DEPENDABLE COMPUTING AND FAULT TOLERANCE :

CONCEPTS AND TERMINOLOGY

Jean-Claude Laprie*

LAAS-CNRS

7, Avenue du Colonel Roche, 31400 Toulouse, France

ABSTRACT

This paper provides a conceptual framework for expressing the attributes of what constitutes dependable and reliable computing:

- the impairments to dependability: faults, errors, and failures,

- the means for dependability: fault-avoidance, fault-tolerance, error-removal, and error-forecasting,

- the measures of dependability: reliability, availability, maintainability, and safety.

Emphasis is being put on the dependability impairments and on fault-tolerance.

FOREWORD

This paper 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 [Avi 78, Ran 78, Car 79, Lap 79, And 81, Sie 82, Cri 84] in order to propose clear and widely acceptable definitions for some basic concepts.

The work presented here has been conducted within the framework of the subcommittee "Fundamental Concepts and Terminology", which is common to the IFIP WG 10.4 "Reliable Computing and Fault-Tolerance" and to the IEEE Computer Society TC "Fault-Tolerant Computing".

This paper is an edited version of [Lap 84]. Going backward in time, a milestone was the special meeting devoted to fundamental concepts held in conjunction with FTCS-11. A special session was organized at FTCS-12, devoted to the presentation of viewpoints elaborated in several places and institutions [And 82, Avi 82, Kop 82, Lap 82, Lee 82, Rob 82]. The previous versions of this paper were then discussed during the 1983 and 1984 Winter and Summer meetings of the IFIP WG 10.4.

The paper proceeds by refinements: dependability is first introduced as a global concept. Fault-tolerance is then detailed.

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

- search for the minimum number of concepts enabling the dependability attributeds 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 shall retain its ordinary sense (as given by any dictionary),

- emphasis on integration (as opposed to specialization) [Gol 82].

In each section, concise definitions are given first, then they are heavily commented in order to (attempt to) show the wide applicability of the adopted presentation.

Boldface characters are used when a term is defined, italic characters being an invitation to focus the reader's attention.

THE DEPENDABILITY CONCEPT

BASIC DEFINITIONS AND ASSOCIATED TERMINOLOGY

Computer system **dependability** is the *quality of the delivered service* such that *reliance* can *justifiably* be placed on this service [1].

The **service** delivered by a system is the system behavior *as it is perceived* by another special system(s) interacting with the considered system: its user(s).

A system **failure** occurs when the delivered service deviates from the specified service, where the service **specification** is an agreed description of the expected service. The failure occurred because the system was **erroneous**: an **error** is that part of the system state which is liable to lead to failure, i.e. to the delivery of a service not complying with the specified service. The cause -- in its phenomenological sense -- of an error is a **fault**.
Upon occurrence, a fault creates a **latent** error, which becomes **effective** when it is activated; when the error affects the delivered service, a failure occurs. Stated in other terms, an error is the manifestation *in the system* of a fault, and a failure is the manifestation *on the service* of an error.

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

- **fault-avoidance:** how to prevent, by *construction*, fault occurrence,

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

- **error-removal:** how to minimize, by *verification*, the presence of latent errors,

- **error-forecasting:** how to estimate, by *evaluation*, the presence, the creation and the consequences of errors.

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

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

- **service accomplishment,** where the service is delivered as specified,

- **service interruption** where the delivered service is different from the specified service.

The events which constitute the transitions between these two states are the **failure** and the **restoration**. Quantifying this accomplishment-interruption alternation leads to the two main measures of dependability:

- **reliability:** a measure of the continuous service accomplishment (or, equivalently, of the time to failure) from a reference initial instant,

- **availability:** a measure of the service accomplishment *with respect to the alternation* of accomplishment and interruption.

COMMENTS

1. On the introduction of dependability as a generic concept

Why should another term be added to an already long list ? reliability, availability, safety, etc. The reasons are basically two-fold:

- to remedy the existing confusion between reliability in its general meaning (reliable system) and reliability as a mathematical quality (system reliability),

- to show that reliability, maintainability, availability, safety, etc. are quantitative measures corresponding to distinct perceptions of the same attribute of a system: its dependability.

In regard to the term "dependability", it is noteworthy that from an etymological point of view, the term"reliability" would be more appropriate: ability to rely upon. Although dependability is synonymous with reliability, it brings in the notion of dependence at a second level. This may be felt as a negative connotation at first sight, when compared to the positive notion of trust as expressed by reliability, but it does highlight our society's ever increasing dependence upon sophisticated systems in general and especially upon computing systems. Moreover, "to rely" comes from the French "relier", itself from the Latin "religare", to bind back: re-, back and ligare, to fasten, to tie. The necessary solidarity for reaching reliability is present! The French word for reliability, "fiabilite" traces back to the 12th century, to the word "fiablete" whose meaning was

---

[1] This definition is adapted from [Car 82], where "trustworthiness and continuity" has been replaced by "quality".

"character of being trustworthy"; the Latin origin is "fidare", a popular verb meaning "to trust".

Finally, it is interesting that viewing dependability as a more general concept than reliability, availability, etc., and embodying the latter terms, has already been attempted in the past (see e.g. [Hos 60]), although with less generality than here, since the goal was then to define a measure.

2. On the notion of service and its specification

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

Concerning specification, what is essential within the present context is that it is a description of the expected service which is *agreed* upon by two persons or corporate bodies: the system supplier (in a broad sense of the term: designer, builder, vendor, etc.) and its (human) user(s). What precedes does not mean that a service specification will not change once established. This would be simple ignorance of the facts of life, which imply *change*. The changes may be motivated by modifying the service expectation: modification of functionality, correction of some undesired features such as deficiencies in the agreed specification. Once more, what is important is that the specification is (re-) agreed upon.
It is noteworthy that such matters as performance, observability, readiness, etc. can be captured by an appropriately stated specification.

3. On the notions of fault, error and failure

First, some illustrative examples:

- a programmer's mistake is a *fault:* the consequence is a *(latent) error* in the written software (erroneous instruction or piece of data); upon *activation* (activation of the module where the error resides *and* an appropriate input pattern activating the erroneous instruction, instruction sequence or piece of data) the error becomes *effective;* when this effective error produces erroneous data (in value or in the timing of their delivery) which affect the delivered service, a *failure* occurs,

- a short-circuit occurring in an integrated circuit is a *fault;* the consequence (connection stuck at a Boolean value, modification of the circuit function, etc.) is an *error* which will remain latent as long as it is not activated, the continuation of

the process being identical to the previous example,

- an electromagnetic pertubation of sufficient energy is a *fault;* when (for instance) acting on a memory's inputs, it will create an *error* if active when the memory is in the write position; the error will remain latent until the erroneous memory location(s) is (are) read, etc.

- an inappropriate man-machine interaction performed (inadvertently or deliberately) by an operator during the operation of the system is a *fault;* the resulting altered data is an *error,* etc.

- a maintenance or operating manual writer's mistake is a *fault;* the consequence is an *error* in the corresponding manual (erroneous directives) which will remain latent as long as the directives are not acted upon in order to face a given situation, etc.

From the above examples, it is easily understood that the error latency duration may vary considerably, depending upon the fault, the considered system utilization, etc. These examples also explain why an error was defined as *being liable* to lead to a failure. Whether or not an error will effectively lead to a failure depends on several factors:

- the activation conditions according to which a latent error will become effective,

- the system composition, and especially the amount of available redundancy:

  • explicit redundancy (in order to ensure fault-tolerance) which is directly intended to prevent an error from leading to a failure,

  • implicit redundancy (it is in fact difficult to build a system without any form of redundancy) which may have the same (unexpected) result as explicit redundancy,

- the very definition of a failure from the user's viewpoint, e.g. in the notion of "acceptable error rate" (implicitly: before considering that a failure has occurred) in data transmission.

These examples finally enable the introduction of the notion of fault *classes,* which are classically [Avi 78] physical faults and human-made faults. Proposed definitions are as follows:

- **physical faults:** adverse physical phenomena, either internal (physico-chemical disorders: threshold changes, short-circuits, open-circuits...) or external (environmental perturbations: electro-magnetic perturbations, temperature, vibration, ...),

- **human-made faults:** imperfections which may be:

- **design faults**, committed either a) during the system initial design (broadly speaking, from requirement specification to implementation) or during subsequent modifications, or b) during the establishment of operating or maintenance procedures,

- **interaction faults**: inadvertent or deliberate violations of operating or maintenance procedures.

Faults, errors and failures are all *undesired circumstances*. At first sight, the three notions are necessary: a) the occurence of an undersired circumstance affecting the service -- failure -- is felt by the user(s) and assessed, b) an (internal) system undesired circumstance -- error -- is detected, c) the undesired circumstance able to give rise to a system error -- fault -- and later on to a service failure is either avoided or tolerated. Assignment of the terms fault, error and failure to the phenomenological cause, the system and the service undesired circumstances simply takes into account current usage: fault-avoidance or -tolerance, error detection, failure rate.
Moreover, it seems important to differentiate between the cause with respect to activation and propagation -- error causing failure --, and the cause with respect to the (suspected) originating phenomenon(a) -- fault causing error -- .

It could be argued that with such a reasoning (fault viewed as the phenomenological cause of error) one can go "a long way back". For instance, if we look back at two of the previously given examples:

- why did the programmer make a mistake?

- why did the short occur in the integrated circuit?
In fact, *recursion stops at the cause which is intended to be avoided or tolerated.* If fault - *avoidance* is meaningful within this context (when a cause is avoided, its effects are of little interest), it may not be so for fault - *tolerance;* in fact, it is the cause which is tolerated, through processing its effects: a fault is thus the *adjudged cause* of an error [Mor 83].
Furthermore, such a view is consistent with the distinction between human and physical faults in that a computing system is a human creation and as such any fault is ultimately human-made since it represents human inability to master the complexity of the phenomena which govern the behavior of a system. In an absolute way, distinguishing between physical and human-made faults (especially design faults affecting the system) may be considered as unnecessary; however it is of importance when considering (current) methods and techniques for procuring and validating dependability. If the above-mentioned recursion is not stopped, *a fault is nothing else than a failure of a system having interacted or interacting with the considered system;* examples follow:

- a design fault is identifiable as a designer failure,

- an internal physical fault is due to a latent error (the "physics reliability" community rarely characterizes failures as "sudden, nonpredictable and irreversible") originating from the hardware production,

- physical external faults and (human-made) interaction faults are identifiable as failures due to another design fault: the inability to foresee all the situations the system will be faced with during its operational life.

Up to now, a system has been considered as a whole, emphasizing its externally perceived behavior; a definition of a **system** complying with this "black box" view is: an entity having interacted, interacting, or liable to interact with other entities, thus other systems. The **behavior** is them simply what the system *does* [Zie 76]. *What enables it to do what it does* is the **structure** of the system or its **organization**. Adopting the spirit of [And 81], a system, from a structural ("white box") viewpoint, is a set of components bound together in order to interact; a component is another system, etc. The recursion stops when a system is considered as being atomic: any further internal structure cannot be discerned, or is not of interest and can be ignored. The term "component" has to be understood in a broad sense: layers of a computing system as well as intra-layer components; in addition, a component being itself a system, it embodies the interrelation(s) of the components of which it is composed.
From these definitions, the discussion of whether "failure" applies to a system or to a component is simply irrelevant, since a component is itself a system. When atomic systems are dealt with, the classical notion of "elementary" failure comes naturally. It is also noteworthy that from the preceding view of system structure, the notions of service and specification apply equally naturally *to the components*. This is especially interesting in the design process, when using off-the-shelf components, either hardware of software [Hor 84]: what is of actual interest is the service they are able to provide, not their detailed (internal) behavior.

This structured view enables *fault pathology* to be made more precise; the creation and action mechanisms of faults, errors and failures may be summarized as follows:

1) A *fault* creates one or several latent errors in the component where it occurs; physical faults can directly affect the physical layer components only, whereas human-made faults may affect any component.

2) The properties governing *errors* may be stated as follows:

   a) a latent error becomes effective once it is ac-

tivated,

b) an error may cycle between its latent and effective states,

c) an effective error may, and in general does, propagate from one component to another; by propagating, an error creates other (new) errors.

From these properties it may be deduced that an effective error within a component may originate from:

- activation of a latent error within the same component,

- an effective error propagating within the same component or from another component.

3) A component *failure* occurs when an error affects the service delivered (as a response to request(s)) by the component.

4) These properties apply to any component of a system.

In the preceding, the intransitive form of "propagate" was intentionnally used: an error does not propagate itself, it just propagates. Although "propagate" was retained due to its wide use, better words would probably be "spread", or "breed".

Three final comments:

i) A given error in a given component may be subsequent to different faults. For instance: an error in a physical component (e.g. stuck at ground voltage) may result from:

- a physical fault (e.g. threshold change) acting at the physical layer comprising the component,

- an information error (e.g. erroneous microinstruction), caused by a design fault (e.g. programmer mistake), propagating top-down through the layers and leading to a short between two circuit outputs for a duration long enough to provoke a short-circuit having the same effect as the threshold change.

ii) It is noteworthy that the notion of failure cannot be separated from time granularity: an error which "passes through" the interface between the system and its user(s) may or may not be viewed as a failure by the system user(s) depending on the time granularity of the latter; this remark is of practical importance when considering the solutions currently adopted for fault-tolerance as a function of the application.

iii) The adjective "deliberate" in the definition of human-made interaction faults is clearly intended to include "undesired accesses" in the sense of computer security and privacy; however, the corresponding methods and techniques will not be addressed in the sequel.

4. On fault-avoidance and -tolerance, error-removal and forecasting

All the "how to's" which appear in the basic definitions are in fact goals which cannot be fully reached, as all the corresponding activities are human activities, and thus imperfect. These imperfections bring in dependencies which explain why it is only the *combined* utilization of the above methods (preferably at each step of the design and implementation process) which can lead to a dependable computing system. These dependencies can be sketched as follows: in spite of construction rules (imperfect in order to be workable), faults occur; hence the need for error-removal; error-removal is itself imperfect, as are the off-the-shelf components of the system, hence the need for error-forecasting; our increasing dependence on computing systems brings in fault-tolerance, which in turn necessitates construction rules, and thus error-removal, error-forecasting, etc. It has to 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 acceptable delays). These tools have themselves to be dependable, and so on.

The preceding reasoning explains why in the given definitions error-removal and error-forecasting are gathered into validation. Classically speaking (see e.g. [And 82, Avi 78, Lap 79]), fault-avoidance and error-removal are conceptually gathered into fault-prevention, error-forecasting being left with no definite (conceptual) room. Validation is then limited to what has been termed as verification; in that case these two terms are often associated, e.g. "V and V" [Boe 79], the distinction being related to the difference between "building the system right" (related to verification) and "building the right system" (related to validation). What is proposed is simply an extension of this concept: the answer to the question "am I building the right system?" being complemented by "for how long will it be right?". Besides highlighting the need for validating the procedures and mechanisms of fault-tolerance, considering error-removal and error-forecasting as two constituents of the same activity -- validation -- is moreover of great interest as it enables a better understanding of the notion of *coverage*, and thus of an important problem introduced by the above recursion(s): *the validation of the validation*, or how to reach confidence in the methods and tools used in building confidence in the system. **Coverage** refers here to a measure of the representativity of the situations to which the system is

submitted during its validation with respect to the actual situations it will be confronted with during its operational life. Finally, "validation" stems from "validity", which encapsulates two notions:

- validity *at a given moment*, which relates to error-removal,

- validity *for a given duration*, which relates to error-forecasting.

5. On the dependability measures

The term "probability" has intentionally not been employed in the given definitions, so as to keep the discussion informal, and to reinforce the physical significance of the defined measures. However, as the considered circumstances are non-deterministic, random variables are associated with them, and the measures which are dealt with are probabilities; this is strictly speaking correct: a probability can be defined mathematically as a measure.

Only two basic measures have been considered, reliability and availability, whereas a third one, **maintainability** is usually considered, which may be defined as a measure of the continuous service interruption, or equivalently, of the time to restoration. This measure is no less important than those previously defined; it was not introduced earlier because it may, at least conceptually, be deduced from the other two. It is noteworthy that availability embodies the failure frequency and the accomplishment time duration at each alternation accomplishment-interruption.

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

- **benign** failures, where the consequences are of the same order of magnitude (generally in terms of cost) as those of the service delivered in the absence of failure,

- **malign** or castrophic failures, where the consequences are not commensurable with those of the service delivered in the absence of failure.

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

- it comes second to repairing (in the broad sense of the term, including legal aspects) the consequences of the catastrophe,

- the lengthy period prior to being allowed to operate the system again (investigation commissions) would lead to meaningless numerical values.
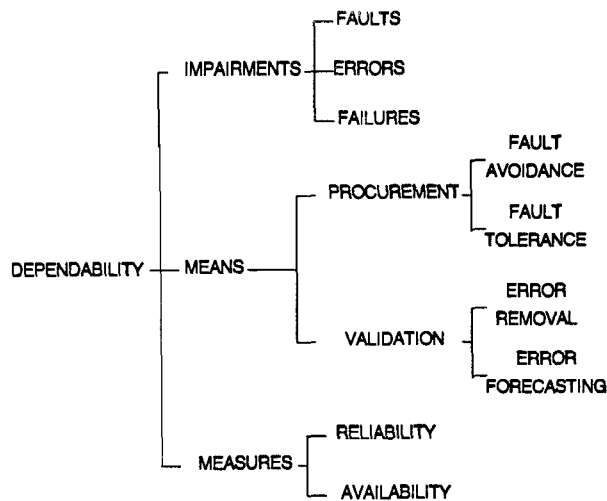
However, a "hybrid" reliability-availability measure can be defined: a measure of the service accomplishment with respect to the alternation of accomplishment and interruption after benign failure. This measure is of interest in that it provides a quantification of the system availability *before* occurrence of a catastrophic failure, and as such enables quantification of the so-called "reliability- (or availability-) safety trade-off".

SUMMARY

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

- the dependability **impairments,** which are undesired (not unexpected) circumstances causing or resulting from un-dependability, whose definition is very simply derived from the definition of dependability: reliance cannot, or will not, be any more justifiably placed on the service,

- the dependability **means,** which are the methods, tools and solutions enabling a) to provide with the ability to deliver a service on which reliance can be placed, and b) to reach confidence in this ability,

- the dependability **measures,** which enable the service quality resulting from the impairments and the means opposing them to be appraised.

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

```
                      ┌─ FAULTS
         ┌─ IMPAIRMENTS ─┼─ ERRORS
         │            └─ FAILURES
         │                              FAULT
         │                         ┌─ AVOIDANCE
         │            ┌─ PROCUREMENT ─┤
         │            │            └─ FAULT
         │            │              TOLERANCE
DEPENDABILITY ─┼─ MEANS ─┤
         │            │              ERROR
         │            │         ┌─ REMOVAL
         │            └─ VALIDATION ─┤
         │                         └─ ERROR
         │                           FORECASTING
         │            ┌─ RELIABILITY
         └─ MEASURES ─┤
                      └─ AVAILABILITY
```

FAULT-TOLERANCE

DEFINITIONS

Fault-tolerance is carried out by **error process-ing**, which may be automatic or operator-assisted; two constituent phases can be identified:

a) **effective error processing**, aimed at bringing the effective error back to a latent state, if possible before occurrence of a failure,

b) **latent error processing**, aimed at ensuring that the latent error does not become effective again.

Effective error processing may take on two forms:

- **error recovery**, where an error-free state is sub-stituted for the erroneous state; this substitution may in turn make on two forms [And 81]:

  • **backward error recovery**, where the er-roneous state transformation consists of bringing the system back to an already oc-cupied state prior to the error becoming effective,

  • **forward error recovery**, where the errone-ous state transformation consists of find-ing a new state (which never occurred be-fore, or which has not occurred since the same erroneous state occurred),

- **error compensation**, where the erroneous state contains enough redundancy to enable the

delivery of an error-free service from the er-roneous (internal) state.

When error recovery is employed, the erroneous state needs to be (urgently) identified as being erroneous pri-or to being transformed, which is the purpose of **error detection**. On the other hand, compensation may be applied systematically, even in the absence of effective error(s), providing **error masking** (to the user(s)).

Latent error processing is carried out by making the error passive and reconfiguring the system if it is no longer capable of delivering the same service as before.

If it is estimated that effective error processing could directly remove the error, or more generally that its likelihood of recurring is low enough, then latent er-ror processing is not undertaken. As long as latent er-ror processing is not undertaken, the error is regarded as being **volatile**; undertaking it implies that the error is considered as **solid**.

Error processing is generally completed by **maintenance** (except of course for non-maintained sys-tems), aimed at *removing* latent errors. Maintenance actions can be put into two classes:

- **corrective maintenance**, aimed at removing those latent errors which have become effective and have been processed,

- **preventive maintenance**, aimed at removing la-tent errors before they become effective in the considered system; these errors can result from:

  • physical faults having occurred since the last preventive maintenance actions,

  • design faults having led to effective errors in other similar systems.

COMMENTS

1. On the tolerated fault classes

The preceding definitions apply to physical faults as well as to design faults: the class(es) of faults which can actually be tolerated depend(s) on the fault hy-pothesis which is being considered in the design process, and thus depend on the *independency* of redundancies with respect to the process of fault creation and activa-tion.

It is noteworthy that fault-tolerance is (also) a re-cursive concept; it is essential that the mechanisms aimed at implementing fault-tolerance be protected against the faults which can affect them: voter replica-tion, self-checking checkers [Car 68], "stable" memory for recovery programs and data [Lam 81].

8

2. On error processing

The goal of error processing is the preservation of data integrity, which in turn requires the data contained in the components involved in this preservation to be consistent [Wen 78, Gra 78].

In effective error processing,

a) backward and forward error recovery are not exclusive: backward recovery may be attempted first; if the effective error persists, forward recovery may then be attempted,

b) in forward error 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 [And 81].

Some previous definitions of compensation consider it within the context of interaction faults in distributed systems only (see e.g. [Ran 78]); the given definition clearly embodies more classical situations such as error-correcting codes and majority voting. It is hoped that the essential idea has been captured, its applicability being a matter of definition of the system boundaries.

In error masking, the systematic application of compensation ensures in itself that any effective error (provided of course it corresponds to the fault hypothesis of the design process) has been brought back to a latent state. However, this can at the same time correspond to a redundancy decrease which is not known. So, practical implementations of masking generally involve error detection, which enables compensation to be applied again (in case an effective error has been detected) in order to check whether latent error processing has to be undertaken or not. It is noteworthy that due to the masking effect, this second application of compensation can be performed within an acceptable delay, off-line with respect to the current computation which was in progress when the error became effective.

Of importance is the signalling of a component failure to its users. This is often accounted for within the framework of *exceptions* [And 81, Cri 82]. From a strict terminology viewpoint, this naming may be regretted in the sense that it may induce a contradiction with the wish of seeing fault-tolerance as a natural feature of computing systems, and as such introduced from the very beginning of the design process (not as an "exceptional" attribute).

The definition given for the fact that an error may be volatile or solid may lead to the feeling that this is a subjective notion. Generally, latent error processing does not immediately follow effective error processing, the delay resulting basically from the fact that the estimation of whether an error is volatile or solid is a complex task involving identification of the fault class which gave rise to the error. As different fault classes can lead to very similar errors, this identification generally involves either waiting for a given lapse of time or logging a given number of error occurrences. An additional, correlated, reason is that latent error processing is in fact a voluntary system change, which, as such, will lead to lowering the system redundancy. Thus, the above-mentioned subjectivity in fact relates to the difficulty in accurately identifying the class of fault having caused an error, as well as diagnosing its effects. Moreover, the given definition is consistent with

a) the difficulty in distinguishing the effects of a temporary (transient or intermittent) physical fault from those of a design fault (see e.g. [Avi 82]),

b) the fact that the techniques for tolerating design faults [Elm 72, Ran 75, Che 78] may be seen as attempts to make design-induced errors volatile.

The knowledge of some system properties may limit the necessary amount of redundancy. Examples of these properties are regularities of structural nature: error detecting and correcting codes, robust data structures [Tay 80], multiprocessors and local area networks [Hay 76]. The faults which are tolerated are then dependent upon the properties which are accounted for, since they intervene directly in the fault hypotheses.

The operational time overhead necessary for effective error processing is radically different according to the two distinguished forms:

- in error detection and recovery, the time overhead is longer when an error becomes effective than when it was latent (it is then related to the provision of recovery points, thus in fact to preparing for effective error processing),

- in error masking, the time overhead is always the same, and in practice the duration of error compensation is much shorter than error recovery, due to the larger amount of available (structural) redundancy.

This remark

a) is of high practical importance in that it often conditions the choice of the adopted fault-tolerance method 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 still existing; roughly speaking, the more the structural redundancy, the less the time overhead incurred.

3. On maintenance

The frontier between latent error processing and corrective maintenance is relatively arbitrary; especially, corrective maintenance may be considered as an (ultimate) means of achieving fault-tolerance. However, the given definitions were adopted for the ability to embody (a) on-line or off-line maintainable fault-tolerant systems, as well as non-fault-tolerant systems, and (b) preventive as well as corrective maintenance.

It is noteworthy within the present context that the current discussions about the irrelevance of the use of the term "maintenance" when applied to software simply forget the etymology of the word: the association of maintenance with repairing hardware is actually a (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.

CONCLUSION

The contents of this paper are devoid of any "Tablets of Stone" pretension: the efforts undertaken are only worthwhile in so far as they manage to embody as wide as possible a range of concepts and therefore those efforts have to keep abreast of technology. Naturally, 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.

The independence of the basic definitions with respect to any fault class should facilitate the bringing together of activities which are often considered as separate, such as:

- VLSI testing and software testing,

- hardware reliability (with respect to physical faults) and software reliability (with respect to design faults), to say nothing of hardware relia-

bility with respect to design faults,

- tolerance to physical faults and tolerance to design faults,

- computer system security, safety and reliability.

ACKNOWLEDGEMENTS

REFERENCES

And 81 T. Anderson, P. A. Lee, *Fault tolerance: principles and practice*, Englewood Cliffs, New Jersey: Prentice Hall, 1981.

And 82 T. Anderson, P. A. Lee, "Fault-tolerance terminology proposals", *proc. 12th Int. Symp. on Fault-tolerant Computing*, Los Angeles, June 1982, pp. 29-33.

Avi 78 A. Avizienis, "Fault-tolerance, the survival attribute of digital systems", *Proceedings of the IEEE*, Vol. 66, No. 10, Oct. 1978, pp. 1109-1125.

Avi 82 A. Avizienis, "The four-universe information system model for the study of fault-tolerance", *proc. 12th Int. Symp. on Fault-Tolerant Computing*, Los Angeles, June 1982, pp. 6-13.

Boe 79 B. W. Boehm, "Guidelines for verifying and validating software requirements and design specifications", *proc. EURO IFIP 79*, London, Sep. 1979, pp. 711-719.

Car 68 W. C. Carter, P. R. Schneider, "Design of dynamically checked computers", *proc. IFIP '68 Cong.*, Amsterdam, 1968, pp. 878-883.

Car 79 W. C. Carter, "Fault detection and recovery algorithms for fault-tolerant systems", *proc. EUROIFIP 79*, London, Sept. 1979, pp. 725-734.

Car 82 W. C. Carter, "A time for reflection", *proc. 12th Int. Symp.*

on *Fault-Tolerant Computing*, Los Angeles, June 1982, p. 41.

Che 78 L. Chen, A. Avizienis, "N-version programming: a fault-tolerance approach to reliability of software operation", *proc. 8th Int. Symp. on Fault-Tolerant Computing*, Toulouse, France, June 1982, p. 3-9.

Cri 82 F. Cristian, "Exception handling and software fault-tolerance", *IEEE Trans. on Computers*, Vol. C-31, No. 6, June 1982, pp. 531-540.

Cri 84 F. Cristian, "A rigorous approach to fault-tolerant programming", *proc. 4th Jerusalem Conference on Information Technology*, Jerusalem, May 1984, pp. 192-201.

Elm 72 W. R. Elmendorf, "Fault-tolerant programming", *proc. 2th Int. Symp. on Fault-Tolerant Computing*, Newton, Massachusetts, June 1972, p. 79-83.

Gol 82 J. Goldberg, "A time for integration", *proc. 12th Int. Symp. on Fault-Tolerant Computing*, Los Angeles, June 1982, pp. 82.

Gra 78 J. N. Gray, "Notes on data base operating systems", in *operating systems*, Lecture Notes in Computer Science 105, Berlin; Springer-Verlag, 1978, pp. 394-479.

Hay 76 J. P. Hayes, "A graph model for fault-tolerant computing systems", *IEEE Trans. on Computers*, Vol. C-25, No. 9, Sept. 1976, pp. 875-884.

Hor 84 E. Horowitz, J. B. Munson, "An expansive view of reusable software", *IEEE Trans. on Software Engineering*, Vol. SE.-10, No. 5, Sept. 1984, pp. 477-487.

Hos 60 J.E. Hosford, "Measures of dependability", *Operations Research*, Vol. 8, No. 1, 196, pp. 204-206.

Kop 82 H. Kopetz, "The failure-fault (FF) model", *proc. 12th Int. Symp. on Fault-Tolerant Computing*, Los Angeles, June 1982, pp. 14-17.

Lam 81 B. W. Lampson, "Atomic Transactions", in *Distributed Systems-Architecture and Implementation*, Lecture Notes in Computer Science 105, Berlin: Springer-Verlag, 1981, Chap. 11.

Lap 79 J. C. Laprie, A. Costes, R. Troy, "Dependability: requirements and solutions", *proc. SEE Cong. on Electrical and Electronical System Dependability*, Toulouse, France, Oct. 1979; in French.

Lap 82 J. C. Laprie, A. Costes, "Dependability: a unifying concept for reliable computing", *proc. 12th Int. Symp. on Fault-Tolerant Computing*, Los Angeles, June 1982, pp. 18-21.

Lap 84 J. C. Laprie, "Dependable Computing and fault-tolerance: concepts and terminology", IFIP WG 104. Summer 1984 meeting, Kissimmee, Florida; LAAS Research Report No. 84.035, June 1984.

Lee 82 P. A. Lee, D. E. Morgan, Eds., "Fundamental concepts of fault-tolerant computing, progress report", *proc. 12th Int. Symp. on Fault-Tolerant Computing*, Los Angeles, June

1982, pp. 34-38.

Mor 83 D. Morgan, W. C. Carter, A. Hopkins, "Report to IFIP WG 10.4 - Concepts and terminology, Draft 1", IFIP WG 10.4 Summer 83 meeting, Como, Italy, June 1983.

Ran 75 B. Randell, "System structure for software fault-tolerance", *IEEE Trans. on Software Engineering*, Vol. SE-1, No. 2, June 1975, pp. 220-232.

Ran 78 B. Randell, P. A. Lee, P. C. Treleaven, "Reliability issues in computing system design", *Computing Surveys*, Vol. 10, No. 2, June 1978, pp. 123-165.

Rob 82 A. S. Robinson, "A user oriented perspective of fault tolerant systems models and terminologies", *proc. 12th Int. Symp. on Fault-Tolerant Computing*, Los Angeles, June 1982, pp. 22-28.

Sie 82 D. P. Siewiorek, R. S. Swarz, *The theory and practice of reliable system design*, Digital Press, 1982.

Tay 80 D. J. Taylor, D. E. Morgan, J. P. Black, "Redundancy in data structures: improving software fault-tolerance", *IEEE Trans. on Software Engineering*, Vol. SE-6, No. 6, Nov. 1980, pp. 383-594.

Wen 78 J. M. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, C.B. Weinstock, "SIFT: the design and analysis of a fault-tolerant computer for aircraft control", *Proceedings of the IEEE*, Vol. 66, No. 10, Oct. 78, pp. 1255-1268.

Zie 76 B.P. Ziegler, *Theory of Modeling and Simulation*, New York: John Wiley, 1976.