DEPENDABILITY AND ITS THREATS: A TAXONOMY

Algirdas Avižienis*, Jean-Claude Laprie**, Brian Randell***
** Vytautas Magnus University, Kaunas, Lithuania and University of California, Los Angeles, USA — aviz@adm.vdu.lt*
*** LAAS-CNRS, 7 avenue du Colonel Roche, 31077 Toulouse Cedex 04, France, jean-claude laprie@laas.fr*
**** School of Computing Science, University of Newcastle upon Tyne, UK brian.randell@newcastle.ac.uk*

Abstract: This paper gives the main definitions relating to dependability, a generic concept including as special case such attributes as reliability, availability, safety, confidentiality, integrity, maintainability, etc. Basic definitions are given first. They are then commented upon, and supplemented by additional definitions, which address the threats to dependability (faults, errors, failures), and the attributes of dependability. The discussion on the attributes encompasses the relationship of dependability with security, survivability and trustworthiness.

Key words: Dependability, availability, reliability, safety, confidentiality, integrity, maintainability, security, survivability, trustworthiness, faults, errors, failures.

1. ORIGINS AND INTEGRATION OF THE CONCEPTS

The delivery of correct computing and communication services has been a concern of their providers and users since the earliest days. In the July 1834 issue of the *Edinburgh Review*, Dr. Dionysius Lardner published the article "Babbage's calculating engine", in which he wrote:

> *"The most certain and effectual check upon errors which arise in the process of computation, is to cause the same computations to be made by separate and independent computers; and this check is rendered still more decisive if they make their computations by different methods".*

It must be noted that the term "computer" in the previous quotation refers to a person who performs computations, and not the "calculating engine".

The first generation of electronic computers (late 1940's to mid-50's) used rather unreliable components, therefore practical techniques were employed to improve their reliability, such as error control codes, duplexing with comparison, triplication with voting, diagnostics to locate failed components, etc. At the same time J. von Neumann [von Neumann 1956], E. F. Moore and C. E. Shannon [Moore & Shannon 1956], and their successors developed theories of using redundancy to build reliable logic structures from less reliable components, whose faults were masked by the presence of multiple redundant components. The theories of masking redundancy were unified by W. H. Pierce as the concept of *failure tolerance* in 1965 [Pierce 1965].

In 1967, A. Avižienis integrated masking with the practical techniques of error detection, fault diagnosis, and recovery into the concept of fault-tolerant systems [Avižienis 1967]. In the reliability modeling field, the major event was the introduction of the coverage concept by Bouricius, Carter and Schneider [Bouricius *et al*. 1969]. Work on software fault tolerance was initiated by Elmendorf [Elmendorf 1972], later it was complemented by recovery blocks [Randell 1975], and by N-version programming [Avižienis & Chen, 1977].

The formation of the *IEEE-CS TC on Fault-Tolerant Computing* in 1970 and of *IFIP WG 10.4 Dependable Computing and Fault Tolerance* in 1980 accelerated the emergence of a consistent set of concepts and terminology. Seven position papers were presented in 1982 at FTCS-12 in a special session on fundamental concepts of fault tolerance [FTCS 1982], and J.-C. Laprie formulated a synthesis in 1985 [Laprie 1985]. Further work by members of IFIP WG 10.4, led by J.-C. Laprie, resulted in the 1992 book *Dependability: Basic Concepts and Terminology* [Laprie 1992], in which the English text was also translated into French, German, Italian, and Japanese.

In this book, intentional faults (malicious logic, intrusions) were listed along with accidental faults (physical, design, or interaction faults). Exploratory research on the integration of fault tolerance and the defenses against deliberately malicious faults, i.e., security threats, was started in the mid-80's [Dobson & Randell 1986], [Joseph & Avižienis 1988], [Fray *et al*. 1986].

The first IFIP Working Conference on Dependable Computing for Critical Applications (DCCA) was held in 1989. This and the six Working Conferences that followed fostered the interaction of the dependability and security communities, and advanced the integration of security (confidentiality, integrity and availability) into the framework of dependable computing. Since 2000, the DCCA Working Conference together with the FTCS became parts of the International Conference on Dependable Systems and Networks (DSN).

2. THE BASIC CONCEPTS

In this section we present a basic set of definitions (in **bold** typeface) that will be used throughout the entire discussion of the taxonomy of dependable 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.

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 four fundamental properties: functionality, performance, dependability, and cost. Those four properties are collectively influenced by two other properties: usability and adaptability. 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. Dependability and cost have separate specifications. 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 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.

The **service** delivered by a system (the *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 **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.

It is usual to have a hierarchical view of a system structure. The relation *is composed of*, or *is decomposed into*, induces a hierarchy; however it relates only to the list of the system components. A hierarchy that takes into account the system behavior is the relation *uses* [Parnas 1974, Ghezzi *et al.*

1991] or *depends upon* [Parnas 1972, Cristian 1991]: a component *a* uses, or depends upon, a component *b* if the correctness of *b*'s service delivery is necessary for the correctness of *a*'s service delivery.

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: Failures, Errors, Faults

Correct service is delivered when the service implements the system function. A **service 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 4.

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**. 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 4 also can be partial failures.

2.3 Dependability and its Attributes

The general, qualitative, 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, quantitative, definition that provides the criterion for deciding if the service is dependable is: **dependability** of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable to the user(s).

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;
- **confidentiality:** absence of unauthorized disclosure of information;
- **integrity:** absence of improper system alterations;
- **maintainability**: ability to undergo, modifications, and repairs.

Security is the concurrent existence of a) availability for authorized users only, b) confidentiality, and c) integrity with 'improper' meaning 'unauthorized'.

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

The taxonomy of the attributes of dependability is presented in Section 5.

2.4 The Means to Attain Dependability

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

- **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 dependability specifications are adequate and that the system is likely to meet them.

The schema of the complete taxonomy of dependable computing as outlined in this section is shown in Figure 2.1.

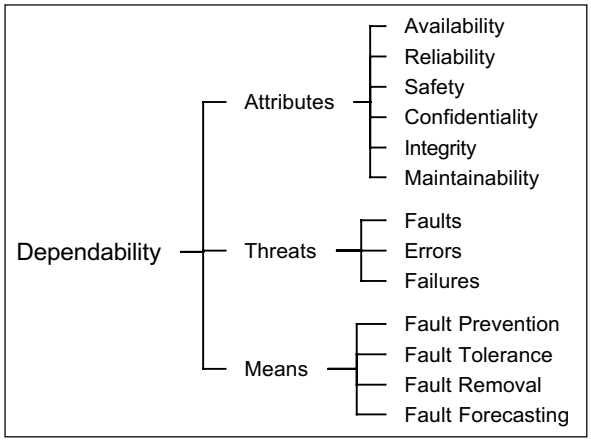

Figure 2.1: The dependability tree

3. THE TAXONOMY OF FAULTS

3.1 System Life Cycle: Phases and Environments

In this and the next 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 be deployed for use in its user's environment. During the development phase the system is interacting 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.

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. *the administrators* (including maintainers): entities (humans, 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. *the users:* entities that receive service at the service interfaces;
4. *the providers:* entities that deliver services to the system at its service interfaces;
5. *the fixed resources:* entities that are not users, but provide specialized services to the system, such as information sources (e.g., GPS, time, etc.), communication links, power sources, cooling airflow, etc.
6. *the intruders*: malicious entities that have no authority but attempt to intrude into the system and to alter service or halt it, alter the system's functionality or performance, or to access confidential information. They are hackers, malicious insiders, agents of hostile governments or organizations, and info-terrorists.

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 Figure 3.1.

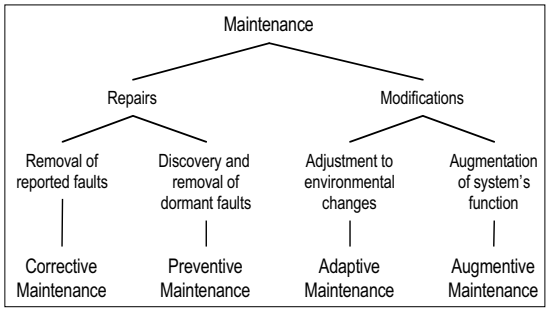

Figure 3.1: The various forms of maintenance

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.

3.2 A Taxonomy of Faults

All faults that may affect a system during its life are classified according to eight basic viewpoints that are shown in Figure 3.2. These fault classes are called *elementary faults.*

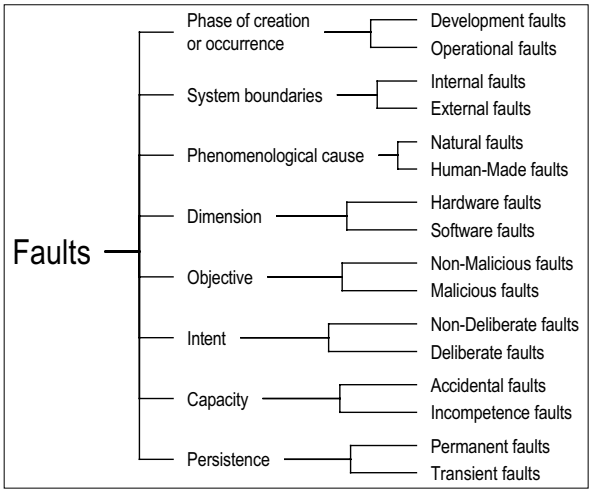

Figure 3.2: The elementary fault classes

The classification criteria are as follows:

1. The *phase* of system life during which the faults originate:

- **development faults** that 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** that occur during service delivery of the use phase.

2. The *location* of the faults with respect to the system boundary:

- **internal faults** that originate inside the system boundary;
- **external faults** that originate outside the system boundary and propagate errors into the system by interaction or interference.

3. The *phenomenological cause* of the faults:

- **natural faults** that are caused by natural phenomena without human participation;
- **human-made faults** that result from human actions.

4. The *dimension* in which the faults originate:
- **hardware (physical) faults** that originate in, or affect, hardware;
- **software (information) faults** that affect software, i.e., programs or data.

5. The *objective* of introducing the faults:
- **malicious faults** that are introduced by a human with the malicious objective of causing harm to the system;
- **non-malicious faults** that are introduced without a malicious objective.

6. The *intent* of the human(s) who caused the faults:
- **deliberate faults** that are the result of a harmful decision;
- **non-deliberate faults** that are introduced without awareness.

7. The *capacity* of the human(s) who introduced the faults:
- **accidental faults** that are introduced inadvertently;
- **incompetence faults** that result from lack of professional competence by the authorized human(s), or from inadequacy of the development organization.

8. The *temporal persistence* of the faults:
- **permanent faults** whose presence is assumed to be continuous in time;
- **transient faults** whose presence is bounded in time.

If all combinations of the eight elementary fault classes were possible, there would be 256 different *combined fault classes*. In fact, the number of likely combinations is 31; they are shown in Figures 3.3 and 3.4.

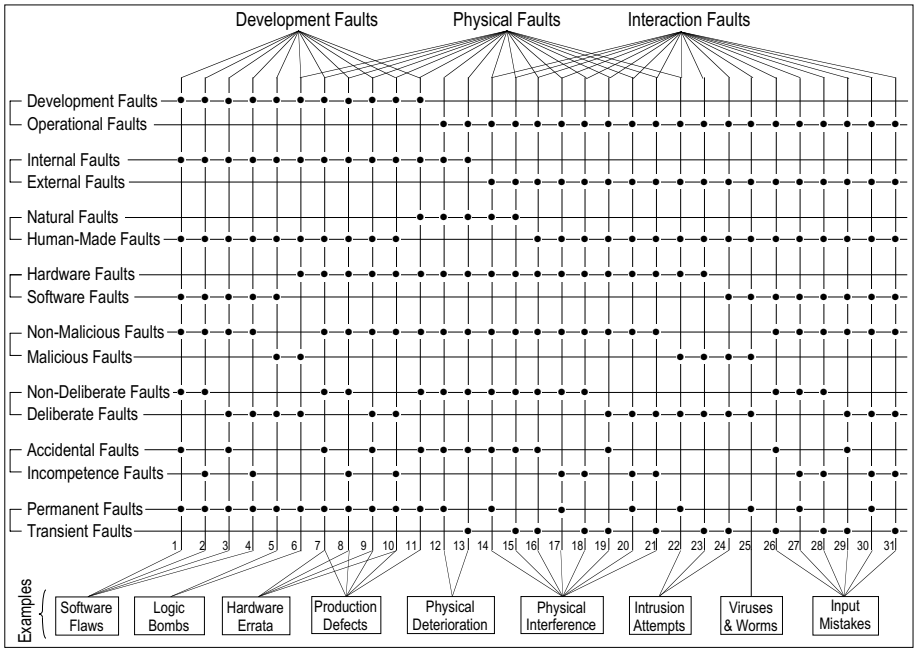

Figure 3.3: The classes of combined faults

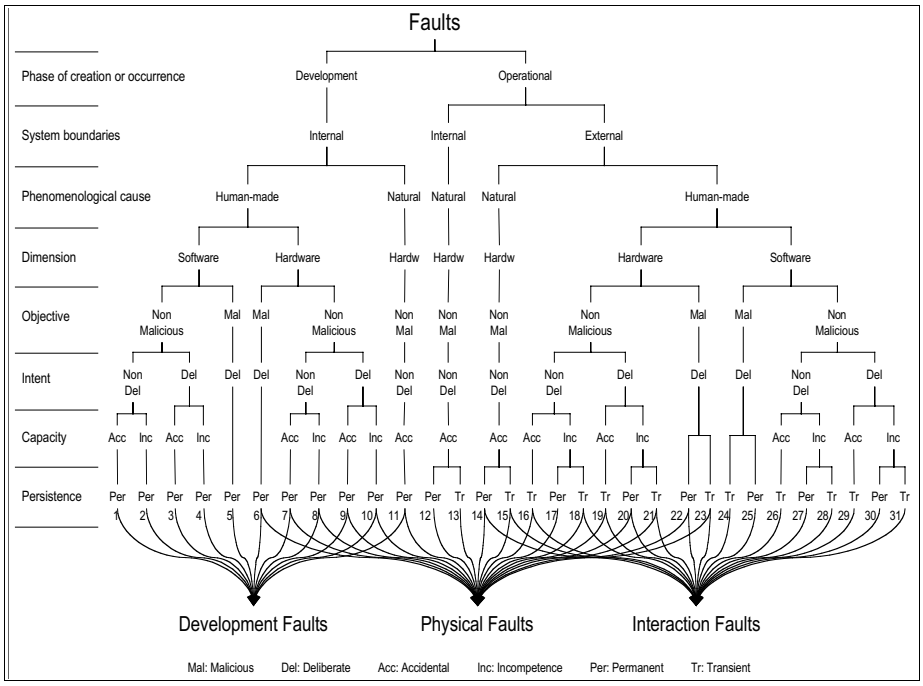

Figure 3.4: Tree representation of combined faults

The combined faults of Figures 3.3 and 3.4 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 Figure 3.3 identify the names of some illustrative fault classes.

3.3 On Human-Made Faults

The definition of human-made faults (that result from harmful 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**.

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 intent to cause harm to the system during its use (#5-#6), or directly during use (#22-#25)

- ***non-malicious faults*** (#1-#4, #7-#21, #26-#31), introduced without malicious objectives.

Malicious human-made faults are introduced by a developer with the malicious objective to alter the functioning of the system during use. The goals of such faults are: (1) to disrupt or halt service, (2) to access confidential information, or (3) to improperly modify the system. They are grouped into two classes:

- *potentially harmful components* (#5, #6): Trojan horses, trapdoors, logic or timing bombs;
- deliberately introduced software or hardware *vulnerabilities or human-made faults.*.

The goals of malicious faults are: (1) to disrupt or halt service (thus provoking *denials-of-service*), (2) to access confidential information, or (3) to improperly modify the system. They fall into two classes:

1. **Malicious logic faults**, that encompass development faults such as *Trojan horses*, logic or timing *bombs*, and *trapdoors*, as well as operational faults such as *viruses*, *worms*, or *zombies*. Definitions for these faults are as follows [Landwehr *et al*. 1994, Powell & Stroud 2003]:
   - **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.
2. **Intrusion attempts**, that are operational external faults. 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, etc.

Non-malicious human-made faults can be partitioned according to the developer's intent:

- *non-deliberate* faults that are due to *mistakes*, that is, *unintended actions* of which the developer, operator, maintainer, etc. is not aware,
- *deliberate* faults that are due to *bad decisions*, that is, *intended actions* that are wrong and causes faults.

Deliberate, non-malicious, development faults result generally from tradeoffs, either a) aimed at preserving acceptable performance, at facilitating system utilization, or b) induced by economic considerations. Deliberate, non-malicious interaction faults may result from the action of an operator either aimed at overcoming an unforeseen situation, or deliberately violating an operating procedure without having realized the possibly damaging consequences of this action. Deliberate non-malicious faults share the property that often it is recognized that they were faults only *after* an unacceptable system behavior, thus a failure, has ensued; the developer(s) or operator(s) did not realize that the consequence of their decision was a fault.

It is often 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 non-malicious 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 both classes of non-malicious human-made faults into (1*) accidental faults*, and (2) *incompetence faults.* The structure of this human-made fault taxonomy is shown in Figure 3.5.

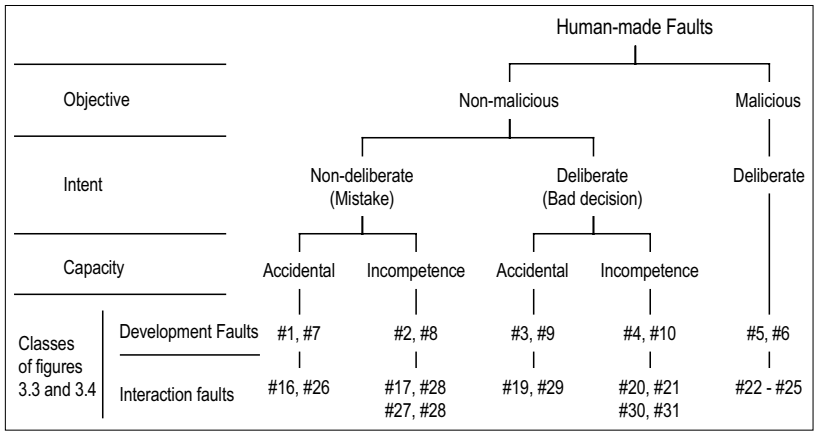

Figure 3.5: Classification of human-made faults

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 will 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 human-made failure of the AAS air traffic control system described in Section 4.2.

The purpose of this fault taxonomy is to present a complete and structured view of the universe of faults. The explicit introduction of incompetence faults in the taxonomy serves as a reminder that incompetence at individual and organizational level is a serious threat in the human-made development and use of dependable systems..

The non-malicious development faults exist in hardware and in software. In hardware, especially in microprocessors, some development faults are discovered after production has started [Avižienis & He 1999]. Such faults are called "errata" and are listed in specification updates [Intel 2001]. The finding of errata 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. The best known of such "secondary" human-made faults is the Pentium division erratum [Meyer 1994].

Designing a system always recurs to some extent to incorporating off-the-shelf (OTS) components. The use of OTS components introduces additional dependability 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** [Huang *et al*. 1995], i.e., progressively accrued error conditions resulting in performance degradation or complete failure. Examples are [Castelli *et al*. 2001] memory bloating and leaking, unterminated threads, unreleased file-locks, data corruption, storage space fragmentation, accumulation of round-off errors.

3.4 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 Figures 3.3 and 3.4. 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. [Gray 2001]. Such faults can occur during configuration changes performed during adaptive or augmentative maintenance performed concurrently with system operation (e.g., introduction of a new software version on a network server); they are then called **reconfiguration faults** [Wood 1994].

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 non-malicious, as can be the external fault 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.

3.5 On Physical Faults

Physical faults shown on Figure 3.3 fall into three categories: purely natural faults (#12-#15), physical development faults (#6-#11), and physical interaction faults (#16-#23). Development and interaction faults have been discussed in the preceding sections. The purely 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 service interfaces (power transients, noisy input lines, etc.).

4. THE TAXONOMY OF FAILURES AND ERRORS

4.1 Service Failures

In Section 2.2 a ***service failure*** (called simply "failure" in this section) 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 omissions 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 happen to be recognized only after its occurrence, for instance via its consequences. So failures can be subjective, disputable, i.e., require judgment to identify and characterize.

The failure modes characterize incorrect service according to four viewpoints: a) the failure domain, b) the detectability of failures, c) the consistency of failures, and d) 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: a) the content can be in numerical or non-numerical sets (e.g., alphabets, graphics, colors, sounds), and b) a timing failure may be **early** or **late**, depending on whether the service is delivered too early or too late. Late failures with correct information are **performance failures** [Cristian 1991]; these can relate to the two aspects of the notion of performance: responsiveness or throughput [Muntz 2000]. Failures when both information and timing are incorrect fall into two classes:

- **halt failure**, or simply **halt**, when the service is *halted* (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).

Figure 4.1 summarizes the failure modes with respect to the failure domain viewpoint.

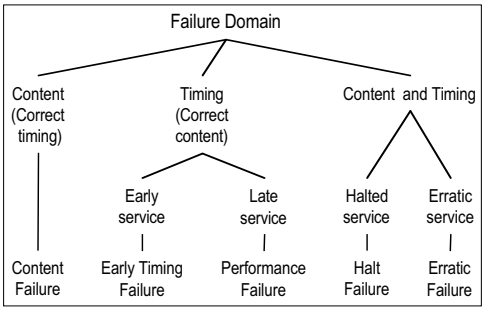

Figure 4.1: Failure modes with respect to the failure domain viewpoint

The detectability viewpoint addresses the signaling of losses of functions to the user(s). The losses result in reduced modes of service. 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: a) signaling a loss of function when they no failure has actually occurred, that is a **false alarm**, b) not signaling a function loss, that is an *unsignaled failure*. Upon detecting the loss of one or more functions, the system retains a specified reduced set of functions and 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[1]; inconsistent failures are usually called, after [Lamppost *et al*. 1982], *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 attributes for the considered application(s). Examples of criteria for determining the classes of failure severities are:

- for availability, the outage duration,
- for safety, the possibility of human lives to be endangered,
- for confidentiality, the type of information that may be unduly disclosed,

[1] Some users may actually perceive correct service.

- 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 as the benefit 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.

Figure 4.2 summarizes the failure modes.

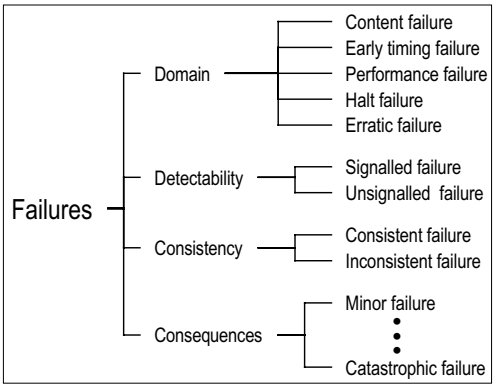

Figure 4.2: Failure modes

Systems that are designed and implemented so that they fail only in specific modes of failure described in the dependability 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 system**; the situations of stuck service and of silence lead respectively to **fail-passive systems** and to **fail-silent systems** [Powell *et al*. 1988]. 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 for service restoration after a failure occurred: a) automatic or operator-assisted recovery, restart or reboot, b) corrective maintenance. Correction of development faults is usually performed off-line, 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**[2].

4.2 Development Failures

A **development failure** causes the development process to be terminated before the system is accepted for use. 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:

1. Too *numerous specification changes initiated by the user*. They have the same impact on the development process as the detection of specification faults, requiring re-design with possibility of new development faults being introduced.
2. *Inadequate design*. The functionality and/or performance goals cannot be met.
3. *Too many faults*. Introduction of an excessive number of development faults and/or inadequate capability of fault removal during development.
4. *Insufficient dependability*. The dependability forecasting by analytical and experimental means shows that the specified dependability goals cannot be met.
5. *Faulty (too low) estimates of development costs*, either in funds, or in time needed, or both. They are usually due to an underestimate of the complexity of the system to be developed.

Budget and/or schedule *overruns* occur when the development is completed, but the funds or time needed to complete the effort exceed the original estimates. The overruns are *partial development failures*, i.e., failures of lesser severity than project termination. 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 than required in the original system specification.

Development failures, overruns, and downgrades have a very negative impact on the user community, as exemplified by Figure 4.3.

[2] Service shutdowns are also called *planned outage*, as opposed to outages consecutive to failures, that are then called *unplanned outages*.

	1994	2002
Number of surveyed projects	8,380	13,522
Successful projects (completed on-time and on-budget, with all features and functions as initially specified)	16%	34%
Challenged projects (completed and operational but over-budget, over the time estimate, and offers fewer features and functions than originally specified)	53%	51%
Canceled projects	31%	15%
Overruns for challenged projects	89%	82%
Left functions for challenged projects	61%	52%
Total estimated budget for software projects in the USA, in $ billion	250	225
Estimated lost value for software projects in the USA, in $ billion	81	38

(a) Large software projects [www.standishgroup.com]

The Advanced Automation System (AAS) was intended to replace the aging air traffic control systems in the USA [Hunt & Kloster 1987]. In 1984 the FAA awarded competitive design phase contracts to IBM and Hughes Aircraft Co. In July 1988 an acquisition phase contract of $3.5 billion was awarded to IBM, and the program cost, including supporting efforts, was estimated by the FAA to be $4.8 billion. In 1994 FAA estimated that the program would cost $7 billion, with key segments as much as eight years behind schedule. The AAS as originally conceived, was terminated in June 1994, and an investigation showed that $2.6 billion were spent, of which $1.5 billion was completely wasted. Of the five program segments, only the simplest one was completed, one was restructured under a new contract, and three were terminated. The main causes of development failure were reported to be (1) overambitious plans, (2) poor oversight of software development, (3) FAA's inability to stabilize requirements, and (4) a poor statement of work in the original contract [US DOT, 1998].

(b) The AAS system

Figure 4.3: Development failures

4.3 Dependability Failures

It is 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 specification* is agreed upon that states the goals for each dependability attribute: availability, reliability, safety, confidentiality, integrity, and manageability.

The specification explicitly identifies the *classes of faults* that are expected and the *use environment* in which the system will operate. The dependability 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 failure** occurs when the given system fails more frequently or more severely than acceptable to the user(s).

Dependability and cost are not part of the functional specification. For this reason we call them the *meta-functional specification*. We object to the often used term “non-functional”, since that also means “failed”. A complete system specification consists of both, as shown in Figure 4.4.

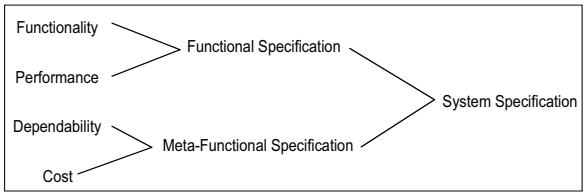

Figure 4.4: Elements of the system specification

The dependability specification can contain also 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 AAS complete outage limit of 3 seconds per year was changed to 5 minutes per year for the new contract in 1994 [US DOT 1998].

4.4 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 failure depends on two factors:

1. The structure of the system, and especially the nature of any redundancy that exists in it:
   - *intentional* 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 4.1: content *vs.* timing errors, detected *vs.* latent errors, consistent *vs.* inconsistent errors when the service goes to two or more users, minor *vs.* catastrophic 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.

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

The creation and manifestation mechanisms of faults, errors, and failures are illustrated by Figure 4.5, and summarized as follows:

1. A fault is **active** when it produces an error, otherwise it is **dormant**. An active fault is either a) an internal fault that was previously dormant and that has been activated by the computation process or environmental conditions, or b) 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 one component (C1) to another component (C2) that receives service from C1 (i.e., *external* propagation) occurs when, through internal propagation, an error reaches the service interface of component C1. At this time, service delivered by C2 to C1 becomes incorrect, and the ensuing failure of C1 appears as an external fault to C2 and propagates the error into C2.
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. A failure of a component causes a permanent or transient fault in the system that contains the component. Failure of a system causes a permanent or transient external fault for the other system(s) that interact with the given system.

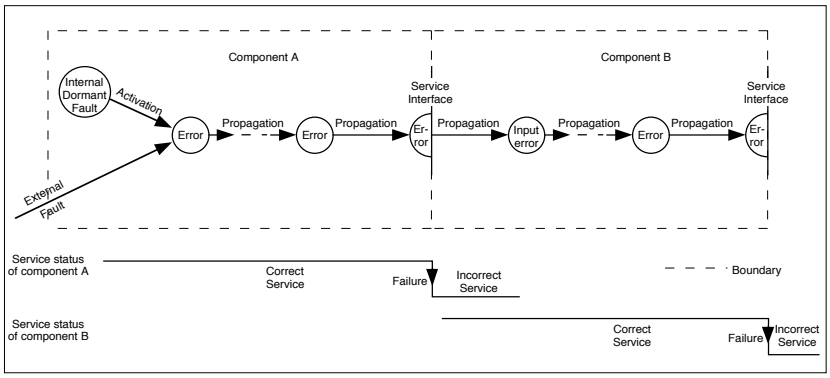

Figure 4.5: Error propagation

These mechanisms enable the 'fundamental chain' to be completed, as indicated by Figure 4.6.

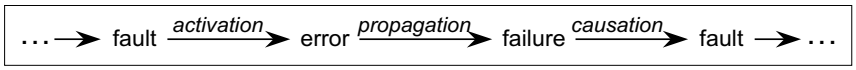

Figure 4.6: The fundamental chain of dependability threats

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. Propagation, and thus instantiation(s) of the chain, can occur via the two fundamentals dimensions associated to the definitions of systems given in Section 2.1: interaction and composition.

Some illustrative examples of fault pathology are given in Figure 4.7. From those examples, it is easily understood that 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 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 requests, that occur rarely and can be very difficult to reproduce [Gray 86]. 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 e.g. marginal timing and synchronization.

- 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 logic 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.

Figure 4-7: Examples illustrating fault pathology

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.**

Situations involving multiple faults and/or failures are frequently encountered. 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 a) **independent faults**, that are attributed to different causes, and b) **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 lead to similar errors [Avižienis & Kelly 1984], or that related faults lead to distinct errors. The failures caused by similar errors are **common-mode failures**.

5- DEPENDABILITY AND ITS ATTRIBUTES

In Section 2, we have presented two alternate definitions of dependability:

1. A qualitative definition: the ability to deliver service that can justifiably be trusted.
2. A quantitative definition: the ability of a system to avoid failures that are more frequent or more severe than is acceptable to the user(s).

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" [ISO 1992].
- "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" [IEC 1992].

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 [CCITT 1984], at a time when availability was the main concern to telephone operating companies. However, the willingness to grant dependability with 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 [Hosford 1960] for dependability: "the probability that a system will operate when needed". The second definition, from [IEC 1992], introduces the notion of reliance, and as such is much closer to our definitions.

Other concepts similar to dependability exist, as survivability and trustworthiness. They are presented and compared to dependability in Figure 5.1.

A side-by-side comparison leads to the conclusion that all three concepts are essentially equivalent in their goals and address similar threats. Trustworthiness omits the explicit listing of internal faults, although its goal implies that they also must be considered. Such faults are implicitly considered in survivability via the (component) failures. Survivability was present in the late sixties in the military standards, where it was defined as a

system capacity to resist hostile environments so that the system can fulfill its mission (see, e.g., MIL-STD-721 or DOD-D-5000.3); it was redefined recently, as described in Figure 5.1. Trustworthiness was used in a study sponsored by the National Research Council, referenced in Figure 5.1. One difference must be noted. Survivability and trustworthiness have the threats explicitly listed in the definitions, while both definitions of dependability leave the choice open: the threats can be either all the faults of Figure 3.3 and Figure 3.4, or a selected subset of them, e.g., 'dependability with respect to development faults', etc.

Concept	Dependability	Survivability	Trustworthiness
Goal	1) ability to deliver service that can justifiably be trusted 2) ability of a system to avoid failures that are more frequent or more severe than is acceptable to the user(s)	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)	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	"Survivable network systems" [Ellison *et al.* 1999]	"Trust in cyberspace" [Schneider 1999]

Figure 5.1: Dependability, survivability and trustworthiness

The attributes of dependability that have been defined in Section 2 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, confidentiality may or may not be required according to the application. The extent to which a system possesses the attributes of dependability 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 a) relate to the notion of authorized actions only, and, b) focus on information (e.g., prevention of

the unauthorized amendment or deletion of information [CEC 1991], assurance of approved data alterations [Jacob 1991]): a) naturally, when a system implements an authorization policy, 'improper' encompasses 'unauthorized', b) 'improper alterations' encompass actions that prevent (correct) upgrades of information, and c) '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.

Security has not been introduced 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" [CEC 1991, Pfleeger 2000]. A 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 Figure 5.2.

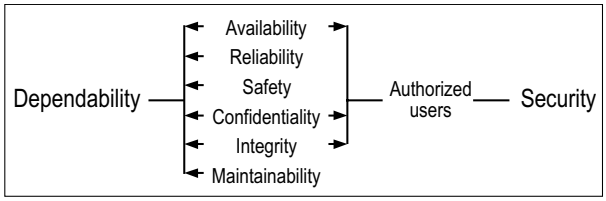

Figure 5.2: Dependability and security

In their definitions, availability and reliability emphasize the avoidance of failures, while safety and security emphasize the avoidance of a specific class of failures (catastrophic failures, unauthorized access or handling of information, respectively). Reliability and availability are thus closer to each other than they are to safety on one hand, and to security on the other; reliability and availability can thus be grouped together, and be collectively defined as the avoidance or minimization of service outages.

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 specializing secondary attribute is **robustness**, i.e., dependability with respect to external faults, that characterizes a system reaction to a specific class of faults.

The notion of secondary attributes is especially relevant for security, when we distinguish among various types of information [Cachin *et al.* 2000]. 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,
- **non-repudiability**: availability and integrity of the identity of the sender of a message (non-repudiation of the origin), or of the receiver (non-repudiation of reception).

Dependability classes are generally defined via the analysis of failure frequencies and severities, and of outage durations, for the dependabilty attributes that are of concern for a given application. This analysis may be conducted directly, or indirectly, via risk assessment (see, e.g., [Grigonis 2001] for availability, [RTCA/EUROCAE 1992] for safety, and [ISO/IEC 1999] for security).

The variations in the emphasis placed on the different attributes of dependability directly influence the balance of the techniques (fault prevention, tolerance, removal and forecasting) to be employed in order to make the resulting system dependable. This problem is all the more difficult as some of the attributes are conflicting (e.g., availability and safety, availability and security), necessitating that trade-offs be made. Regarding the three main development dimensions of a computing system besides functionality, i.e., cost, performance and dependability, the problem is further exacerbated by the fact that the dependability dimension is less understood than the cost-performance development space [Siewiorek & Johnson 1992].

6. CONCLUSION

Increasingly, individuals and organizations are developing or procuring sophisticated computing systems on whose services they need to place great reliance — 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. The notion of dependability provides a very convenient means of subsuming these various concerns within a single conceptual framework. Dependability includes as special cases such properties as availability, reliability, safety, confidentiality, integrity, maintainability. It also provides the means of addressing the problem that what a user usually needs from a system is an *appropriate balance* of these properties.

A major strength of the dependability concept, as it is formulated in this paper, is its integrative nature, that enables to put into perspective the more classical notions of reliability, availability, safety, security, maintainability, that are then seen as attributes of dependability. The fault-error-failure model is central to the understanding and mastering of the various threats that may affect a system, and it enables a unified presentation of these threats, while preserving their specificities via the various fault classes that can be defined. The model provided for the means for dependability is extremely useful, as those means are much more orthogonal to each other than the more classical classification according to the attributes of dependability, with respect to which the design of any real system has to perform trade-offs due to the fact that these attributes tend to conflict with each other.

What has been presented is an attempt to document a minimum consensus within the community in order to facilitate fruitful technical interactions. The associated terminology effort is not an end in itself: words are only of interest because they unequivocally label concepts, and enable ideas and viewpoints to be shared.

REFERENCES

[Avižienis 1967] A. Avižienis, "Design of fault-tolerant computers", in *Proc. 1967 Fall Joint Computer Conf., AFIPS Conf. Proc. Vol. 31,* pp. 733-743, 1967.

[Avižienis & Chen, 1977] A. Avižienis and L. Chen, "On the implementation of N-version programming for software fault tolerance during execution", in *Proc. IEEE COMPSAC 77*, pp. 149-155, Nov. 1977.

[Avižienis & He 1999] A. Avižienis, Y. He, "Microprocessor entomology: a taxonomy of design faults in COTS microprocessors", in Dependable Computing for Critical Applications 7, C.B. Weinstock and J. Rushby, eds, IEEE CS Press, 1999, pp. 3-23.

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

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

[Cachin *et al*. 2000] C. Cachin, J. Camenisch, M. Dacier, Y. Deswarte, J. Dobson, D. Horne, K. Kursawe, J.-C. Laprie, J.C. Lebraud, D. Long, T. McCutcheon, J. Muller, F. Petzold, B. Pfitzmann, D. Powell, B. Randell, M. Schunter, V. Shoup, P. Veríssimo, G. Trouessin, R.J. Stroud, M. Waidner, I. Welch, "Malicious- and Accidental-Fault Tolerance in Internet Applications: reference model and use cases", LAAS report no. 00280, MAFTIA, Project IST-1999-11583, Aug. 2000, 113p.

[Castelli *et al*. 2001] V. Castelli, R.E. Harper, P. Heidelberger, S.W. Hunter, K.S. Trivedi, K. Vaidyanathan, W.P. Zeggert, "Proactive management of software aging", *IBM J. Res.& Dev.*, vol. 45, no. 2, March 201, pp. 311-332.

[CCITT 1984] Termes et définitions concernant la qualité de service, la disponibilité et la fiabilité, Recommandation G 106, CCITT, 1984; in French.

[CEC 1991] *Information Technology Security Evaluation Criteria,* Harmonized criteria of France, Germany, the Netherlands, the United Kingdom, Commission of the European Communities, 1991.

[Cristian 1991] F. Cristian, “Understanding Fault-Tolerant Distributed Systems”, *Com. of the ACM*, vol. 34, no. 2, pp. 56-78, 1991.

[Dobson & Randell 1986] J.E. Dobson and B. Randell. Building reliable secure computing systems out of unreliable insecure compnents. In *Proc. of the 1986 IEEE Symp. Security and Privacy*, pp. 187-193, April 1986.

[Ellison *et al*. 1999] R.J. Ellison, D.A. Fischer, R.C. Linger, H.F. Lipson, T. Longstaff, N.R. Mead, “Survivable network systems: an emerging discipline”, Technical Report CMU/SEI-97-TR-013, November 1997, revised May 1999.

[Elmendorf 1972] W.R. Elmendorf, “Fault-tolerant programming”, in *Proc. 2nd IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-2)*, Newton, Massachusetts, June 1972, pp. 79-83.

[Fray *et al*. 1986] J.-M. Fray, Y. Deswarte, D. Powell, “Intrusion tolerance using fine-grain fragmentation-scattering”, in *Proc. 1986 IEEE Symp. on Security and Privacy*, Oakland, April 1986, pp. 194-201.

[FTCS 1982] Special session. Fundamental concepts of fault tolerance. In *Digest of FTCS-12*, pages 3-38, June 1982.

[Ghezzi *et al*. 1991] C. Ghezzi, M. Jazayeri, D. Mandrioli, *Fundamentals of Software Engineering*, Prentice-Hall, 1991.

[Gray 2001] J. Gray, “Functionality, Availability, Agility, Manageability, Scalability -- the New Priorities of Application Design”, *in Proc. HPTS 2001*, Asilomar, April 2001.

[Grigonis 2001] R. Grigonis, “Fault-resilience for communications convergence", *Special Supplement to CMP Media's Converging Communications Group*, Spring 2001.

[Hosford 1960] J.E. Hosford, “Measures of dependability”, *Operations Research*, vol. 8, no. 1, 1960, pp. 204-206.

[Huang *et al*. 1995] Y. Huang, C. Kintala, N. Kolettis, N.D. Fulton, “Software rejuvenation: analysis, module and applications”, in *Proc. 25th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-25)*, Pasadena, California, June 1995, pp. .

[Hunt & Kloster 1987] V.R. Hunt & G.V. Kloster, editors, "The FAA's Advanced Automation Program", special issue, *Computer*, February 1987.

[IEC 1992] Industrial-process measurement and control — Evaluation of system properties for the purpose of system assessment. Part 5: Assessment of system dependability, Draft, Publication 1069-5, International Electrotechnical Commission (IEC) Secretariat, Feb. 1992.

[Intel 2001] Intel Corp. Intel Pentium III Processor Specification Update, May 2001. Order No.244453-029.

[ISO 1992] Quality Concepts and Terminology, Part one: Geberic Terms and Definitions, Document ISO/TC 176/SC 1 N 93, Feb. 1992.

[ISO/IEC 1999] *Common Criteria for Information Technology Security Evaluation*, ISO/IEC Standard 15408, August 1999.

[Jacob 1991] J. Jacob. “The Basic Integrity Theorem”, in *Proc. Int. Symp. on Security and Privacy*, pp. 89-97, Oakland, CA, USA, 1991.

[Joseph & Avižienis 1988] M.K. Joseph and A. Avižienis, “A fault tolerance approach to computer viruses”, in *Proc. of the 1988 IEEE Symposium on Security and Privacy*, pages 52-58, April 1988.

[Lamport *et al*. 1982] L. Lamport, R. Shostak, M. Pease, “The Byzantine generals problem”, *ACM Trans.on Programming Languages and Systems*, vol. 4, no. 3, July 1982, pp. 382-401.

[Landwehr *et al*. 1994] C.E. Landwher, A.R. Bull, J.P. McDermott, W.S. Choi, “A Taxonomy of Computer Program Security Flaws”, *ACM Computing Surv.*, vol. 26, no. 3, pp. 211-254, 1994.

[Laprie 1985] J.-C. Laprie. Dependable computing and fault tolerance: concepts and terminology. In *Proc. 15th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-15)*, Ann Arbor, June 1985, pp. 2-11, .

[Laprie 1992] J.-C. Laprie, editor, *Dependability: Basic Concepts and Terminology*, Springer-Verlag, 1992.

[Meyer 1994] Meyer, Michael, "A 'Lesson' for Intel: How It Mishandled the Pentium Flap", *Newsweek*, December 12, 1994, p.58.

[Moore & Shannon 1956] E.F. Moore and C.E. Shannon, "Reliable circuits using less reliable relays", *J. Franklin Institute*, 262:191-208 and 281-297, Sept/Oc. 1956.

[Muntz 2000] R.R. Muntz, "Performance measurement and evaluation", in *Encyclopedia of Computer Science*, A.Ralston, E.D. Reilly, D. Hemmendinger, eds, Nature Publishing Group, 2000.

[Parnas 1972] D. Parnas, "On thecriteria to be used in decomposing systems into modules", Comunications of the ACM, vol. 15, no. 12, Dec. 1972, pp. 1053-1058.

[Parnas 1974] D. Parnas, "On a 'buzzword': hierarchical structure", in *Proc. Information Processing 74"*.

[Pfleeger 2000] C.P. Pfleeger, "Data security", in *Encyclopedia of Computer Science*, A.Ralston, E.D. Reilly, D. Hemmendinger, eds, Nature Publishing Group, 2000, pp. 504-507.

[Pierce 1965] W.H. Pierce, *Failure-Tolerant Computer Design*, Academic Press, 1965.

[Powell *et al*. 1988] D. Powell, G. Bonn, D. Seaton, P. Veríssimo, F. Waeselynck, "The Delta-4 approach to dependability in open distributed computing systems", in *Proc. 18th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-18)*, Tokyo, Japan, June 1988, pp. 246-251.

[Powell & Stroud 2003] D. Powell, R. Stroud, editors, "Conceptual Model and Architecture of MAFTIA", MAFTIA, Project IST-1999-11583, Jan. 2003, 123p.

[Randell 1975] B. Randell, "System structure for software fault tolerance", *IEEE Transactions on Software Engineering*, SE-1:1220-232, 1975.

[RTCA/EUROCAE 1992] *Software considerations in airborne systems and equipment certification*, DO-178-B/ED-12-B, Requirements and Technical Concepts for Aviation/European Organisation for Civil Aviation Equipment, 1992.

[Schneider 1999] F. Schneider, ed., *Trust in Cyberspace*, National Academy Press, 1999.

[Siewiorek & Johnson 1992] D.P. Siewiorek, D. Johnson, "A design methodology for high reliability systems: the Intel 432", in D.P. Siewiorek, R.S. Swarz, *Reliable Computer Systems, Design and Evaluation*, Digital Press, 1992, pp. 737-767.

[US DOT 1998] USA Department of Transportation, Office of Inspector General, Audit Report: Advance Automation System, Report No. AV-1998-113, April 15, 1998.

[von Neumann 1956] J. von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components", in C. E. Shannon and J. McCarthy, editors, *Annals of Math Studies*, numbers 34, pages 43-98. Princeton Univ. Press, 1956.

[Wood 1994] A. Wood, "NonStop availability in a client/server environment", Tandem Technical Report 94.1, March 1994.