell 1975]. This phenomenon could only be reached by exploiting a powerful conceptual tool for managing complexity in a flexible and effective way, that is, by devising hierarchies of sophisticated abstract machines [Tanenbaum 1990]. This translates into implementing software with high-level computer languages lying on top of other software strata: the device-driver layers, basic services kernel, operating system, runtime support of the involved programming languages, and so forth.

Partitioning the complexity into stacks of software layers allowed the implementors to focus exclusively on the high-level aspects of their problems, and hence allowed them to manage greater and greater degrees of complexity. However, though made transparent, this complexity is still part of the overall system being developed. A number of complex algorithms are executed by the hardware at the same time, resulting in the simultaneous progress of many system states under the hypothesis that neither the involved abstract machine nor the actual hardware will be affected by faults. Unfortunately, since in real life faults do occur, the corresponding deviations are likely to jeopardize the system's function. Moreover, faults can propagate from one layer to another, unless appropriate means are taken to either remove or tolerate these faults, or to avoid creating them in the first place. Furthermore, faults may also occur in the *application layer*, that is, in the abstract machine on top of the software hierarchy.[3] These faults, possibly having their origin at design time, during operation, or while interacting with the environment, do not differ in terms of their consequences from those faults originating, in the hardware or operating system. A well-known example is the case of Ariane 5 flight 501 [Inquiry Board Report 1996], in which the consequences of a fault in the application ultimately caused the system to crash. In general we observe that the higher the level of abstraction, the higher the complexity of the algorithms that come into play and consequent error proneness of the involved (real or abstract) machines. As such, we conclude that full tolerance of faults and complete fulfilment of the dependability design goals of a complex software application must include means to avoid, remove, or tolerate faults working at all, including application, layers. This article focuses on runtime detection and recovery of faults through mechanisms either residing in or cooperating with the application layer.

2.2. Software Fault-Tolerance in the Application Layer

The need for software fault tolerance provisions, located in the application layer, is supported by studies showing that the majority of failures experienced by modern computer systems are due to software faults, including those located in the application layer [Lyu 1998a, 1998b; Avižienis et al. 2004a, 2004b]; for instance, NRC reported that 81% of the total number of outages of US switching systems in 1992 were due to software faults [NRC 1993]. Moreover, modern application software systems are increasingly networked and distributed. Such systems, such as client-server applications, are often characterized by a loosely coupled architecture whose global structure is in general

[3]In what follows, the application layer is intended as the programming and execution context in which a complete, self-contained program that performs a specific function directly for the user is expressed or running.

more prone to failures.[4] Due to the complex and temporal nature of the interleaving of messages and computations in distributed software systems, no amount of verification, validation, and testing can eliminate all faults in an application and give complete confidence in the availability and data consistency of applications of this kind [Huang and Kintala 1995]. Under these assumptions, the only alternative (and effective) means for increasing software reliability is that of incorporating in the application software provisions for software fault tolerance [Randell 1975].

Another argument that justifies the addition of software fault tolerance means in the application layer is given by the widespread adoption of reusable software components. Approaches such as object orientation, component-based middleware, and service orientation have provided the designer with effective tools to compose systems out of, for example, COTS object libraries, third-party components, and Web services. For instance, many object-oriented applications are indeed built from reusable components, the sources of which are unknown to the application developers. The aforementioned approaches fostered the capability of dealing with higher levels of complexity in software and at the same time eased and therefore encouraged software reuse. This has a big, positive impact on development costs, but turns the application into a sort of collection of reused, preexisting "blocks" made by third parties. The reliability of these components and therefore their impact on the overall reliability of the user application is often unknown, to the extent that Green defines as "art" creating reliable applications using off-the-shelf software components [Green 1997]. The case of the Ariane 501 flight is a well-known example that shows how improper reuse of software may have severe consequences [Inquiry Board Report 1996].[5]

However, probably the most convincing argument for not excluding the application layer from a fault tolerance strategy is the so-called "end-to-end argument," a system design principle introduced by Saltzer et al. [1984]. This principle states that, rather often, functions such as reliable file transfer can be completely and correctly implemented only with the knowledge and help of the application standing at the endpoints of the underlying system (e.g., the communication network).

This does not mean that everything should be done at application level; fault tolerance strategies in the underlying hardware and operating system can have a strong impact on system performance. However, an extraordinarily reliable communication system, guaranteeing that no packet is lost, duplicated, corrupted, or delivered to the wrong addressee, does not reduce the burden on the application program of ensuring reliability: For instance, for reliable file transfer, the application programs that perform the transfer must still supply a file-transfer-specific end-to-end reliability guarantee.

Hence one can conclude that:

> Pure hardware-based or operating system-based solutions to fault-tolerance, though often characterized by a higher degree of transparency, are not fully capable of providing complete end-to-end tolerance to faults in the user application. Furthermore, relying solely on the hardware and the operating system develops only partially satisfying solutions; requires a large amount of extra resources and costs; and is often characterised by poor service portability [Saltzer et al. 1984; Siewiorek and Swarz 1992].

[4]As Leslie Lamport effectively summarized, "a distributed system is one in which I cannot get something done because a machine I've never heard of is down."

[5]The Ariane 5 program reused the extensively tested software used in Ariane 4. As such, the software had been thoroughly tested and complied to Ariane 4 specifications. Unfortunately, with the specifications for Ariane 5 were different. A dormant design fault had never been unraveled, simply because the operating conditions of Ariane 4 were different from those of Ariane 5. This failure entailed a loss of about 370 million euros [Le Lann 1996].

2.3. Strategies, Problems, and Key Properties

The aforesaid conclusions justify the strong need for ALFT; as a consequence of this need, several approaches to ALFT have been devised during the last three decades (see Section 3 for a brief survey). Such a long research period hints at the complexity of the design problems underlying ALFT engineering, which include:

(1) how to incorporate fault tolerance in the application layer of a computer program;
(2) which fault-tolerance provisions to support; and
(3) how to manage the fault-tolerant code.

Problem 1 is also known as the problem of the *system-structure-to-software fault tolerance*, first proposed by Randell [1975]. It states the need of appropriate structuring techniques such that the incorporation of a set of fault-tolerance provisions in the application software might be performed in a simple, coherent, and well-structured way. Indeed, poor solutions to this problem result in a huge degree of *code intrusion*: In such cases, the application code addressing functional requirements and that addressing fault-tolerance requirements are intermixed into one large and complex application software. We next give three consequent obstacles to software fault-tolerance design.

—Amalgamating these two types of application code greatly complicates the task of the developer and requires expertise in both the application domain and in fault tolerance. Negative repercussions on the development times and costs are to be expected.
—The maintenance of the resulting code, both for the functional part and for the fault-tolerance provisions, is more complex, costly, and error prone.
—Furthermore, the overall complexity of the software product is increased, which is detrimental to its resilience to faults.

One can conclude that, with respect to the first problem, an ideal system structure should guarantee an adequate *Separation between the functional and fault-tolerance Concerns* (SC).

Moreover, the design choice of which fault-tolerance provisions to support can be conditioned by the adequacy of the syntactical structure at "hosting" the various provisions. The well-known quotation by B. L. Whorf succinctly captures this concept:

> Language shapes the way we think, and determines what we can think about.

Indeed, as explained in Section 2.3.1, a nonoptimal answer to problem 2 may:

—require a high degree of redundancy, and
—rapidly consume large amounts of the available redundancy,

thus increasing the costs and reducing reliability. One can conclude that devising a syntactical structure offering straightforward support to a large set of fault-tolerance provisions can be an important aspect of an ideal system structure for ALFT. In the following, this property will be called *syntactical adequacy* (SA).

Finally, one can observe that another important aspect of an ALFT architecture is the way the fault-tolerant code is managed, at compile time as well as at runtime. Evidence for this statement can be found by observing that a number of important choices pertaining to the adopted fault-tolerance provisions, such as the parameters of a temporal redundancy strategy, are a consequence of an analysis of the environment in which the application is to be deployed and run.[6] In other words, depending on the

[6]For instance, if an application is to be moved from a domestic environment to another one characterized by a higher electro-magnetic interference (EMI), it is reasonable to assume that, for example, the number of replicas of some protected resource should be increased accordingly.

target environment, the set of (external) impairments that might affect the application can vary considerably. Now, while it may be in principle straightforward to port an existing code to another computer system, *porting the service* supplied by that code may require a proper adjustment of the aforementioned choices, namely the parameters of the adopted provisions [De Florio and Blondia 2007a]. Effective support towards managing the parametrization of the fault-tolerant code, and of its maintenance in general, could guarantee *fault-tolerance software reuse*. Therefore, offline and online (dynamic) management of fault-tolerance provisions and their parameters may be an important requirement for any satisfactory solution of problem 3. As further motivated in Section 2.3.1, ideally the fault-tolerant code should *adapt* itself to the current environment. Furthermore, any satisfactory management approach should not overly increase the complexity of the application, as this would be detrimental to dependability. Let us call this property *adaptability* (A).

Let us refer collectively to properties SC, SA, and A as the *structural attributes* of ALFT.

The various approaches to ALFT surveyed in Section 3 provide different system structures to solve the aforementioned problems. Three structural attributes are used in that section in order to provide a qualitative assessment with respect to various application requirements. These structural attributes constitute, in a sense, a base with which to perform this assessment. One of the major conclusions of the survey therein is that none of the surveyed approaches is capable of providing the best combination of values of the three structural attributes in every application domain. For specific domains such as object-oriented distributed applications, satisfactory solutions have been devised, at least for SC and SA Nonethless, only partial solutions exist, for instance, when dealing with the class of distributed or parallel applications not based on the object model.

The aforesaid obstruction has, as a matter of fact, been efficaciously captured by Lyu, who calls this situation "the software bottleneck" of system development [Lyu 1998b]: In other words, there is evidence of an urgent need for systematic approaches to assure software reliability within a system [Lyu 1998b] while effectively addressing the previously described problems. In the cited paper, Lyu remarks how "developing the required techniques for software reliability engineering is a major challenge to computer engineers, software engineers and engineers of related disciplines."

2.3.1. Fault-Tolerance, Redundancy, and Complexity. A well-known result by Shannon [1993] tells us that from any unreliable channel, it is possible to set-up a more reliable one by increasing the degree of information redundancy. This means that it is possible to tradeoff reliability and redundancy of a channel. The authors observe that the same can be said for a fault-tolerant system because fault tolerance is, in general, the result of some strategy effectively exploiting some form of redundancy: time-, information-, and/or hardware redundancy [Johnson 1989]. This redundancy has a cost penalty attached, however. Its ability to address a weak failure semantics and to span many failure behaviors effectively translates into higher reliability. Regardless, redundancy:

(1) requires large amounts of extra resources, and therefore implies a high cost penalty; and
(2) consumes large amounts of extra resources, which translates into rapid exhaustion of the extra resources.

For instance, Lamport et al. [1982] set the minimum level of redundancy required for tolerating Byzantine failures to a value greater than that required for tolerating, for example, value failures. Using the simplest of the algorithms described in the cited paper, a 4-modular-redundant (4-MR) system can only withstand a single Byzantine failure, while the same system may exploit its redundancy to withstand up to three crash faults (though no other kind of fault) [Powell 1997]. In other words:

> After the occurrence of a crash fault, a 4-MR system with strict Byzantine failure semantics has exhausted its redundancy and is no more dependable than a no[nr]edundant system supplying the same service, while the crash failure semantics system is able to survive to the occurrence of that and two other crash faults. On the other hand, the latter system, subject to just one Byzantine fault, would fail regardless its redundancy.

We can conclude that for any given level of redundancy, trading the complexity of the failure mode against the number and types of faults tolerated may be an important capability for an effective fault-tolerant structure. Dynamic adaptability to different environmental conditions[7] may provide a satisfactory answer to this need, especially when this additional complexity does not burden (and hence jeopardize) the application. Ideally, such complexity should be part of a custom architecture and not part of the application. On the other hand, an embedding in the application of complex failure semantics covering many failure modes implicitly promotes complexity, as it may require the implementation of many recovery mechanisms. This complexity is detrimental to the dependability of the system, as it is in itself a significant source of design faults. Furthermore, the isolation of this complexity outside the user application may allow cost-effective verification, validation, and testing. These processes may be unfeasible at the application level.

The authors conjecture that a satisfactory solution to the design problem of management of the fault-tolerant code (presented in Section 2.3) may translate into an optimal management of the failure semantics (with respect to the involved penalties). In other words, we conjecture that linguistic structures characterized by high *adaptability* (A) may be better suited to cope with the preceding problems.

3. CURRENT APPROACHES TO APPLICATION-LEVEL FAULT TOLERANCE

One of the conclusions drawn in Section 1 is that for the system to be made fault tolerant, we must also include provisions for fault tolerance in the application layer of a computer program. In this context, the problem of which system structure to use for ALFT has been proposed. This section provides a critical survey of the state-of-the-art on embedding fault-tolerance means in the application layer.

According to the literature, at least six classes of method can be used for embedding fault-tolerance provisions in the application layer of a computer program. This section describes these approaches and points out positive and negative aspects of each with respect to the structural attributes defined in Section 2.3, as well as to various application domains. A nonexhaustive list of the systems and projects implementing these approaches is also given. Conclusions are drawn in Section 4, where the need for more effective approaches is recognized.

Two of the previously mentioned approaches derive from well-established research in software fault-tolerance; Lyu [1998b, 1996, 1995] refers to them as single-version and multiple-version software fault tolerance. They are dealt with in Section 3.1. A third approach, described in Section 3.2, is based on metaobject protocols. It is derived from the domain of object-oriented design and can also be used for embedding services other than those related to fault tolerance. A fourth approach translates into developing

[7]The following quote by J. Horning [1998] captures very well how relevant may be the role of the environment with respect to achieving the required quality of service: "What is the most often overlooked risk in software engineering? That the environment will do something the designer never anticipated."

new, custom high-level distributed programming languages or enhancing preexistent languages of that kind. It is described in Section 3.3. Aspect-oriented programming as a fault-tolerance structuring technique is discussed in Section 3.4. Finally, Section 3.5 describes an approach based on a special recovery task monitoring the execution of the user task.

3.1. Single- and Multiple-Version Software Fault-Tolerance

A key requirement for the development of fault-tolerant systems is the availability of *replicated resources* in hardware or software. A fundamental method employed to attain fault tolerance is *multiple computation*, namely, N-fold ($N \geq 2$) replications in the three domains next given.

—*Time*. This refers to repetition of computations.
—*Space*. This is replication as the adoption of multiple hardware channels (also called "lanes").
—*Information*. This means the adoption of multiple versions of software.

Following Avižienis [1985], it is possible to characterize at least some of the approaches towards fault tolerance by means of a notation resembling the one used to classify queueing systems models [Kleinrock 1975]. Specifically,

$$n\mathrm{T}/m\mathrm{H}/p\mathrm{S},$$

the meaning of which is "n executions, on m hardware channels, of p programs." The nonfault-tolerant system, or 1T/1H/1S, is called a *simplex* in the cited paper.

3.1.1. Single-Version Software Fault-Tolerance. Single-version software fault-tolerance (SV) is basically the embedding into the user application of a simplex system of error detection or recovery features, for example, atomic actions [Jalote and Campbell 1985], checkpoint-and-rollback [Elnozahy et al. 2002], or exception handling [Cristian 1995]. The adoption of SV in the application layer requires the designer to concentrate, in one physical location (i.e., the source code of the application), both the specification of what to do in order to perform some user computation and the strategy such that faults are tolerated when they occur. As a result, the size of the problem addressed is increased. *A fortiori*, this translates into an increase in size of the user application, which induces loss of transparency, maintainability, and portability while increasing development times and costs.

A partial solution to this loss in portability and these higher costs is given by the development of libraries and frameworks created under strict software engineering processes. In the following, two examples of this approach are presented: the EFTOS library and the SwiFT system.

—*The EFTOS Library.* EFTOS [De Florio et al. 1998a] (i.e., "Embedded, Fault-Tolerant Supercomputing") is the name of ESPRIT-IV project 21012. The aims of this project were to investigate approaches to add fault tolerance to embedded high-performance applications in a flexible and cost-effective way. The EFTOS library was first implemented on a Parsytec CC system [Parsytec 1996], a distributed-memory MIMD supercomputer consisting of processing nodes based on PowerPC 604 microprocessors at 133MHz, dedicated high-speed links, I/O modules, and routers.

Through adoption of the EFTOS library, the target embedded parallel application is plugged into a hierarchical, layered system whose structure and basic components (depicted in Figure 2) are described as follows.

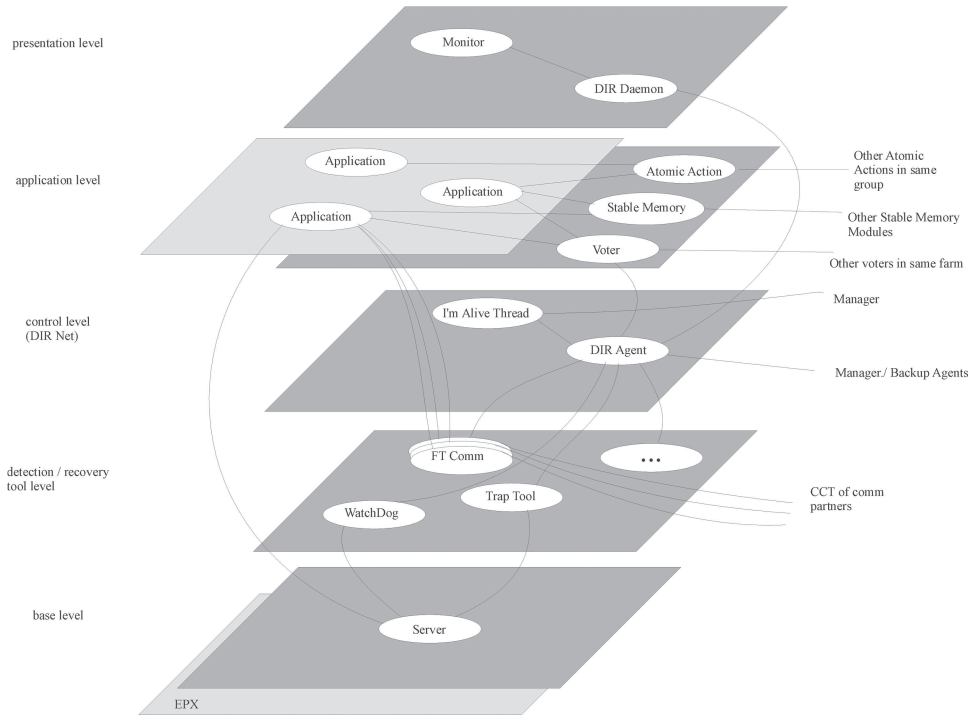


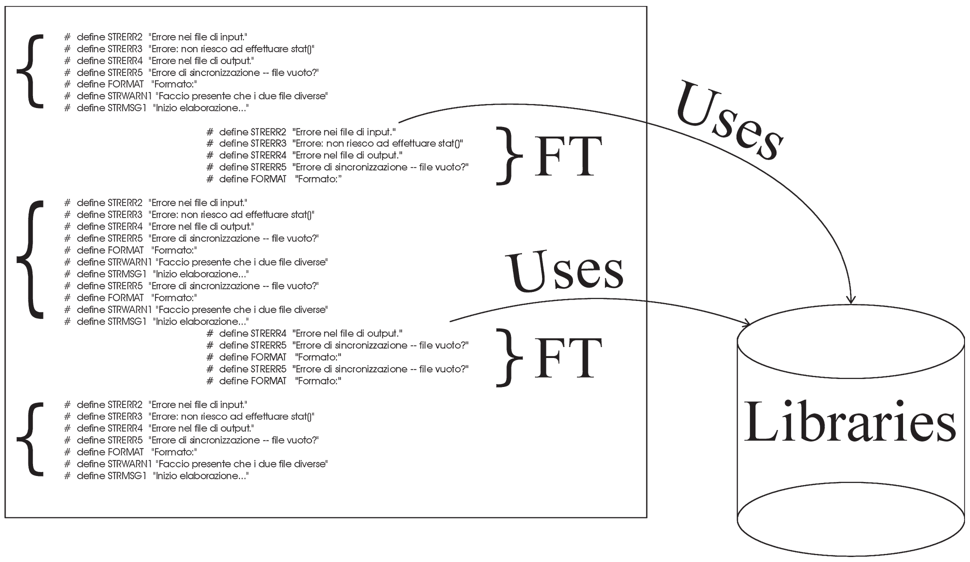
Fig. 2. The structure of the EFTOS library. Light gray has been used for the operating system and the user application, while dark gray layers pertain to EFTOS.

—At the base level, there is a distributed net of "servers," whose main task is mimicking some possibly missing (with respect to POSIX standards) operating system functionalities such as remote thread creation.

—One level upward (at the detection tool layer) there exists a set of parameterizable functions managing error detection, referred to as "Dtools." These basic components are plugged into the embedded application to make it more dependable. EFTOS supplies a number of these Dtools (e.g., a watchdog timer thread and a trap-handling mechanism) plus an API for incorporating user-defined EFTOS-compliant tools.

—At the third level (control layer), a distributed application called "DIR net" (its name stands for "detection, isolation, and recovery network") is used to coherently combine the Dtools. This is done to ensure consistent fault-tolerance strategies throughout the system, and to play the role of a backbone handling information to and from the fault-tolerance elements [De Florio et al. 2000; De Florio 1998]. The DIR net can be regarded as a fault-tolerant network of crash-failure detectors connected to other peripheral error detectors. Each node of the DIR net is "guarded" by an <I'm Alive> thread that requires the local component to periodically send "heartbeats" (signs of life). A special component, called RINT, manages error recovery by interpreting a custom language called RL [De Florio and Deconinck 2002; De Florio 1997a].

—At the fourth level (application layer), the Dtools and components of the DIR net are combined into dependable mechanisms; that is, methods to guarantee fault-tolerant communication [Efthivoulidis et al. 1998], tools implementing a virtual stable memory [De Florio et al. 2001], a distributed voting mechanism called the "voting farm" [De Florio 1997b; De Florio et al. 1998a, 1998b], and so forth.



```
{  #  define STRERR2  "Errore nei file di input."
   #  define STRERR3  "Errore: non riesco ad effettuare stat()"
   #  define STRERR4  "Errore nel file di output."
   #  define STRERR5  "Errore di sincronizzazione -- file vuoto?"
   #  define FORMAT   "Formato:"
   #  define STRWARN1 "Faccio presente che i due file diverse"
   #  define STRMSG1  "Inizio elaborazione..."

                        #  define STRERR2  "Errore nei file di input."
                        #  define STRERR3  "Errore: non riesco ad effettuare stat()"
                        #  define STRERR4  "Errore nel file di output."                  } FT
                        #  define STRERR5  "Errore di sincronizzazione -- file vuoto?"
                        #  define FORMAT   "Formato:"

{  #  define STRERR2  "Errore nei file di input."
   #  define STRERR3  "Errore: non riesco ad effettuare stat()"
   #  define STRERR4  "Errore nel file di output."
   #  define STRERR5  "Errore di sincronizzazione -- file vuoto?"
   #  define FORMAT   "Formato:"
   #  define STRWARN1 "Faccio presente che i due file diverse"
   #  define STRMSG1  "Inizio elaborazione..."
   #  define STRERR5  "Errore di sincronizzazione -- file vuoto?"
   #  define FORMAT   "Formato:"
   #  define STRWARN1 "Faccio presente che i due file diverse"
   #  define STRMSG1  "Inizio elaborazione..."
                        #  define STRERR4  "Errore nel file di output."
                        #  define STRERR5  "Errore di sincronizzazione -- file vuoto?"
                        #  define FORMAT   "Formato:"                                    } FT
                        #  define STRERR5  "Errore di sincronizzazione -- file vuoto?"
                        #  define FORMAT   "Formato:"

{  #  define STRERR2  "Errore nei file di input."
   #  define STRERR3  "Errore: non riesco ad effettuare stat()"
   #  define STRERR4  "Errore nel file di output."
   #  define STRERR5  "Errore di sincronizzazione -- file vuoto?"
   #  define FORMAT   "Formato:"
   #  define STRWARN1 "Faccio presente che i due file diverse"
   #  define STRMSG1  "Inizio elaborazione..."
```

Fig. 3. A fault-tolerant program according to an SV system.

—The highest level (presentation layer) is given by a hypermedia distributed application which monitors the structure and state of the user application [De Florio et al. 1998c]. This application is based on a special CGI script [Kim 1996] called DIR Daemon, which continuously takes its inputs from the DIR net, translates them into HTML, and remotely controls a Netscape browser [Zawinski 1994] so that it renders this HTML data.

—*The SwiFT System.* SwiFT [Huang et al. 1996] stands for "software-implemented fault tolerance." It includes a set of reusable software components consisting of `watchd`, a general-purpose UNIX daemon watchdog timer; `libft`, a library of fault-tolerance methods including a single-version implementation of recovery blocks and N-version programming (see Section 3.1.2); `libckp`, which is a user-transparent checkpoint-and-rollback library; a file replication mechanism called `REPL`; and `addrejuv`, a special "reactive" feature of `watchd` [Huang et al. 1995] that allows for software rejuvenation.[8] SwiFT has been successfully used and proved an efficient and economical means to increase the level of fault tolerance in a software system where residual faults are present and their toleration is less costly than their full elimination [Lyu 1998b]. A relatively small overhead is introduced in most cases [Huang and Kintala 1995].

Conclusions. Figure 3 shows the main characteristics of the SV approach: The functional and fault-tolerant codes are intertwined and the developer has to deal with the two concerns at the same time, even with the help of libraries of fault-tolerance provisions. In other words, SV requires the application developer to be an expert in fault tolerance as well, because he (she) has to integrate in the application a number of fault-tolerance provisions among those available in a set of ready-made basic tools. His (her) responsibility is to do so in a coherent, effective, and efficient way. As observed in

[8]Software rejuvenation [Huang et al. 1995; Bao et al. 2003] offers tools for periodical and graceful termination of an application with immediate restart, so that possible erroneous internal states due to transient faults are wiped out before they cause a failure.

Section 2.3, the resulting code is a mixture of functional- and custom error-management code that does not always offer an acceptable degree of portability and maintainability. The functional and nonfunctional design concerns are not kept apart from SV, hence one can conclude that (qualitatively) SV exhibits poor separation of concerns (SC). This in general has a bad impact on design and maintenance costs.

As for syntactical adequacy (SA), we observe that, following SV, the fault-tolerance provisions are offered to the user through an interface based on a general-purpose language such as C or C++. As a consequence, very limited SA can be achieved by SV as a system structure for ALFT.

Furthermore, no support is provided for offline and online configuration of the fault-tolerance provisions. Consequently, we regard the adaptability (A) of this approach as insufficient.

On the other hand, tools in SV libraries and systems give the user the ability to deal with fault-tolerance "atoms" without having to worry about their actual implementation. Moreover, these tools provide a good ratio of cost over improvement of the dependability attributes, sometimes introducing a relatively small overhead. Using these tool sets the designer can reuse existing, long-tested, and sophisticated pieces of software without having each time to "reinvent the wheel."

Finally, it is important to remark that, in principle, SV poses no restrictions on the class of applications that may be tackled with it.

3.1.2. Multiple-Version Software Fault-Tolerance. This section describes multiple-version software fault-tolerance (MV), an approach which requires N ($N \geq 2$) independently designed versions of software. MV systems are therefore xT/yH/NS systems. In MV, the same service or functionality is supplied by N pieces of code that have been designed and developed by different independent software teams.[9] The aim of this approach is to reduce the effects of design faults due to human mistakes committed at design time. The most frequently used configurations are NT/1H/NS; that is, N sequentially applicable alternate programs using the same hardware channel, and 1T/NH/NS, which is based on the parallel execution of alternate programs on N (possibly diverse) hardware channels.

Two major approaches exist: The first is known as a recovery block [Randell 1975; Randell and Xu 1995], and is dealt with next. Then, the second approach, the so-called N-version programming [Avižienis 1995, 1985], is described.

—*The Recovery Blocks Technique.* Recovery blocks are usually implemented as NT/1H/NS systems. The technique addresses residual software design faults. It aims at providing fault-tolerant functional components which may be nested within a sequential program. Other versions of the approach, implemented as 1T/NH/NS systems, are suited for parallel or distributed programs [Scott et al. 1985; Randell and Xu 1995].

The recovery blocks technique is similar to the hardware fault-tolerance approach known as "standby sparing," which is described, for example, in Johnson [1989]. The approach is summarized in Figure 4: On entry to a recovery block, the current state of the system is checkpointed. A primary alternate is executed. When it ends, an acceptance test checks whether the primary alternate successfully accomplished its objectives. If it has not, a backward recovery step reverts the system state back to its original value and a secondary alternate takes over the task of the primary alternate. When the secondary

[9]This requirement is well explained by Randell [1975]: "All fault-tolerance must be based on the provision of useful redundancy, both for error detection and error recovery. In software the redundancy required is not simple replication of programs but *redundancy of design*." Footnote 5 briefly reports on the consequences of a well-known case of redundant design.

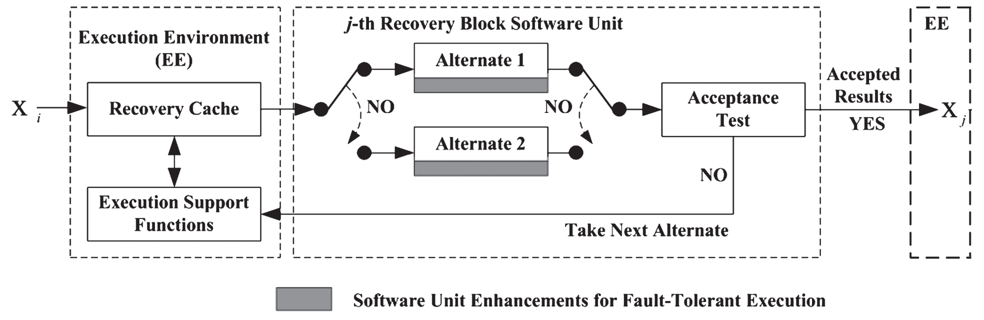

Execution Environment (EE)
X i
Recovery Cache
Execution Support Functions
j-th Recovery Block Software Unit
Alternate 1
Alternate 2
NO
NO
Acceptance Test
Accepted Results
YES
NO
Take Next Alternate
EE
X j
Software Unit Enhancements for Fault-Tolerant Execution

Fig. 4. The recovery block model with two alternates. The execution environment is charged with the management of the recovery cache and execution support functions (used to restore the state of the application when the acceptance test is not passed), while the user is responsible for supplying both alternates and the acceptance test.

alternate ends, the acceptance test is executed again. The strategy goes on until either an alternate fulfils its tasks or all alternates are executed without success. In such a case, an error routine is executed. Recovery blocks can be nested: In this case the error routine invokes recovery in the enclosing block [Randell and Xu 1995]. An exception triggered within an alternate is managed as a failed acceptance test. A possible syntax for recovery blocks is as follows.

```
ensure       acceptance test
by           primary alternate
else by      alternate 2
     .
     .
else by      alternate N
else error
```

Note how this syntax does not explicitly show the recovery step that should be carried out transparently by a runtime executive.

The effectiveness of recovery blocks rests to a great extent on the coverage of the error detection mechanisms adopted, the most crucial component of which is the acceptance test. A failure of the acceptance test is a failure of the whole recovery blocks strategy. For this reason, the acceptance test must be simple, must not introduce huge runtime overheads, must not retain data locally, and so forth. It must be regarded as the ultimate means for detecting errors, though not an exclusive one. Assertions and runtime checks, possibly supported by underlying layers, need to buttress the strategy and reduce the probability of an acceptance test failure. Another possible failure condition for the recovery blocks approach is given by an alternate failing to terminate. This may be detected by a time-out mechanism that could be added to the recovery blocks. This addition obviously further increases the complexity.

The SwiFT library described in Section 3.1.1 implements recovery blocks in the C language as follows.

```
#include <ftmacros.h>
...
ENSURE(acceptance-test) {
           primary alternate;
} ELSEBY {
   alternate 2;
} ELSEBY {
   alternate 3;
}
...
ENSURE;
```

Unfortunately, this approach does not cover any of the aforementioned requirements for enhancing the error detection coverage of the acceptance test. This would clearly require a runtime executive that is not part of this strategy. Other solutions, based on enhancing the grammar of preexisting programming languages such as Pascal [Shrivastava 1978] and Coral [Anderson et al. 1985], have some impact on portability. In both cases, code intrusion is not avoided. This translates into difficulties when trying to modify or maintain the application program without interfering much with the recovery structure, and vice-versa: as when trying to modify or maintain the recovery structure without interfering much with the application program. Hence, one can conclude that recovery blocks are characterized by unsatisfactory values of the structural attribute SC. Furthermore, a system structure for ALFT based exclusively on recovery blocks does not satisfy attribute SA.[10] Finally, regarding attribute A, one can observe that recovery blocks comprise a rigid strategy that does not allow offline configuration nor (*a fortiori*) code adaptability.

On the other hand, recovery blocks have been successfully adopted throughout the last 30 years in many different application fields. It has been successfully validated by a number of statistical experiments and through mathematical modeling [Randell and Xu 1995]. Its adoption as the sole fault-tolerance means while developing complex applications has resulted, in some cases [Anderson et al. 1985], in a failure coverage of over 70%, with acceptable overheads in memory space and CPU time.

A negative aspect in any MV system is given by development and maintenance *costs* that grow as a monotonic function of x, y, z in any xT/yH/zS system.

—**N**-*Version Programming.* N-version programming (NVP) systems are built from generic architectures based on redundancy and consensus. Such systems usually belong to class 1T/NH/NS, and less often to class NT/1H/NS. Specifically, NVP is defined by its author Avižienis [1985] as "the independent generation of $N \geq 2$ functionally equivalent programs from the same initial specification." These N programs, called versions, are developed to be executed in parallel. This system constitutes a fault-tolerant software unit that depends on a generic decision algorithm to determine a consensus or majority result from the individual outputs of two or more versions of the unit.

Such a strategy (depicted in Figure 5) has been developed under the fundamental conjecture that independent designs translate into random component failures, namely, statistical independence. Such a result would guarantee that correlated failures do not translate into immediate exhaustion of the available redundancy, as would happen, for example, using N copies of the same version. Replicating software would also mean replicating any dormant software fault in the source version; see, for example, the accidents with the Therac-25 linear accelerator [Leveson 1995] or Ariane 5 flight 501 [Inquiry Board Report 1996]. According to Avižienis, independent generation of the versions significantly reduces the probability of correlated failures. Unfortunately,

[10]Randell himself states that, given the ever-increasing complexity of modern computing, there is still an urgent need for "richer forms of structuring for error recovery and for design diversity" [Randell and Xu 1995].

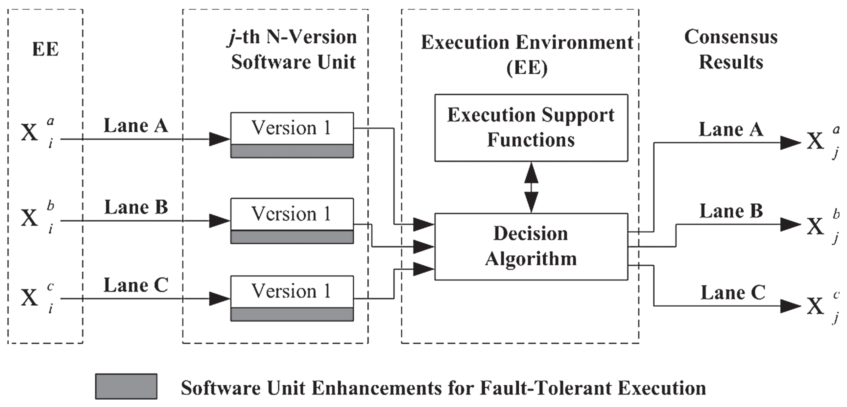


Fig. 5. The N-version software model when $N = 3$. The execution environment is charged with the management of the decision algorithm and with the execution support functions. The user is responsible for supplying the N versions. Note how the decision algorithm box takes care also of multiplexing its output onto the three hardware channels, also called "lanes."

a number of experiments [Eckhardt et al. 1991] and theoretical studies [Eckhardt and Lee 1985] have shown that this assumption is not always correct.

The main differences between recovery blocks and NVP are next described.

—Recovery blocks (in original form) constitutes a sequential strategy, whereas NVP allows concurrent execution.

—Recovery blocks require the user to provide a fault-free, application-specific, effective acceptance test, while NVP adopts a generic consensus or majority voting algorithm that can be provided by the execution environment (EE).

—Recovery blocks allow different correct outputs from the alternates, while the general-purpose character of the consensus algorithm of NVP calls for a single correct output.[11]

The two models collapse when the acceptance test of recovery blocks is done as in NVP; that is, when the acceptance test is a consensus on the basis of the outputs of the different alternates.

Conclusions. As with recovery blocks, NVP has been successfully adopted for many years in various application fields, including safety-critical airborne and spaceborne applications. The generic NVP architecture, based on redundancy and consensus, addresses parallel and distributed applications written in any programming paradigm. A generic, parameterizable architecture for real-time systems that supports the NVP strategy straightforwardly is GUARDS [Powell et al. 1999].

It is worth remarking that the EE (also known as N-version executive) is a complex component that needs to manage a number of basic functions; for instance, execution of the decision algorithm, the assurance of input consistency for all versions,

[11]This weakness of NVP can be narrowed, if not solved, adopting the approach used in the so-called "voting farm" [De Florio et al. 1998a, 1998b; De Florio 1997b]: a generic voting tool designed by one of the authors of this article within the framework of his participation in project EFTOS (see Section 3.1.1). Specifically, such a tool works with opaque objects that are compared by means of a user-defined function. This function returns an integer value representing a "distance" between any two objects to be voted. The user may choose between a set of predefined distance functions or to develop an application-specific distance function. Doing the latter, a distance may be endowed with the ability to assess that bitwise different objects are semantically equivalent. Of course, the user is still responsible for supplying a bug-free distance function, although assisted in this simpler task by a number of template functions supplied with this tool.

interversion communication, version synchronization, and the enforcement of timing constraints [Avižienis 1995]. On the other hand, this complexity is not part of the application software (the N versions) and does not need to be aware of the fault-tolerance strategy. An excellent degree of transparency can be reached, thus guaranteeing a good value for attribute SC. Furthermore, as mentioned in Section 2.3, the cost and time required by a thorough verification, validation, and testing of this architectural complexity may be acceptable, while charging them to each application component is certainly not a cost-effective option.

Regarding attribute SA, the same considerations provided when describing recovery blocks hold for NVP: Also in this case a single fault-tolerance strategy is followed. For this reason we assess NVP as unsatisfactory regarding attribute SA.

Offline adaptability to "bad" environments may be reached by increasing the value of N, though this requires developing new versions: a costly activity in terms of both time and cost. Furthermore, the architecture does not allow any dynamic management of the fault-tolerance provisions. We conclude that attribute A is poorly addressed by NVP.

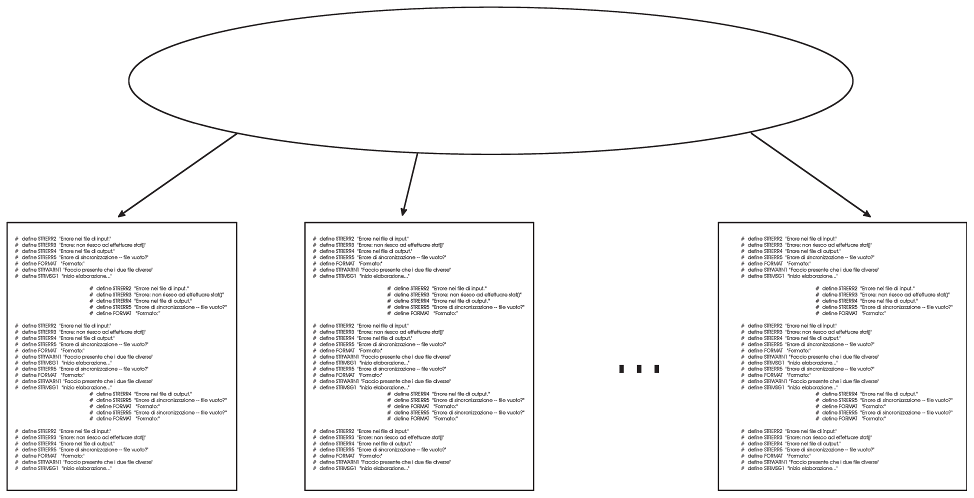
Portability is restricted by the portability of the EE and of each of the N versions. Maintainability actions may also be problematic: They need to be replicated and validated N times, as well as performed according to the NVP paradigm, so as not to impact negatively on the statistical independence of failures. Clearly, the same considerations apply to recovery blocks as well. In other words, the adoption of multiple-version software fault-tolerance provisions always implies a penalty on maintainability and portability.

Limited NVP support has been developed for "conventional" programming languages such as C. For instance, `libft` (see Section 3.1.1) implements NVP as follows.

```
#include <ftmacros.h>
...
NVP
VERSION{
          block 1;
          SENDVOTE(v_pointer, v_size);
}
VERSION{
          block 2;
          SENDVOTE(v_pointer, v_size);
}
...
ENDVERSION(timeout, v_size);
if (!agreeon(v_pointer)) error_handler();
ENDNVP;
```

Note that this particular implementation extinguishes the potential transparency that in general characterizes NVP, as it requires including some nonfunctional code. This translates into an unsatisfactory value for attribute SC. Note also that the execution of each block is, in this case, carried out sequentially.

It is important to note how the adoption of NVP as a system structure for ALFT requires a substantial increase in development and maintenance costs: Both the 1T/NH/NS and NT/1H/NS systems have a cost function growing quadratically with N. The author of the NVP strategy remarks how such costs are repaid by the gain in trustworthiness. This is certainly true when dealing with systems possibly subjected to catastrophic failures let us recall once more the case of the Ariane 5 flight 501 [Inquiry



```
# define STRERR2  "Errore nei file di input."
# define STRERR3  "Errore: non riesco ad effettuare stat()"
# define STRERR4  "Errore nel file di output."
# define STRERR5  "Errore di sincronizzazione -- file vuoto?"
# define FORMAT   "Formato:"
# define STRWARN1 "Faccio presente che i due file diverse"
# define STRMSG1  "Inizio elaborazione..."

                       # define STRERR2  "Errore nei file di input."
                       # define STRERR3  "Errore: non riesco ad effettuare stat()"
                       # define STRERR4  "Errore nel file di output."
                       # define STRERR5  "Errore di sincronizzazione -- file vuoto?"
                       # define FORMAT   "Formato:"

# define STRERR2  "Errore nei file di input."
# define STRERR3  "Errore: non riesco ad effettuare stat()"
# define STRERR4  "Errore nel file di output."
# define STRERR5  "Errore di sincronizzazione -- file vuoto?"
# define FORMAT   "Formato:"
# define STRWARN1 "Faccio presente che i due file diverse"
# define STRMSG1  "Inizio elaborazione..."
# define STRERR5  "Errore di sincronizzazione -- file vuoto?"
# define FORMAT   "Formato:"
# define STRWARN1 "Faccio presente che i due file diverse"
# define STRMSG1  "Inizio elaborazione..."
                       # define STRERR4  "Errore nel file di output."
                       # define STRERR5  "Errore di sincronizzazione -- file vuoto?"
                       # define FORMAT   "Formato:"
                       # define STRERR5  "Errore di sincronizzazione -- file vuoto?"
                       # define FORMAT   "Formato:"

# define STRERR2  "Errore nei file di input."
# define STRERR3  "Errore: non riesco ad effettuare stat()"
# define STRERR4  "Errore nel file di output."
# define STRERR5  "Errore di sincronizzazione -- file vuoto?"
# define FORMAT   "Formato:"
# define STRWARN1 "Faccio presente che i due file diverse"
# define STRMSG1  "Inizio elaborazione..."
```

```
# define STRERR2  "Errore nei file di input."
# define STRERR3  "Errore: non riesco ad effettuare stat()"
# define STRERR4  "Errore nel file di output."
# define STRERR5  "Errore di sincronizzazione -- file vuoto?"
# define FORMAT   "Formato:"
# define STRWARN1 "Faccio presente che i due file diverse"
# define STRMSG1  "Inizio elaborazione..."

                       # define STRERR2  "Errore nei file di input."
                       # define STRERR3  "Errore: non riesco ad effettuare stat()"
                       # define STRERR4  "Errore nel file di output."
                       # define STRERR5  "Errore di sincronizzazione -- file vuoto?"
                       # define FORMAT   "Formato:"

# define STRERR2  "Errore nei file di input."
# define STRERR3  "Errore: non riesco ad effettuare stat()"
# define STRERR4  "Errore nel file di output."
# define STRERR5  "Errore di sincronizzazione -- file vuoto?"
# define FORMAT   "Formato:"
# define STRWARN1 "Faccio presente che i due file diverse"
# define STRMSG1  "Inizio elaborazione..."
# define STRERR5  "Errore di sincronizzazione -- file vuoto?"
# define FORMAT   "Formato:"
# define STRWARN1 "Faccio presente che i due file diverse"
# define STRMSG1  "Inizio elaborazione..."
                       # define STRERR4  "Errore nel file di output."
                       # define STRERR5  "Errore di sincronizzazione -- file vuoto?"
                       # define FORMAT   "Formato:"
                       # define STRERR5  "Errore di sincronizzazione -- file vuoto?"
                       # define FORMAT   "Formato:"

# define STRERR2  "Errore nei file di input."
# define STRERR3  "Errore: non riesco ad effettuare stat()"
# define STRERR4  "Errore nel file di output."
# define STRERR5  "Errore di sincronizzazione -- file vuoto?"
# define FORMAT   "Formato:"
# define STRWARN1 "Faccio presente che i due file diverse"
# define STRMSG1  "Inizio elaborazione..."
```

```
# define STRERR2  "Errore nei file di input."
# define STRERR3  "Errore: non riesco ad effettuare stat()"
# define STRERR4  "Errore nel file di output."
# define STRERR5  "Errore di sincronizzazione -- file vuoto?"
# define FORMAT   "Formato:"
# define STRWARN1 "Faccio presente che i due file diverse"
# define STRMSG1  "Inizio elaborazione..."

                       # define STRERR2  "Errore nei file di input."
                       # define STRERR3  "Errore: non riesco ad effettuare stat()"
                       # define STRERR4  "Errore nel file di output."
                       # define STRERR5  "Errore di sincronizzazione -- file vuoto?"
                       # define FORMAT   "Formato:"

# define STRERR2  "Errore nei file di input."
# define STRERR3  "Errore: non riesco ad effettuare stat()"
# define STRERR4  "Errore nel file di output."
# define STRERR5  "Errore di sincronizzazione -- file vuoto?"
# define FORMAT   "Formato:"
# define STRWARN1 "Faccio presente che i due file diverse"
# define STRMSG1  "Inizio elaborazione..."
# define STRERR5  "Errore di sincronizzazione -- file vuoto?"
# define FORMAT   "Formato:"
# define STRWARN1 "Faccio presente che i due file diverse"
# define STRMSG1  "Inizio elaborazione..."
                       # define STRERR4  "Errore nel file di output."
                       # define STRERR5  "Errore di sincronizzazione -- file vuoto?"
                       # define FORMAT   "Formato:"
                       # define STRERR5  "Errore di sincronizzazione -- file vuoto?"
                       # define FORMAT   "Formato:"

# define STRERR2  "Errore nei file di input."
# define STRERR3  "Errore: non riesco ad effettuare stat()"
# define STRERR4  "Errore nel file di output."
# define STRERR5  "Errore di sincronizzazione -- file vuoto?"
# define FORMAT   "Formato:"
# define STRWARN1 "Faccio presente che i due file diverse"
# define STRMSG1  "Inizio elaborazione..."
```

Fig. 6. A fault-tolerant program according to an MV system.

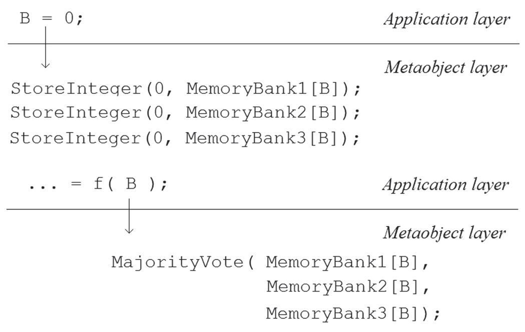
Board Report 1996]. Clearly, risk of rapid exhaustion of redundancy (due to a burst of correlated failures) caused by a single or few design faults is serious. It justifies and calls for the adoption of other fault-tolerance provisions within and around the NVP unit in order to deal with the case of its failure..

Figure 6 shows the main characteristics of the MV approach: Several replicas (portions) of the functional code are produced and managed by a control component. In recovery blocks this component is often coded side-by-side with the functional code, while in NVP this is usually a custom hardware box.

3.1.3. A Hybrid Case: Data Diversity. A special hybrid case is given by data diversity [Ammann and Knight 1988]. A data diversity system is a 1T/NH/1S (less often a NT/1H/1S). It can be concisely described as an NVP system in which N equal replicas are used as versions, but each replica receives a different minor perturbation of the input data. Under the hypothesis that the function computed by the replicas is non-chaotic, that is, does not produce very different output values when fed slightly different inputs, data diversity may be a cost-effective way to fault tolerance. Clearly in this case the vote, mechanism does not run a simple majority vote but some vote fusion algorithm [Lorczak et al. 1989]. A typical application of data diversity is that of a real-time control program, where sensor resampling or a minor perturbation in the sampled sensor value may be able to prevent a failure. Being substantially an NVP system, data diversity reaches the same values for the structural attributes. The greatest advantage of this technique is its drastically decreased design and maintenance costs, since design diversity is avoided.

3.2. Metaobject Protocols and Reflection

Some of the negative aspects pointed out while describing single- and multiple-version software approaches can be, in some cases, weakened (if not solved) by means of a generic structuring technique. This method allows one to reach in some cases an adequate degree of flexibility, transparency, and separation of design concerns: the adoption of metaobject protocols (MOPs) [Kiczales et al. 1991]. The idea is to "open" the implementation of the runtime executive of an object-oriented language (e.g., C++ or Java) so



```
B = 0;

StoreInteger(0, MemoryBank1[B]);
StoreInteger(0, MemoryBank2[B]);
StoreInteger(0, MemoryBank3[B]);

... = f( B );

MajorityVote( MemoryBank1[B],
              MemoryBank2[B],
              MemoryBank3[B]);
```

Fig. 7. A MOP may be used to realize, for example, triple-redundant memories in a fully transparent way.

that the developer can adopt and program different, custom semantics, adjusting the language to the needs of the user and environment. Using MOPs, the programmer can modify the behavior of fundamental features such as methods invocation, object creation and destruction, and member access. The transparent management of spatial and temporal redundancy [Taylor et al. 1980] is a case where MOPs seem particularly adequate: For instance, an MOP programmer may easily create "triple-redundant" memory cells to protect his (her) variables against transient faults, as depicted in Figure 7.

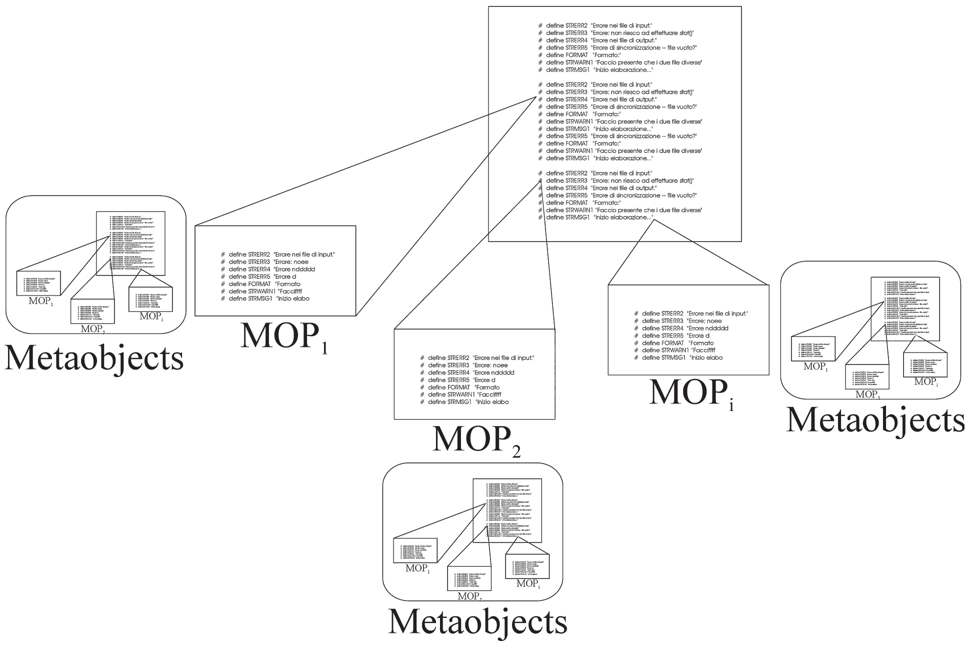
The key concept behind MOPs is that of computational reflection, or the causal connection between a system and a metalevel description representing the structural and computational aspects of that system [Maes 1987]. MOPs offer the metalevel programmer a representation of a system as a set of metaobjects. Metaobjects are those objects that represent and reflect properties of "real" objects, namely, those objects that constitute the functional part of the user application. Metaobjects can, for instance, represent the structure of a class, object interaction, or code of an operation. This mapping process is called reification [Robben 1999].

The causality relation of MOPs could also be extended to allow for a dynamic reorganization of the structure and operation of a system; for instance, to perform reconfiguration and error recovery. The basic object-oriented feature of inheritance can be used to enhance the reusability of the FT mechanisms developed with this approach.

3.2.1. Project FRIENDS. An architecture supporting this approach is the one developed in the framework of project FRIENDS [Fabre and Pérennou 1998, 1996]. The name FRIENDS stands for "flexible and reusable implementation environment for your next dependable system." This project aims at implementing a number of fault-tolerance provisions (e.g., replication, group-based communication, synchronization, voting, etc. [van Achteren 1997]) at metalevel. In FRIENDS a distributed application is a set of objects interacting via the proxy model, a proxy being a local intermediary between each object and any other (possibly replicated) object. FRIENDS uses the metaobject protocol provided by Open C++, a C++ preprocessor that provides control over instance creation and deletion, state access, and invocation of methods.

Other ALFT architectures exploiting the concept of metaobject protocols within custom programming languages are reported in Section 3.3.

Conclusions. MOPs are indeed a promising system structure for embedding different nonfunctional concerns in the application level of a computer program. MOPs work at *language* level, providing a means to modify the semantics of basic object-oriented language building blocks, such as object creation and deletion, calling and termination of class methods, and so forth. This appears to match perfectly with a proper subset of



```
#  define STRERR2  "Errore nei file di input."
#  define STRERR3  "Errore: non riesco ad effettuare stat()"
#  define STRERR4  "Errore nel file di output."
#  define STRERR5  "Errore di sincronizzazione -- file vuoto?"
#  define FORMAT   "Formato:"
#  define STRWARN1 "Faccio presente che i due file diverse"
#  define STRMSG1  "Inizio elaborazione..."

#  define STRERR2  "Errore nei file di input."
#  define STRERR3  "Errore: non riesco ad effettuare stat()"
#  define STRERR4  "Errore nel file di output."
#  define STRERR5  "Errore di sincronizzazione -- file vuoto?"
#  define FORMAT   "Formato:"
#  define STRWARN1 "Faccio presente che i due file diverse"
#  define STRMSG1  "Inizio elaborazione..."
#  define STRERR5  "Errore di sincronizzazione -- file vuoto?"
#  define FORMAT   "Formato:"
#  define STRWARN1 "Faccio presente che i due file diverse"
#  define STRMSG1  "Inizio elaborazione..."

#  define STRERR2  "Errore nei file di input."
#  define STRERR3  "Errore: non riesco ad effettuare stat()"
#  define STRERR4  "Errore nel file di output."
#  define STRERR5  "Errore di sincronizzazione -- file vuoto?"
#  define FORMAT   "Formato:"
#  define STRWARN1 "Faccio presente che i due file diverse"
#  define STRMSG1  "Inizio elaborazione..."
```

```
#  define STRERR2  "Errore nei file di input."
#  define STRERR3  "Errore: noee
#  define STRERR4  "Errore nddddd
#  define STRERR5  "Errore d
#  define FORMAT   "Formato
#  define STRWARN1 "Facciffff
#  define STRMSG1  "Inizio elabo
```

```
#  define STRERR2  "Errore nei file di input."
#  define STRERR3  "Errore: noee
#  define STRERR4  "Errore nddddd
#  define STRERR5  "Errore d
#  define FORMAT   "Formato
#  define STRWARN1 "Facciffff
#  define STRMSG1  "Inizio elabo
```

```
#  define STRERR2  "Errore nei file di input."
#  define STRERR3  "Errore: noee
#  define STRERR4  "Errore nddddd
#  define STRERR5  "Errore d
#  define FORMAT   "Formato
#  define STRWARN1 "Facciffff
#  define STRMSG1  "Inizio elabo
```

Fig. 8. A fault-tolerant program according to a MOP system.

the possible fault-tolerance provisions, especially those such as transparent object redundancy that can be straightforwardly managed with the metaobject approach. When dealing with these fault-tolerance provisions, MOPs provide a perfect separation of the design concerns, namely, optimal SC. Some other techniques, specifically those which might be described as the most coarse-grained (e.g., distributed recovery blocks [Kim and Welch 1989]), appear to be less suited to efficient implementation via MOPs. These techniques work at a distributed, macroscopic level.

Figure 8 synthesizes the main characteristics of the MOP approach: The fault-tolerance programmer defines a number of metaobject protocols and associates them with method invocations or other grammar cases. Each time the functional program enters a certain grammar case, the corresponding protocol is transparently executed. Each protocol has access to a representation of the system through its metaobjects, by means of which it can also perform actions on the corresponding "real" objects.

MOPs appear to constitute a promising technique for a transparent, coherent, and effective adoption of some existing FT mechanisms and techniques. A number of studies confirm that MOPs reach efficiency in some cases [Kiczales et al. 1991; Masuhara et al. 1992], though no experimental or analytical evidence thus far allows to estimate the practicality and generality of this approach: "[w]hat reflective capabilities are needed for what form of fault-tolerance, and to what extent these capabilities can be provided in more-or-less conventional programming languages, and allied to other structuring techniques [e.g., recovery blocks or NVP] remain to be determined" [Randell and Xu 1995]. In other words, it is still an open question as to whether MOPs represent a practical solution towards effective integration of most of the existing fault-tolerance mechanisms in user applications.

The aforesaid situation reminds the authors of another, regarding the "quest" for a novel computational paradigm for parallel processing which would be capable of dealing effectively with the widest class of problems, as the von Neumann paradigm does for sequential processing, though with the highest degree of efficiency and least amount of change in the original (sequential) user code. In that context, the concept of computational *grain* came up; some techniques were inherently looking at the problem "with coarse-grained glasses," that is, at macroscopic level, while others were considering the problem exclusively at microscopical level. One can conclude that MOPs offer an elegant system structure to embed a set of nonfunctional services (including fault-tolerance provisions) in an object-oriented program. It is still unclear whether this set is general enough to host, efficaciously, many forms of fault tolerance, as remarked, for instance, in Randell and Xu [1995] and Lippert and Videira Lopes [2000]. It is therefore difficult to establish a qualitative assessment of attribute SA for MOPs.

The runtime management MOP libraries may be used to reach satisfactory values for attribute A. To the best of the authors' knowledge, this feature is not present in any language supporting MOPs.

As evident, the target application domain is the one of object-oriented applications written with languages extended with a MOP, such as Open C++.

As a final remark, we observe how the cost of MOP-compliant fault-tolerant software design should include those related to acquisition of the extra competence and experience in MOP design tools, reification, and custom programming languages.

3.3. Enhancing or Developing Fault-Tolerance Languages

Another approach is given by working at the language level, enhancing a preexisting programming language or developing an ad hoc distributed programming language so that it hosts specific fault-tolerance provisions. The following two sections cover these topics.

3.3.1. Enhancing Preexisting Programming Languages. Enhancing a preexisting programming language means augmenting the grammar of a widespread language such as C or Pascal so that it directly supports features that can be used to enhance the dependability of its programs (e.g., recovery blocks [Shrivastava 1978]).

In the following, four classes of system based on this approach are presented: Arjuna, Sina, Linda, and FT-SR. All of them constitute provisions to develop distributed fault-tolerant systems.

—*The Arjuna Distributed Programming System.* Arjuna is an object-oriented system for portable distributed programming in C++ [Parrington 1990; Shrivastava 1995]. It can be considered as a clever blending of useful and widespread tools, techniques, and ideas; as such, it is a good example of the evolutionary approach towards application-level software fault-tolerance. It exploits remote procedure calls [Birrell and Nelson 1984] and UNIX daemons. On each node of the system, an object server connects client objects to objects supplying services. The object server also takes care of spawning objects when they are not yet running (in this case they are referred to as "passive objects"). Arjuna also exploits a "naming service," by means of which client objects request a service "by name." This transparency effectively supports object migration and replication.

As in other systems, Arjuna makes use of stub generation to specify remote procedure calls and remote manipulation of objects. A nice feature of this system is that the stubs are derived automatically from the C++ header files of the application, which avoids the need of a custom interface description language.

Arjuna offers the programmer means for dealing with atomic actions (via the two-phase commit protocol) and persistent objects. The core class hierarchy of Arjuna appears to the programmer as follows [Parrington 1990]: StateManager LockManager User-Defined Classes Lock User-Defined Lock Classes AtomicAction AbstractRecord RecoveryRecord LockRecord and other management record types; etc.

Johnson, B.W. “Fault Tolerance”
The Electrical Engineering Handbook
Ed. Richard C. Dorf
Boca Raton: CRC Press LLC, 2000

93
Fault Tolerance

93.1 Introduction
93.2 Hardware Redundancy
93.3 Information Redundancy
93.4 Time Redundancy
93.5 Software Redundancy
93.6 Dependability Evaluation

Barry W. Johnson
University of Virginia

93.1 Introduction

Fault tolerance is the ability of a system to continue correct performance of its tasks after the occurrence of hardware or software faults. A **fault** is simply any physical defect, imperfection, or flaw that occurs in hardware or software. Applications of fault-tolerant computing can be categorized broadly into four primary areas: long-life, critical computations, maintenance postponement, and high availability. The most common examples of long-life applications are unmanned space flight and satellites. Examples of critical-computation applications include aircraft flight control systems, military systems, and certain types of industrial controllers. Maintenance postponement applications appear most frequently when maintenance operations are extremely costly, inconvenient, or difficult to perform. Remote processing stations and certain space applications are good examples. Banking and other time-shared systems are good examples of high-availability applications. Fault tolerance can be achieved in systems by incorporating various forms of redundancy, including hardware, information, time, and software redundancy [Johnson, 1989].

93.2 Hardware Redundancy

The physical replication of hardware is perhaps the most common form of fault tolerance used in systems. As semiconductor components have become smaller and less expensive, the concept of hardware redundancy has become more common and more practical. There are three basic forms of hardware redundancy. First, *passive* techniques use the concept of fault masking to hide the occurrence of faults and prevent the faults from resulting in **errors**. Passive approaches are designed to achieve fault tolerance without requiring any action on the part of the system or an operator. Passive techniques, in their most basic form, do not provide for the detection of faults but simply mask the faults. An example of a passive approach is triple modular redundancy (TMR), which is illustrated in Fig. 93.1. In the TMR system three identical units perform identical functions, and a majority vote is performed on the output.

The second form of hardware redundancy is the *active* approach, which is sometimes called the *dynamic* method. Active methods achieve fault tolerance by detecting the existence of faults and performing some action to remove the faulty hardware from the system. In other words, active techniques require that the system perform reconfiguration to tolerate faults. Active hardware redundancy uses fault detection, fault location, and fault recovery in an attempt to achieve fault tolerance. An example of an active approach to hardware redundancy is standby sparing, which is illustrated in Fig. 93.2. In standby sparing one or more units operate as spares and replace the primary unit when it fails.

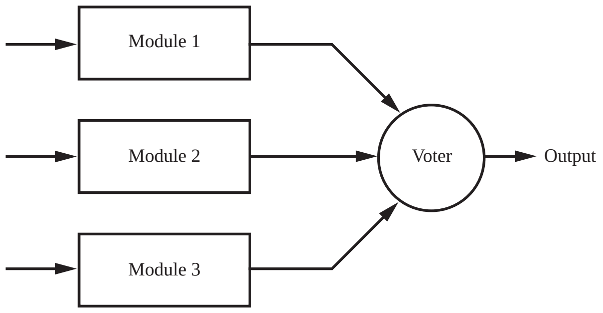


FIGURE 93.1 Fault masking using triple modular redundancy (TMR).

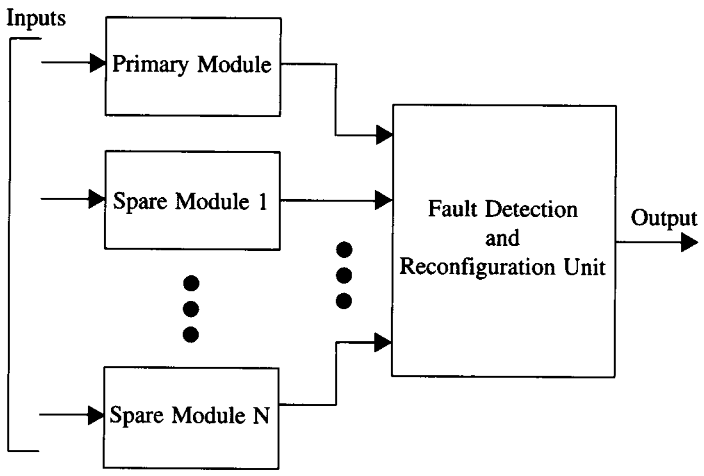


FIGURE 93.2 General concept of standby sparing.

The final form of hardware redundancy is the *hybrid* approach. Hybrid techniques combine the attractive features of both the passive and active approaches. Fault masking is used in hybrid systems to prevent erroneous results from being generated. Fault detection, fault location, and fault recovery are also used in the hybrid approaches to improve fault tolerance by removing faulty hardware and replacing it with spares. Providing spares is one form of providing redundancy in a system. Hybrid methods are most often used in the critical-computation applications where fault masking is required to prevent momentary errors, and high reliability must be achieved. The basic concept of the hybrid approach is illustrated in Fig. 93.3.

93.3 Information Redundancy

Another approach to fault tolerance is to employ redundancy of information. Information redundancy is simply the addition of redundant information to data to allow fault detection, fault masking, or possibly fault tolerance. Good examples of information redundancy are error detecting and error correcting codes, formed by the addition of redundant information to data words or by the mapping of data words into new representations containing redundant information [Lin and Costello, 1983].

In general, a *code* is a means of representing information, or data, using a well-defined set of rules. A *code word* is a collection of symbols, often called digits if the symbols are numbers, used to represent a particular piece of data based upon a specified code. A *binary code* is one in which the symbols forming each code word consist of only the digits 0 and 1. A code word is said to be *valid* if the code word adheres to all of the rules that define the code; otherwise, the code word is said to be *invalid.*

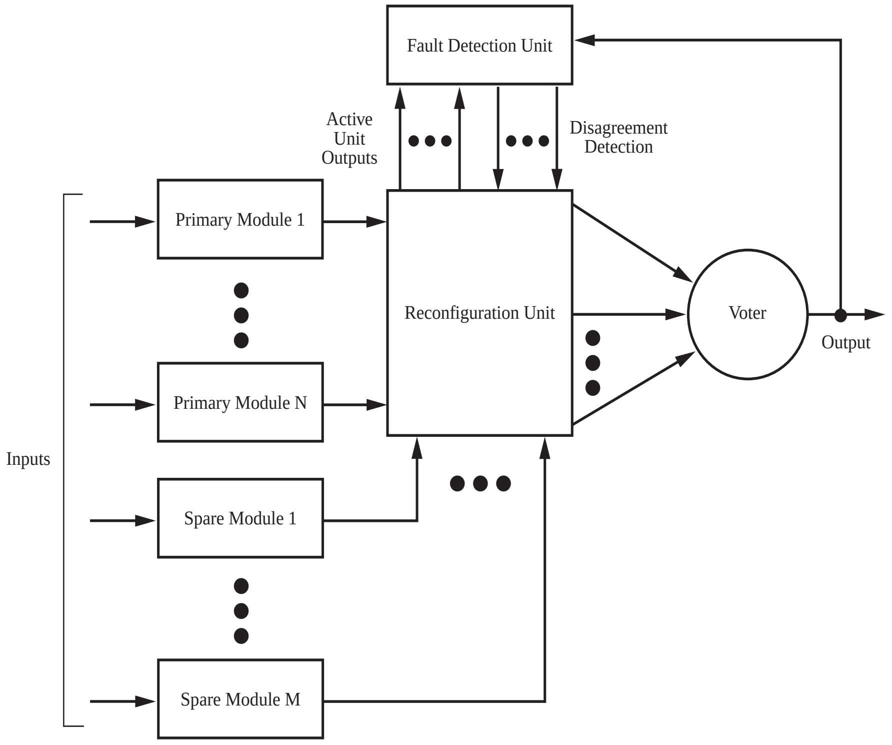


FIGURE 93.3 Hybrid redundancy approach.

The *encoding operation* is the process of determining the corresponding code word for a particular data item. In other words, the encoding process takes an original data item and represents it as a code word using the rules of the code. The *decoding operation* is the process of recovering the original data from the code word. In other words, the decoding process takes a code word and determines the data that it represents.

It is possible to create a binary code for which the valid code words are a subset of the total number of possible combinations of 1s and 0s. If the code words are formed correctly, errors introduced into a code word will force it to lie in the range of illegal, or invalid, code words, and the error can be detected. This is the basic concept of the *error detecting codes.* The basic concept of the *error correcting code* is that the code word is structured such that it is possible to determine the correct code word from the corrupted, or erroneous, code word.

A fundamental concept in the characterization of codes is the *Hamming distance* [Hamming, 1950]. The *Hamming distance* between any two binary words is the number of bit positions in which the two words differ. For example, the binary words 0000 and 0001 differ in only one position and therefore have a Hamming distance of 1. The binary words 0000 and 0101, however, differ in two positions; consequently, their Hamming distance is 2. Clearly, if two words have a Hamming distance of 1, it is possible to change one word into the other simply by modifying one bit in one of the words. If, however, two words differ in two bit positions, it is impossible to transform one word into the other by changing one bit in one of the words.

The Hamming distance gives insight into the requirements of error detecting codes and error correcting codes. We define the *distance* of a code as the minimum Hamming distance between any two valid code words. If a binary code has a distance of two, then any single-bit error introduced into a code word will result in the erroneous word being an invalid code word because all valid code words differ in at least two bit positions. If a code has a distance of 3, then any single-bit error or any double-bit error will result in the erroneous word being an invalid code word because all valid code words differ in at least three positions. However, a code distance of 3 allows any single-bit error to be corrected, if it is desired to do so, because the erroneous word with a single-bit error will be a Hamming distance of 1 from the correct code word and at least a Hamming

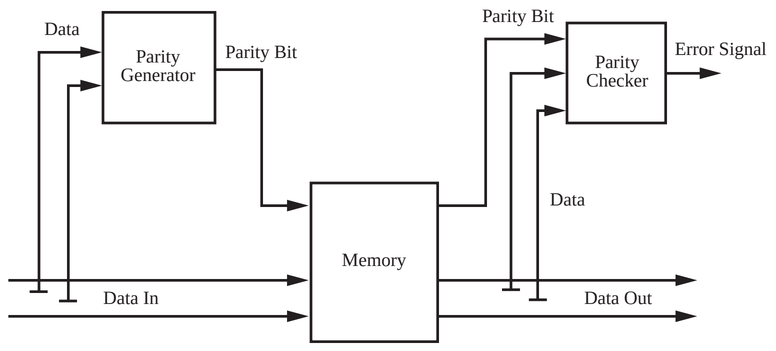


FIGURE 93.4 Use of parity coding in a memory application.

distance of 2 from all others. Consequently, the correct code word can be identified from the corrupted code word.

In general, a binary code can correct up to c bit errors and detect an additional d bit errors if and only if

$$2c - \mathrm{d} - 1 \leq \mathrm{H}_{\mathrm{d}}$$

where H_d is the distance of the code [Nelson and Carroll, 1986]. For example, a code with a distance of 2 cannot provide any error correction but can detect single-bit errors. Similarly, a code with a distance of 3 can correct single-bit errors or detect a double-bit error.

A second fundamental concept of codes is *separability.* A *separable code* is one in which the original information is appended with new information to form the code word, thus allowing the decoding process to consist of simply removing the additional information and keeping the original data. In other words, the original data is obtained from the code word by stripping away extra bits, called the code bits or check bits, and retaining only those associated with the original information. A *nonseparable code* does not possess the property of separability and, consequently, requires more complicated decoding procedures.

Perhaps the simplest form of a code is the parity code. The basic concept of parity is very straightforward, but there are variations on the fundamental idea. Single-bit parity codes require the addition of an extra bit to a binary word such that the resulting code word has either an even number of 1s or an odd number of 1s. If the extra bit results in the total number of 1s in the code word being odd, the code is referred to as *odd parity.* If the resulting number of 1s in the code word is even, the code is called *even parity.* If a code word with odd parity experiences a change in one of its bits, the parity will become even. Likewise, if a code word with even parity encounters a single-bit change, the parity will become odd. Consequently, a single-bit error can be detected by checking the number of ls in the code words. The single-bit parity code (either odd or even) has a distance of 2, therefore allowing any single-bit error to be detected but not corrected. Figure 93.4 illustrates the use of parity coding in a simple memory application.

Arithmetic codes are very useful when it is desired to check arithmetic operations such as addition, multiplication, and division [Avizienis, 1971]. The basic concept is the same as all coding techniques. The data presented to the arithmetic operation is encoded before the operations are performed. After completing the arithmetic operations, the resulting code words are checked to make sure that they are valid code words. If the resulting code words are not valid, an error condition is signaled. An arithmetic code must be invariant to a set of arithmetic operations. An arithmetic code, A, has the property that $A(b{*}c) = A(b){*}A(c)$, where b and c are operands, * is some arithmetic operation, and $A(b)$ and $A(c)$ are the arithmetic code words for the operands b and c, respectively. Stated verbally, the performance of the arithmetic operation on two arithmetic code words will produce the arithmetic code word of the result of the arithmetic operation. To completely define an arithmetic code, the method of encoding and the arithmetic operations for which the code is invariant must be specified. The most common examples of arithmetic codes are the *AN* codes, residue codes, and the inverse residue codes.

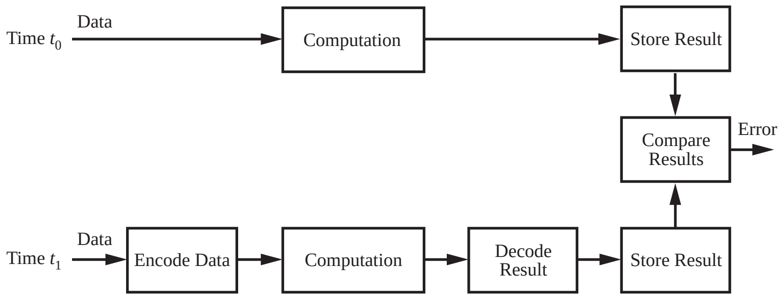


FIGURE 93.5 Time redundancy concept.

93.4 Time Redundancy

Time redundancy methods attempt to reduce the amount of extra hardware at the expense of using additional time. In many applications, the time is of much less importance than the hardware because hardware is a physical entity that impacts weight, size, power consumption, and cost. Time, on the other hand, may be readily available in some applications. The basic concept of time redundancy is the repetition of computations in ways that allow faults to be detected. Time redundancy can function in a system in several ways. The fundamental concept is to perform the same computation two or more times and compare the results to determine if a discrepancy exists. If an error is detected, the computations can be performed again to see if the disagreement remains or disappears. Such approaches are often good for detecting errors resulting from transient faults but cannot provide protection against errors resulting from permanent faults.

The main problem with many time redundancy techniques is assuring that the system has the same data to manipulate each time it redundantly performs a computation. If a transient fault has occurred, a system's data may be completely corrupted, making it difficult to repeat a given computation. Time redundancy has been used primarily to detect transients in systems. One of the biggest potentials of time redundancy, however, now appears to be the ability to detect permanent faults while using a minimum of extra hardware. The fundamental concept is illustrated in Fig. 93.5. During the first computation or transmission, the operands are used as presented and the results are stored in a register. Prior to the second computation or transmission, the operands are encoded in some fashion using an encoding function. After the operations have been performed on the encoded data, the results are then decoded and compared to those obtained during the first operation. The selection of the encoding function is made so as to allow faults in the hardware to be detected. Example encoding functions might include the complementation operator and an arithmetic shift.

93.5 Software Redundancy

Software faults are unusual entities. Software does not break as hardware does, but instead software faults are the result of incorrect software designs or coding mistakes. Therefore, any technique that detects faults in software must detect design flaws. A simple duplication and comparison procedure will not detect software faults if the duplicated software modules are identical, because the design mistakes will appear in both modules.

The concept of N self-checking programming is to first write N unique versions of the program and to develop a set of acceptance tests for each version. The acceptance tests are essentially checks performed on the results produced by the program and may be created using consistency checks and capability checks, for example. Selection logic, which may be a program itself, chooses the results from one of the programs that passes the acceptance tests. This approach is analogous to the hardware technique known as hot standby sparing. Since each program is running simultaneously, the reconfiguration process can be very fast. Provided that the software faults in each version of the program are independent and the faults are detected as they occur by the acceptance tests, then this approach can tolerate $N - 1$ faults. It is important to note that the assumptions of fault independence and perfect fault coverage are very big assumptions to make in almost all applications.

The concept of *N*-version programming was developed to allow certain design flaws in software modules to be tolerated [Chen and Avizienis, 1978]. The basic concept of *N*-version programming is to design and code the software module *N* times and to vote on the *N* results produced by these modules. Each of the *N* modules is designed and coded by a separate group of programmers. Each group designs the software from the same set of specifications such that each of the *N* modules performs the same function. However, it is hoped that by performing the *N* designs independently, the same mistakes will not be made by the different groups. Therefore, when a fault occurs, the fault will either not occur in all modules or it will occur differently in each module, so that the results generated by the modules will differ. Assuming that the faults are independent the approach can tolerate $(N - 1)/2$ faults where N is odd.

The recovery block approach to software fault tolerance is analogous to the active approaches to hardware fault tolerance, specifically the cold standby sparing approach. *N* versions of a program are provided, and a single set of acceptance tests is used. One version of the program is designated as the primary version, and the remaining $N - 1$ versions are designated as spares, or secondary versions. The primary version of the software is always used unless it fails to pass the acceptance tests. If the acceptance tests are failed by the primary version, then the first secondary version is tried. This process continues until one version passes the acceptance tests or the system fails because none of the versions can pass the tests.

93.6 Dependability Evaluation

Dependability is defined as the quality of service provided by a system [Laprie, 1985]. Perhaps the most important measures of dependability are reliability and availability. Fundamental to reliability calculations is the concept of failure rate. Intuitively, the *failure rate is* the expected number of **failures** of a type of device or system per a given time period [Shooman, 1968]. The failure rate is typically denoted as λ when it is assumed to have a constant value. To more clearly understand the mathematical basis for the concept of a failure rate, first consider the definition of the reliability function. The **reliability** $R(t)$ of a component, or a system, is the conditional probability that the component operates correctly throughout the interval $[t_0, t]$ given that it was operating correctly at the time t_0.

There are a number of different ways in which the failure rate function can be expressed. For example, the failure rate function $z(t)$ can be written strictly in terms of the reliability function $R(t)$ as

$$z(t) = \left(-\frac{dR(t)/dt}{R(t)} \right)$$

Similarly, $z(t)$ can be written in terms of the unreliability $Q(t)$ as

$$z(t) = -\frac{dR(t)/dt}{R(t)} = \frac{dQ(t)/dt}{1 - Q(t)}$$

where $Q(t) = 1 - R(t)$. The derivative of the unreliability, $dQ(t)/dt$, is called the *failure density function.*

The failure rate function is clearly dependent upon time; however, experience has shown that the failure rate function for electronic components does have a period where the value of $z(t)$ is approximately constant. The commonly accepted relationship between the failure rate function and time for electronic components is called the bathtub curve and is illustrated in Fig. 93.6. The bathtub curve assumes that during the early life of systems, failures occur frequently due to substandard or weak components. The decreasing part of the bathtub curve is called the early-life or infant mortality region. At the opposite end of the curve is the wear-out region where systems have been functional for a long period of time and are beginning to experience failures due to the physical wearing of electronic or mechanical components. During the intermediate region, the failure rate function is assumed to be a constant. The constant portion of the bathtub curve is called the useful-life phase

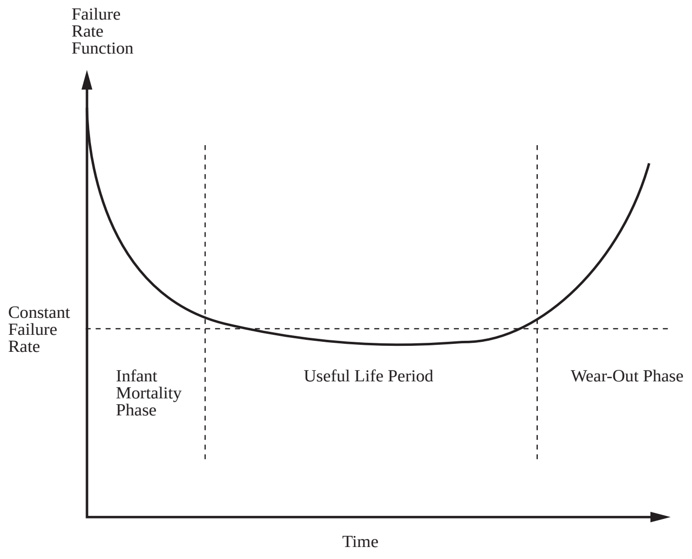


FIGURE 93.6 Bathtub curve relationship between the failure rate function and time.

of the system, and the failure rate function is assumed to have a value of λ during that period. λ is referred to as the failure rate and is normally expressed in units of failures per hour.

The reliability can be expressed in terms of the failure rate function as a differential equation of the form

$$\frac{dR(t)}{dt} = -z(t)R(t)$$

The general solution of this differential equation is given by

$$R(t) = e^{-'z(t)dt}$$

If we assume that the system is in the useful-life stage where the failure rate function has a constant value of λ, the solution to the differential equation is an exponential function of the parameter λ given by

$$R(t) = e^{-\lambda t}$$

where λ is the constant failure rate. The exponential relationship between the reliability and time is known as the *exponential failure law,* which states that for a constant failure rate function, the reliability varies exponentially as a function of time.

In addition to the failure rate, the mean time to failure (MTTF) is a useful parameter to specify the quality of a system. The MTTF is the expected time that a system will operate before the first failure occurs. The MTTF can be calculated by finding the expected value of the time of failure.

From probability theory, we know that the expected value of a random variable, *X*, is

$$E[X] = \int_{-\infty}^{\infty} x f(x) dx$$

where $f(x)$ is the probability density function. In reliability analysis we are interested in the expected value of the time of failure (MTTF), so

$$\text{MTTF} = \int_0^\infty t f(t) dt$$

where $f(t)$ is the failure density function, and the integral runs from 0 to ∞ because the failure density function is undefined for times less than 0. We know, however, that the failure density function is

$$f(t) = \frac{dQ(t)}{dt}$$

so, the MTTF can be written as

$$\text{MTTF} = \int_0^\infty t \frac{dQ(t)}{dt} dt$$

Using integration by parts and the fact that $dQ(t)/dt = -dR(t)/dt$ we can show that

$$\text{MTTF} = \int_0^\infty t \frac{dQ(t)}{dt} dt = -\int_0^\infty t \frac{dR(t)}{dt} dt = \left[-tR(t) + \int R(t) dt\right]\Big|_0^\infty = \int_0^\infty R(t) dt$$

Consequently, the MTTF is defined in terms of the reliability function as

$$\text{MTTF} = \int_0^\infty R(t) dt$$

which is valid for any reliability function that satisfies $R(\infty) = 0$.

The mean time to repair (MTTR) is simply the average time required to repair a system. The MTTR is extremely difficult to estimate and is often determined experimentally by injecting a set of faults, one at a time, into a system and measuring the time required to repair the system in each case. The MTTR is normally specified in terms of a repair rate, μ, which is the average number of repairs that occur per time period. The units of the repair rate are normally number of repairs per hour. The MTTR and the rate, μ, are related by

$$\text{MTTR} = \frac{1}{\mu}$$

It is very important to understand the difference between the MTTF and the mean time between failure (MTBF). Unfortunately, these two terms are often used interchangeably. While the numerical difference is small in many cases, the conceptual difference is very important. The MTTF is the average time until the first failure of a system, while the MTBF is the average time between failures of a system. If we assume that all repairs to a system make the system perfect once again just as it was when it was new, the relationship between the MTTF and the MTBF can be determined easily. Once successfully placed into operation, a system will operate, on the average, a time corresponding to the MTTF before encountering the first failure. The system will then require some time, MTTR, to repair the system and place it back into operation once again. The system will then be perfect once again and will operate for a time corresponding to the MTTF before encountering its next failure. The time between the two failures is the sum of the MTTF and the MTTR and is the MTBF. Thus, the difference between the MTTF and the MTBF is the MTTR. Specifically, the MTBF is given by

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

In most practical applications the MTTR is a small fraction of the MTTF, so the approximation that the MTBF and MTTF are equal is often quite good. Conceptually, however, it is crucial to understand the difference between the MTBF and the MTTF.

An extremely important parameter in the design and analysis of fault-tolerant systems is fault coverage. The fault coverage available in a system can have a tremendous impact on the reliability, safety, and other attributes of the system. Fault coverage is mathematically defined as the conditional probability that, given the existence of a fault, the system recovers [Bouricius et al., 1969]. The fundamental problem with fault coverage is that it is extremely difficult to calculate. Probably the most common approach to estimating fault coverage is to develop a list all of the faults that can occur in a system and to form, from that list, a list of faults from which the system can recover. The fault coverage factor is then calculated appropriately.

Reliability is perhaps one of the most important attributes of systems. The reliability of a system is generally derived in terms of the reliabilities of the individual components of the system. The two models of systems that are most common in practice are the series and the parallel. In a series system, each element of the system is required to operate correctly for the system to operate correctly. In a parallel system, on the other hand, only one of several elements must be operational for the system to perform its functions correctly.

The series system is best thought of as a system that contains no redundancy; that is, each element of the system is needed to make the system function correctly. In general, a system may contain *N* elements, and in a series system each of the *N* elements is required for the system to function correctly. The reliability of the series system can be calculated as the probability that none of the elements will fail. Another way to look at this is that the reliability of the series system is the probability that all of the elements are working properly. The reliability of a series system is given by

$$R_{\text{series}}(t) = R_1(t)R_2(t)\ldots R_N(t)$$

or

$$R_{\text{series}}(t) = \prod_{i=1}^{N} R_i(t)$$

An interesting relationship exists in a series system if each individual component satisfies the exponential failure law. Suppose that we have a series system made up of *N* components, and each component, *i*, has a constant failure rate of λ_i. Also assume that each component satisfies the exponential failure law. The reliability of the series system is given by

$$R_{\text{series}}(t) = e^{-\lambda_1 t}e^{-\lambda_2 t}\ldots e^{-\lambda_N t}$$

$$R_{\text{series}}(t) = e^{-\sum_{i=1}^{N}\lambda_i t}$$

The distinguishing feature of the basic parallel system is that only one of *N* identical elements is required for the system to function. The reliability of the parallel system can be written as

$$R_{\text{parallel}}(t) = 1.0 - Q_{\text{parallel}}(t) = 1.0 - \prod_{i=1}^{N} Q_i(t) = 1.0 - \prod_{i=1}^{N}(1.0 - R_i(t))$$

It should be noted that the equations for the parallel system assume that the failures of the individual elements that make up the parallel system are independent.

M-of-*N* systems are a generalization of the ideal parallel system. In the ideal parallel system, only one of *N* modules is required to work for the system to work. In the *M*-of-*N* system, however, *M* of the total of *N* identical modules are required to function for the system to function. A good example is the TMR configuration where two of the three modules must work for the majority voting mechanism to function properly. Therefore, the TMR system is a 2-of-3 system.

In general, if there are *N* identical modules and *M* of those are required for the system to function properly, then the system can tolerate $N - M$ module failures. The expression for the reliability of an *M*-of-*N* system can be written as

$$R_{M\text{-of-}N}(t) = \sum_{i=0}^{N-M} \binom{N}{i} R^{N-i}(t)(1.0 - R(t))^{i}$$

where

$$\binom{N}{i} = \frac{N!}{(N-i)!\,i!}$$

The **availability**, $A(t)$, of a system is defined as the probability that a system will be available to perform its tasks at the instant of time t. Intuitively, we can see that the availability can be approximated as the total time that a system has been operational divided by the total time elapsed since the system was initially placed into operation. In other words, the availability is the percentage of time that the system is available to perform its expected tasks. Suppose that we place a system into operation at time $t = 0$. As time moves along, the system will perform its functions, perhaps fail, and hopefully be repaired. At some time $t = t_{\text{current}}$, suppose that the system has operated correctly for a total of t_{op} hours and has been in the process of repair or waiting for repair to begin for a total of t_{repair} hours. The time t_{current} is then the sum of t_{op} and t_{repair}. The availability can be determined as

$$A(t_{\text{current}}) = \frac{t_{\text{op}}}{t_{\text{op}} + t_{\text{repair}}}$$

where $A(t_{\text{current}})$ is the availability at time t_{current}.

If the average system experiences *N* failures during its lifetime, the total time that the system will be operational is *N*(MTTF) hours. Likewise, the total time that the system is down for repairs is *N*(MTTR) hours. In other words, the operational time, t_{op}, is *N*(MTTF) hours and the downtime, t_{repair}, is *N*(MTTR) hours. The average, or steady-state, availability is

$$A_{SS} = \frac{N(\text{MTTF})}{N(\text{MTTF}) + N(\text{MTTR})}$$

We know, however, that the MTTF and the MTTR are related to the failure rate and the repair rate, respectively, for simplex systems, as

$$\text{MTTF} = \frac{1}{\lambda}$$

$$\text{MTTR} = \frac{1}{\mu}$$

Therefore, the steady-state availability is given by

$$A_{SS} = \frac{1/\lambda}{1/\lambda + 1/\mu} = \frac{1}{1 + \lambda/\mu}$$

Defining Terms

Availability, $A(t)$: The probability that a system is operating correctly and is available to perform its functions at the instant of time t.

Dependability: The quality of service provided by a particular system.

Error: The occurrence of an incorrect value in some unit of information within a system.

Failure: A deviation in the expected performance of a system.

Fault: A physical defect, imperfection, or flaw that occurs in hardware or software.

Fault avoidance: A technique that attempts to prevent the occurrence of faults.

Fault tolerance: The ability to continue the correct performance of functions in the presence of faults.

Maintainability, $M(t)$: The probability that an inoperable system will be restored to an operational state within the time t.

Performability, $P(L,t)$: The probability that a system is performing at or above some level of performance, L, at the instant of time t.

Reliability, $R(t)$: The conditional probability that a system has functioned correctly throughout an interval of time, $[t_0,t]$, given that the system was performing correctly at time t_0.

Safety, $S(t)$: The probability that a system will either perform its functions correctly or will discontinue its functions in a well-defined, safe manner.

Related Topics

98.1 Introduction • 98.4 Relationship between Reliability and Failure Rate

References

A. Avizienis, "Arithmetic error codes: Cost and effectiveness studies for application in digital system design," *IEEE Transactions on Computers,* vol. C-20, no. 11, pp. 1322–1331, November 1971.

W. G. Bouricius, W. C. Carter, and P. R. Schneider, "Reliability modeling techniques for self-repairing computer systems," in *Proceedings of the 24th ACM Annual Conference,* pp. 295–309, 1969.

L. Chen and A. Avizienis, "N-version programming: A fault tolerant approach to reliability of software operation," in *Proceedings of the International Symposium on Fault Tolerant Computing,* pp. 3–9, 1978.

R. W. Hamming, "Error detecting and error correcting codes," *Bell System Technical Journal,* vol. 26, no. 2, pp. 147–160, April 1950.

B. W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems,* Reading, Mass.: Addison-Wesley, 1989.

J-C. Laprie, "Dependable computing and fault tolerance: Concepts and terminology," in *Proceedings of the 15th Annual International Symposium on Fault-Tolerant Computing,* Ann Arbor, Mich.: pp. 2–11, June 19–21, 1985.

S. Lin and D. J. Costello, Jr., *Error Control Coding: Fundamentals and Applications,* Englewood Cliffs, N.J.: Prentice-Hall, 1983.

V. P. Nelson and B. D. Carroll, *Tutorial: Fault-Tolerant Computing,* Washington, D.C.: IEEE Computer Society Press, 1986.

M. L. Shooman, *Probabilistic Reliability: An Engineering Approach,* New York: McGraw-Hill, 1968.

Further Information

The *IEEE Transactions on Computers, IEEE Computer* magazine, and the *Proceedings of the IEEE* have published numerous special issues dealing exclusively with fault tolerance technology. Also, the IEEE International Symposium on Fault-Tolerant Computing has been held each year since 1971. Finally, the following textbooks are available, in addition to those referenced above:

P. K. Lala, *Fault Tolerant and Fault Testable Hardware,* Englewood Cliffs, N.J.: Prentice-Hall, 1985.
D. K. Pradhan, *Fault-Tolerant Computing: Theory and Techniques,* Englewood Cliffs, N.J.: Prentice-Hall, 1986.
D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable Systems Design,* 2nd ed., Bedford, Mass.: Digital Press, 1992.