Software reliability and system reliability

Jean-Claude Laprie, Karama Kanoun

Chapter

2

Software Reliability and System Reliability

Jean-Claude Laprie and Karama Kanoun
LAAS-CNRS, Toulouse, France

2.1 Introduction

This chapter is mainly aimed at showing that, by using deliberately simple mathematics, *the classical reliability theory can be extended in order to be interpreted from both hardware and software viewpoints.* This is referred to as *X-ware* [Lapr89, Lapr92b] throughout this chapter. It will be shown that, even though the action mechanisms of the various classes of faults may be different from a physical viewpoint according to their causes, a single formulation can be used from the reliability modeling and statistical estimation viewpoints. A single formulation has several advantages, both theoretical and practical, such as (1) easier and more consistent modeling of hardware-software systems and of hardware–software interactions, (2) adaptability of models for hardware dependability to software systems and vice versa, and (3) mathematical tractability.

Section 2.2 gives a general overview of the dependability concepts. Section 2.3 is devoted to the failure behavior of an X-ware system, disregarding the effect of restoration actions (the quantities of interest are thus the time to the next failure or the associated failure rate), considering in turn atomic systems and systems made up of components. In Sec. 2.4, we deal with the behavior of an X-ware system with service restoration, focusing on the characterization of the sequence of the times to failure (i.e., the failure process); the measures of interest are thus the failure intensity, reliability, and availability. Section 2.5 outlines the state of art in dependability evaluation and specification. Finally, Sec. 2.6 summarizes the results obtained.

2.2 The Dependability Concept

2.2.1 Basic definitions

The basic definitions for dependability impairments, means, and attributes are given in Fig. 2.1, and the main characteristics of dependability are summarized in the form of a tree as shown in Fig. 2.2 [Lapr92a, Lapr93].

2.2.2 On the impairments to dependability

Of primary importance are the impairments to dependability, as we have to know what we are faced with. The creation and manifestation mechanisms of *faults, errors,* and *failures* may be summarized as follows:

1. A fault is *active* when it produces an error. An active fault is either an internal fault previously *dormant* and activated by the computation process or an external fault. Most internal faults cycle between their 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 recognized 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). During operation, the presence of active faults is determined only by the detection of errors.
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 (1) for the system which contains the component and (2) 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.

Some examples illustrative 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.* The consequence (connection stuck at a boolean value, modification of the

Dependability is defined as the trustworthiness of a computer system such that *reliance can justifiably be placed on the service it delivers.* The service delivered by a system is its behavior *as it is perceptible* by its user(s); a user is another system (human or physical) *interacting* with the former.

Depending on the application(s) intended for the system, a different emphasis may be put on the various facets of dependability, that is, dependability may be viewed according to different, but complementary, *properties,* which enable the *attributes* of dependability to be defined:

- The *readiness for usage* leads to **availability.**
- The *continuity of service* leads to **reliability.**
- The *nonoccurrence of catastrophic consequences on the environment* leads to **safety.**
- The *nonoccurrence of the unauthorized disclosure of information* leads to **confidentiality.**
- The *nonoccurrence of improper alterations of information* leads to **integrity.**
- The *ability to undergo repairs and evolutions* leads to **maintainability.**

Associating availability and integrity with respect to authorized actions, together with confidentiality, leads to **security.**

A system **failure** occurs when the delivered service deviates from fulfilling the system's **function,** the latter being what the system *is intended for.* An **error** is that part of the system state which is *liable to lead to subsequent failure:* an error affecting the service is an indication that a failure occurs or has occurred. The *adjudged or hypothesized cause* of an error is a **fault.**

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

- **Fault prevention:** how to prevent fault occurrence or introduction.
- **Fault removal:** how to reduce the presence (number, seriousness) of faults.
- **Fault tolerance:** how to ensure a service capable of fulfilling the system's function in the presence of faults.
- **Fault forecasting:** how to estimate the present number, future incidence, and consequences of faults.

The notions introduced can be grouped into three classes:

- The **impairments** to dependability: faults, errors, failures; they are undesired—but not in principle unexpected—circumstances causing or resulting from undependability (whose definition is very simply derived from the definition of dependability: reliance cannot or will no longer be placed on the service).
- The **means** for dependability: fault prevention, fault removal, fault tolerance, fault forecasting; these are the methods and techniques enabling one (1) to provide the ability to deliver a service on which reliance can be placed and (2) to reach confidence in this ability.
- The **attributes** of dependability: availability, reliability, safety, confidentiality, integrity, maintainability; these attributes (1) enable the properties which are expected from the system to be expressed and (2) allow the system quality resulting from the impairments and the means opposing them to be assessed.

Figure 2.1 Dependability basic definitions.

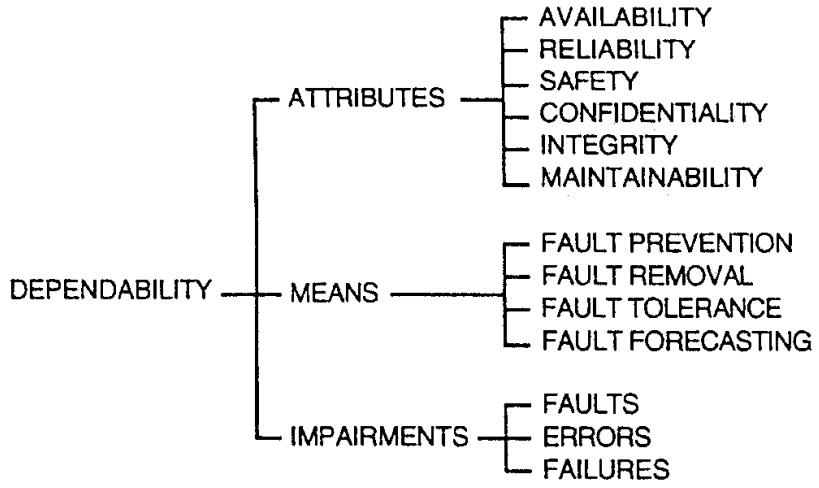


Figure 2.2 The dependability tree.

circuit function, etc.) is a *fault* which will remain dormant as long as it has not been activated, the continuation of the process being identical to that of the previous example.

- An inappropriate human–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.*
- 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.

Figure 2.3 summarizes the fault classification; the upper part indicates the viewpoint according to which they are classified, and the lower part gives the likely combinations according to these viewpoints, as well as the usual labeling of these combinations—*not their definition.*

It is noteworthy that the very notion of fault is *arbitrary,* and in fact a facility provided for stopping the recursion induced by the causal relationship between faults, errors, and failures—hence the definition given: *adjudged or hypothesized* cause of an error. This cause may vary depending upon the viewpoint chosen: fault tolerance mechanisms, maintenance engineers, repair shop, developer, semiconductor physicist, etc. In fact, *a fault is nothing other than the consequence of a failure of some other system* (including the developer) *that has delivered or is now delivering a service to the given system.* A computing system is a human artifact and, as such, any fault in it or affecting it is ultimately human-made since it represents the human inability to master all the phenomena which govern the behavior of a system. Going further, *any fault can be viewed as a permanent design fault.* This is indeed true in

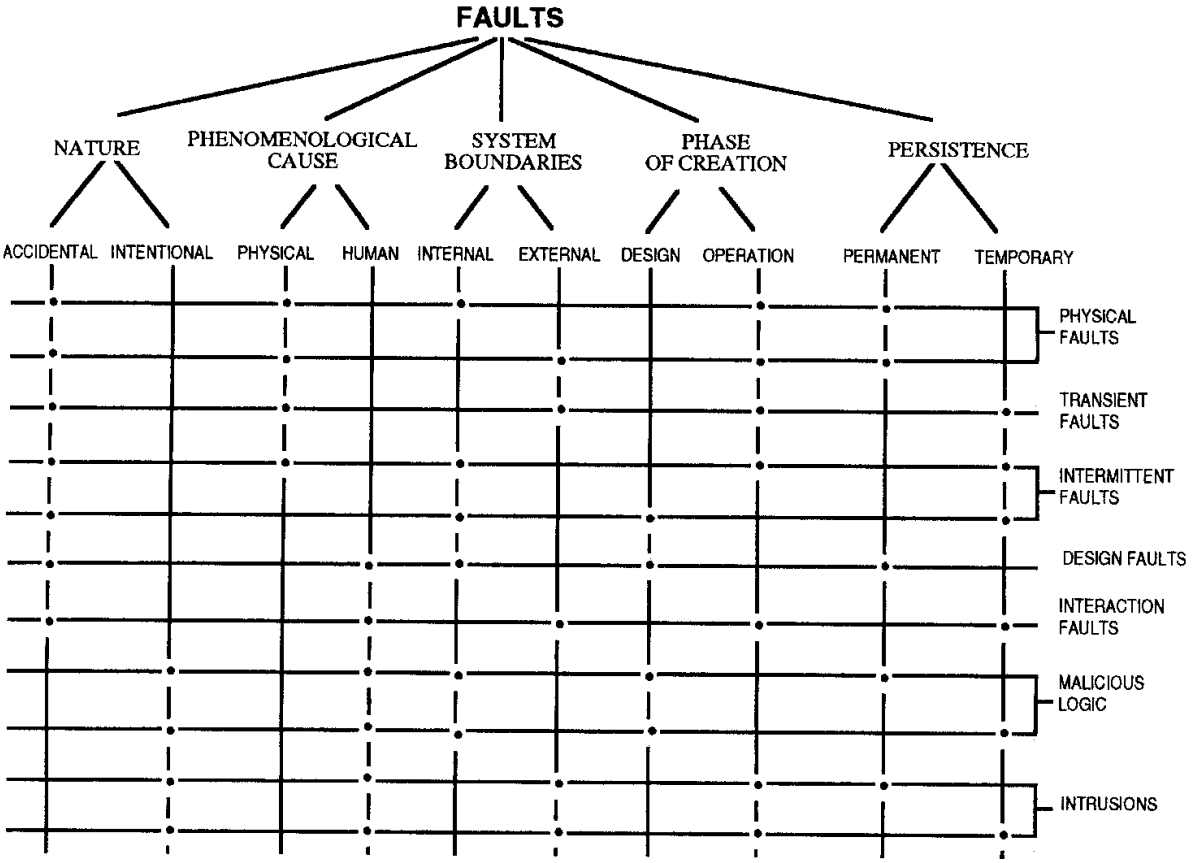


Figure 2.3 Fault classes.

an absolute sense, but not very helpful for system developers and assessors; hence the usefulness of the various fault classes when considering the (current) methods and techniques for procuring and validating dependability.

A system may not, and generally does not, always fail in the same way. The ways a system can fail are its *failure modes.* These can be characterized according to three viewpoints as shown in Fig. 2.4.

Given below are two additional comments regarding the words, or labels, *fault, error,* and *failure:*

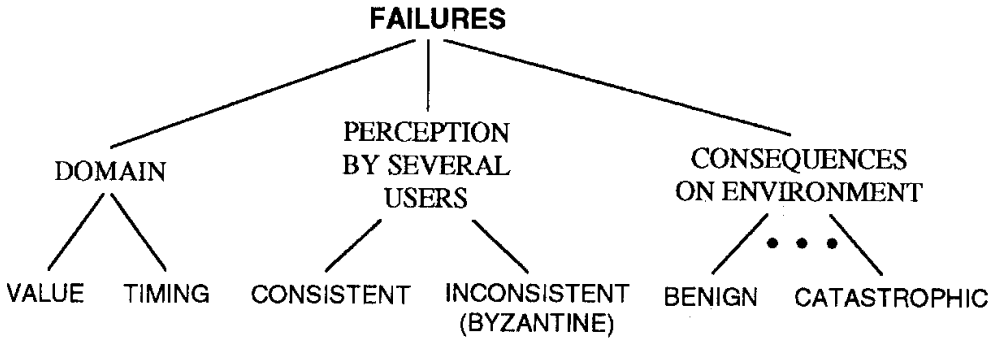


Figure 2.4 Failure classification.

1. Their exclusive use in this book (except Chap. 9) 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, defect, deficiency, flaw) and to failures (e.g., breakdown, malfunction, denial of service).
2. The assignment of the particular terms *fault, error,* and *failure* simply takes into account current usage: (1) fault prevention, tolerance, and diagnosis, (2) error detection and correction, and (3) failure rate.

2.2.3 On the attributes of dependability

The definition given for *integrity*—the avoidance of improper alterations of information—generalizes the usual definitions (e.g., prevention of unauthorized amendment or deletion of information [EEC91] or ensuring approved alteration of data [Jaco91]) which are directly related to a specific class of faults, that is, intentional faults (deliberately malevolent actions). Our definition encompasses accidental faults as well (i.e., faults appearing or created fortuitously), and the use of the word *information* is intended to avoid being limited strictly to data: integrity of programs is also an essential concern; regarding accidental faults, error recovery is indeed aimed at restoring the system's integrity.

Confidentiality, not security, has been introduced as a basic attribute of dependability. Security is usually defined (see e.g., [EEC91]) as the combination of confidentiality, integrity, and availability, where the three notions are understood with respect to unauthorized actions. A definition of security encompassing the three aspects of [EEC91] is: the prevention of unauthorized access and/or handling of information; security issues are indeed dominated by intentional faults, but not restricted to them: an accidental (e.g., physical) fault can cause an unexpected leakage of information.

The definition given for *maintainability*—ability to undergo repairs and evolutions—deliberately goes beyond *corrective* maintenance, which relates to repairability only. Evolvability clearly relates to the two other forms of maintenance, that is, (1) *adaptive* maintenance, which adjusts the system to environmental changes and (2) *perfective* maintenance, which improves the system's function by responding to customer- and designer-defined changes. The frontier between repairability and evolvability is not always clear, however (for instance, if the requested change is aimed at fixing a specification fault [Ghez91]). Maintainability actually *conditions* dependability when considering the whole operational life of a system: systems which do not undergo adaptive or perfective maintenance are likely to be exceptions.

The properties allowing the dependability attributes to be defined may be emphasized to a greater or lesser extent depending on the application intended for the computer system concerned. For instance, availability is always required (although to a varying degree, depending on the application), whereas reliability, safety, and confidentiality may or may not be required according to the application. The variations in the emphasis to be put on the attributes of dependability have a direct influence on the appropriate balance of the means to be employed for the resulting system to be dependable. This problem is all the more difficult to address because certain attributes are antagonistic (e.g., availability and safety, availability and confidentiality), and call for tradeoffs. Given the three main design dimensions of a computer system (i.e., cost, performance and dependability), the problem is further exacerbated by the fact that the dependability dimension is not so well mastered as the cost-performance design space [Siew92].

2.2.4 On the means for dependability

All the how-tos which appear in the basic definitions given in Fig. 2.1 are in fact goals which cannot be fully reached, as all the corresponding activities are carried out by humans and therefore are prone to imperfections. These imperfections bring in *dependencies* which explain why it is only the *combined* utilization of the above methods—preferably at each step in the design and implementation process—that can best lead to a dependable computing system. These dependencies can be sketched as follows: despite fault prevention by means of design methodologies and construction rules (imperfect so as to be workable), faults are created—hence the need for fault removal. Fault removal itself is imperfect, just like all of the off-the-shelf system components—hardware or software—hence the importance of fault forecasting. Our increasing dependence on computing systems brings in the requirement for fault tolerance, which in turn is 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). In turn, these tools themselves have to be dependable.

The preceding reasoning illustrates the close interactions between fault removal and fault forecasting and supports 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" [Boeh79]). 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.* 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 the additional question "how long will it be right?" (fault forecasting). In addition, fault removal is usually closely associated with fault prevention, together forming *fault avoidance,* that is, 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, as it leads to a better understanding of the notion of coverage, and thus of the 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. Here *coverage* refers 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 operational life.† Imperfect coverage strengthens the relation between fault removal and fault forecasting, as it can be considered that the need for fault forecasting stems from an imperfect coverage of fault removal.

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 fulfills the system function‡
- *Incorrect service,* where the delivered service does not fulfill the system function

A failure is thus a transition from a correct to an incorrect service, while the transition from an incorrect service to a correct one is a *restoration.* Quantifying the correct-incorrect service alternation

---

* It is noteworthy that these assignments are sometimes reversed, as in the field of communication protocols (see, for example, [Rudi85]).

† 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

‡ 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 opinion, nonfaulty systems hardly ever exist, there are only systems which have not yet failed.

enables reliability and availability to be defined as *measures* of dependability:

- *Reliability.* A measure of the *continuous* delivery of the 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.

As a measure, safety can be viewed as an extension of reliability. Let us group together the state of a correct service with that of an 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

For multiperforming systems, several services can be distinguished, together with several modes of service delivery, ranging from full capacity to complete disruption, which can be seen as distinguishing less and less correct service deliveries. The performance-related measures of dependability for such systems are usually referred to as *performability* [Meye78, Smit88].

2.3 Failure Behavior of an X-ware System

Section 2.3.1 characterizes the behavior of atomic systems: discrete- and continuous-time reliability expressions are derived. The behavior of systems made up of components is addressed in Sec. 2.3.2, where structural models of a system according to different types of relations are first derived, enabling a precise definition of the notion of *interpreter;* behavior of single-interpreter and of multi-interpreter systems are then successively considered.

2.3.1 Atomic systems

The simplest functional model of a system is regarded as performing a mapping of its input domain I into its output space O.

An execution run of the system consists of selecting a sequence of input points. A trajectory in the input domain—not necessarily composed of contiguous points—can be associated with such a sequence. Thus, each element in I is mapped to a unique element in O if it is assumed that the state variables are considered as part of I and/or O.

According to Sec. 2.2.2, a system failure may result from:

- The *activation* of a fault internal to the system, previously *dormant;* an internal fault may be a physical or design fault.

- The *occurrence* of an external fault, originating from either the physical or the human environment of the system.

Two subspaces in the input space can thus be identified:

- I_{fi}, subspace of the faulty inputs
- I_{af}, subspace of the inputs activating internal faults

The *failure domain* of the system is $I_F = I_{fi} \cup I_{af}$. When the input trajectory meets I_F, an error occurs which leads to failure.

Thus, at each selection of an input point, there is a nonzero probability for the system to fail. Let p be this probability, assumed identical for the time being whatever the input point selected:

$$p = P\{\text{system failure at an input point selection} \mid \text{no failure at the previous input point selections}\}$$

Let $R_d(k)$ be the probability of no system failure during an execution run comprising k input point selections. We then have

$$R_d(k) = (1 - p)^k \tag{2.1}$$

where $R_d(k)$ is the *discrete-time system reliability.*

Let t_e be the execution duration associated with an input selection; t_e is supposed for the moment to be identical irrespective of the input point selected. Let t denote the time elapsed since the start of execution: $t = kt_e$.

The notion of (isolated) input points is not very well suited for a number of situations, such as control systems, executive software, and hardware. Thus, let us assume that there exists a finite limit for p/t_e when t_e becomes vanishingly small. Let λ be this limit:

$$\lambda = \lim_{t_e \to 0} \frac{p}{t_e}$$

It turns out that the distribution becomes the exponential distribution:

$$R(t) = \lim_{t_e \to 0} R_d(k) = \exp(-\lambda t) \tag{2.2}$$

where $R(t)$ is the *continuous-time system reliability,* and λ is its failure rate.

Let us now relax the identity assumption of p and t_e with respect to the input point selections; let the following be defined

$p(j) = P\{$system failure at the jth input point selection | no failure at the previous input point selections$\}$

$t_e(j)$ = execution duration associated with the jth input selection

We then obtain

$$R_d(k) = \prod_{j=1}^{k} [1 - p(j)] \qquad t = \sum_{j=1}^{k} t_e(j) \qquad \lambda(j) = \lim_{\forall j\, t_e(j) \to 0} \frac{p(j)}{t_e(j)}$$

$$R_d(k) = \prod_{j=1}^{k} \{1 - \lambda(j)\, t_e(j) + o[t_e(j)]\} \tag{2.3}$$

$$R(t) = \lim_{\forall j\, t_e(j) \to 0} R_d(k) = \exp\left(-\int_0^t \lambda(\tau)\, d\tau\right)$$

Equation (2.3) is nothing other than the general expression of a system's reliability: see App. B (Sec. B.2), which describes the reliability theory in detail.

It could be argued that the preceding formulations are in fact better suited to design faults or to external faults than to physical faults, since they are based on the existence of a failure domain, which may be nonexistent with respect to physical faults as long as no hardware component fails. It should be remembered that, from the point of view of physics reliability, there is no sudden, unpredictable failure. In fact, a hardware failure is due to anomalies (errors) at the electronic level, caused by physicochemical defects (faults). Or to put it differently, there are no fault-free systems—either hardware or software—there are only systems which have not yet failed. However, the notion of operational fault (i.e., which develops during system operation, and thus did not exist at the start of operational life) is—although arbitrarily [Lapr92a]—a convenient and usual notion. Incorporating the notion of operational fault in the previous formulation can be done as follows. Let j_0 be the number of input point selections such that $p(j) = 0$ for $j < j_0$, $p(j) = p$ for $j \geq j_0$, and $u(j_0)$ the associated probability, i.e., $u(j_0) = P\{p(j) = 0, j < j_0; p(j) = p, j \geq j_0\}$.

The expression of discrete-time reliability becomes:

$$R_d(k) = \sum_{j_0=0}^{k} (1-p)^{k-j_0}\, u(j_0), \text{ with } \sum_{j_0=0}^{k} u(j_0) = 1$$

Going through the same steps as before, we get: $R(t) = \lim_{t_e \to 0} R_d(k) = \exp(-\lambda t)$.

What precedes shows that, although the action mechanisms of the various classes of faults may be different from a physical viewpoint according to their causes, a single formulation can be used from a probability modeling perspective. This formulation applies whatever the fault class considered, either internal or external, physical or design-induced. In the case of software, randomness results, at least from the trajectory in the input space that will activate the fault(s). In addition, it is now known that most of the software faults still present in operation, after validation, are "soft" faults, in the sense that their activation conditions are extremely difficult to reproduce, hence the difficulty of diagnosing and removing them,* which adds to the randomness.

The constancy of the conditional failure probability at execution with respect to the execution sequence is directly related to the constancy of the failure rate with respect to time, as evidenced by Eqs. (2.1) to (2.3). In other words, the points of an input trajectory are not correlated *with respect to the failure process.* This statement is all the more likely to be true if the failure probability is low, i.e., if the quality of the system is high, and thus applies more to systems in operational life than to those under development and validation. It is an abstraction, however, thus immediately raising the question of how well this abstraction reflects reality.

As far as hardware is concerned, it has been long shown that (see, for example, [Cart70]) the failure rates of electronic components as estimated from experimental data actually decrease with time—even after the period of infant mortality. However, the decrease is generally low enough to be neglected. Going further, the interpretation of the failure data for satellites, as described in [Hech87], establishes a distinction between (1) stable operating conditions where the failure rate is slowly decreasing (namely, a constant failure rate is a satisfactory assumption) and (2) varying operating conditions leading to failure rates significantly decreasing with time.

Similar phenomena have been noticed concerning software: (1) constant failure rates for given, stable, operating conditions [Nage82] and (2) high influence of system load [Cast81]. Also, a series of experimental studies conducted on computing systems have confirmed the significant influence of the system load on both hardware and software failure processes [Iyer82].

The influence of varying operating conditions can be introduced by considering that *both the input trajectory and the failure domain* I_F

* By way of example, a large survey was conducted on Tandem systems [Gray86]. From the examination of several dozens of spooler error logs, it was concluded that only one software fault out of 132 was not a soft fault.

may be subject to variations. The variation of the failure domain deserves some comments. It may be due to two phenomena [Iyer82b]:

1. Accumulation of physical faults which remain dormant under low load conditions and are progressively activated as the load increases [Meye88].
2. Creation of temporary faults resulting from the presence of seldom-occurring combinations of conditions. Examples are (1) pattern-sensitive faults in semiconductor memories, change in parameters of a hardware component (effect of temperature variation, delay in timing due to parasitic capacitance, etc.) or (2) situations occurring when system load rises beyond a certain threshold, such as marginal timing and synchronization. The latter situation may affect software as well as hardware: the notion of temporary faults—especially intermittent faults—also applies to software [Gray86]. Experimental work [McCo79] has shown that the failure rates relative to temporary faults decrease significantly with time.

From a probabilistic viewpoint, the failure probability at execution in discrete time, or the failure rate in continuous time, may be considered as random variables. The system reliability then results from the mixture of two distributions:

1. *In discrete time,* the distribution of the number of nonfailed executions in given operating conditions, thus with a given, constant, failure probability at execution, and the distribution of the probability of failure at execution.
2. *In continuous time,* the distribution of the time to failure with a given, constant, failure rate, and the distribution of the failure rate.

Let $g_d(p)$ and $g_d(\lambda)$ be the probability density functions of the distributions of the probability of failure at execution and of the failure rate, which may take G values relative to the realizations p_i, $i = 1, \ldots, G$, of p, and λ_i, $i = 1, \ldots, G$, of λ. The expression of discrete-time reliability becomes

$$R_d(k) = \sum_{i=1}^{G} (1 - p_i)^k \, g_d(p_i)$$

Going through similar steps as before, we obtain

$$t = k \sum_{i=1}^{G} t_{ei} \, g_d(p_i) \qquad \lambda_i = \lim_{t_{ei} \to 0} \frac{p_i}{t_{ei}}, \; i = 1, \ldots, G$$

where t_{ei} and λ_i are the execution time and the failure rate, respectively, for executions carried out under operating condition i, $i = 1, \ldots, G$. Finally we get

$$R(t) = \lim_{\forall i\, t_{ei} \to 0} R_d(k) = \sum_{i=1}^{G} g_d(\lambda_i) \exp(-\lambda_i t)$$

This is the mixed exponential distribution.

When p is a continuous random variable with density function $g_c(p)$, or λ is a continuous random variable with density function $g_c(\lambda)$, we get

$$R_d(k) = \int_0^1 (1-p)^k g_c(p)\, dp \qquad R(t) = \int_0^\infty \exp(-\lambda t)\, g_c(\lambda)\, d\lambda$$

From the properties of the mixture of distributions (see, for example, [Barl75]), the system failure rate is nonincreasing with time, whatever the distributions g_d and g_c.

A model is of no use without data. This is where statistics come into play. Let M instances of the system be considered, executed independently. The term *independently* is a keyword in the following. This is a conventional assumption with respect to physical faults when several sets of hardware are run in parallel, supplied with the same input pattern sequences. Of course, this approach cannot be transposed to software. The independence of the executions of several systems means that they are supplied with *independent input sequences.* This reflects operational conditions when considering a base of deployed systems; for instance, the input sequences supplied to the same text-processing software by users in different places performing completely different activities are likely to exhibit independence with respect to residual fault activation.

Let $M(k)$ and $M(t)$ be the number of nonfailed instances after k executions of each instance, and after an elapsed time of t, respectively, since the start of the experiment ($M(0) = M$); an instance failing at the jth execution, $j = 1, \ldots, k$, or at time τ, $\tau \in [0, t]$, is no longer executed. The independency of execution of the various instances enable these executions to be considered as Bernoulli trials, and the discrete-time reliability is thus $R_d(k) = E[M(k)]/M(0)$, and the continuous-time reliability is $R(t) = E[M(t)]/M(0)$. These equations are none other than the basic equations for the statistical interpretation of reliability as stated in the general systems reliability theory (e.g., [Shoo73, Kozl70]). Statistical estimators of $R_d(k)$ and $R(t)$ are then: $\hat{R}_d(k) = M(k)/M(0)$ and $\hat{R}(t) = M(t)/M(0)$.

The preceding shows that the equations forming the core of the statistical estimation of reliability for a set of hardware systems apply

equally to software systems, *provided the experimental conditions are in agreement with the underlying assumptions.*

2.3.2 Systems made up of components

2.3.2.1 System models. Adopting the spirit of [Ande81], a system may be viewed from a structural viewpoint as a set of components bound together in order to interact; a component itself is a system, decomposed (again) into components, etc. The recursion ends when a system is considered *atomic:* no further internal structure can be discerned or is of interest, and can be ignored. The model corresponding to the relation "is composed of" is a tree, whose nodes are the components; the levels of a tree obviously constitute a hierarchy. Such a model does not enable the interactions between the components to be represented: the presence of arcs in a graphic representation would present only the relation "is composed of" existing between a node and the set of its immediate successors. The set of the elements of a level of the tree gives only the list of the system components, with more or less *detail* according to the tree level considered: the lower the level, the more detailed the list becomes. To obtain a more representative view of the system, the relations existing between the components have to be presented. This is achieved through interaction diagrams where the nodes are the system components and the arcs represent a common interface. An arc exists when two elements *can* interact. Although the relation "interacts with" is an essential relation when describing a system, it obviously does not infer any hierarchy.

The use of the relations in modeling a system is given in Fig. 2.5 for an intentionally simple system, where the components S_1, S_2, and S_3 can, for instance, be the application software, the executive software, and the hardware.

The respective roles of a system and of its user(s) with respect to the notion of service are fixed: the system is the producer of the service, and the user is the consumer. Therefore, there exists a natural hierarchy between the system and its user: the user *uses the service of* (or *delivered by*) the system.

With respect to the set of components of one given level of the decomposition tree, the relation "uses the service of"—a special form of the relation "interacts with"—allows for an accurate definition of a special class of components: if and only if *all* the components of the level may be hierarchically situated through the relation "uses the service," then they are *layers.* Or equivalently, components of a given detail level are layers (1) if any two components of that level play a fixed role with respect to this relation, i.e., either consumer or producer; similarly, (2) if the graph of the relation is a single branch tree. Conversely, if their

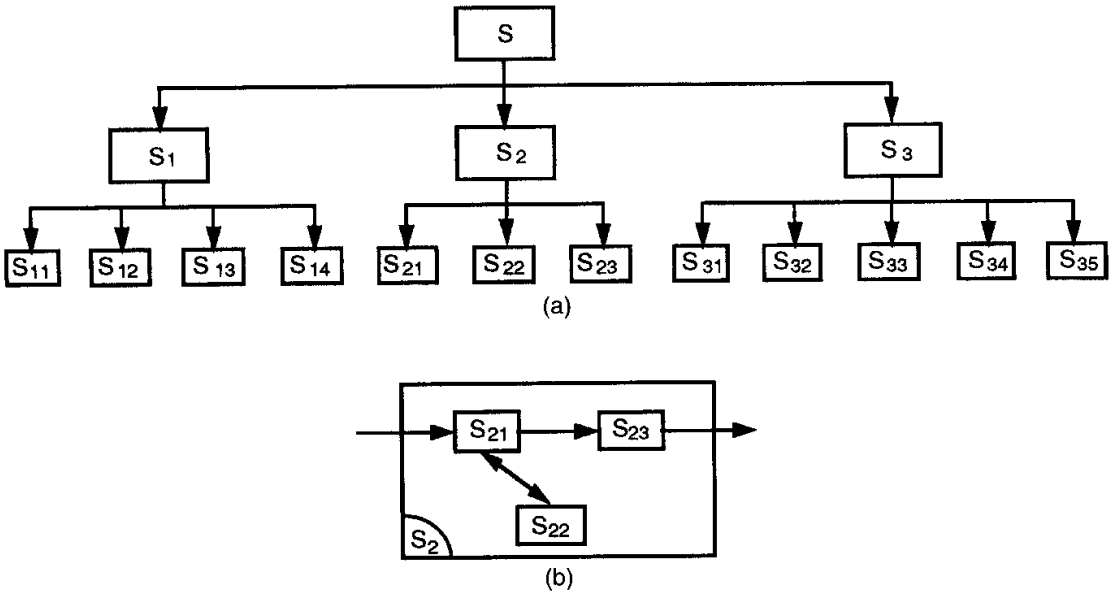


Figure 2.5 Components of a system. (*a*) System model according to the relation "is composed of." (*b*) System model according to the relation "interacts with."

respective consumer and producer roles can change, or if their interactions are not governed by such a relation, they are simply components. It is noteworthy that the notion of service has naturally—and implicitly—been generalized with respect to the layers: the service delivered by a given layer is its behavior as perceived by the upper layer, where the term *upper* has to be understood with respect to the relation "uses the service of." Also worth noting is the fact that the relation "uses the service of" induces an ordering of the time scales of the various layers: time granularity usually does not decrease with increasing layers. Considering the previous example in Fig. 2.5 leads to the model in Fig. 2.6.

Structuring into many layers may be considered for design purposes. Their actual relationship at execution is generally different: compilation removes—at least partially—the structuring, and several layers may, and generally are, executed on a single one. A third type of rela-

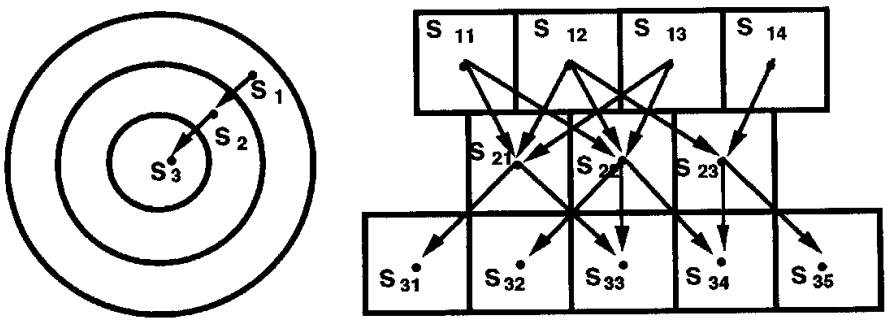


Figure 2.6 Layers of a system.

tion thus has to be considered: "is interpreted by." The *interpretive interface* [Ande81] between two layers or sets of layers is characterized by the provision of objects and operations to manipulate those objects. A system may then be viewed as a hierarchy of *interpreters,* where a given interpreter may be viewed as providing a concrete representation of abstract objects in the above interpreter; this concrete representation is itself an abstract object for the interpreter beneath the considered one.

Considering again the previous example, the hardware layer interprets the application as well as the executive software layers. However, the executive software may be viewed as an *extension* of the hardware interpreter—e.g., through (1) the provision of "supervisor call" instructions or (2) the prevention of invoking certain "privileged" instructions. This leads to the system model depicted in Fig. 2.7.

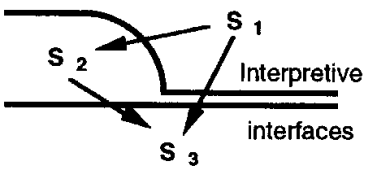
Finally, note that the expression "abstraction level" has not been used so as to avoid any confusion: all the hierarchies defined are, strictly speaking, abstractions.

2.3.2.2 Behavior of a single-interpreter system. A system is assumed to be composed of C components, of respective failure rates λ_i, $i = 1, \ldots, C$. The system behavior with respect to the execution process is modeled through a Markov chain with the following parameters:

- S: number of the states of the chain, a state being defined by the components under execution
- $1/\gamma_j$: mean sojourn time in state j, $j = 1, \ldots, S$
- $q_{jk} = P\{$system makes a transition from state j to state k | start or end of execution of one or several components$\}$, $j = 1, \ldots, S$, $k = 1, \ldots, S$, $\sum_{k=1}^{S} q_{jk} = 1$

A system failure is caused by the failure of any of its components. The system failure rate ξ_j in state j is thus the sum of the failure rates of the components under execution in this state, denoted by

$$\xi_j = \sum_{i=1}^{C} \delta_{i,j} \lambda_i, j = 1, \ldots, S \tag{2.4}$$



System model according to the relation «is interpreted by»

Figure 2.7 System interpreters.

where $\delta_{i,j}$ is equal to 1 if component i is under execution in state j, or else it is equal to 0.

The system failure behavior may be modeled by a Markov chain with $S + 1$ states, where the system delivers correct service in the first S states (components are under execution without failure occurrence); state $S + 1$ is the failure state, which is an absorbing state. Let $\mathbf{A} = [a_{jk}]$, $j = 1, \ldots, S$, $k = 1, \ldots, S$ be the transition matrix associated with the nonfailed states. This matrix is such that its diagonal terms a_{jj} are equal to $-(\gamma_j + \xi_j)$, and its nondiagonal terms a_{jk}, $j \neq k$, are equal to $q_{jk}\,\gamma_j$. The matrix $\mathbf{A}$ may be viewed as the sum of two matrices $\mathbf{A}'$ and $\mathbf{A}''$ such that: (1) the diagonal terms of $\mathbf{A}'$ are equal to $-\gamma_j$, its nondiagonal terms being equal to $q_{jk}\,\gamma_j$, and (2) the diagonal terms of $\mathbf{A}''$ are equal to $-\xi_j$, its nondiagonal terms being equal to 0.

The system behavior can thus be viewed as resulting from the superimposition of two processes: the execution process, of parameters γ_j and q_{jk} (transition matrix $\mathbf{A}'$), and the failure process, governed by the failure rates ξ_j (transition matrix $\mathbf{A}''$).

A natural assumption is that the failure rates are small with respect to the rates governing the transitions from the execution process or, equivalently, that a large number of transitions resulting from the execution process will take place before the occurrence of a failure—a system that would not satisfy this assumption would be of little interest in practice. This assumption is expressed as follows: $\gamma_j >> \xi_j$.

Adopting a Markov approach for modeling the system behavior resulting from the compound execution-failure process is based on this assumption. Similar models have been proposed in the past for software systems [Litt81, Cheu80, Lapr84], with less generality than here, however, since those models assumed a sequential execution (one component only executed at a time).*

By definition, the system failure rate $\lambda(t)$ is given by

$$\lambda(t) = \lim_{dt \to 0} \frac{1}{dt} P\{\text{failure between } t \text{ and } t + dt \mid \text{no failure between initial instant and } t\}$$

* In these references, the Markov approach was justified:

- Heuristically in [Cheu80, Lapr84], by analogy with performance models in the first reference, and with availability models in the second,
- From a weaker assumption, semi-Markov, in [Litt81]. It is shown there that the compound process of execution and failure converges toward a Poisson process, and that the contribution of the distribution functions of the component execution times is limited to their first moments.

Let $Pj(t)$ denote the probability for the system to be in state j. It follows that

$$\lambda(t) = \frac{\left(\sum_{j=1}^{S} \xi_j P_j(t)\right)}{\left(\sum_{j=1}^{S} P_j(t)\right)} \tag{2.5}$$

The consequence of the assumption $\gamma_j >> \xi_j$ is that the execution process converges toward equilibrium before failure occurrence. The vector $\alpha = [\alpha_j]$ of the equilibrium probabilities is the solution of $\alpha \cdot \mathbf{A}' = \mathbf{0}$, with $\sum_{j=1}^{S} \alpha_j = 1$. Thus, $P_j(t)$ converges towards α_j before failure occurs, and Eq. (2.5) becomes*:

$$\lambda = \sum_{j=1}^{S} \alpha_j \xi_j \tag{2.6}$$

Equation (2.6) may be rewritten as follows, to account for Eq. (2.4):

$$\lambda = \sum_{j=1}^{S} \alpha_j \sum_{i=1}^{C} \delta_{i,j} \lambda_i = \sum_{i=1}^{C} \lambda_i \sum_{j=1}^{S} \delta_{i,j} \alpha_j \tag{2.7}$$

Let

$$\pi_i = \sum_{j=1}^{S} \delta_{i,j} \alpha_j$$

Equation (2.6) becomes

$$\lambda = \sum_{i=1}^{C} \pi_i \lambda_i \tag{2.8}$$

This equation has a simple physical interpretation:

- α_j represents the average proportion of time spent in state j in the absence of failure; thus π_i is the average proportion of time when

* Another approach to this result is as follows. A given system component will be executed; thus the transition graph between the nonfailed states is strongly connected. As a result, the matrix **A** is irreducible and has one real negative eigenvalue whose absolute value σ is lower than the absolute values of the real parts of the other eigenvalues [Page80]. Asymptotically, the system failure behavior is then a Poisson process of rate σ. In our case, the asymptotic behavior is relative to the execution process; therefore, (1) it is reached rapidly and (2) $\sigma = \lambda$, thus system reliability is: $R(t) = \exp(-\lambda t)$.

component i is under execution in the absence of failure. It is noteworthy that the sum of the π_i's can be larger than 1:

$$0 \leq \sum_{i=1}^{C} \pi_i \leq C$$

- λ_i is the failure rate of component i assuming a continuous execution.

The term $\pi_i \lambda_i$ can therefore be considered as the *equivalent* failure rate of component i.

Equation (2.8) deserves a few comments. First let us consider hardware systems. It is generally considered that all components are continuously active. This corresponds to making all the π_i's equal to 1, leading to the usual equation

$$\lambda = \sum_{i=1}^{C} \lambda_i$$

Consider software systems. The key question is how to estimate the component failure rates. There are two basic—and opposite—approaches: (1) exploiting results of repetitive-run experiments without experiencing failures (i.e., through statistical testing [Curr86, Thev91]) and (2) exploiting failure data using a reliability growth model, the latter being applied to each software component, as performed in [Kano87, Kano91a, Kano93b]. It is important to stress the data representativeness in either approach; a condition is that the data are collected in a representative environment (i.e., being relative to components in interaction, real or simulated, with the other system components). If this condition is not fulfilled, a distinction has to be made between the interface failure rates (characterizing the failures occurring during interactions with other components) and the internal component failure rates, as in [Litt81], where the expression of a component failure rate has the form

$$\lambda_i = \zeta_i + \gamma_i \sum_j q_{ij} \rho_{ij}$$

the ζ_i being the internal component failure rates and ρ_{ij} the interface failure probabilities. This leads to a complexity in the estimation of the order of C^2 instead of C.

An important question is how to account for different environments. If this question is interpreted as estimating the reliability of a software system of a base of deployed software systems, then the approach indicated in Sec. 2.3.1 where the failure rate was considered as a random variable can be extended here; the π_i's being considered as random

variables as well. Another interpretation of the previous question is: knowing the reliability in a given environment, how can the reliability be estimated in another environment? Let us consider sequential software systems. The parameters characterizing the execution process are then defined as follows:

- $1/\gamma_i$: mean execution time of component i, $i = 1, \ldots, C$.
- $q_{ij} = P\{$ component j starts execution $|$ end of execution of component $i\}$, $i = 1, \ldots, C, j = 1, \ldots, C$, $\sum_{j=1}^{C} q_{ij} = 1$

The Markov chain modeling the compound execution-failure process is a $(C + 1)$ state chain, state i being defined by the execution of component i, and the π_i's reduce to the α_i's (in the case of sequential software, $\delta_{ii} = 1$ and all others are zeros). We have $\lambda_i = p_i \gamma_i$, $i = 1, \ldots, C$, where p_i is the failure probability at execution of component i; hence

$$\lambda = \sum_{i=1}^{C} \pi_i \gamma_i p_i = \sum_{i=1}^{C} \eta_i p_i \tag{2.9}$$

where $\eta_i = \pi_i \gamma_i$ is the visit rate of state i at equilibrium. The η_i's have a simple physical interpretation, as $1/\eta_i$ is the mean recurrence time of state i (i.e., the mean time duration between two executions of component i in the absence of failure). Equation (2.9) is of interest as it enables a distinction to be made between (1) continuous time–execution process and (2) discrete time–failure process conditioned upon execution. If the p_i's are intrinsic to the considered software and independent of the execution process, then it is possible to infer the software failure rate for a given environment from the knowledge of the η_i's for this environment and the knowledge of the p_i's. The condition for this assumption to be verified in practice is that it is possible to find a suitable decomposition into components: the notion of component for a software is highly arbitrary, and the higher the number of components considered for a given software, the smaller the state space of each component, so the higher the likelihood of providing a satisfactory coverage of the input space for the component. A limit to such an approach is that the higher the number of components, the more difficult the estimation of the η_i's becomes. Also, time granularity (and near decomposability [Cour77]) can offer criteria to find suitable decompositions.

2.3.2.3 Behavior of a multi-interpreter system. When a system is viewed as a hierarchy of interpreters, as defined in Sec. 2.3.2.1, the execution relative to the selection of an input point for the highest interpreter (which directly interprets the requests originating from the system

user) is supported by a sequence of input point selections in the next lower interpreter, and so on up to the lowest considered interpreter. Assume the system is composed of I interpreters, the first interpreter being the top of the hierarchy and the Ith interpreter its base.

Each interpreter may be faulty and submitted to erroneous inputs. Failure of any interpreter during the computations relative to the input point selection of the next higher interpreter will lead to the failure of the latter, and thus by propagation to the top interpreter's failure (i.e., to system failure). Adopting the terminology of the conventional system reliability theory, the hierarchy interpreters constitute a series system. Intuitively, it may be deduced that the system failure rate is equal to the sum of the failure rates of interpreters of the hierarchy. If $\lambda_i(t)$, $i = 1, \ldots, I$ denotes the failure rate of interpreter i, we then have (the demonstration is left as an exercise to you):

$$\lambda(t) = \sum_{i=1}^{I} \lambda_i(t) \tag{2.10}$$

Now consider that each interpreter is composed of C_i components, $i = 1, \ldots, I$. At execution, a component of interpreter i will use services of one or more components of interpreter $i + 1$, and so on. We may therefore define trees of utilization of services provided by components of interpreter $i + 1$ by the components of interpreter i, as indicated in Fig. 2.8.

Thus, with each pair of adjacent interpreters, it is possible to associate a service utilization matrix $\mathbf{U}_{i,i+1} = [\mathbf{U}_{jk}], j = 1, \ldots, C_i, k = 1,., C_{i+1}$. $\mathbf{U}_{i,i+1}$ is a connectivity matrix whose terms U_{jk} are such that $U_{jk} = 1$ if, during execution, component j of interpreter i utilizes the services of component k of interpreter $i + 1$, or else $U_{jk} = 0$.

Let us define the following failure rate vectors:

- $\Lambda_i = [\lambda_{i,j}], i = 1, \ldots, I, j = 1, \ldots, C_i$, where λ_{ij} is the failure rate of component j of interpreter i.
- $\Omega_i = [\omega_{i,j}], i = 1, \ldots, I, j = 1, \ldots, C_i$; $\omega_{i,j}$ is the aggregated failure rate of component j of interpreter i; the term *aggregated* means that the failure rates of components of interpreters $i + 1, \ldots, I$ needed for execution are accounted for.

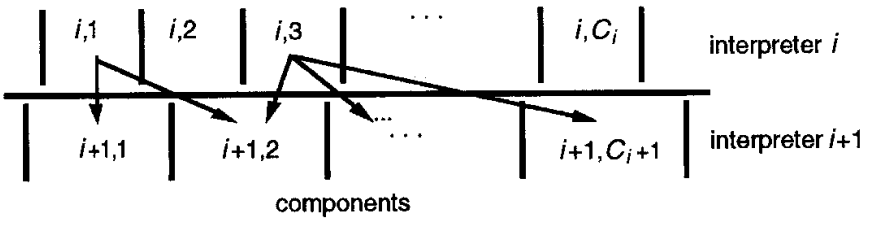


Figure 2.8 Utilization trees between components of interpreters.

The vectors $\mathbf{\Omega}_i$ are solutions of the following matrix equation:

$$\mathbf{\Omega}_i = \mathbf{\Lambda}_i + \mathbf{U}_{i,i+1}\,\mathbf{\Omega}_{i+1},\ i = 1, \ldots, I-1,\ \mathbf{\Omega}_I = \mathbf{\Lambda}_I$$

It then follows that:

$$\mathbf{\Omega}_1 = \sum_{k=1}^{I} \mathbf{V}_k\,\mathbf{\Lambda}_k$$

$\mathbf{V}_k$ is the accessibility matrix of the top interpreter to interpreter k: $\mathbf{V}_1$ is the identity matrix of dimensions $(C_1 \times C_1)$, $\mathbf{V}_k = \mathbf{U}_{1,2} \otimes \mathbf{U}_{2,3}, \ldots, \otimes \mathbf{U}_{k-1,k}$, $k = 2, \ldots, I$; where the symbol $\otimes$ denotes the boolean product of matrices (a given component can contribute only once through its failure rate).

When applying Eq. (2.8) to the components of the upper interpreter in the hierarchy, we obtain the following system failure rate:

$$\lambda = \sum_{j=1}^{C_1} \pi_{1,j}\,\omega_{1,j}$$

where $\pi_{1,j}$ is the proportion of time during which component j of the top interpreter is being executed, with an idle component period being characterized by $\omega_{1,j} = 0$.

Consider the important case in practice of a system composed of two interpreters: a software interpreter and a hardware interpreter. It is assumed that the software components are executed sequentially, and that all hardware components are together involved in the execution; it is further assumed that the system is in stable operating conditions. In the following, indices S and H relate to software and hardware, respectively. Applying the above approach leads to the following equations:

$$\begin{aligned} \omega_{S,j} &= \lambda_{S,j} + \sum_{k=1}^{C_H} \lambda_{H,k} \\ \lambda &= \sum_{j=1}^{C_S} \pi_{S,j}\,\omega_{S,j} = \sum_{j=1}^{C_S} \pi_{S,j}\,\lambda_{S,j} + \sum_{k=1}^{C_H} \lambda_{H,k} \end{aligned} \tag{2.11}$$

The intuitive result expressed in Eq. (2.11) has thus been obtained through use of a rigorous approach.

2.4 Failure Behavior of an X-ware System with Service Restoration

In Sec. 2.3, the behavior of atomic and multicomponent systems was characterized without taking into account the effects of service restoration, thereby allowing expressions of the failure rate of such systems

and of the reliability to be derived. In this section, service restoration is taken into account, thus allowing the system behavior resulting from the compound action of failure and restoration processes to be modeled. Restoration activities may consist of a pure restart (supplying the system with an input pattern different from the one which led to failure) or they can be performed after introduction of modifications (corrections only or/and specification changes).

System behavior is first characterized by the evolution of its failure intensity in Sec. 2.4.1. Section 2.4.2 introduces the various maintenance policies that can be carried out. Sections 2.4.3 and 2.4.4 address reliability and availability modeling, respectively.

2.4.1 Characterization of system behavior

The nature of the operations to be performed in order for the service to be restored (i.e., delivered again to its user(s)) after a failure has occurred enables stable reliability or reliability growth to be identified. This may be defined as follows:

- *Stable reliability.* The system's ability to deliver a proper service is *preserved* (stochastic identity of the successive times to failure).
- *Reliability growth.* The system's ability to deliver proper service is *improved* (stochastic increase of the successive times to failure).

Practical interpretations 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: (1) in the case of a hardware failure, the failed part is substituted for another one, identical and nonfailed; (2) in the case of a software failure, the system is restarted with an input pattern that differs 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 hardware) and removed.

Reliability *decrease* (stochastic decrease of the successive times to failure) is both theoretically and practically possible. In this case, it is hoped that the decrease is limited in time and that reliability is globally growing over a long observation time.

Reliability decrease may originate from (1) introduction of new faults during corrective actions, whose probability of activation is greater than that of the removed fault(s); (2) introduction of a new version with modified functionalities; (3) change in the operating conditions (e.g., an intensive testing period; see [Kano87], where such a situation is depicted); (4) dependencies between faults: some software faults can be masked by others, that is, they cannot be activated as long

as the latter are not removed [Ohba84]; removal of the masking faults will lead to an increase in the failure intensity.

The reliability of a system is conveniently illustrated by the failure intensity, as it is a measure of the frequency of the system failures as noticed by its user(s). Failure intensity is typically first decreasing (reliability growth) due to the removal of residual design faults either in the software or hardware. It may become stable (stable reliability) after a certain period of operation; the failures due to internal faults occurring in this period are due either to physical faults or to unremoved design faults. Failure intensity generally exhibits an increase (reliability decrease) upon the introduction of new versions incorporating modified functionalities; then it tends toward an asymptote again, and so on. It is noteworthy that such a behavior is not restricted to the operational life of a system but also applies to situations occurring during the development phase of a system—for example, (1) during incremental development [Curr86] or (2) during system integration [Leve89, Tohm89].

Typical variations of the failure intensity may be represented as indicated in Fig. 2.9, curve a. Such a curve depends on the granularity of the observations, and may be felt as resulting from the smoothing of more noticeable variations (curve *b*); in turn, it may be smoothed into a continuously decreasing curve *c*. Although such a representation is very general and covers many practical situations (see, for example, [Kenn92]), there are situations which exhibit discontinuities important enough that the smoothing process cannot be considered as reasonable (e.g., upon introduction of a new system generation).

2.4.2 Maintenance policies

The rate of reliability growth (i.e., failure intensity decrease) is closely related to the correction and maintenance policies retained for the sys-

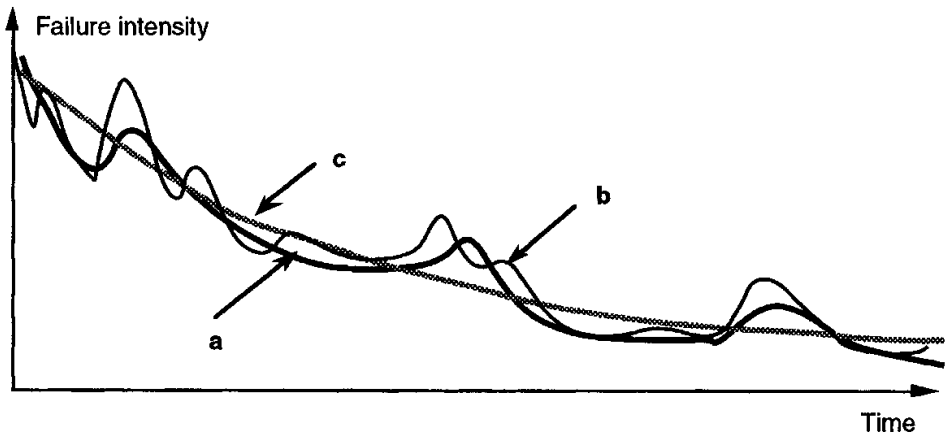


Figure 2.9 Typical variations of a system's failure intensity.

tem [Kano89]. These policies may consist of either (1) system modification after each failure, or (2) system modification after a given number of failures, or even (3) preventive maintenance (i.e., introduction of modifications without any failure observed on the system considered). The status of a system between two modifications will be called a *version.*

A policy that accepts as special cases the specific cases mentioned above is as follows: the jth system modification takes place after a_j failures have occurred since the $(j-1)$th system modification, which means that version j experiences a_j failures.

Concerning the times to failure:

- Let $X_{j,i}$ denote the time between service restoration following the $(i-1)$th failure and the ith failure of version j.
- Let Z_j denote the time between service restoration following the a_jth failure of version j and service interruption for the introduction of the jth modification, that is, for the introduction of version $(j+1)$.

Considering now the times to restoration, two types of service restoration are to be distinguished: service restoration due to system restart after failure and service restoration after introduction of a new version. Let $Y_{j,i}$ denote the restart duration after the ith failure of version j, and W_j denote the duration necessary for the introduction of the jth modification; the modification itself may have been performed off-line. Finally, let T_j denote the time between two version introductions. We have:

$$T_j = \sum_{i=1}^{a_j} (X_{j,i} + Y_{j,i}) + Z_j + W_j, j = 1,2, \ldots$$

The relationship between the various time intervals is given in Fig. 2.10.

The number of failures between two modifications (a_j) characterize the policy for maintenance and service restoration. It depends on sev-

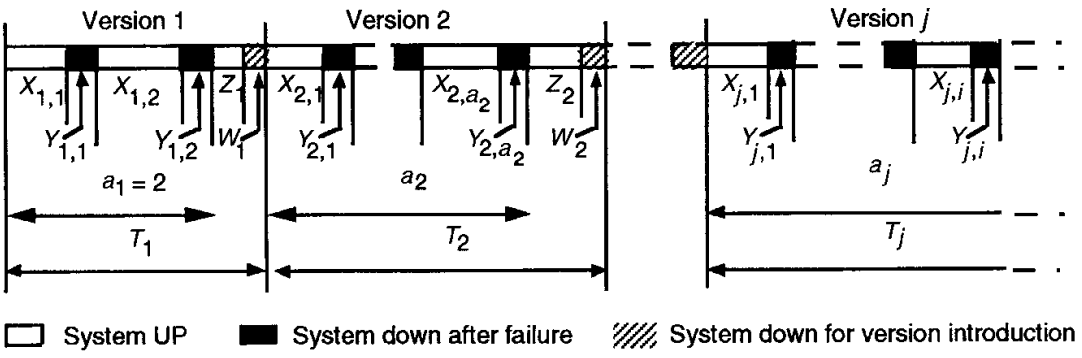


Figure 2.10 Relationship between the various time intervals.

eral factors such as (1) the failure rate of the system, (2) the nature of the faults (e.g., time needed to diagnose the fault and the consequence of the failure due to the activation of this fault), (3) the considered phase in the life cycle (the policy may vary for a given system within the same phase), and (4) the availability of the maintenance team.

Three (extreme) special cases of this general policy are noteworthy:

1. $a_j = 1$ and $Z_j = 0\ \forall j$. Service is restored only after a system modification has been performed. This case relates to (1) a usual hypothesis for several software reliability (growth) models or (2) the case of critical systems after a (potentially) dangerous failure occurrence.
2. $a_1 = \infty$ and $Z_j = 0\ \forall j$. Service is restored without any system modification ever being performed (stable reliability). This case relates to (1) hardware, when maintenance consists of replacing a failed part with an identical (new) one and (2) software, when no maintenance is performed; service restoration always corresponds to a restart with an input pattern different from the one having led to failure.
3. $a_j = 0$. The $(j + 1)$th version is introduced before any failure occurrence since the last modification. This case corresponds to preventive maintenance, either corrective, adaptive, or perfective.

Although this policy is more general than those usually considered, it is a simplification of real life, and does not explicitly model such phenomena as interweaving of failures and corrections [Kano88] and failure rediscoveries [Adam84].

2.4.3 Reliability modeling

We focus here on the failure process and therefore do not consider the times to restoration or the (possible) time interval between a failure and the introduction of a modification (i.e., we assume the $Y_{j,i}$'s, W_j's, and Z_j's are zero, which means that the failure instants are also restoration instants). Let:

- $t_0 = 0$ denote the considered initial instant (the system is assumed nonfailed).
- $n = 1,2, \ldots$ denote the number of failures. As the nth system failure is in fact the ith failure of version j, the relationship between n, i, and j is

$$n = \left(\sum_{k=1}^{j} a_{k-1} \right) + i, j = 1,2, \ldots, i = 1, \ldots a_j, a_0 = 0$$

- t_n, $n = 1,2, \ldots$ denote the instant of failure occurrence.

- $f_{\chi_j}(t), j = 1, 2, \ldots$ denote the probability density functions (pdf) of the times to failure Xj,i and $sf_{\chi_j}(t)$ denote their survival function (the one's complement of its distribution function). The $X_{j,i}$'s are assumed stochastically identical for a given version.
- $\phi_n(t)$ and $\Phi_n(t)$ denote, respectively, the pdf and the distribution function of the instants of failure occurrence, $n = 1,2,\ldots$.
- $N(t)$ denote the number of failures having occurred in $[0,t]$ and $H(t)$ denote its expectation: $H(t) = E[N(t)]$.

Performing derivations adapted from the renewal theory (see, for example, [Gned69]) is relatively straightforward, provided that the $X_{j,i}$'s are assumed stochastically independent. This assumption, although usual in both hardware and software models, is again a simplification of real life. The T_j's can reasonably be considered as stochastically independent, as resuming execution after the introduction of a modification generally involves a so-called cold restart; however, it must be stated that imperfect maintenance, the consequences of which were noticed a long time ago [Lewi64], is also a source of stochastic dependency. The stochastic independence of the $X_{j,i}$'s for a given j depends on (1) the extent to which the internal state of the system has been affected and (2) the nature of operations undertaken for execution resumption (i.e., whether or not they involve state cleaning).

The following is then obtained under the stochastic independence assumption

$$\phi_n(t) = \left(\circledast_{k=1}^{j-1} f_{\chi_k}(t)^{*a_k} \right) * (f_{\chi_j}(t)^{*i}), j = 1,2, \ldots, i = 1, \ldots, a_j, a_0 = 0 \qquad (2.12)$$

where $*$ stands for the convolution operation, $f_{\chi_k}(t)^{*a_k}$, the a_k-fold convolution of $f_{\chi_k}(t)$ by itself, and $\circledast_{k=1}^{j} f_{\chi_k}(t)$, the convolution of $f_{\chi_1}(t), \ldots, f_{\chi_j}(t)$. In Eq. (2.12) the first term covers $j - 1$ versions and the second term covers the i failures of version j. We have

$$P\{N(t) \geq n\} = P\{t_n < t\} = \Phi_n(t)$$

$$P\{N(t) = n\} = P\{t_n < t < t_{n+1}\} = \Phi_n(t) - \Phi_{n+1}(t)$$

$$H(t) = \sum_{n=1}^{\infty} n\, P\{N(t) = n\} = \sum_{n=1}^{\infty} n\, [\Phi_n(t) - \Phi_{n+1}(t)]$$

$$H(t) = \sum_{n=1}^{\infty} n\, \Phi_n(t) - \sum_{n=1}^{\infty} (n-1)\, \Phi_n(t) = \sum_{n=1}^{\infty} \Phi_n(t)$$

Let $h(t)$ denote the rate of occurrence of failure [Asch84], or ROCOF, $h(t) = dH(t)/dt$, whence

$$h(t) = \sum_{n=1}^{\infty} \phi_n(t) = \sum_{j=1}^{\infty} \sum_{i=1}^{a_j} \left(\circledast_{k=1}^{j-1} (f_{\chi_k}(t)^{*a_k}) * (f_{\chi_j}(t)^{*i}) \right) \tag{2.13}$$

As we do not consider simultaneous occurrences of failure, the failure process is regular or orderly, and the ROCOF is then the *failure intensity* [Asch84].

When considering reliability growth, a usual measure of reliability is *conditional reliability* [Goel79, Musa84]; since the system has experienced $n - 1$ failures, conditional reliability is the survival function associated with failure n. It is defined as follows:

$$\begin{aligned} R_n(\tau) &= P\{Xj, i+1 > \tau \mid t_{n-1}\} = sf_{\chi_j}(\tau), \text{ for } i < a_j \\ R_n(\tau) &= P\{Xj+1, 1 > \tau \mid t_{n-1}\} = sf_{\chi_{j+1}}(\tau), \text{ for } i = a_j \end{aligned} \tag{2.14}$$

This measure is mainly of interest when considering a system in its development phase, as we are then concerned with the time to next failure. However, when dealing with a system in operational life, the interest is in failure-free time intervals τ whose starting instants are not necessarily conditioned on system failures; that is, they are likely to occur at any time t. In this case, we are concerned with the reliability over a given time interval independently of the number of failures experienced, that is, *interval reliability*. Interval reliability is then the probability for the system to experience no failure during the time interval $[t, t + \tau]$.

Consider the following exclusive events:

$$E_0 = \{t + \tau < t_1\} \qquad E_n = \{t_n < t < t + \tau < t_{n+1}\}, n = 1,2, \ldots$$

Event E_n means that exactly n failures occurred prior to instant t, and that no failure occurs during the interval $[t, t + \tau]$. The absence of failure during $[t, t + \tau]$ is the union of all events E_n, $n = 0,1,2, \ldots$ The interval reliability, owing to the exclusivity of the events E_n, is then

$$R(t, t + \tau) = \sum_{n=0}^{\infty} P\{E_n\}$$

The probability of event E_n is shown as

$$P\{E_n\} = P\{t_n < t < t + \tau < t_n + Xj, i + 1\}$$

$$P\{E_n\} = \int_0^t P\{x < t_n < x + dx\}\, P\{Xj, i + 1 > t + \tau - x\} = \int_0^t sf_{\chi_j}(t + \tau - x)\, \phi_n(x)\, dx$$

The reliability thus has the expression

$$R(t,t+\tau) = sf_{\chi_1}(t+\tau) + \sum_{j=1}^{\infty} \sum_{i=0}^{a_j-1} \int_0^t sf_{\chi_j}(t+\tau-x)\, \phi_{\left(\sum_{k=1}^{j} a_{k-1}\right)+i}(x)\, dx$$

which can be written as

$$R(t,t+\tau) = sf_{\chi_1}(t+\tau) + \sum_{j=1}^{\infty} \sum_{i=0}^{a_j-1} sf_{\chi_j}(t+\tau) * \phi_{\left(\sum_{k=1}^{j} a_{k-1}\right)+i}(t) \quad (2.15)$$

This equation is obviously not easy to use. However, it is not difficult to derive, for $\tau << t$, the following equation from Eqs. (2.13) and (2.15):

$$R(t,t+\tau) = 1 - h(t)\,\tau + o(\tau) \quad (2.16)$$

Besides its simplicity, Eq. (2.16) is highly important in practice, as it applies to systems for which the mission time τ is small with respect to the system lifetime t (e.g., systems on board airplanes).

The above derivation is a (simple) generalization of the renewal theory and of the notion of renewal process to nonstationary processes; in the classical theory (stationary processes), (1) the $X_{j,i}$'s are stochastically identical, that is, $f_{\chi_j}(t) = f_{\chi}(t)\ \forall j$ (the case where the first time to failure has a distribution different from the subsequent ones is referred to as *modified renewal* process in [Cox62, Biro74]) and (2) $H(t)$ and $h(t)$ are the renewal function and the renewal density, respectively.

Consider the case where the $X_{j,i}$'s are exponentially distributed: $f_{\chi_j}(t) = \lambda_j \exp(-\lambda_j t)$. The interfailure occurrence times in such a case constitute a piecewise Poisson process. No assumption is made here on the sequence of magnitude of the λ_j's. However, it is assumed that the failure process is converging toward a Poisson process after r modifications have taken place. This assumption means that either (1) no more modifications are performed or (2) if some modifications are still being performed, they do not significantly affect the failure behavior of the system. Let $\{\lambda_1, \lambda_2, \ldots, \lambda_r\}$ be the sequence of these failure rates (Fig. 2.11).

The Laplace transform $\tilde{h}(s)$ of the failure intensity $h(t)$ (Eq. (2.13)) is:

$$\tilde{h}(s) = \sum_{j=1}^{r-1} \prod_{k=0}^{j-1} \left(\frac{\lambda_k}{\lambda_k + s}\right)^{a_k} \sum_{i=1}^{a_j} \left(\frac{\lambda_j}{\lambda_j + s}\right)^{i} + \frac{\lambda_r}{s} \prod_{k=1}^{r-1} \left(\frac{\lambda_k}{\lambda_k + s}\right)^{a_k}$$

Derivation of $h(t)$ is very tedious (see Prob. 2.6). Thus our study will be limited to summarizing the properties of the failure intensity $h(t)$ that can be derived:

- $h(t)$ is a continuous function of time, with $h(0) = \lambda_1$ and $h(\infty) = \lambda_r$.

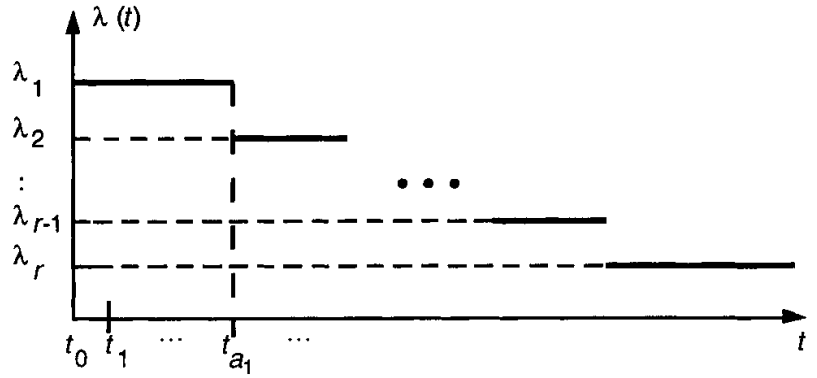


Figure 2.11 Sequence of failure rates.

- When the a_j's are finite,
  - —A condition for $h(t)$ to be a nonincreasing function of time (i.e., a condition for *reliability growth*) is $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_j \geq \ldots \geq \lambda_r$.
  - —The smaller the a_j's, the faster the reliability growth becomes.
  - —If a (local) increase in the failure rates occurs, then the failure intensity correspondingly (locally) increases.
- When $a_1 = \infty$, no correction takes place and we are faced with a classical renewal process; then $h(t) = \lambda_1 \; \forall \; t \in [0, \infty]$, which is the formulation of *stable reliability.*

These results are shown in Fig. 2.12, where the failure intensity is plotted for $a_j = a \; \forall j$.

Typical variations of conditional reliability $R_n(\tau)$ (Eq. (2.14)) are given by Fig. 2.13. When stable reliability is assumed, the underlying process is a classical renewal process with $R_n(\tau) = R_1(\tau), n = 2, 3, \ldots$. For a so-called modified renewal process (the case where the first time to failure has a distribution different from the subsequent ones [Cox62, Biro74]), we usually have $R_n(\tau) < R_1(\tau)$, $n = 2, 3, \ldots$, with $R_n(\tau) = R_2(\tau)$, $n \geq 3$.

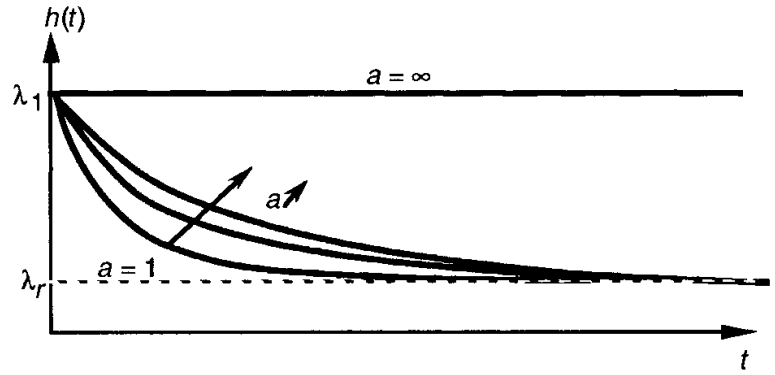


Figure 2.12 Failure intensity.

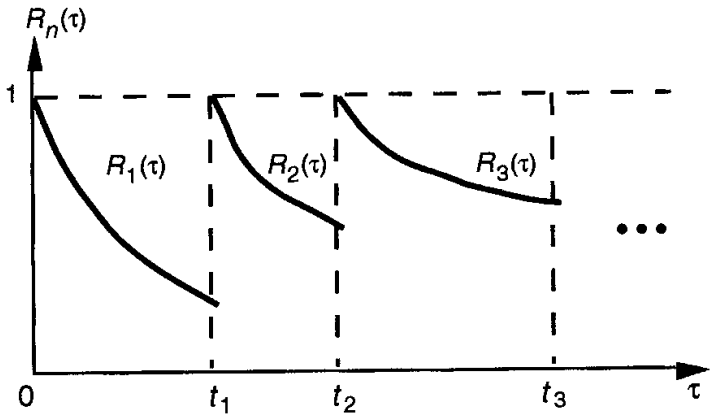


Figure 2.13 Conditional reliability.

Interval reliability $R(t,t+\tau)$ for a given, finite, a (which corresponds to a given curve of Fig. 2.12) can then be derived from Eq. (2.16) for $\tau << t$. Figure 2.14 indicates typical variations of $R(t,t+\tau)$: reliability over mission time τ increases with system lifetime t. In case of stable reliability (i.e., a classical renewal process), interval reliability is independent of the time origin t: the curves of Fig. 2.14 are thus not distinguishable; however, for a modified renewal process, depending on the granularity of time representation versus the mean time to failures, two groups of curves may be distinguished depending on the values of t compared to the mean time to failures.

Although still suffering from some limitations such as the assumed independency between the times to failure necessary for performing the renewal theory derivations, the derivations conducted in this section are more general than what has been previously published. The resulting model can be termed a *knowledge model* [Lapr91] with respect to the reliability growth models which have appeared in the literature, which can be termed *action models.* Support for this terminology, adapted from the Automatic Control theory, lies in the following remarks:

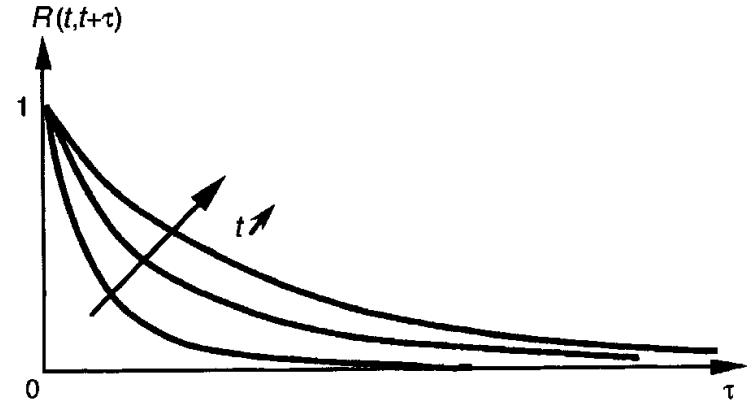


Figure 2.14 Interval reliability.

- The knowledge model allows the various phenomena to be taken into account explicitly and enables a number of properties to be derived; nevertheless, it is too complex in practice and not suitable for predictions.
- The action models, although based on more restrictive assumptions, are simplified models suitable for practical purposes.

The results obtained from the knowledge model thus derived in this section enable the action models reported in the literature to be classified as follows:

1. Models based on the relation between successive failure rates, which can be referred to as *failure rate models;* these models describe the behavior between two failures. Two categories of failure rate models can be distinguished according to the nature of the relationship between the successive failure rates: (1) deterministic relationship, which is the case for most failure rate models; see, for example, [Jeli72, Shoo73, Musa75]; (2) stochastic relationship [Keil83, Litt88]; the corresponding models are known as doubly stochastic reliability growth models in [Mill86].
2. Models based on the failure intensity, thus called *failure intensity models;* these models describe the failure process, and are usually expressed as nonhomogeneous Poisson processes; see, for example, [Crow77, Goel79, Yama83, Musa84, Lapr91].

Most reliability growth models consider reliability growth stricto sensu, without taking into account possible stable reliability or reliability decrease periods: they assume that the failure rate and/or the failure intensity decrease monotically and become asymptotically zero with time. Note, however, that the S-shaped models [Yama83, Tohm89] relate to initial reliability decrease followed by reliability growth, and that the hyperexponential model [Lapr84, Kano87, Lapr91] relates to reliability growth converging toward stable reliability. Finally, it is noteworthy that the models referenced above have been established specifically for software; however, there is no impairment to apply them to hardware [Litt81]; conversely, the Duane's model [Duan64], derived for hardware has been successfully applied to software [Keil83]. Chapter 3 provides a comprehensive survey of these reliability models.

Of prime importance when considering the practical use of the above models is the question of their application to real systems. Failure data can be collected under two forms: (1) times between failures or (2) number of failures per unit of time (failure-count data). Failure rate models are more naturally suited to data in the form of times between failures whereas failure intensity models are more naturally suited to data in

the form of number of failures per unit of time. However, some models accommodate both forms of failure data, such as the logarithmic Poisson model [Musa84] or the hyperexponential model. The collection of data under the form "number of failures per unit of time" is less constraining than the other form, since one does not have to record all failure times; the definition of the unit of time can be varied throughout the life cycle of the system according to the amount of failures experienced, e.g., from a few days to weeks during the development phase, and from a few weeks to months during the operational life.

As action models are based on precise hypotheses (particularly with respect to the reliability trends they can accommodate, as discussed above), it is helpful to process failure data before model application, in order to (1) determine the reliability trend exhibited by the data and (2) select the model(s) whose assumptions are in agreement with the evidenced trend [Kano91b]. Trend tests are given detailed treatment in Chap. 10.

2.4.4 Availability modeling

All the time intervals defined in Sec. 2.4.2 are now considered, i.e., the times to failure and the times to restoration: the $Y_{j,i}$'s, Z_j's, and W_j's are no longer assumed to be zero. For simplicity, it is assumed that the times to restoration after failure are stochastically identical for a given version, i.e., $Y_{j,i} = Y_j$. Let:

- $t''_0 = 0$ denote the considered initial instant (the system is assumed nonfailed).
- n stand for the number of service restorations that took place before instant t.
- t'_n and t''_n, $n = 1,2, \ldots$ be the instants when correct service is no longer delivered and when service is restored, respectively, either:
  - —Upon (respectively after) failure, with $n = \sum_{k=1}^{j} (a_{k-1} + 1) + i$, $j = 1,2, \ldots, i = 1, \ldots a_j$, $a_0 = 0$
  - —Upon (respectively after) stopping the operation of system in order to introduce a modification, with $n = \sum_{k=1}^{j+1} (a_{k-1} + 1), j = 1,2, \ldots,$ $a_0 = 0$
- $f_{Y_j}(t), f_{Z_j}(t), f_{W_j}(t), j = 1,2, \ldots$ denote the probability density functions (pdf's) of the Y_j's, Z_j's, and W_j's, respectively, and $sf_{Z_j}(t)$ the survival function of the Z_j's.
- $\psi_n(t)$ be the pdf of the instants of service restoration, $n = 1,2, \ldots$

Derivation of availability is performed as in the case of reliability, with the pdf of the instants of service restoration $\psi_n(t)$ replacing the pdf

of the instants of failure occurrence $\phi_n(t)$. As in Sec. 2.4.3, we assume that the various time intervals under consideration are stochastically independent, which leads to:

- For $n = \sum_{k=1}^{j} (a_{k-1} + 1) + i$

$$\psi_n(t) = \left(\circledast_{k=1}^{j-1} (f_{X_k}(t) * f_{Y_k}(t))^{*a_k} * (f_{Z_k}(t) * f_{W_k}(t)) \right) * (f_{X_j}(t) * f_{Y_j}(t))^{*i}$$

- For $n = \sum_{k=1}^{j+1} (a_{k-1} + 1)$

$$\psi_n(t) = \circledast_{k=1}^{j} (f_{X_k}(t) * f_{Y_k}(t))^{*a_k} * (f_{Z_k}(t) * f_{W_k}(t))$$

Let us consider the event $E_n = \{t''_n < t < t'_{n+1}\}$, $n = 0,1,2,\ldots$. The event E_n means that exactly n service restorations took place before instant t, and that the system is nonfailed at instant t. The pointwise availability $A(t)$, denoted simply by availability in the following, is then, due to the exclusivity of events E_n:

$$A(t) = \sum_{n=0}^{\infty} P\{E_n\}$$

The probability of event E_n is:

- For $n = \sum_{k=1}^{j} (a_{k-1} + 1) + i : P\{E_n\} = P\{t''_n < t < t + \tau < t''_n + Xj,i + 1\}$

$$P\{E_n\} = \int_0^t P\{x < t''_n < x + dx\}\, P\{Xj,i + 1 > t - x\} = \int_0^t sf_{X_j}(t - x)\, \psi_n(x)\, dx$$

- For $n = \sum_{k=1}^{j+1} (a_{k-1} + 1)$

$$P\{E_n\} = P\{t''_n < t < t + \tau < t''_n + Z_j\} = \int_0^t sf_{Z_j}(t - x)\, \psi_n(x)\, dx$$

The expression of $A(t)$ is then

$$A(t) = sf_{X_1}(t) + \sum_{j=1}^{\infty} \times \left(sf_{X_j}(t) * \sum_{i=0}^{a_j - 1} \psi_{\left(\sum_{k=1}^{j} (a_{k-1} + 1) + i\right)}(t) + sf_{Z_j}(t) * \psi_{\left(\sum_{k=1}^{j+1} (a_{k-1} + 1)\right)}(t) \right)$$

Statistical estimation of the availability of a set of systems is given by the ratio of nonfailed systems at time t to the total number of systems in the set. When field data are related to times to failure and to times to restoration, considering the average availability rather than availability facilitates the estimation process, as the average availability is the expected proportion of time a system is nonfailed (see, for example, [Barl75]). The average availability over $[0,t]$ is defined by

$$A_{av}(t) = \frac{1}{t}\int_0^t A(\tau)\, d\tau$$

Denoting respectively UT_i the observed times where the system is operational, a statistical estimator of $A_{av}(t)$ is given by the following analytical expression

$$\hat{A}_{av}(t) = \frac{1}{t}\sum_{i=1}^{n} UT_i$$

So far, no assumption has been made on the pdf's of the various times considered. To derive properties of the availability, consider the case where:

- A modification takes place after each failure, i.e., $a_j = 1, Z_j = W_j = 0\ \forall j$,
- The $X_{j,i}$'s and the Y_j's are exponentially distributed: $f_{X_j}(t) = \lambda_j \exp(-\lambda_j t)$, $f_{Y_j}(t) = \mu_j \exp(-\mu_j t)$, and that both corresponding piecewise Poisson processes converge toward Poisson processes after r modifications have taken place.

Then the Laplace transform $\tilde{A}(s)$ of availability is

$$\tilde{A}(s) = \sum_{j=1}^{r-1}\left(\frac{1}{\lambda_{j+s}}\right)\prod_{k=1}^{j-1}\left(\frac{\lambda_k}{\lambda_{k+s}}\right)\left(\frac{\mu_k}{\mu_{k+s}}\right) + \frac{1}{s}\,\frac{\mu_r\,\lambda_r}{\lambda_r+\mu_r+s}\left(\frac{1}{\lambda_{r+s}}\right)\prod_{k=1}^{r-1}\left(\frac{\mu_k}{\mu_{k+s}}\right)\left(\frac{\lambda_k}{\lambda_{k+s}}\right)$$

The times to failure are large with respect to the times to restoration, i.e., $\lambda_j/\mu_j \ll 1$. Performing an asymptotic development with respect to λ_j/μ_j for the unavailability $\bar{A}(t) = 1 - A(t)$ leads to

$$\bar{A}(t) = \frac{\lambda_r}{\mu_r} + \sum_{j=1}^{r-1}\alpha_j \exp(-\lambda_j t) - \frac{\lambda_1}{\mu_1}\exp(-\mu_1 t) \qquad (2.17)$$

with $$\alpha_j = \sum_{k=j}^{r-1}\frac{\lambda_k}{\mu_k}\,\frac{\prod_{i=1}^{k-1}\lambda_i}{\prod_{i=1,i\neq j}^{k}(\lambda_i-\lambda_j)} - \frac{\lambda_r}{\mu_r}\prod_{k=1,k\neq j}^{r-1}\frac{\lambda_k}{\lambda_k-\lambda_j}$$

The α_j is linked by the following equation:

$$\sum_{j=1}^{r-1} \alpha_j = \frac{\lambda_1}{\mu_1} - \frac{\lambda_r}{\mu_r}$$

The following properties can be derived from Eq. (2.17), thus confirming and generalizing what had previously been established in [Cost78, Lapr84] through modeling via multistate Markov and semi-Markov chains:

P1. When reliability becomes stable, unavailability becomes constant: $\overline{A}(\infty) \approx \lambda_r/\mu_r$.

P2. If $(\lambda_1/\mu_1) \geq (\lambda_r/\mu_r)$, then there is an overshoot of unavailability in comparison with the asymptotic value (λ_r/μ_r).

P3. There is a single unavailability maximum (availability minimum) if $(\lambda_{j+1}/\mu_{j+1}) \leq (\lambda_j/\mu_j)$, for $j = 1, \ldots$; conversely, local maxima (minima) occur.

P4. If the piecewise Poisson process of the interfailure occurrence times is continuously nonincreasing from λ_1 to λ_r, and if the times to failure are large with respect to the times to restoration, then $\overline{A}_{\max} \approx (\lambda_1/\mu_1)$.

P5. The time to reach the maximum unavailability (minimum availability) is of the order of magnitude of the mean time to restoration $(1/\mu_1)$.

P6. The changes in availability are significantly more influenced by the stochastic changes in the times to failure than by the stochastic changes in the times to restoration, which can thus be assumed as stochastically identical over the system's life.

Figure 2.15 gives the typical shape of system unavailability in the presence of reliability growth.

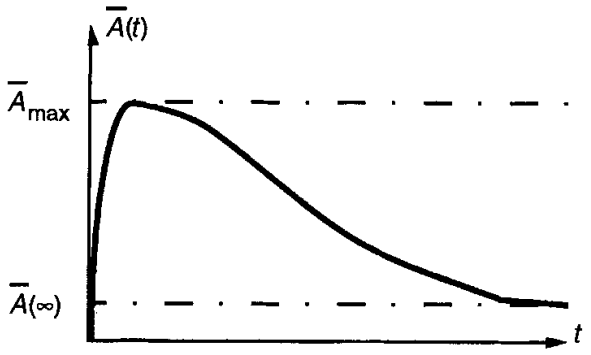


Figure 2.15 Typical system unavailability.

In case of stable reliability, ($\lambda_j = \lambda_1, j = 1, \ldots, r$), assuming $\mu_j = \mu_1$, $j = 1, \ldots, r$, the maximum corresponds to the asymptotic unavailability. Thus we have $\overline{A}(\infty) = \overrightarrow{A}_{max}$.

Although the availability of operational computing systems is usually significantly influenced in the field by reliability growth (for instance, the data displayed in [Wall84] for AT&T's ESS-4 show that unavailability is decreased by a factor of 250 during 3.5 years of operation), the only published *action* model for availability in the presence of reliability growth is the hyperexponential model [Lapr91].

2.5 Situation with Respect to the State of the Art in Reliability Evaluation

Hardware evaluation and software evaluation have followed courses which could hardly have been more distant from each other.

Hardware evaluation has focused on the operational life, placing the emphasis on the influence of the system structure on dependability. A number of significant results have been derived with respect to (1) the role and influence of the coverage of the fault-tolerance strategies and mechanisms [Bour69, Arno73]; (2) the construction [Beya81] and processing [Gros84, Bobb86, Cour77] of large, stiff, Markov models; (3) the definition and evaluation of performance-related measures of dependability, which are usually gathered in the concept of performability [Meye78, Smit88]. These results have been integrated into software packages, such as ARIES, HARP, SAVE, and SURF. As far as elementary data are concerned, reliance has generally been placed on databases such as the MIL-HDBK-217, which are limited to permanent faults, whereas it is currently agreed that temporary faults constitute a major source of failure [Siew92]. In addition, it has largely been ignored that the reliability of hardware parts grows significantly during the whole system's life, as shown, for example, in the experimental data displayed in [Baue85].

Software evaluation has mainly focused on the development phase, and especially on the reliability growth of single-component (black-box) systems. Many models have been proposed (see surveys such as [Rama82, Yama85] and Chap. 3). Less attention has been paid to accounting for the structure of a software system, where most of the corresponding work has been restricted to the failure process, either for non-fault-tolerant [Litt81, Lapr84] or for fault-tolerant software systems [Hech79, Grna80, Lapr84, Arla88]; only our recent work [Lapr91, Kano93a] deals with the evaluation of the reliability growth of a system from the reliability growth of its components. In [Lapr91] a so-called transformation approach based on a Markov interpretation of the hyperexponential reliability growth model is derived: the hyperex-

ponential model is regarded as resulting from the transformation of a traditional Markov model (stable reliability) into another Markov model, which, through a suitable addition of states, enables reliability growth phenomena to be accounted for. This stable reliability–reliability growth transformation—is shown to be applicable to the Markov models of systems made up of components. In particular, it enables the reliability and the availability of systems to be evaluated from the reliability growth of their components. This approach has been applied to model the reliability growth of fault-tolerant software systems (recovery blocks, N-version programming and N self-checking programming; see Chap. 14) in [Kano93a].

From the practical utilization viewpoint, the situation can be summarized as follows:

- Hardware evaluation is fairly well included in the design process; although the estimations are usually carried out more as an adjunct to the design methodology than an integral part of producing an optimized design, predictive evaluations of operational dependability are routinely performed during the design of a system.
- In the vast majority of software developments, evaluation (if any) is extremely limited, and most of the published work devoted to applications on real data is in fact post mortem work.

However, when dealing with the assessment of dependability, the users of computing systems are interested in figures resulting from modeling and evaluation of *systems,* composed of hardware and software, with respect to both *physical and design faults.* This statement may be supported by considering:

1. The requirements for systems in terms of dependability; an example is provided by the requirements for electronic switching systems [Clem87] (Fig. 2.16), in which it is explicitly stated that they apply to both hardware and software, especially in terms of reliability and availability.
2. The sources of failure of computing systems; an example of sources of failures is given in Fig. 2.17, showing results of a survey on the sources of unavailability for electronic switching systems and transaction processing systems [Toy85]. Clearly, an evaluation which would be performed with respect to hardware failures only would not be representative of the actual behavior of the considered systems, and this situation is exacerbated even more when considering (hardware) fault-tolerant systems, where software is the dependability bottleneck, as it contributes to more than half the system failures [Gray90].

The hardware and software for switching systems must be designed to meet the requirements shown in the table below.

Operation	• Continuous (20 years)
Time shared channels	• Thousands
Recovery time	• Critical
Value of reliability	• High 1 call per 100,000 call cut off
Downtime	• <3 minutes/year
Synchronization	• Network sync
System growth Database changes Program changes Maintenance	All must be done with the system on-line and operational

Figure 2.16 Requirements for ESSs.

Faced with these user requirements, the evaluations of the system's dependability in terms of hardware and software are not common practice compared to the large amount of evaluations considering only hardware. Hardware and software evaluations generally address stable reliability [Rohn72, Avey80, Angu82, Star87, Pign88, Duga94], with a few exceptions accounting for reliability growth [Cost78, Lapr91]. Evaluations dedicated to reliability growth are based on either multistate Markov modeling [Cost78] or on Markov modeling combined with the above-mentioned transformation approach [Lapr91]. For evaluations related to stable reliability, various modeling techniques are used: semi-Markov modeling [Star87] or Markov modeling used either (1) alone [Rohn72, Avey80, Angu82] or (2) together with other techniques such as fault tree analysis [Duga94b] or block diagrams [Pign88].

In addition, standardization bodies and regulating agencies are increasingly aware of the need to perform evaluations encompassing

Unavailability sources	Electronic switching	Transaction processing
Hardware failures	20%	40%
Environment		5%
Software failures	15%	30%
Recovery deficiencies	35%	
Incorrect procedures	30%	20%
Others, such as updating		5%

Figure 2.17 Sources of failures.

Studies have been conducted for ESA into software reliability requirements for ESA Space programmes. These studies, conducted in 1986, concluded:

a) Software failure is a process that appears to the observer to be "random", therefore the term "reliability" is meaningful when applied to a system which includes software, and the process can be modeled as stochastic.
b) The definition of "Reliability" is identical for a system which includes software as it is for a purely hardware system. It is the classical definition: "the probability of successful operation for a given period under given conditions".
c) The specification of numerical levels of reliability for a complete system is meaningless unless the reliability of the software which it contains is similarly quantified and the level of reliability achieved by that software is verified.

Figure 2.18 Statement from ESA.

both hardware and software. Two such examples are given in Figs. 2.18 and 2.19. The example in Fig. 2.18 is extracted from an invitation to tender from ESA, the European Space Agency [ESA88], and reports the conclusions derived from the studies conducted for ESA in 1986. The example in Fig. 2.19, extracted from the British Standard [BStd86] (Part 4) is more general since it addresses all types of systems containing software. Unfortunately, despite the existence of such recommendations made several years ago, the specification of software reliability is far from being a common activity.

Clearly, the results presented in this chapter show that the current limitation to the practicality of dependability evaluations of hardware

Specification of reliability for systems containing software (Section 3.1.7)

The mechanisms for the specification of reliability requirements, from the point of view of the failure of an item to perform its function, should be no different for systems containing software than for any other. It is imperative that there should be no discrimination between failures due to logical faults in the system from those due to physical breakdown. The following areas, however, require special consideration:

a) In the prediction of reliability, allowance should be made for the contribution made by logical errors to the overall unreliability of the system. At the moment, prediction of such a contribution, particularly before testing begins, is neither accurate nor well understood. Experience from similar systems is the most likely source of such data.
b) The contribution of logical errors to system unavailability should be considered. Although there is no 'repair' in the strict sense, depending upon the design, considerable time may elapse after a after a failure before operation restart.
c) The principal area in which failures due to logical errors differ from physical failure is that such failures do not require spares, routine maintenance, and repair resources generally required for hardware. However, the inclusion of software in the system may indicate the need for a software facility and requirement for this facility should be separately considered.

Figure 2.19 Recommendations from British Standards.

and software systems is not due to theoretical impediments. The lack of credibility of *predictive* software reliability evaluations lies therefore in the capabilities of the current reliability growth models—the action models in our terminology—with regard to the failure data upon which their predictions are based. When the predictions and failure data are homogeneous, they apply to the same phase of the system life (either development or operation) and are commensurate; then the application of those models is greatly facilitated, and the evaluations done are meaningful. The problem lies in heterogeneous situations, typically predicting operational dependability from failure data collected during development. The current models, which are fundamentally performing extrapolations, then fall out (a typical situation is that the software will exhibit a reliability in operation that is hopefully much better than predicted from failure data collected during its development), and new approaches are needed, such as those proposed in (1) [Lapr92c] via the so-called product-in-a-process approach, aimed at enhancing the reliability prediction for a given software when exploiting field data collected for former, similar software or (2) [Haml93] via the amplification of software reliability testing.

2.6 Summary

In this chapter, we addressed the problem of software reliability and system reliability. Following our discussion of the dependability concept, we dealt with system behavior up to the (next) failure. We also focused on the sequence of failures when considering restoration actions consecutive to various forms of maintenance. The assumptions made for derivation purposes were carefully stated and analyzed, and we commented at length on the results obtained in order to relate them to the existing body of results. Furthermore, we described a generalization of the classical, hardware-oriented, reliability theory in order to incorporate software as well. This chapter is then concerned with reliability growth phenomena; since most published material on reliability growth is software-oriented, this chapter can be regarded as a generalization to include hardware as well. Finally, we were devoted to position the results obtained with respect to the state of art in dependability evaluation, considering both hardware and software.

Problems

2.1 Some illustrative examples of sequences fault–error–failure are given in Sec. 2.2.2. Considering physical and design faults, give similarities and differences with respect to fault creation mode and manifestation mode.

2.2 Give other examples of the sequence fault–error–failure.

2.3 In Sec. 2.3.1, a general formulation of the reliability $R(t)$ as a function of the distribution of the failure rate λ ($g_c(\lambda)$) is given. It is stated that a gamma distribution for λ leads to a Pareto distribution for $R(t)$. Demonstrate this result.

2.4 Considering a multi-interpreter system viewed as a hierarchy of interpreters, Eq. (2.10) gives the relationship between the failure rate of the system and the failure rates of its interpreters. Demonstrate this result.

2.5 The results of this chapter show that the same approaches can be applied to hardware and software systems; for those who are familiar with Markov modeling, construct the Markov chain of a multi-interpreter system with a hardware and a software interpreters. The hardware interpreter is made of two hardware redundant components (failure rate of a component λ_h, repair rate μ) and the software interpreter is made of one component (failure rate λ_s, restart rate δ). Derive the reliability of the system.

2.6 Section 2.4.3 gives the expression of the Laplace transform of the failure intensity, $\tilde{h}(s)$, related to the sequence of failure rates presented in Fig. 2.11. Assuming operation restart after correction only ($a_j = 1 \ \forall \ j$), derive the equation given the temporal expression of the failure intensity function $h(t)$.

2.7 As in the previous exercise, assuming operation restart after correction only ($a_j = 1 \ \forall \ j$), derive the expression of the conditional reliability, $R_n(t)$ from Eq. (2.14), and derive the plot of $R(t)$ (Fig. 13).

2.8 As in the previous exercises, assuming operation restart after correction only ($a_j = 1 \ \forall \ j$), derive the expression of the interval reliability, $R(t, t + \tau)$ (Eq. (2.16) and Fig. 14).

2.9 Considering the following sequence of failure rates,

$$\lambda_0 = \lambda, \lambda_1 = [(n-1)/n]\ \lambda, \lambda_2 = [(n-2)/n]\ \lambda \ldots \lambda_{n-1} = [1/n]\ \lambda, \lambda_n = 0$$

derive the expressions of the failure intensity, the conditional reliability and the interval reliability.