Chapter 4

Dependability Concepts[1]

This chapter does not deal with Delta-4 *per se* but is aimed at giving informal but precise definitions characterizing the various attributes of computing systems dependability. It is a contribution to the work undertaken within the "Reliable and Fault Tolerant Computing" scientific and technical community [Anderson and Lee 1981, Avizienis 1978, Avizienis and Laprie 1986, Carter 1979, Cristian et al. 1985, FTCS12, Jessep 1977, Laprie 1985, Laprie 1989, Melliar-Smith and Randell 1977, Randell et al. 1978, Siewiorek and Johnson 1982] in order to propose clear and widely acceptable definitions for some basic concepts. Readers already familiar with this terminology may wish to skip this chapter.

Dependability is first introduced as a global concept that subsumes the usual attributes of reliability, availability, safety, security. The basic definitions given in the first section are then commented, and supplemented by additional definitions, in the subsequent sections. A glossary is given in annex, which recapitulates the definitions given throughout the chapter. The presentation has been structured so as to avoid forward referencing. Underlining is used when a term is defined, italic characters being an invitation to focus the reader's attention. The guidelines that have governed this presentation can be summed up as follows:

- search for the minimum number of concepts enabling the dependability attributes to be expressed;
- use of terms which are identical to — whenever possible — or as close as possible to those generally used; as a rule, a term which has not been defined retains its ordinary sense (as given by any dictionary);
- emphasis on integration [Goldberg 1982, Randell and Dobson 1986] (as opposed to specialization) through the independence of the given definitions with respect to the classes of faults.

This contents of this chapter can be seen as a minimum consensus within the community in order to facilitate fruitful interactions; in addition the material presented is hoped to be suitable a) for being used by other bodies (including standards organizations), and b) for educational purposes. In this view, the associated terminology effort is not an end in itself: words are only of interest in so far as they transmit ideas, subject them to criticism, and enable viewpoints to be shared. There is no pretension of this chapter representing *the* state-of-the-art or "Tablets of Stone": the presented concepts have to evolve with technology, and with our progress in understanding and mastering the design and the assessment of dependable computer systems.

---

1 This chapter is a result of work partially financed by the Esprit basic research action project PDCS (Predictably Dependable Computing Systems). It is also the basis of pre-standardization work being carried out by the IFIP 10.4 working group on Dependable Computing and Fault-Tolerance. It has been included in this book about the Delta-4 architecture in order to provide a well-defined terminological and conceptual framework.

4.1. Basic Definitions

Dependability is defined as the trustworthiness of a computer system such that reliance can justifiably be placed the service it delivers [Carter 1982]. The service delivered by a system is its behavior *as it is perceived* by its user(s); a user is another system (human or physical) which *interacts* with the former.

Depending on the application(s) intended for the computer system under consideration, different emphasis may be put on different facets of dependability, i.e., dependability may be viewed according to different, but complementary, *properties,* which enable the *attributes* of dependability to be defined:

- with respect to the *readiness for usage*, dependable means available;
- with respect to the *continuity of service,* dependable means reliable;
- with respect to the *avoidance of catastrophic consequences on the environment,* dependable means safe;
- with respect to the *prevention of unauthorized handling of information*, dependable means secure.

A system failure occurs when the delivered service no longer complies with the specification, the latter being an *agreed* description of the system's expected function and/or service. An error is that part of the system state that is liable to lead to failure: an error affecting the service, i.e., becoming user-visible, is an indication that a failure occurs or has occurred. The *adjudged or hypothesised* cause of an error is a fault.

The development of a dependable computing system calls for the *combined* utilization of a set of methods that can be classed into:

- fault prevention: how to prevent fault occurrence or introduction;
- fault tolerance: how to provide a service complying with the specification in spite of faults;
- fault removal: how to reduce the presence (number, seriousness) of faults;
- fault forecasting: how to estimate the present number, the future incidence, and the consequences of faults.

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

Reliance on the system's service, and justification for reliance, are based on the *assessment* of the system, conducted primarily with respect to the *attributes* of dependability.

The notions introduced up to now can be grouped into three classes (figure 1):

- the impairments to dependability: faults, errors, failures; they are undesired — but not in principle unexpected — circumstances causing or resulting from un-dependability (whose definition is very simply derived from the definition of dependability: reliance cannot, or will not any longer, be placed on the service);
- the means for dependability: fault prevention, fault tolerance, fault removal, fault forecasting; these are the methods, tools, and solutions enabling one a) to provide the ability to deliver a service on which reliance can be placed, and b) to reach confidence in this ability.

- the attributes of dependability: reliability, availability, safety, security; these enable a) which properties are expected from the system to be expressed, and b) the system quality resulting from the impairments and the means opposing to them to be assessed.

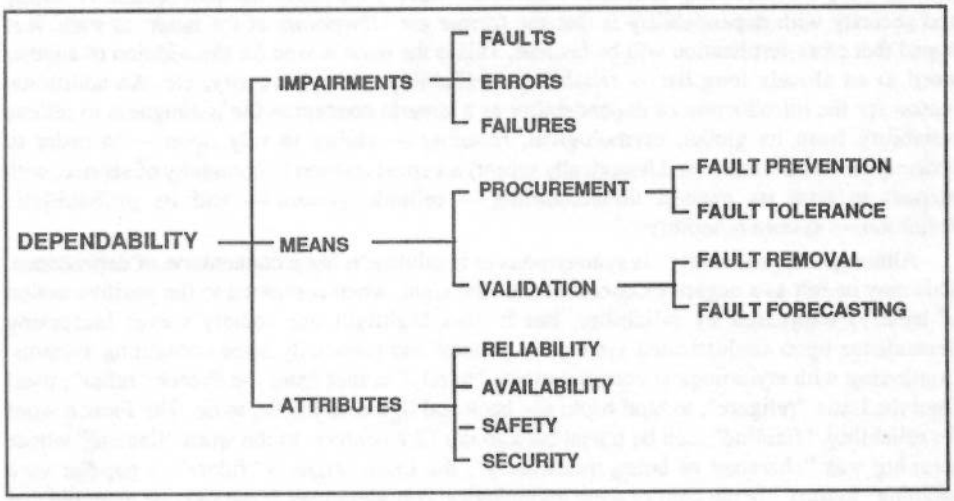

Fig. 1 - The Dependability Tree

4.2. On the Introduction of Dependability as a Generic Concept

A natural tendency of any emerging scientific or technical discipline is, in a first step, to restrict its field of investigation in order to make — rapid — progress in solving the associated problems. Then comes a time when its interactions with other disciplines can no longer be ignored. A great temptation is then to declare those other disciplines as being "special cases" of the considered discipline. This usually results in large debates, often conducted by the adherents of each discipline in their own jargon. This was the case for reliability, safety, and security of computing systems. Initially, the main concern was to have computing systems work: *reliability*. The utilization of computing systems in critical applications brought in the concern for *safety*. The safest system is often the one that does not do anything, which is not very helpful; so, people concerned with safety tend to consider reliability a subset of safety. The advent of distributed systems has exacerbated the *security* issues. Again, a secure system is intended to fulfil functionalities; in addition, security violations can be catastrophic; so, people concerned with security tend to consider safety and reliability as subsets of security.

However, the relations between reliability, safety and security are more complex than a simple dependence. Let us consider the example of the so-called "softbombs", i.e., faults deliberately introduced in a computing system in order to provoke, at a moment chosen by the "terrorist" — and under his/her control — a system failure, of consequences preferably felt by the user as non-catastrophic (until he becomes aware of the failure causes). This example clearly involves reliability, safety, and security, in a very intricate and varying manner

depending upon the viewpoint considered[2]. What is certain is that the user cannot, or should not, place reliance on the service delivered by such a system, which is not *dependable,* in the primary sense of the word.

The preceding discussion clearly shows that it is *not* this chapter's intention to contribute to the controversy concerning whether reliability is a broader concept than safety or vice-versa, and similarly when security is added. What is essential in the relationship of reliability, safety and security with dependability is that the former are *viewpoints* of the latter: as such, it is hoped that cross-fertilization will be favored. This is the main reason for the addition of another word to an already long list — reliability, availability, safety, security, etc. An additional reason for the introduction of dependability as a generic concept is the willingness to relieve reliability from its global, etymological, meaning — ability to rely upon — in order to concentrate on its widely (and historically recent) accepted relation to continuity of service, with respect to both its general understanding — reliable system — and its probabilistic definition — system reliability[3].

Although "dependability" is synonymous to reliability, it has a connotation of dependence. This may be felt as a negative connotation at first sight, when compared to the positive notion of trust as expressed by reliability, but it does highlight our society's ever increasing dependence upon sophisticated systems in general and especially upon computing systems. Continuing with etymological considerations, "to rely" comes from the French "relier", itself from the Latin "religare", to bind back: re-, back and ligare, to fasten, to tie. The French word for reliability, "fiabilité", can be traced back to the 12th century, to the word "fiableté" whose meaning was "character of being trustworthy"; the Latin origin is "fidare", a popular verb meaning "to trust". In the light of these etymological considerations, it can only be regretted that the definition of reliability currently employed in many engineering fields[4] has substituted the notion of "ability" for the notion of "trust", for (at least) the two following reasons:

a) from the viewpoint of the system user, what is of real interest is not so much the *ability* to provide functionalities, as the *service* which is *actually* delivered to the user;

b) from the viewpoint of the system producer who is willing to admit the possible existence of faults in his design, the interpretation of the term "ability" must be questionable, despite the fact that it has been adopted in software engineering glossaries (see, e.g., [IEEE 729]).

4.3. On System Function, Behavior, Structure and Specification

Up to now, a system has been — implicitly — considered as a whole, emphasizing its externally perceived behavior. A definition complying with this "black box" view is: an entity having interacted or interfered, interacting or interfering, or likely to interact or interfere with other entities, i.e., with other systems. These other systems have been, are, or will constitute

---

2 It is also noteworthy that the events reported in the section on "Risk to the Public in Computer Systems" of the ACM Software Engineering Notes relate to reliability, to safety, and to security.

3 It is interesting to note that:
   a) most books having the word "reliability" in their title actually deal with how to evaluate, measure, predict the reliability of systems, not really with how to build reliable systems;
   b) viewing dependability as a more general concept than reliability, availability, etc., and embodying the latter terms, has already been attempted in the past (see, e.g., [Hosford 1960]); this was however attempted with less generality than here, since the goal was to define a measure embodying availability and reliability, and security was not of concern.

4 For example, "Reliability: The ability of an item to perform a required function under given conditions for a given time interval" [IEC 191].

the environment of the considered system[5,6]. A system user is that part of the environment that *interacts* with the considered system: the user provides inputs to and receives outputs from the system, its distinguishing feature being to *use the service* delivered by the system.

The function of a system is what the system *is intended for* [Kuipers 1985]. The behavior of a system is what the system *does*. What *makes it do what it does* is the structure of the system [Ziegler 1976]. Adopting the spirit of [Anderson and Lee 1981], a system, from a structural ("glassbox") viewpoint, is a set of components bound together in order to interact; a component is another system, etc. The recursion stops when a system is considered as being atomic: any further internal structure cannot be discerned, or is not of interest and can be ignored. The term "component" has to be understood in a broad sense: layers[7] of a system as well as intra-layer components; in addition, a component being itself a system, it embodies the interrelation(s) of the components of which it is composed. A more classical definition of system structure is what a system *is*. Such a definition fits in perfectly when representing a system without accounting explicitly for any impairments to dependability, and thus in the case where the structure is considered as *fixed*. We do not want to restrict ourselves to systems whose structure is fixed. In particular, we need to allow for structural changes caused by, or resulting from, dependability impairments. It thus appears that a structure may have states[8,9]. Hence a definition for the notion of state: a condition of being with respect to a set of circumstances, *whether of behavior or of structure*.

From its very definition (the user-perceived behavior), the service delivered by a system is clearly an *abstraction* of its behavior. It is noteworthy that this abstraction is highly dependent on the application that the computer system supports. An example of this dependence is the important role played in this abstraction by time: the time granularities of the system and of its user(s) are generally different, and the difference varies from one application to another one.

The specification of a system may describe the system's expectations in terms of either or both its expected function and its expected service; there is usually not a single specification, but several ones, according to:

- varying degrees of detail: requirement specification, design specification, realization specification, etc.;
- different viewpoints [Anderson and Lee 1981]: functional relationship between inputs and outputs, performance criteria (e.g., limits on response time), attributes of dependability (reliability, availability, safety, security).

Clearly, a system may fail with respect to some of these multiple specifications, and still comply with other ones, leading to the notion of *degraded* mode of operation.

It is essential that a specification be *agreed upon* by two persons or corporate bodies — in fact, legal personnae: the system supplier (in a broad sense of the term: designer, builder,

---

5 Giving recursive definitions is not for recursion's sake. The aim is to emphasise relativity with respect to the adopted viewpoint. So is it for the notion of system: a given system's boundaries may vary depending on whether it is viewed by its designer(s), by its user(s), by its maintenance crew, etc.

6 The passive, present and future forms are employed to stress that a system's environment may vary with time, especially with respect to the phases of its life-cycle. For instance, the notion of "programming environment" fits into the given definition, as well as the physical environment a system is confronted with during operation.

7 In the sense of protocols, i.e., a given layer using the services provided by lower layer(s), including hardware, and delivering services to the upper layer(s).

8 It could therefore be said that a "structure" has also a "behavior", especially with respect to the dependability impairments, even if the considered velocities of evolution with respect a) to the user's request on one hand, and b) to the impairments on the other, are —hopefully— different.

9 The given definition enables other types of systems with varying structures to be embodied, e.g., adaptive —especially knowledge-based— systems.

vendor, etc.) and its human user(s)[10]. The agreement is necessary so that the specification can serve as a basis for adjudicating whether the delivered service is acceptable or not, or, equivalently, whether a failure has occurred or not. What can be judged as an acceptable service with respect to a specification at a given level of detail may not comply with the specification at a less detailed level, because of mistakes occurred when detailing the specification, resulting in fact in *specification faults*. Specification faults may in turn affect any of the various specifications. More generally, a specification cannot be claimed to be immutable once established. This would be simple ignorance of the facts of life, which imply *change*. The changes may be motivated by modifying the system requirements: modification of the expected function and/or service, or correction of some faults[11]. Once more, what is important is that the specification is (again) agreed upon. Finally, it is noteworthy that such matters as environment, exposure time, observability, etc., can — and should — be captured by an appropriately stated specification.

Based on the preceding view of system structure, the notions of function, of service and of their specification apply equally naturally to the components. This is especially interesting in the design process, when off-the-shelf components, either hardware or software are used: what is more of interest to the designer is the function and/or the service they are able to provide, rather than their detailed (internal) behavior.

4.4. The Impairments to Dependability

4.4.1. Faults

Faults and their sources are extremely diverse. They can be classified according to three main viewpoints that are their nature, their origin and their persistence.

The *nature* of faults leads one to distinguish:

- accidental faults, which occur or are created fortuitously;
- intentional faults, which occur or are created deliberately.

The *origin* of faults may itself be decomposed into three viewpoints:

- the *phenomenological causes*, which lead one to distinguish [Avizienis 1978]:
  - physical faults, which are due to adverse physical phenomena,
  - human-made faults, which result from human imperfections;
- the *system boundaries*, which lead one to distinguish:
  - internal faults, which are those parts of the state of a system which, when invoked by the computation activity, will produce an error,
  - external faults, which result from interference to the system from its physical environment (electromagnetic perturbations, radiation, temperature, vibration, etc.), or from interaction with its human environment;
- the *phase of creation* with respect to the system's life, which leads one to distinguish:

---

10 The agreement may be implicit, as when purchasing a system that comes with its specification and user's manual, or when using off-the-shelf systems.

11 We are thus faced with a circular problem: a reference is needed for adjudicating whether a delivered service is acceptable or not, and this reference may be faulty. Improving specifications has long been devoted a significant amount of attention, including proposals for life-cycle models aimed at this objective [Boehm 1988].

- design faults, which result from imperfections arising either a) during the initial design of the system (broadly speaking, from requirement specification to implementation) or during subsequent modifications, or b) during the establishment of the procedures for operating or maintaining the system;
- operational faults, which occur during the system's exploitation.

A distinction can also be made with respect to temporal *persistence* of faults, leading to:

- permanent faults, whose *presence* is not related to pointwise conditions whether they be internal (computation activity) or external (environment),
- temporary faults, whose presence is related to such conditions, and are thus present for a limited amount of time.

Security issues are dominated by — but not restricted to — intentional faults, which are clearly *human-made* faults. Intentional faults can be either internal or external; typical examples are:

- concerning internal faults, the incorporation of malicious logic (e.g., the so-called "Trojan horses"), which is an intentional *design* fault;
- concerning external faults, an intrusion that is an intentional *operational external* fault.

To be "successful", intentional faults may take advantage of accidental faults, e.g., an intrusion exploiting a security breach due to an accidental design fault; there are interesting and obvious similarities between this example and an accidental temporary external fault "exploiting" a lack of shielding.

It could be argued that introducing the *phenomenological causes* in the classification criteria of faults may lead recursively "a long way back", e.g., why do programmers make mistakes? why do integrated circuits fail? The very notion of fault is *arbitrary,* and is in fact a facility provided for stopping the recursion. Hence the definition given: *adjudged or hypothesised* cause of an error. This cause may vary depending upon the chosen viewpoint: fault tolerance mechanisms, maintenance engineers, repair shop, designer, semiconductor physicist, etc. In our view, recursion stops at *the cause that is intended to be avoided or tolerated.* This view provides consistency with the distinction between human-made and physical faults: a computing system is a human artifact and as such any fault in it or affecting it is ultimately human-made since it represents human inability to master all the phenomena that govern the behavior of a system. In an absolute sense, a distinction between physical faults and human-made faults (especially design faults) may be considered unnecessary; however, this distinction is of importance when considering the (current) methods and techniques for procuring and validating dependability. If the recursion mentioned above is not stopped, then *a fault is nothing other than the consequence of a failure of some other system* (including the designer) *that has delivered or is now delivering a service to the given system.*

Examples of the preceding discussion follow:

- a design fault results from a designer failure;
- a physical internal fault is due to a hardware component failure, which is itself the consequence of (an) error(s) at the electrical or electronic level (the "physics reliability" community rarely characterizes failures as "sudden and unpredictable"), in turn originating from physico-chemical disorders, again originating from the hardware production, or from — the limits of — our knowledge of semiconductor physics;
- a physical or human-made external fault is in fact a design fault: the inability to foresee all the situations the system will be faced with during its operational life, or the refusal to consider some of them (e.g., for economic reasons); for instance:

- in the case of an electromagnetic perturbation: is it an external fault or a design fault, i.e., the lack of adequate shielding?
- in the case of a failure caused by an operator typing a single inappropriate character: is it an interaction fault or a design fault, i.e., the lack of confirmation asked by the system [Norman 1983]?

The temporal persistence viewpoint deserves the following comments:

1) Temporary external faults originating from the physical environment are often termed transient faults.
2) Temporary internal faults are often termed intermittent faults; such faults result from the presence of rarely occurring combinations of conditions; examples are a) "pattern sensitive" faults in semiconductor memories, changes in the parameters of a hardware component (effect of temperature variation, delay in timing due to parasitic capacitance, etc.), or b) situations — affecting either hardware or software — occurring when system load goes beyond a certain level, such as marginal timing and synchronization. In fact, the term "fault" in such cases is actually an abstraction for *fault conditions*. The very notion of intermittent faults is in an absolute sense arbitrary: such faults are nothing other than (permanent) faults whose conditions of activation cannot be reproduced or which occur rarely enough; however, as already pointed out for the distinction between physical faults and design faults, their consideration is a useful facility. Permanent faults whose conditions of activation can be reproduced are often termed recurrent faults.

From the above discussions, it appears that *any fault is a permanent design fault*. This is indeed true in an absolute sense, but is not very helpful for the designers and assessors of a system.

Figure 2 summarises the various classes of faults that have been dealt with, with respect to the various viewpoints that have been considered.

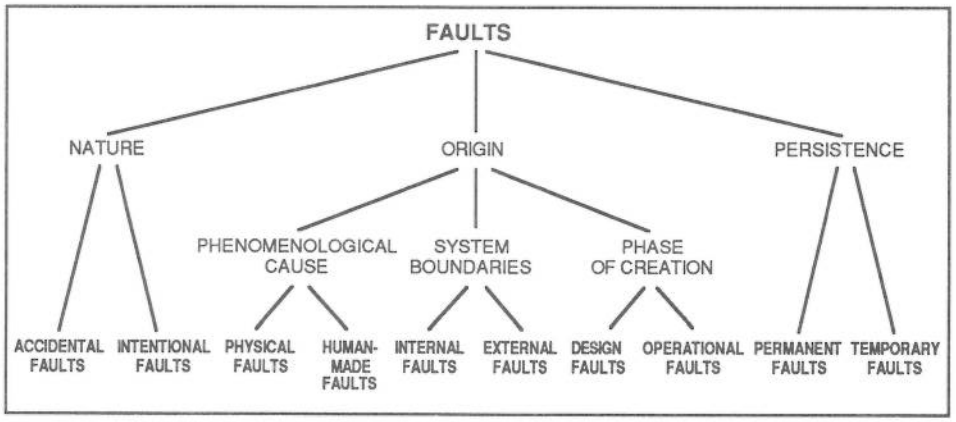

Fig. 2 - The Classes of Faults according to Various Viewpoints

If all the combinations of fault classes according to the 5 viewpoints of figure 2 were possible, there would be 32 different fault classes. In fact, the number of likely combinations is more restricted: 10 combinations are indicated by the rows of table 1, which also gives the usual labelling of these combinations — not their definition.

Table 1 - The Classes of Faults resulting from Combinations according to the Various Viewpoints

Nature		Origin						Persistence		Usual Labelling
		Phenomenological cause		System Boundaries		Phase of creation				
Accidental Faults	Intentional Faults	Physical Faults	Human-made Faults	Internal Faults	External Faults	Design Faults	Operational Faults	Permanent Faults	Temporary Faults	
✓		✓		✓			✓	✓		Physical faults
✓		✓			✓		✓		✓	Transient faults
✓		✓		✓			✓		✓	Intermittent faults
✓			✓	✓		✓			✓	
✓			✓	✓		✓		✓		Design faults
✓			✓		✓		✓		✓	Interaction faults
	✓		✓	✓		✓		✓		Malicious Logic
	✓		✓	✓		✓			✓	
	✓		✓		✓		✓	✓		Intrusions
	✓		✓		✓		✓		✓	

4.4.2. Errors

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

1) The system composition, and especially the nature of the existing redundancy:
   - *intentional* redundancy (introduced to provide fault tolerance) which is explicitly intended to prevent an error from leading to failure,
   - *unintentional* redundancy (it is practically difficult if not impossible to build a system without any form of redundancy[12]) which may have the same — unexpected — result as intentional redundancy.
2) The system activity: an error may be overwritten before creating damage.
3) The definition of a failure from the user's viewpoint: what is a failure for a given user may be a bearable nuisance for another one. Examples are a) accounting for the user's time granularity: an error which "passes through" the system-user interface may or may not be viewed as a failure depending on the user's time granularity, b) the notion of "acceptable error rate" — implicitly before considering that a failure has occurred — in data transmission. This discussion explains why it is often desirable to explicitly mention in the specification such conditions as the maximum outage time (related to the user time granularity).

4.4.3. Failures

A system may not, and generally does not, always fail in the same way. The ways a system can fail are its failure *modes*, which may be characterized according to three viewpoints: domain, perception by the system users, and consequences on the environment.

The *failure domain* viewpoint leads one to distinguish:

- value failures: the value of the delivered service does not comply with the specification;

12 A classical problem in hardware testing is the removal of such "false redundancies", whose effect may be to mask faults, and as such to make the task of test pattern generation more complicated.

- timing failures: the timing of the service delivery does not comply with the specification.

Such general definitions (non compliance with the specification) apply to arbitrary failures. Refined modes of failures can be distinguished. For instance, the notion of timing failure may be refined into *early* timing failure or *late* timing failure, depending on whether the service is delivered too early or too late. A class of failures relating to both value and timing is the stopping failures: system activity, if any, is not any more perceptible to the users, and a constant value service is delivered; the constant value delivered may vary according to the application, and thus the specification, e.g., last correct value, predetermined value, etc. A special case of stopping failures is constituted by omission failures [Cristian et al. 1985, Ezhilchelvan and Shrivastava 1986]: no service is delivered. Such a failure can be seen as a common limiting case for both value failures (null value) and timing failures (infinitely late failure); a *persistent* omission failure is a crash failure. A system whose failures can only be — or more generally are to an acceptable extent — stopping failures, is a fail-stop system[13], and a system whose failures can only be — are to an acceptable extent — crash failures, is a fail-silent system [Powell et al. 1988]; on the opposite, a system whose failures may be arbitrary is a fail-uncontrolled system [Powell et al. 1988].

When a system has several users, the *failure perception* viewpoint leads one to distinguish:

- consistent failures: all system users have the same perception of the failures;
- inconsistent failures: the system users may have different perceptions of a given failure; inconsistent failures are usually termed, after [Lamport et al. 1982], *Byzantine failures*.

The failure severities result from grading the *consequences of the failures* upon the system environment. They thus enable the failure modes to be ordered. A special case of great interest is that of systems whose failure modes can be grouped into two classes whose severities differ considerably:

- benign failures, where the consequences are of the same order of magnitude (generally in terms of cost) as the benefit provided by service delivery in the absence of failure;
- catastrophic failures, where the consequences are incommensurably greater than the benefit provided by service delivery in the absence of failure.

A system whose failures can only be — or more generally are to an acceptable extent — benign failures is a fail-safe system. The notion of failure severity enables the notion of criticality to be defined: the criticality of a system is the highest severity of its (possible) failure modes[14].

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

---

13 The concept of fail-stop processor has been defined in [Schlichting and Schneider 1983] in the context of distributed systems. The definition of fail-stop system we give, when interpreted in the context of distributed systems where information is exchanged by messages, is consistent with the definition of fail-stop processor.

14 As an example, the criticality levels accepted by the aviation community are defined as follows [RTCA 178A]:

- critical: functions for which the occurrence of any failure would prevent the continued safe flight and landing of the aircraft;
- essential: functions for which the occurrence of any failure would reduce the capability of the aircraft or the ability of the crew to cope with adverse operating conditions;
- non essential: functions for which failure could not significantly degrade aircraft capability or crew ability.

4.4.4. Fault Pathology

The creation and manifestation mechanisms of faults, errors, and failures may be summarised as follows:

1) A fault is active when it produces an error. An active fault is either a) an internal fault that was previously dormant and that has been activated by the computation process (including the simultaneous existence of the fault conditions for an intermittent fault), or b) an external fault. An internal fault may cycle between its dormant and active states. Physical faults can directly affect the hardware components only, whereas human-made faults may affect any component.
2) An error may be latent or detected. An error is latent when it has not been recognised as such; an error is detected by a detection algorithm or mechanism. An error may disappear before being detected. An error may, and in general does, propagate; by propagating, an error creates other — new — error(s).
3) A failure occurs when an error "passes through" the system-user interface and affects the service delivered by the system. A component failure results in a fault a) for the system that contains the component, and b) as viewed by the other component(s) with which it interacts; the failure modes of the failed component then become fault types for the components interacting with it.

These mechanisms enable the "fundamental chain" to be completed:

$$\cdots \longrightarrow \text{failure} \longrightarrow \text{fault} \longrightarrow \text{error} \longrightarrow \text{failure} \longrightarrow \text{fault} \longrightarrow \cdots$$

Some illustrative examples of fault pathology:

- the result of a programmer's *error* is 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 erroneous data affect the delivered service (in value and/or in the timing of their delivery), a *failure* occurs;
- a short circuit occurring in an integrated circuit is a *failure* (with respect to the service specification of the circuit); the consequence (connection stuck at a Boolean value, modification of the circuit function, etc.) is a *fault* which will remain dormant as long as it is not activated, the continuation of the process being identical to the previous example;
- an electromagnetic perturbation of sufficient energy is a *fault;* this fault may
  a) directly create an *error*, e.g., by electromagnetic interference with the electrical charges circulating along wires,
  b) create another, (internal) fault; for instance, if the perturbation acts on a memory's inputs in the write position in changing some bit values, these errors will subsequently remain as faults in the memory; the latter faults will remain dormant until the particular memory location(s) are read; the error-failure sequence from the external transient fault to the internal fault still exists, at the electronic level;
- an inappropriate man-machine interaction performed by an operator during the operation of the system is a *fault* (from the system viewpoint); the resulting altered processed data is an *error;* etc.

- a maintenance or operating manual writer's error may result in a fault in the corresponding manual (faulty directives) which will remain dormant as long as the directives are not acted upon in order to deal with a given situation, etc.

From the above examples, it is easily understood that the fault dormancy may vary considerably, depending upon the fault, the given system's utilization, etc.

The man-made faults can be either accidental or intentional. The previous example relating to a programmer's error and its consequences may be rephrased as follows: a logic bomb is created by a malicious programmer; it will remain dormant up to being activated (e.g., at some predetermined date); it then produces an error which 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, a special type of failure.

These examples were deliberately kept simple. Real life is, as usual, much more complicated; four examples:

a) a given fault in a given component may result from 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 design fault — e.g., faulty microinstruction — propagating "top-down" through the layers and causing a short between two circuit outputs for a duration long enough to provoke a short-circuit having the same consequence as the threshold change;

b) a fault of a given class may, through propagation, create a fault of another class; for instance, the fault having led to an error during the execution of the microinstruction in the preceding example could have been a transient fault;

c) some viewpoints may become — temporarily at least — of lesser importance during the propagation process; for instance, when dealing with external faults producing input errors during execution of a software component (thus, invoking it in its so-called exceptional input domain [Cristian 1980]), the fact that the fault is physical or human-made may not be of importance for the failure behavior of the given component;

d) 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 bypass access control) which is inserted into a computer system, either accidentally or intentionally, is a design fault; this fault may remain dormant until some malicious human makes use of it to enter the system; the intruder login is an intentional interaction fault; when the intruder is logged in (while he or she should not be), he or she may deliberately create an error, e.g., in modifying some file (integrity attack); when this file is used by an authorized user, the service will be affected, and a failure will occur.

Two additional comments, relative to the words, or labels, "fault", "error", and "failure":

a) their exclusive use in this paper does not preclude the use in special situations of words which designate, briefly and unambiguously, a specific class of impairment; this is especially applicable to faults (e.g., bug[15], defect, deficiency) and to failures (e.g., breakdown, malfunction, denial-of-service);

---

15 Including specialization of the term "bug", as in [Gray 1986], which distinguishes "Heisenbugs" (for intermittent software faults, from the Heisenberg uncertainty principle) from "Bohrbugs" (for permanent software faults, "like the Bohr atom, solid, easily detected by standard techniques, and hence boring").

b) the assignment made of the particular terms fault, error, failure simply takes into account current usage: i) fault avoidance, tolerance, and diagnosis, ii) error detection and correction, iii) failure rate.

Finally, it has to be stressed that the definitions given in this section are *syntactic*; accordingly, the criteria for the various classifications performed have been emphasised, and are in our view more important than the classes themselves.

4.5. The Means for Dependability

4.5.1. Dependencies between the Means for Dependability

All the "how to's" that appear in the basic definitions given in section 4.1 are in fact goals that cannot be fully reached, as all the corresponding activities are human activities, and thus imperfect. These imperfections bring in *dependencies* that explain why it is only the *combined* utilization of the above methods — preferably at each step of the design and implementation process — that can best lead to a dependable computing system. These dependencies can be sketched as follows: in spite of fault prevention by means of design methodologies and construction rules (imperfect in order to be workable), faults occur. Hence the need for fault removal. Fault removal is itself imperfect, as are the off-the-shelf components — hardware or software — of the system, hence the importance of fault forecasting. Our increasing dependence on computing systems brings in the requirement for fault tolerance, which is in turns based on construction rules; hence fault removal, fault forecasting, etc. It must be noted that the process is even more recursive than it appears from the above: current computer systems are so complex that their design and implementation need computerized tools in order to be cost-effective (in a broad sense, including the capability of succeeding within an acceptable time scale). These tools have themselves to be dependable, and so on.

The preceding reasoning illustrates the close interactions between fault removal and fault forecasting, and motivates their gathering into the single term *validation*. This is despite the fact that validation is often limited to fault removal, and associated with one of the main activities involved in fault removal, verification: e.g., in "V and V" [Boehm 1979]; in such a case the distinction is related to the difference between "building the system right" (related to verification) and "building the right system" (related to validation)[16]. What is proposed here is simply an extension of this concept: the answer to the question "am I building the right system?" (fault removal) being complemented by "for how long will it be right?" (fault forecasting)[17]. In addition, fault removal is usually closely associated with fault prevention, forming together fault avoidance, i.e., how to *aim at* a fault-free system. Besides highlighting the need for validating the procedures and mechanisms of fault tolerance, considering fault removal and fault forecasting as two constituents of the same activity — validation — is of great interest in that it enables a better understanding of the notion of coverage, and thus of an important problem introduced by the above recursion: *the validation of the validation*, or how to reach confidence in the methods and tools used in building confidence in the system. Coverage refers here to a measure of the representativity of the situations to which the system is submitted during its validation compared to the actual situations it will be confronted with during its

---

16 It is noteworthy that these assignments are sometimes reversed, as in the domain of communication protocols (see, e.g., [Rudin 1985]).

17 Validation stems from "validity", which encapsulates two notions:
- validity at a given moment, which relates to fault removal;
- validity for a given duration, which relates to fault forecasting.

operational life[18]. Imperfect coverage strengthens the relation between fault removal and fault forecasting, as it can be considered that the need for fault forecasting stems from imperfect coverage of fault removal.

In the remainder of this section, we examine in turns fault tolerance, fault removal and fault forecasting; fault prevention is not dealt with as it clearly relates to "general" system engineering.

4.5.2. Fault Tolerance

Fault tolerance is carried out by error processing and by fault treatment [Anderson and Lee 1981]. Error processing is aimed at removing errors from the computational state, if possible before failure occurrence; fault treatment is aimed at preventing faults from being activated — again.

Error processing can be carried out in two ways:

- error recovery, where an error-free state is substituted for the erroneous state; this substitution may take on two forms [Anderson and Lee 1981]:
  - backward recovery, where the erroneous state transformation consists of bringing the system back to a state already occupied prior to error occurrence; this involves the establishment of recovery points, which are points in time during the execution of a process for which the then current state may subsequently need to be restored;
  - forward recovery, where the erroneous state transformation consists of finding a new state, from which the system can operate (frequently in a degraded mode);
- error compensation, where the erroneous state contains enough redundancy to enable the delivery of an error-free service from the erroneous (internal) state.

When error recovery is employed, the erroneous state needs to be (urgently) identified as being erroneous prior to being transformed; this is the purpose of error detection, hence the term of *error detection-and-recovery* that is usually employed. The association into a component of its functional processing capability together with error detection mechanisms leads to the notion of self-checking component, either in hardware [Carter and Schneider 1968, Nicolaidis et al. 1989, Wakerly 1978] or in software [Yau and Cheung 1975]; one of the important benefits of the self-checking component approach is the ability to give a clear definition of *error confinement areas* [Siewiorek and Johnson 1982]. When error compensation is performed in a system made up of self-checking components partitioned into classes executing the same tasks (the so-called "active redundancy"), then state transformation is nothing else than switching within a class from a failed component to a non-failed one, hence the corresponding approach to fault tolerance: *error detection-and-compensation*[19]. On the other hand, compensation may be applied systematically, even in the absence of errors, then providing error masking (e.g., in majority vote). However, this can at the same time correspond to an unknown decrease in

---

18 The notion of coverage as defined here is very general; it may be made more precise by indicating its field of application, e.g.:
- coverage of a software test with respect to its 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 design assumption with respect to reality.

19 Error detection and compensation may be seen as a limiting case of error detection-and-recovery, where recovery is performed using the present (erroneous) state of the system instead of substituting an error-free state to the erroneous state.

redundancy. So, practical implementations of masking generally involve error detection, which may then be performed *after* the state transformation.

Backward and forward error recovery are not exclusive: backward recovery may be attempted first; if the error persists, forward recovery may then be attempted. In forward recovery, it is necessary to *assess the damage* caused by the detected error, or by errors propagated before detection; damage assessment can — in principle — be ignored in the case of backward recovery, provided that the mechanisms enabling the transformation of the erroneous state into an error-free state have not been affected [Anderson and Lee 1981].

The operational time overhead necessary for error processing is radically different according to the adopted error processing form:

- in error recovery, the time overhead is longer upon error occurrence than before; especially, in backward recovery it is related to the provision of recovery points, thus in fact to preparing for error processing;
- in error compensation, the time overhead required by compensation is the same, or almost the same, whether errors are present or not.

In addition, the duration of error compensation is much shorter than the duration of error recovery, due to the larger amount of (structural) redundancy.

This remark

a) is of high practical importance in that it often conditions the choice of the adopted fault tolerance strategy with respect to the user time granularity;

b) has introduced a relation between operational time overhead and structural redundancy; more generally, a redundant system always provides redundant behavior, incurring at least some operational time overhead; the time overhead may be small enough not to be perceived by the user, which means only that the *service* is not redundant; an extreme opposite form is "time redundancy" (redundant *behavior* obtained by repetition) which needs to be at least initialized by a structural redundancy, limited but existing; roughly speaking, the more the structural redundancy, the less the time overhead incurred.

The first step in *fault treatment* is fault diagnosis, which consists of determining the cause(s) of error(s), in terms of both location and nature. Then come the actions aimed at fulfilling the main purpose of fault treatment: preventing the fault(s) from being activated again, thus aimed at making it(them) passive, i.e., fault passivation. This is carried out by removing the component(s) identified as being faulty from further executions. If the system is no longer capable of delivering the same service as before, then a *reconfiguration* may take place.

If it is estimated that error processing could directly remove the fault, or if its likelihood of recurring is low enough, then fault passivation need not be undertaken. As long as fault passivation is not undertaken, the fault is regarded as a soft fault; undertaking it implies that the fault is considered as hard, or solid. At first sight, the notions of soft and hard faults may seem to be respectively synonymous to the previously introduced notions of temporary and permanent faults. Indeed, tolerance of temporary faults does not require fault treatment, since error recovery should in this case directly remove the effects of the fault, which has itself vanished, provided that a permanent fault has not been created in the propagation process. In fact, the notions of soft and hard faults are useful due to the following reasons:

- distinguishing a permanent fault from a temporary fault is a difficult and complex task, since a) a temporary fault vanishes after a certain amount of time, usually before fault diagnosis is undertaken, and b) faults from different classes may lead to very similar errors; so, the notion of soft or hard fault in fact incorporates the subjectivity associated with these difficulties, including the fact that a fault may be declared as a soft fault when the fault diagnosis is unsuccessful;

- the ability of those notions to incorporate subtleties of the modes of action of some transient faults; for instance, can it be said that the dormant internal fault resulting from the action of alpha particles (due to the residual ionization of circuit packages), or of heavy ions in space, on memory elements (in the broad sense of the term, including flip-flops) is a *temporary* fault? Such a dormant fault is however a *soft* fault.

The preceding definitions apply to physical faults as well as to design faults: the class(es) of faults that can actually be tolerated depend(s) on the fault hypothesis that is being considered in the design process, and thus relies on the *independence* of redundancies with respect to the process of fault creation and activation. An example is provided by considering tolerance of physical faults and tolerance of design faults. A (widely-used) method to attain fault tolerance is to perform multiple computations through multiple channels. When tolerance of physical faults is foreseen, the channels may be identical, based on the assumption that hardware components fail independently; such an approach is not suitable for the tolerance to design faults where the channels have to provide *identical services* through *separate designs and implementations* [Avizienis 1978, Elmendorf 1972, Randell 1975], i.e., through design diversity [Avizienis and Kelly 1984].

An important aspect in the coordination of the activity of multiple components is that of preventing errors to propagate and to affect the operation of non-failed components. This aspect becomes particularly important when a given component needs to communicate some information to other components that is private to that component. 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 non-failed components must reach an *agreement* as to how the information they obtain should be employed in a mutually consistent way. Specific attention has been devoted to this problem in the field of distributed systems (see, e.g., clock synchronization [Kopetz and Ochsenreiter 1987, Lamport and Melliar-Smith 1985] or membership protocols [Cristian 1988]). It is important to realize, however, that the inevitable presence of structural redundancy in any fault-tolerant system implies distribution at one level or another, and that the agreement problem therefore subsists. Geographically localized fault-tolerant systems may employ solutions to the agreement problem that would be deemed too costly in a "classical" distributed system of components communicating by messages (e.g., inter-stages [Lala 1986], multiple stages for interactive consistency [Frison and Wensley 1982]).

The knowledge of some system properties may limit the necessary amount of redundancy, leading to the so-called "low-cost fault tolerance". Examples of these properties are regularities of a structural nature: error detecting and correcting codes [Peterson and Weldon 1972], robust data structures [Taylor et al. 1980], multiprocessors and computer networks [Pradhan 1986, Rennels 1986], algorithm-based fault tolerance [Huang and Abraham 1982]. The faults that are tolerated are then dependent upon the properties accounted for, as they intervene directly in the fault hypotheses.

Of importance is the signalling of a component failure to its users. This may be accounted for within the framework of *exceptions* [Anderson and Lee 1981, Cristian 1980, Melliar-Smith and Randell 1977]. *Exception handling* facilities provided in some languages may constitute a convenient way for implementing error recovery, especially forward recovery[20].

---

20 The use of the term "exception", due to its origin of coping with exceptional situations —not only errors— is to be carefully used in the framework of fault tolerance: it could appear as contradicting the view that fault tolerance is a natural attribute of computing systems, considered from the very initial design phases, and not an "exceptional" attribute.

Fault tolerance is (also) a recursive concept: it is essential that the mechanisms aimed at implementing fault tolerance be protected against the faults that can affect them. Examples are voter replication, self-checking checkers [Carter and Schneider 1968], "stable" memory for recovery programs and data [Lampson 1981].

Fault tolerance is not restricted to accidental faults. Protection against intrusions traditionally involves cryptography [Denning 1982]. Some mechanisms of error detection are directed towards both intentional and accidental faults (e.g., memory access protection techniques) and schemes have been proposed for the tolerance to both intrusions and physical faults [Fraga and Powell 1985, Rabin 1989], as well as for tolerance to malicious logic [Joseph and Avizienis 1988].

4.5.3. Fault Removal

Fault removal is composed of three steps: verification, diagnosis, correction. Verification is the process of checking whether the system adheres to properties, termed the *verification conditions* [Cheheyl et al. 1981]; 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 process has to be resumed to check that fault removal had no undesired consequences; the verification performed at this stage is usually termed (non-)regression verification. The verification conditions can take two forms:

- general conditions, which apply to a given class of systems, and are therefore — relatively — independent of the specification, e.g., absence of deadlock, conformance to design and realization rules;
- conditions specific to the considered system, directly deduced from its specification.

The verification techniques can be classed according to whether or not they involve exercizing the system. Verifying a system without actual execution is static verification. The verification can be conducted:

- on the system itself, in the form of a) *static analysis* (e.g., inspections or walk-through [Myers 1979], data flow analysis [Osterweil and Fodsick 1976], complexity analysis [McCabe 1976], compiler checks, etc.) or b) *proof-of-correctness* (inductive assertions [Craigen 1987, Hoare 1969]);
- on a model of the system behavior (e.g., Petri nets, finite state automata), leading to *behavior analysis* [Diaz 1982].

Verifying a system through exercizing it constitutes dynamic verification; the inputs supplied to the system can be either symbolic in the case of symbolic execution, or valued in the case of testing.

Testing exhaustively a system with respect to all its possible inputs is generally impractical. The methods for the determination of the test patterns can be classed according to two viewpoints:

- *criteria* for selecting the test inputs, which may relate to either the function or the structure of the system, leading respectively to functional testing and structural testing; in both cases, the criteria may relate to
  - the system's ability to deliver service (e.g., path sensitization [Rapps and Weyuker 1985], input boundary values in software [Ntafos 1988]),
  - revealing specific classes of faults (e.g., stuck-at-faults in hardware production [Roth et al. 1967], physical faults affecting the instruction set of a microprocessor [Thatte and Abraham 1978], design faults in software [DeMillo et al. 1978, Goodenough and Gerhart 1975, Howden 1987]);

- *generation* of the test inputs, which may be deterministic or probabilistic:
  - in deterministic testing, test patterns are predetermined by a selective choice according to the adopted criteria,
  - 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 adopted criteria [David and Thévenod-Fosse 1981, Duran and Ntafos 1984].

Observing the test outputs and deciding whether they satisfy or not the verification conditions is known as the *oracle* problem [Adrion et al. 1982]. 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 [David 1986], or a "partial oracle" when testing for design faults of software [Weyuker 1982]). 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 [Levendel 1986] or from a reference system ("golden unit"). For design 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", see, e.g., [Bishop 1988]).

Some verification methods may be used in conjunction, e.g., symbolic execution may be used a) for facilitating the determination of the testing patterns [Adrion et al. 1982], or b) as a proof-of-correctness method [Carter et al. 1978].

As verification has to be performed all along a system's development, the above techniques apply naturally to the various forms taken by a system during its development: prototype, component, etc. Verifying that the system cannot do *more* than what is specified is especially important with respect to intentional faults [Gasser 1988].

Designing a system in such a way as to facilitate its verification is the design for verifiability. This is especially developed for hardware with respect to physical faults, where the corresponding techniques are then termed *design for testability* [McCluskey 1986, Williams 1983].

Fault removal during the operational phase of a system's life is corrective maintenance, aimed at preserving or improving the system's ability to deliver a service complying with the specification[21]. Corrective maintenance can take two forms:

- curative maintenance, aimed at removing faults which have produced one or more errors and have been reported;
- preventive maintenance, aimed at removing faults before they produce errors; these faults can be
  - physical faults having occurred since the last preventive maintenance actions,
  - design faults having led to errors in other similar systems [Adams 1984].

These definitions[22] apply to non-fault-tolerant systems as well as to fault-tolerant systems, which can be maintainable on-line (without interrupting service delivery) or off-line. It is finally

---

21 The other forms of maintenance usually distinguished are [Ramamoorthy et al. 1984]:
- *adaptive maintenance*, which adjusts the system to environmental changes (e.g., change of operating systems or system data-bases);
- *perfective maintenance*, which improves the system's function by responding to customer — and designer — defined changes, which may involve removal of specification faults.

22 It is noteworthy that current discussions about the irrelevance of the use of the term "maintenance" when applied to software simply forget the etymology of the word: in the Middle Ages, maintenance designated the actions performed in order to keep an army battleworthy, thus including the corrective, adaptive and perfective forms of maintenance. The association of maintenance with repairing hardware is actually a

noteworthy that the frontier between corrective maintenance and fault treatment is relatively arbitrary; especially, curative maintenance may be considered as an — ultimate — means of achieving fault tolerance.

4.5.4. 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:

- *non-probabilistic*, e.g., determining the minimal cutset or pathset of a fault tree, conducting a failure mode and effect analysis;
- *probabilistic*, aimed at determining the conformance of the system to dependability objectives expressed in terms of probabilities associated to some of the attributes of dependability, which may then be defined as *measures* of dependability.

The life of a system is perceived by its user(s) as an alternation between two states of the delivered service with respect to the specification:

- correct service, where the delivered service *complies with* the specification[23];
- incorrect service, where the delivered service *does not comply with* the specification.

A failure is thus a transition from correct to incorrect service, and the transition from incorrect to correct service is a restoration. Quantifying the alternation of correct-incorrect service delivery enables reliability and availability to be defined as measures of dependability:

- reliability: a measure of the *continuous* delivery of correct service — or, equivalently, of the *time to* failure;
- availability: a measure of the delivery of correct service *with respect to the alternation* of correct and incorrect service.

A third measure, maintainability, is usually considered, which may be defined as a measure of the time to restoration from the last experienced failure, or equivalently, of the continuous delivery of incorrect service.

As a measure, safety can be seen as an extension of reliability. Let us group the state of correct service together with the state of incorrect service subsequent to benign failures into a safe state (in the sense of being free from catastrophic damage, not from danger); safety is then a measure of continuous "safeness", or equivalently, of the time to catastrophic failure. Safety can thus be considered as reliability with respect to the catastrophic failures. A direct extension of availability, i.e., a measure of safeness with respect to the alternation of safeness and incorrect service after catastrophic failure, would not provide a significant measure. When a catastrophic failure has occurred, the consequences are generally so important that service restoration is not of prime importance for — at least — the two following reasons:

- it comes second to repairing (in the broad sense of the term, including legal aspects) the consequences of the catastrophe;
- the lengthy period before being allowed to operate the system again (commissions of enquiry, etc.) would lead to insignificant numerical values.

A "hybrid" reliability-availability-type measure can however be defined: a measure of correct service delivery with respect to the alternation of correct service and incorrect service

---

(recent) deviation; associating "to maintain" with the notion of *service* would enable this etymological meaning to be revived, while at the same time removing the very source of discussion.

23 We deliberately restrict the use of "correct" to the service delivered by a system, and do not use it for the system itself: in our view, non-faulty systems do not exist, there are only systems that may have not yet failed.

after benign failure. This measure is of interest in that it provides indeed a quantification of the system availability *before* occurrence of a catastrophic failure, and as such enables quantification of the so-called "reliability- (or availability-) safety tradeoff".

In the case of multi-performing systems, several modes of service delivery can be distinguished, ranging from full capacity to complete disruption, which can be seen as distinguishing less and less correct service deliveries. Performance-related measures of dependability for such systems are usually termed performability [Meyer 1978, Smith et al. 1988].

When performing a probabilistic evaluation, the approaches differ significantly according to whether the system is considered as being in stable reliability or in reliability growth, which may be defined as follows [Laprie et al. 1990]:

- stable reliability: the system's ability to deliver correct service is *preserved* (stochastic identity of the successive times to failure);
- reliability growth: the system's ability to deliver correct service is *improved* (stochastic increase of the successive times to failure)[24];

Practical interpretations of stable reliability and of reliability growth are as follows:

- stable reliability: at a given restoration, the system is identical to what it was at the previous restoration; this corresponds to the following situations:
  - in the case of a hardware failure, the failed part is changed for another one, identical and non-failed,
  - in the case of software failure, the system is restarted with an input pattern different from the one having led to failure;
- reliability growth: the fault whose activation has led to failure is diagnosed as a design fault (in software or in hardware) and is removed.

Evaluation of the dependability of systems in stable reliability is usually composed of two main phases:

- *construction* of the model of the system from the elementary stochastic processes which model the behavior of the components of the system and their interactions;
- *processing* the model in order to obtain the expressions and the values of the dependability measures of the system.

Evaluation can be conducted with respect to a) physical faults [Trivedi 1984], b) design faults [Arlat et al. 1988, Littlewood 1979], or c) a combination of both [Laprie 1984, Pignal 1988]. The dependability of a system is highly dependent on its environment, either in the broad sense of the term [Hecht and Fiorentino 1987], or more specifically its load [Castillo and Siewiorek 1981, Iyer et al. 1982]. When evaluating fault-tolerant systems, the coverage of error processing and fault treatment mechanisms has a very significant influence [Arnold 1973, Bouricius et al. 1969]; its evaluation can be performed either through modeling [Dugan and Trivedi 1989] or fault-injection [Arlat et al. 1989].

Many models of reliability growth have been proposed, devoted to hardware [Duane 1964], software, or both [Laprie et al. 1990]. Most of them are devoted to software, and they are aimed at evaluating either the reliability [Miller 1986a, Yamada and Osaki 1985], or the number of (remaining) faults [Goel and Okumoto 1979, Tohma et al. 1989]; as these models

---

24 *Reliability decrease* (the system's ability to deliver correct service is degraded, and there is thus a stochastic decrease of the successive times to failure) is theoretically, and practically, possible, e.g., upon introduction of new faults during corrective actions, whose probability of activation is greater than for the removed fault(s). In such a case, it has to be hoped that the decrease is limited in time, and that reliability is globally growing over a long observation period of time.

are aimed at predicting the future reliability from the failure data accumulated in the past, particular attention has been devoted to the prediction problem [Littlewood 1988].

4.6. The Attributes of Dependability

The attributes have been defined in §4.1 according to different properties, which may be more or less emphasised depending on the application intended for the computer system under consideration. This may be refined as follows:

- readiness for usage is always required as a property, although to a varying degree depending on the application;
- continuity of service, avoidance of catastrophic consequences on the environment, preservation of confidentiality may or may not be required according to the application.

An additional property, which may be viewed as a prerequisite for the other properties to be fulfilled, is integrity, i.e., the condition of being unimpaired, in the broad sense of the term: a) for either data or programs, and b) with respect to either accidental or intentional faults[25].

The variations in the emphasis to be put on the attributes of dependability have a direct influence on the appropriate balance of the techniques addressed in the previous section to be employed so that the resulting system be dependable. This problem is all the more difficult since some of the attributes are antagonistic (e.g., availability and safety, availability and security), necessitating tradeoffs to be performed. Considering the three main design dimensions of a computer system, i.e., cost, performance and dependability, the problem is still exacerbated by the fact that the dependability dimension is less understood than the cost-performance design space [Siewiorek and Johnson 1982].

The *assessment* of whether a system is truly dependable — justified reliance on the delivered service — or not thus goes beyond the validation techniques as they have been addressed in the previous section for, at least, the three following reasons:

- checking with certainty 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. which are by far superior to what is generally achievable;
- performing an evaluation of a system according to some attributes of dependability with respect to some classes of faults is currently considered as non feasible or yielding non-significant results: probability-theoretic bases do not exist or are not widely accepted; examples are safety with respect to accidental design faults[26], security with respect to intentional faults;
- the specifications "against" which validation is performed are not generally non-faulty — as any system.

The consequence is an emphasis put on the development and production process when assessing a system: methods and techniques utilized and how they are employed; in some

---

25 This definition of integrity incorporates the notion of fault secureness as defined in the theory of self-checking circuits [Anderson and Metze 1972; Wakerly 1978].

26 The following sentence is excerpted from [RTCA 178A]: "During the preparation of this document, techniques for estimating the post-verification probabilities of software errors were examined. The objective was to develop numerical requirements for such probabilities for digital computer-based equipment and systems certification. The conclusion reached, however, was that currently available methods do not yield results in which confidence can be placed to the level required for this purpose."

cases, a *grade* is assigned and delivered to the system according to a) the nature of the latter and to b) an assessment of their utilization[27].

27 For instance:
- systems are ranked from the security viewpoint [DoD 5200.28] from A1 ("verified design") to D ("minimal protection");
- software for civil transportation airplanes is classed as Level 1, 2 or 3 according to the criticality of the function to be accomplished by the software: critical, essential, non-essential.

GLOSSARY

Warning: this glossary is provided as an aid for reading the chapter. *Do not* consider it independently of the chapter.

Accidental fault.....Fault occurring or created fortuitously.

Active fault.....Fault producing an error.

Arbitrary failure.....see Failure

Atomic system..... System whose internal structure cannot be discerned, or is not of interest and can be ignored.

Attributes of dependability. Attributes enabling the system quality resulting from the impairments and the means opposing to them to be assessed. Reliability, availability, maintainability, safety, security.

Availability.....Readiness for usage.
Measure of correct service delivery with respect to the alternation of correct and incorrect service.

Avoidance (fault ~).....Methods and techniques aimed at producing a fault-free system.
Fault prevention and fault removal.

Backward recovery..... Form of error recovery where the erroneous state transformation consists of bringing the system back to previously occupied state.

Behavior (system ~).....What a system does.

Benign failure.....Failure whose penalties are of the same order of magnitude as the benefit provided by correct service delivery.

Catastrophic failure.....Failure whose consequences are incommensurably greater than the benefit provided by correct service delivery.

Compensation (error ~).....Form of error processing when erroneous state contains enough information to enable correct service delivery.

Component (system ~).....Another system.

Consistent failure.....Failure perceived similarly by all system users.

Corrective maintenance.... Preservation or improvement during its operational life of a system's ability to deliver a service complying with the specification.
Fault removal during the operational life of a system.

Coverage.....Measure of the representativity of the situations to which a system is submitted during its validation compared to the actual situations it will be confronted with during its operational life.

Crash failure.....Persistent omission failure

Criticality (system ~).....Highest severity of failure modes.

Curative maintenance..... Corrective maintenance aimed at removing faults that have produced errors and which have been reported.

Dependability.....trustworthiness of the delivered service such that reliance can justifiably be placed on this service.

Design diversity..... An approach to the production of systems, involving the provision of identical services from separate designs and implementations.

Design fault.....Human-made internal fault.

Design for verifiability.....Methods and techniques when designing a system that facilitate its verification.

Detection (error ~).....The action of identifying that a system state is erroneous.

Detected error.....Error recognised as such by a detection algorithm or mechanism.

Deterministic testing.....Form of testing where the test patterns are predetermined by a selective choice.

Diagnosis (fault ~).....The action of determining the cause of an error in location and nature.

Dormant fault.....Internal fault not activated by the computation process.

Dynamic verification.....Verification involving exercizing the system.

Environment (system ~).....The other systems interacting or interfering with the given system.

Error..... Part of system state that is liable to lead to failure.
Manifestation of a fault in a system.

External fault.....Fault resulting from environmental interference.

Fail-safe system..... System whose failures can only be, or are to an acceptable extent, benign failures.

Fail-silent system..... System whose failures can only be, or are to an acceptable extent, crash failures.

Fail-stop system.....System whose failures can only be, or are to an acceptable extent, stopping failures.

Fail-uncontrolled system....System whose failures may be arbitrary.

Failure..... Deviation of the delivered service from compliance with the system specification.
Transition from correct service delivery to incorrect service delivery.

Fault.....Adjudged or hypothesised cause of an error.
Error cause that is intended to be avoided or tolerated.
Consequence for a system of the failure of another system that has interacted or is interacting with the considered system.

Forecasting (fault ~).....Methods and techniques aimed at estimating the present number, the future incidence, and the consequences of faults.

Forward recovery.....Form of error recovery where the erroneous state transformation consists of finding a new state.

Function (system ~).....What a system is intended for.

Functional testing.....Form of testing where the testing inputs are selected according to criteria relating to the system's function.

Hard fault or **solid fault**.....Fault requiring passivation.

Human-made fault.....Consequence of human imperfection.

Impairments to dependability Undesired, but not unexpected, circumstances causing or resulting from un-dependability.
Faults, errors, and failures.

Incorrect service..... Service delivered not in compliance with the system specification.

Inconsistent failure.....Failure such that system users may have different perceptions of it.

Integrity.....Condition of being unimpaired.
Intentional fault.....Fault occurring or created deliberately.
Intermittent fault.....Temporary internal fault.
.....Faults whose conditions of activation cannot be reproduced or which occur rarely enough.
Internal fault.....Fault inside a system.
Intrusion.....Intentional operational external fault.
Latent error.....Error not recognised as such.
Maintainability..... Measure of continuous incorrect service delivery.
Measure of the time to restoration from the last experienced failure.
Malicious logic.....Intentional design fault.
Masking (fault ~).....The result of applying error compensation systematically, even in the absence of error.
Means for dependability.....Methods and techniques enabling a) to provide a system with the ability to deliver a service on which reliance can be placed, and b) to reach confidence in this ability.
Fault prevention, fault tolerance, fault removal, fault forecasting.
Measures of dependability..Attributes enabling the service quality resulting from the impairments and the means opposing to them to be appraised.
Reliability, availability, maintainability, safety.
Omission failure.....Failure such that no service is delivered.
Operational fault.....Faults that occur during the system's exploitation.
Passivation (fault ~).....The actions taken in order that a fault cannot be activated.
Performability.....Performance-related measure of dependability.
Permanent fault.....Fault whose presence is not related to pointwise conditions of the system, either internal or external.
Physical fault.....Fault resulting from adverse physical phenomena.
Preventive maintenance.... Corrective maintenance aimed at removing faults before they are activated.
Prevention (fault ~).....Methods and techniques aimed at preventing fault occurrence or introduction.
Processing (error ~).....The actions taken in order to eliminate errors from a system.
Procurement of dependability Methods and techniques intended to provide a system with the ability to deliver a service complying with the system specification.
Fault prevention and fault tolerance.
Correct service..... Service delivered in compliance with the system specification.
Recovery (error ~).....Form of error processing where an error-free state is substituted for an erroneous state.
Recovery point..... Point in time during the execution of a process for which the then current state may subsequently need to be restored.
Recurrent fault..... Permanent fault whose conditions of activation can be reproduced.
Regression verification.....Verification performed after a correction, in order to check that the correction has no undesired consequences.

Reliability..... Dependability with respect to the continuity of service.
Measure of continuous correct service delivery.
Measure of the time to failure.

Reliability growth.....The system's ability to deliver correct service is improved (stochastic increase of the successive times to failure).

Removal (fault ~).....Methods and techniques aimed at reducing the presence (number, seriousness) of faults.

Restoration (service ~).....Transition from incorrect to correct service delivery.

Random testing.....See Statistical testing

Safety..... Dependability with respect to the non occurrence of catastrophic failures.
Measure of continuous delivery of either correct service or incorrect service after benign failure.
Measure of the time to catastrophic failure.

Security..... Dependability with respect to the preservation of confidentiality and integrity.

Self-checking component...Component comprising error detection mechanisms associated with its functional part.

Service..... System behavior as perceived by the system user.

Severity (failure ~).....Grade of the failure consequences upon the system environment.

Soft fault.....Fault for which fault passivation is not undertaken.

Specification (system ~)....Agreed description of the system's requirements.

State (system ~).....A condition of being with respect to a set of circumstances.

Stable reliability.....The system's ability to deliver correct service is preserved (stochastic identity of the successive times to failure).

Static verification.....Verification conducted without exercizing the system.

Statistical testing.....Form of testing where the test patterns are selected according to a defined probability distribution on the input domain.

Structure (system ~).....What makes a system do what it does.

Structural testing.....Form of testing where the testing inputs are selected according to criteria relating to the system's structure.

Symbolic execution.....Dynamic verification performed with symbolic inputs.

System..... Entity having interacted, interacting, or able to interact with other entities.
Set of components bound together in order to interact.

Temporary fault.....Fault that is present for a limited amount of time.

Testing.....Dynamic verification performed with valued inputs.

Timing failure.....Failure such that the timing of service delivery does not comply with the specification.

Tolerance (fault ~).....Methods and techniques aimed at providing a service complying with the specification in spite of faults.

Transient fault.....Temporary physical external fault.

Treatment (fault ~).....The actions taken in order to prevent a fault from being re-activated.

Un-dependability .....Property of a computing system such that reliance cannot, or will not, any more be justifiably placed on the service it delivers.

User (system ~).....Another system (physical, human) interacting with the considered system.

Validation..... Methods and techniques intended to enable confidence to be reached in a system's ability to deliver a service complying with the system specification.
Fault removal and fault forecasting.

Value failure.....Failure such that the value of the delivered service does not comply with the specification.

Verification.....The process of determining whether a system adheres to properties (the verification conditions) which can be a) general, independent of the specification, or b) specific, deduced from the specification.