UNIVERSITY OF
NEWCASTLE

Fundamental Concepts of Dependability

Algirdas Avižienis
UCLA Computer Science Dept.
Univ. of California, Los Angeles
USA

Jean-Claude Laprie
LAAS-CNRS
Toulouse
France

Brian Randell
Dept. of Computing Science
Univ. of Newcastle upon Tyne
U.K.

UCLA CSD Report no. 010028

LAAS Report no. 01-145

Newcastle University Report no.
CS-TR-739

Abstract *Dependability is the system property that integrates such attributes as reliability, availability, safety, security, survivability, maintainability. The aim of the presentation is to summarize the fundamental concepts of dependability. After a historical perspective, definitions of dependability are given. A structured view of dependability follows, according to a) the threats, i.e., faults, errors and failures, b) the attributes, and c) the means for dependability, that are fault prevention, fault tolerance, fault removal and fault forecasting.*

The protection and survival of complex information systems that are embedded in the infrastructure supporting advanced society has become a national and world-wide concern of the highest priority [1]. Increasingly, individuals and organizations are developing or procuring sophisticated computing systems on whose services they need to place great reliance — whether to service a set of cash dispensers, control a satellite constellation, an airplane, a nuclear plant, or a radiation therapy device, or to maintain the confidentiality of a sensitive data base. In differing circumstances, the focus will be on differing properties of such services — e.g., on the average real-time response achieved, the likelihood of producing the required results, the ability to avoid failures that could be catastrophic to the system's environment, or the degree to which deliberate intrusions can be prevented. The notion of dependability provides a very convenient means of subsuming these various concerns within a single conceptual framework.

Our goal is to present a concise overview of the concepts, techniques and tools that have evolved over the past forty years in the field of dependable computing and fault tolerance.

ORIGINS AND INTEGRATION OF THE CONCEPTS

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

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

The first generation of electronic computers (late 1940’s to mid-50’s) used rather unreliable components, therefore practical techniques were employed to improve their reliability, such as error control codes, duplexing with comparison, triplication with voting, diagnostics to locate failed components, etc. At the same time J. von Neumann, E. F. Moore and C. E. Shannon, and their successors developed theories of using redundancy to build reliable logic structures from less reliable components, whose faults were masked by the presence of multiple redundant components. The theories of masking redundancy were unified by W. H. Pierce as the concept of *failure tolerance* in 1965 (Academic Press). In 1967, A. Avizienis integrated masking with the practical techniques of error detection, fault diagnosis, and recovery into the concept of fault-tolerant systems [2]. In the reliability modeling field, the major event was the introduction of the coverage concept by Bouricius, Carter and

Schneider [3]. Seminal work on software fault tolerance was initiated by B. Randell [4], later it was complemented by N-version programming [5].

The formation of the *IEEE-CS TC on Fault-Tolerant Computing* in 1970 and of *IFIP WG 10.4 Dependable Computing and Fault Tolerance* in 1980 accelerated the emergence of a consistent set of concepts and terminology. Seven position papers were presented in 1982 at FTCS-12 in a special session on fundamental concepts of fault tolerance, and J.-C. Laprie formulated a synthesis in 1985 [6]. Further work by members of IFIP WG 10.4, led by J.-C. Laprie, resulted in the 1992 book *Dependability: Basic Concepts and Terminology* (Springer-Verlag), in which the English text was also translated into French, German, Italian, and Japanese.

In this book, intentional faults (malicious logic, intrusions) were listed along with accidental faults (physical, design, or interaction faults). Exploratory research on the integration of fault tolerance and the defenses against deliberately malicious faults, i.e., security threats, was started in the mid-80's [7-9]. The first IFIP Working Conference on Dependable Computing for Critical Applications was held in 1989. This and the six Working Conferences that followed fostered the interaction of the dependability and security communities, and advanced the integration of security (confidentiality, integrity and availability) into the framework of dependable computing [10].

THE DEFINITIONS OF DEPENDABILITY

A systematic exposition of the concepts of dependability consists of three parts: the **threats** to, the **attributes** of, and the **means** by which dependability is attained, as shown in Figure 1.

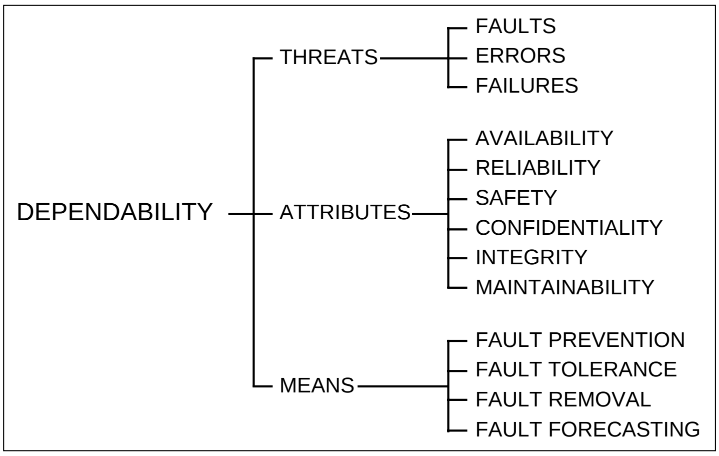

Figure 1 - The dependability tree

Computing systems are characterized by five fundamental properties: functionality, usability, performance, cost, and dependability. **Dependability** of a computing system is the ability to deliver service that can justifiably be trusted. The **service** delivered by a system is its behavior as it is perceived by its user(s); a **user** is another system (physical, human) that interacts with the former at the **service interface**. The **function** of a system is what the system is intended to do, and is described by the functional specification. **Correct service** is delivered when the service implements the system function. A system **failure** is an event that occurs when the delivered service deviates from correct service. A failure is thus a transition from correct service to **incorrect service**, i.e., to not implementing the system function. The delivery of incorrect service is a system **outage**. A transition from incorrect service to correct service is **service restoration**. Based on the definition of failure, an alternate definition of **dependability**, which complements the initial definition in providing a criterion for adjudicating whether the delivered service can be trusted or not: the ability of a system to avoid failures that are more frequent or more severe, and outage durations that are longer, than is acceptable to the user(s). In the opposite case, the system is no longer dependable: it suffers from a dependability failure, that is a **meta-failure**.

THE THREATS: FAULTS, ERRORS, AND FAILURES

A system may fail either because it does not comply with the specification, or because the specification did not adequately describe its function. An **error** is that part of the system state that may cause a subsequent failure: a failure occurs when an error reaches the service interface and alters the service. A **fault** is the adjudged or hypothesized cause of an error. A fault is **active** when it produces an error; otherwise it is **dormant**.

A system does not always fail in the same way. The ways in which a system can fail are its **failure modes**. These can be ranked according to **failure severities**. The modes characterize incorrect service according to four viewpoints:

- the failure domain,
- the controllability of failures,
- the consistency of failures, when a system has two or more users,
- the consequences of failures on the environment.

Figure 2 shows the modes of failures according to the above viewpoints, as well as failure symptoms which result from the combination of the domain, controlability and consistency viewpoints. The failure symptoms can be mapped into the failure severities as resulting from grading the consequences of failures.

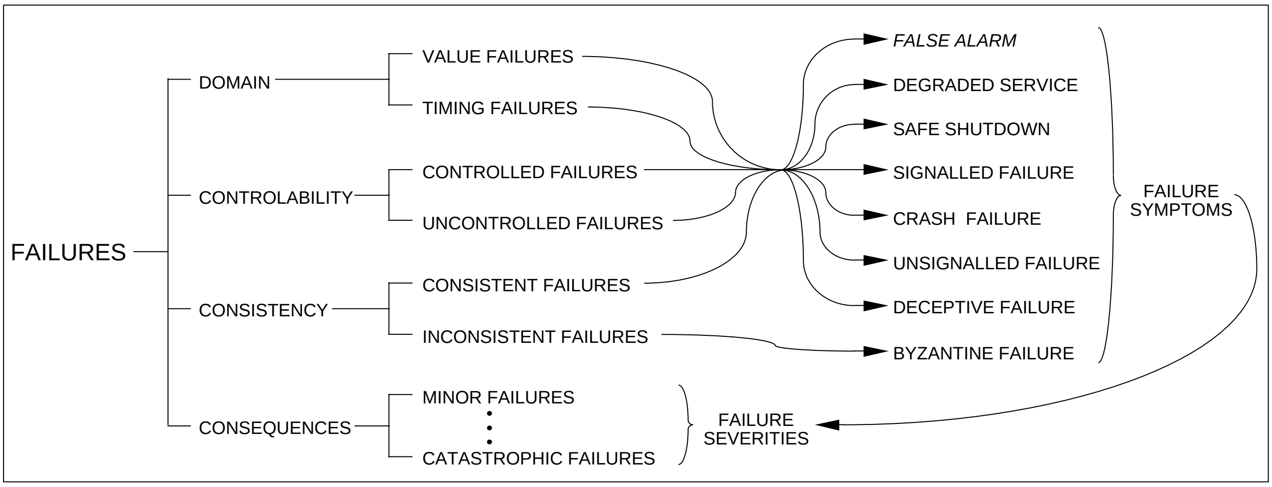

Figure 2 - The failure modes

A system consists of a set of interacting components, therefore the system state is the set of its component states. A fault originally causes an error within the state of one (or more) components, but system failure will not occur as long as the error does not reach the service interface of the system. A convenient classification of errors is to describe them in terms of the component failures that they cause, using the terminology of Figure 2: value vs. timing errors; consistent vs. inconsistent ('Byzantine') errors when the output goes to two or more components; errors of different severities:

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

Faults and their sources are very diverse. Their classification according to six major criteria is presented in Figure 3. It could be argued that introducing *phenomenological causes* in the classification criteria of faults may lead recursively to questions such as ‘why do programmers make mistakes?’, ‘why do integrated circuits fail?’ Fault is a concept that serves to stop recursion. Hence the definition given: *adjudged or hypothesized* cause of an error. This cause may vary depending upon the viewpoint that is chosen: fault tolerance mechanisms, maintenance engineer, repair shop, developer, semiconductor physicist, etc.

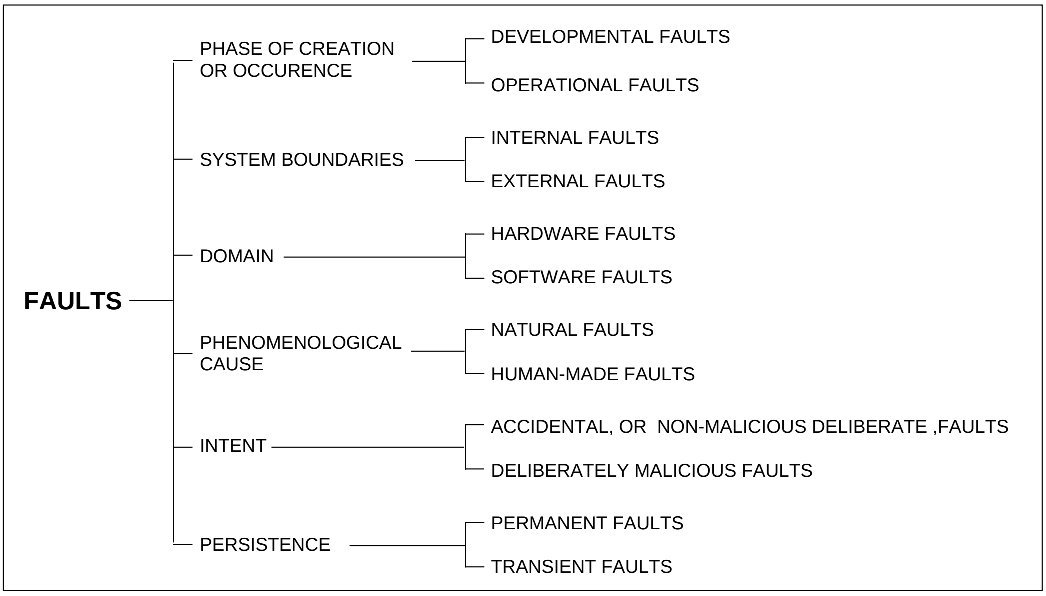

Figure 3 - Elementary fault classes

Combining the elementary fault classes of figure 3 leads to the tree of the upper part of figure 4. The leaves of the tree are gathered into three major fault classes for which defenses need to be devised: **design faults**, **physical faults**, **interaction faults**. The boxes of figure 4 point at generic illustrative fault classes.

Non-malicious deliberate faults can arise during either development or operation. During development, they result generally from tradeoffs, either a) aimed at preserving acceptable performance and facilitating system utilization, or b) induced by economic considerations; such faults can be sources of security breaches, in the form of *covert channels*. Non-malicious deliberate interaction faults may result from the action of an operator either aimed at overcoming an unforeseen situation, or deliberately violating an operating procedure without having realized the possibly damaging consequences of his or her action. Non-malicious deliberate faults share the property that often it is recognized that they were faults only *after* an unacceptable system behavior, thus a failure, has ensued; the specifier(s), designer(s), implementer(s) or operator(s) did not realize that the consequence of some decision of theirs was a fault.

Malicious faults fall into two classes: a) **malicious logics** [11], that encompass developmental faults such as *Trojan horses*, logic or timing *bombs*, and *trapdoors*, as well as operational faults (with respect to the given system) such as *viruses* or *worms*, and b) **intrusions**. There are interesting and obvious similarities between an intrusion that exploits an internal fault and a physical external fault that 'exploits' a lack of shielding. It is in addition noteworthy that a) the external character of intrusions does not exclude the possibility that they may be attempted by system operators or administrators who are exceeding their rights, and that b) intrusions may use physical means to cause faults: power fluctuation, radiation, wire-tapping, etc.

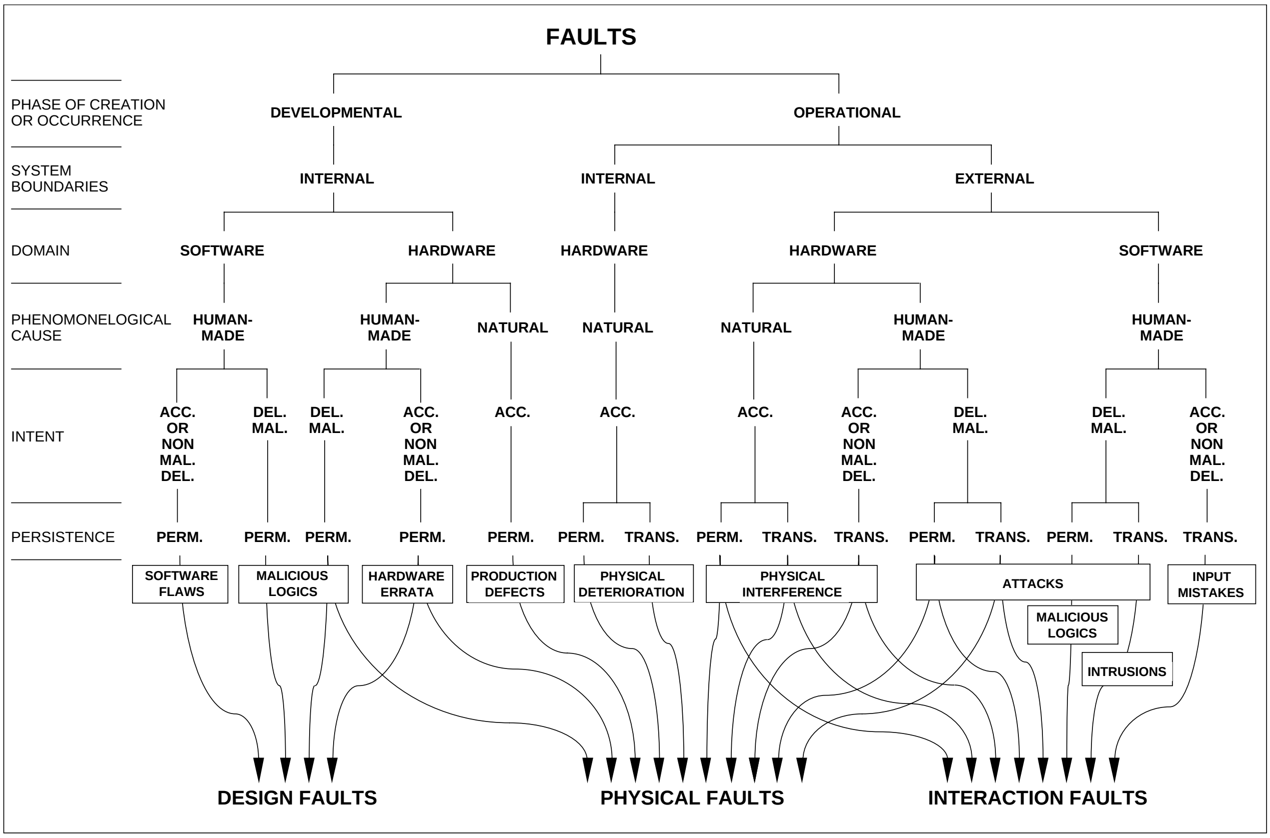

Figure 4 - Combined fault classes

Some design faults affecting software can cause so-called **software aging**, i.e., progressively accrued error conditions resulting in performance degradation or activation of elusive faults. Examples are memory bloating and leaking, unreleased file-locks, storage space fragmentation.

The relationship between faults, errors, and failures is addressed in Appendix 1.

Two final comments about the words, or *labels*, 'fault', 'error', and 'failure':

a) though we have chosen to use just these terms in this document, and to employ adjectives to distinguish different kinds of faults, errors and failures, we recognize the potential convenience of using words that designate, briefly and unambiguously, a specific class of threats; this is especially applicable to faults (e.g. bug, flaw, defect, deficiency, erratum) and to failures (e.g. breakdown, malfunction, denial-of-service);

b) the semantics of the terms fault, error, failure reflect current usage: i) fault prevention, tolerance, and diagnosis, ii) error detection and correction, iii) failure rate, failure mode.

THE ATTRIBUTES OF DEPENDABILITY

Dependability is an integrative concept that encompasses the following basic attributes:

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

Depending on the application(s) intended for the system, different emphasis may be put on different attributes. The description of the required goals of the dependability attributes in terms of the acceptable frequency and severity of the failure modes, and of the corresponding acceptable outage durations (when relevant), for a stated set of faults, in a stated environment, is the **dependability requirement** of the system.

Several other dependability attributes have been defined that are either combinations or specializations of the six basic attributes listed above. **Security** is the concurrent existence of a) availability for authorized users only, b) confidentiality, and c) integrity with 'improper' taken as meaning 'unauthorized'. Characterizing a system reaction to faults, is of special interest, via ,e.g., **robustness**, i.e. dependability with respect to erroneous inputs.

The attributes of dependability may be emphasized to a greater or lesser extent depending on the application: availability is always required, although to a varying degree, whereas reliability, safety, confidentiality may or may not be required. The extent to which a system possesses the attributes of dependability should be interpreted in a relative, probabilistic, sense, and not in an absolute, deterministic sense: due to the unavoidable presence or occurrence of faults, systems are never totally available, reliable, safe, or secure.

Integrity is a prerequisite for availability, reliability and safety, but may not be so for confidentiality (for instance attacks via covert channels or passive listening can lead to a loss of confidentiality, without impairing integrity). The definition given for integrity – absence of improper system state alterations – extends the usual definition as follows: (a) when a system implements an authorization policy, 'improper' encompasses 'unauthorized'; (b) 'improper alterations' encompass actions resulting in preventing (correct) upgrades of information; (c) 'system state' encompasses hardware modifications or damages. The definition given for maintainability goes beyond corrective and preventive maintenance, and encompasses the forms of maintenance aimed at adapting or perfecting the system.

Security has not been introduced as a single attribute of dependability. This is in agreement with the usual definitions of security, which view it as a composite notion, namely 'the combination of (1) confidentiality (the prevention of the unauthorized disclosure of information), (2) integrity (the prevention of the unauthorized amendment or deletion of information), and (3) availability (the prevention of the unauthorized withholding of information)' [12]. A single definition for **security** could be: the absence of unauthorized access to, or handling of, system state.

Besides the attributes defined at the beginning of the section, and discussed above, other, *secondary*, attributes can be defined. An example of specializing secondary attribute is **robustness**, i.e. dependability with respect to external faults, that characterizes a system reaction to a specific class of faults. The notion of secondary attributes is especially relevant for security, when we distinguish among various types of information. Examples of such secondary attributes are:

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

Variations in the emphasis on the different attributes of dependability directly affect the appropriate balance of the techniques (fault prevention, tolerance, removal and forecasting) to be employed in order to make the resulting system dependable. This problem is all the more difficult as some of the attributes conflict (e.g. availability and safety, availability and security), necessitating design trade-offs.

Other concepts similar to dependability exist, as survivability and trustworthiness. They are presented in Appendix 2.

THE MEANS TO ATTAIN DEPENDABILITY

The development of a dependable computing system calls for the combined utilization of a set of four techniques:

- **fault prevention**: how to prevent the occurrence or introduction of faults,
- **fault tolerance**: how to deliver correct service in the presence of faults,
- **fault removal**: how to reduce the number or severity of faults,
- **fault forecasting**: how to estimate the present number, the future incidence, and the likely consequences of faults.

The concepts relating to those techniques are presented in this section. A brief state-of-the-art is presented in Appendix 3.

Fault Prevention

Fault prevention is attained by quality control techniques employed during the design and manufacturing of hardware and software. They include structured programming, information hiding, modularization, etc., for software, and rigorous design rules for hardware. Shielding, radiation hardening, etc., intend to prevent operational physical faults, while training, rigorous procedures for maintenance, ‘foolproof’ packages, intend to prevent interaction faults. Firewalls and similar defenses intend to prevent malicious faults.

Fault Tolerance

Fault tolerance is intended to preserve the delivery of correct service in the presence of active faults. It is generally implemented by error detection and subsequent system recovery.

Error detection originates an error signal or message within the system. An error that is present but not detected is a **latent** error. There exist two classes of error detection techniques:

- **concurrent error detection**, which takes place during service delivery,

- **preemptive error detection**, which takes place while service delivery is suspended; it checks the system for latent errors and dormant faults.

Recovery transforms a system state that contains one or more errors and (possibly) faults into a state without detected errors and faults that can be activated again. Recovery consists of error handling and fault handling. **Error handling** eliminates errors from the system state. It may take three forms:

- **rollback**, where the state transformation consists of returning the system back to a saved state that existed prior to error detection; that saved state is a **checkpoint**,
- **compensation**, where the erroneous state contains enough redundancy to enable error elimination,
- **rollforward**, where the state without detected errors is a new state.

Fault handling prevents located faults from being activated again. Fault handling involves four steps:

- **fault diagnosis**, which identifies and records the cause(s) of error(s), in terms of both location and type,
- **fault isolation**, which performs physical or logical exclusion of the faulty components from further participation in service delivery, i.e., it makes the fault dormant,
- **system reconfiguration**, which either switches in spare components or reassigns tasks among non-failed components,
- **system reinitialization**, which checks, updates and records the new configuration and updates system tables and records.

Usually, fault handling is followed by corrective maintenance that removes faults isolated by fault handling. The factor that distinguishes fault tolerance from maintenance is that maintenance requires the participation of an external agent.

Systematic usage of compensation may allow recovery without explicit error detection. This form of recovery is called **fault masking**. However, such simple masking will conceal a possibly progressive and eventually fatal loss of protective redundancy; so, practical implementations of masking generally involve error detection (and possibly fault handling).

Preemptive error detection and handling (often called BIST: built-in self-test), possibly followed by fault handling is performed at system power up. It also comes into play during operation, under various forms such as spare checking, memory scrubbing, audit programs, or so-called software rejuvenation, aimed at removing the effects of software aging before they lead to failure.

Systems that are designed and implemented so that they fail only in specific modes of failure described in the dependability requirement and only to an acceptable extent, are **fail-controlled systems**, e.g., with stuck output as opposed to delivering erratic values, silence as opposed to babbling, consistent as opposed to inconsistent failures. A system whose failures are to an acceptable extent halting failures only, is a **fail-halt**, or **fail-silent**, **system**. A system whose failures are, to an acceptable extent, all minor ones is a **fail-safe system**.

The choice of error detection, error handling and fault handling techniques, and of their implementation, is directly related to the underlying fault assumption. The classes of faults that can actually be tolerated depend on the fault assumption in the development process. A widely-used method of fault tolerance is to perform multiple computations in multiple channels, either sequentially or concurrently. When tolerance of operational physical faults is required, the channels may be of identical design, based on the assumption that hardware components fail independently. Such an approach has proven to be adequate for elusive design faults, via rollback, however it is not suitable for the tolerance of solid design faults, which necessitates that the channels implement the same function via separate designs and implementations, i.e., through **design diversity** [4,5].

Fault tolerance is a recursive concept: it is essential that the mechanisms that implement fault tolerance should be protected against the faults that might affect them. Examples are voter replication, self-checking checkers, 'stable' memory for recovery programs and data, etc. Systematic introduction of fault tolerance is facilitated by the addition of support systems specialized for fault tolerance such as software monitors, service processors, dedicated communication links.

Fault tolerance is not restricted to accidental faults. Some mechanisms of error detection are directed towards both malicious and accidental faults (e.g. memory access protection techniques) and schemes have been proposed for the tolerance of both intrusions and physical faults, via information fragmentation and dispersal [8], as well as for tolerance of malicious logic, and more specifically of viruses [9], either via control flow checking, or via design diversity.

Fault Removal

Fault removal is performed both during the development phase, and during the operational life of a system.

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

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

Verification techniques can be classified according to whether or not they involve exercising the system. Verifying a system without actual execution is **static verification**, via static analysis (e.g., inspections or walk-through), model-checking, theorem proving. Verifying a system through exercising it constitutes **dynamic verification**; the inputs supplied to the system can be either symbolic in the case of **symbolic execution**, or actual in the case of verification testing, usually simply termed **testing**. An important aspect is the verification of fault tolerance mechanisms, especially a) formal static verification, and b) testing that necessitates faults or errors to be part of the test patterns, a technique that is usually referred to as **fault injection**. Verifying that the system cannot do more than what is specified is especially important with respect to what the system should not do, thus with respect to safety and security. Designing a system in order to facilitate its verification is termed **design for**

verifiability. This approach is well-developed for hardware with respect to physical faults, where the corresponding techniques are termed design for testability.

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

Fault Forecasting

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

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

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

The evolution of dependability over a system's life-cycle is characterized by the notions of stability, growth, and decrease, that can be stated for the various attributes of dependability. These notions are illustrated by **failure intensity**, i.e., the number of failures per unit of time. It is a measure of the frequency of system failures, as noticed by its user(s). Failure intensity typically first decreases (reliability growth), then stabilizes (stable reliability) after a certain period of operation, then increases (reliability decrease), and the cycle resumes.

The alternation of correct-incorrect service delivery is quantified to define reliability, availability and maintainability as measures of dependability:

- **reliability**: a measure of the continuous delivery of correct service — or, equivalently, of the time to failure,
- **availability:** a measure of the delivery of correct service with respect to the alternation of correct and incorrect service,
- **maintainability**: a measure of the time to service restoration since the last failure occurrence, or equivalently, measure of the continuous delivery of incorrect service,
- **safety** is an extension of reliability: when the state of correct service and the states of incorrect service due to non-catastrophic failure are grouped into a safe state (in the sense of being free from

catastrophic damage, not from danger), **safety** is a measure of continuous safeness, or equivalently, of the time to catastrophic failure; safety is thus reliability with respect to catastrophic failures.

Generally, a system delivers several services, and there often are two or more modes of service quality, e.g. ranging from full capacity to emergency service. These modes distinguish less and less complete service deliveries. Performance-related measures of dependability are usually subsumed into the notion of **performability**.

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

When evaluating fault-tolerant systems, the effectiveness of error and fault handling mechanisms, i.e. their coverage, has a drastic influence on dependability measures [3]. The evaluation of coverage can be performed either through modeling or through testing, i.e. **fault injection**.

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

REFERENCES

1. A. Jones, "The challenge of building survivable information-intensive systems", *IEEE Computer,* Vol. 33, No. 8, August 2000, pp. 39-43.
2. A. Avizienis, "Design of fault-tolerant computers", *Proc. 1967 AFIPS Fall Joint Computer Conf.*, AFIPS Conf. Proc. Vol. 31, pp. 733-743, 1967.
3. W.G.Bouricius, W.C. Carter, and P.R. Schneider "Reliability modeling techniques for self-repairing computer systems", *Proc. 24th National Conference of ACM*, 1969, pp. 295-309.
4. B. Randell, "System structure for software fault tolerance", *IEEE Transctions on Software Engineering,* Vol. SE-1, No. 10, June 1975, pp. 1220-232.
5. A. Avizienis and L. Chen, "On the implementation of N-version programming for software fault tolerance during execution", *Proc. IEEE COMPSAC 77*, November 1977, pp. 149-155.
6. J.C. Laprie, "Dependable computing and fault tolerance: concepts and terminology", *Proc. 15th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-15),* Ann Arbor, Michigan, June 1985, pp. 2-11.
7. J.E. Dobson and B. Randell, "Building reliable secure computing systems out of unreliable insecure compnents", *Proc. 1986 IEEE Symp. Security and Privacy*, Oakland, Calif., April 1986, pp. 187-193.
8. J.M. Fray, Y. Deswarte, D. Powell, "Intrusion tolerance using fine-grain fragmentation-scattering", *Proc. 1986 IEEE Symp. Security and Privacy*, Oakland, Calif., April 1986, pp. 194-201.
9. M.K. Joseph and A. Avizienis "A fault tolerance approach to computer viruses", *Proc. 1988 IEEE Symposium on Security and Privacy*, Oakland, Calif., April 1986, pp. 52-58.

10. A. Avizienis, J.-C. Laprie, and B. Randell, "Dependability of computer systems: Fundamental concepts, terminology, and examples", LAAS Report No. , UCLA Report No. , Newcastle No. , October 2000.

11. C.E. Landwehr et al, "A taxonomy of computer program security flaws", *ACM Computing Surveys*, Vol. 26, No. 3, September 1994, pp.211-254.

12. *Information Technology Security Evaluation Criteria (ITSEC),* Commission of The European Communities, Office for Official Publications of the European Communities, June 1991.

APPENDIX 1– THE PATHOLOGY OF FAILURE: RELATIONSHIP BETWEEN FAULTS, ERRORS AND FAILURES

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

1) A fault is **active** when it produces an error, otherwise it is **dormant**. An active fault is either a) an internal fault that was previously dormant and that has been activated by the computation process or environmental conditions, or b) an external fault. **Fault activation** is the application of an input (the activation pattern) to a component that causes a dormant fault to become active. Most internal faults cycle between their dormant and active states.
2) Error propagation within a given component (i.e., *internal* propagation) is caused by the computation process: an error is successively transformed into other errors. Error propagation from one component (C1) to another component (C2) that receives service from C1 (i.e., *external* propagation) occurs when, through internal propagation, an error reaches the service interface of component C1. At this time, service delivered by C2 to C1 becomes incorrect, and the ensuing failure of C1 appears as an external fault to C2 and propagates the error into C2. The presence of an error within a system can arise from the a) activation of an internal fault, previously dormant, b) occurrence of a physical operational fault, either internal or external, or c) propagation of an error from another system (interacting with the given system) via the service interface, that is an **input error**. A failure occurs when an error is propagated to the service interface and unacceptably alters the service delivered by the system.
3) A failure occurs when an error is propagated to the service interface and unacceptably alters the service delivered by the system. A failure of a component causes a permanent or transient fault in the system that contains the component. Failure of a system causes a permanent or transient external fault for the other system(s) that interact with the given system.

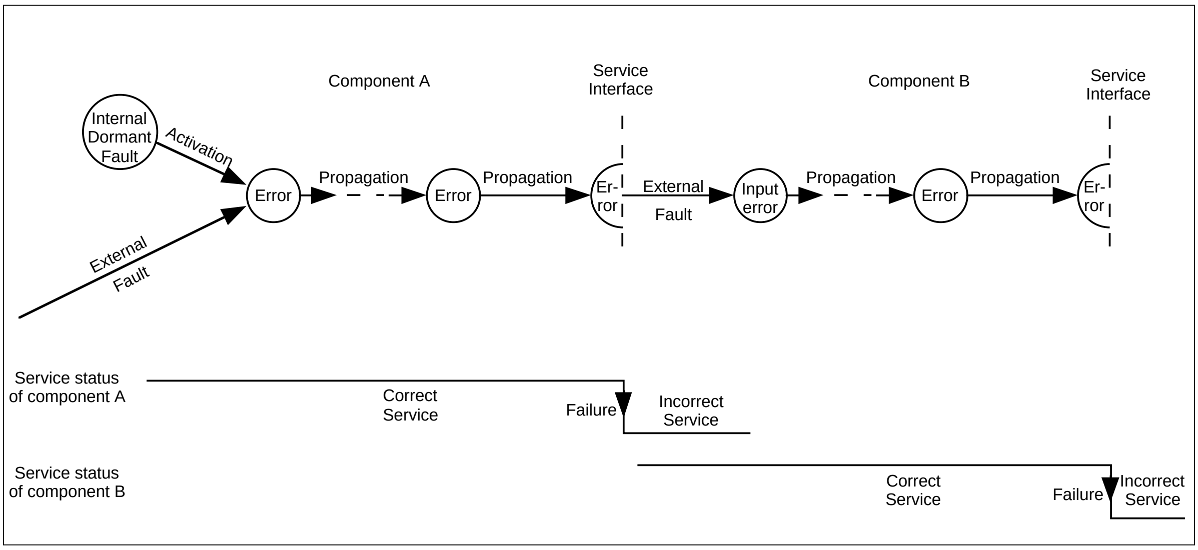

Figure A - Error propagation

These mechanisms enable the ‘fundamental chain’ to be completed, as indicated by figure B.

$$\cdots \longrightarrow \text{fault} \xrightarrow{\textit{activation}} \text{error} \xrightarrow{\textit{propagation}} \text{failure} \xrightarrow{\textit{causation}} \text{fault} \longrightarrow \cdots$$

Figure B - The fundamental chain of dependability threats

The arrows in this chain express a causality relationship between faults, errors and failures. They should be interpreted generically: by propagation, several errors can be generated before a failure occurs.

Some illustrative examples of fault pathology are given in Figure C. From those examples, it is easily understood that fault dormancy may vary considerably, depending upon the fault, the given system's utilization, etc.

- A short circuit occurring in an integrated circuit is a *failure* (with respect to the function of the circuit); the consequence (connection stuck at a Boolean value, modification of the circuit function, etc.) is a *fault* that will remain dormant as long as it is not activated. Upon activation (invoking the faulty component and uncovering the fault by an appropriate input pattern), the fault becomes active and produces an error, which is likely to propagate and create other errors. If and when the propagated error(s) affect(s) the delivered service (in information content and/or in the timing of delivery), a failure occurs.
- The result of an error by a programmer leads to a failure to write the correct instruction or data, that in turn results in a *(dormant) fault* in the written software (faulty instruction(s) or data); upon activation (invoking the component where the fault resides and triggering the faulty instruction, instruction sequence or data by an appropriate input pattern) the fault becomes *active* and produces an error; if and when the error affects the delivered service (in information content and/or in the timing of delivery), a *failure* occurs. This example is not restricted to accidental faults: a logic bomb is created by a malicious programmer; it will remain dormant until activated (e.g. at some predetermined date); it then produces an error that may lead to a storage overflow or to slowing down the program execution; as a consequence, service delivery will suffer from a so-called denial-of-service.
- The result of an error by a specifier' leads to a failure to describe a function, that in turn results in a *fault* in the written specification, e.g. incomplete description of the function. The implemented system therefore does not incorporate the missing (sub-)function. When the input data are such that the service corresponding to the missing function should be delivered, the actual service delivered will be different from expected service, i.e., an *error* will be perceived by the user, and a *failure* will thus occur.
- An inappropriate human-system interaction performed by an operator during the operation of the system is an external *fault* (from the system viewpoint); the resulting altered processed data is an *error;* etc.
- An error in reasoning leads to a maintenance or operating manual writer's failure to write correct directives, that in turn results in a *fault* in the corresponding manual (faulty directives) that will remain dormant as long as the directives are not acted upon in order to address a given situation, etc.

Figure C - Examples illustrating fault pathology

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

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

The similarity of the manifestation of elusive design faults and of transient physical faults leads to both classes being grouped together as **intermittent faults**. Errors produced by intermittent faults are usually termed **soft errors** [1].

The complexity of the mechanisms of creation, activation and manifestation of faults leads to the possibility of several causes. Such complexity leads to the definition, whether for classifying faults uncovered during operation or for stating fault hypotheses during system design, of classes of faults more that are more general than the specific classes considered up to now, in that sense that they may in turn have physical, design or interaction causes, or combinations of those. An example of such a class of faults is the **configuration change faults**: service delivery is altered during adaptive or perfective maintenance operations performed on-line, concurrently with system operation (e.g., introduction of a new software version on a network server).

References

1. D.C. Bossen, M.Y. Hsiao, "A system solution to the memory soft error problem", *IBM J. Res. Develop.*, vol. 24, no. 3, May 1980, pp. 390-397.

APPENDIX 2– DEPENDABILITY, SURVIVABILITY, TRUSTWORTHINESS: THREE NAMES FOR AN ESSENTIAL PROPERTY

The protection of highly complex societal infrastructures controlled by embedded information systems against all classes of faults defined in the section on the threats to dependability, including intelligent attacks, has become a top priority of governments, businesses, and system builders. As a consequence, different names have been given to the same essential property that assures protection. Here we compare the definitions of three widely known concepts: dependability, survivability, and trustworthiness.

A side-by-side comparison leads to the conclusion that all three concepts are essentially equivalent in their goals and address similar threats (figure A). Trustworthiness omits the explicit listing of internal faults, although its goal implies that they also must be considered. Such faults are implicitly considered in survivability via the (component) failures. Survivability was present in the late sixties in the military standards, where it was defined as a system capacity to resist hostile environments so that the system can fulfill its mission (see, e.g., MIL-STD-721 or DOD-D-5000.3); it was redefined recently, as described below. Trustworthiness was used in a study sponsored by the National Research Council, referenced below. One difference must be noted. Survivability and trustworthiness have the threats explicitly listed in the definitions, while both definitions of dependability leave the choice open: the threats can be either all the faults of figure 4 of this article, or a selected subset of them, e.g., ‘dependability with respect to design faults’, etc.

Concept	Dependability	Survivability	Trustworthiness
Goal	1) ability to deliver service that can justifiably be trusted 2) ability of a system to avoid failures that are more frequent or more severe, and outage durations that are longer, than is acceptable to the user(s)	capability of a system to fulfill its mission in a timely manner	assurance that a system will perform as expected
Threats present	1) design faults (e.g., software flaws, hardware errata, malicious logics) 2) physical faults (e.g., production defects, physical deterioration) 3) interaction faults (e.g., physical interference, input mistakes, attacks, including viruses, worms, intrusions)	1) attacks (e.g., intrusions, probes, denials of service) 2) failures (internally generated events due to, e.g., software design errors, hardware degradation, human errors, corrupted data) 3) accidents (externally generated events such as natural disasters)	1) hostile attacks (from hackers or insiders) 2) environmental disruptions (accidental disruptions, either man-made or natural) 3) human and operator errors (e.g., software flaws, mistakes by human operators)
Reference	This article	“Survivable network systems” [1]	“Trust in cyberspace” [2]

Figure A - Dependability, survivability and trustworthiness

References

1. R.J. Ellison, D.A. Fischer, R.C. Linger, H.F. Lipson, T. Longstaff, N.R. Mead, “Survivable network systems: an emerging discipline”, Technical Report CMU/SEI-97-TR-013, November 1997, revised May 1999.
2. F. Schneider, ed., *Trust in Cyberspace*, National Academy Press, 1999

APPENDIX 3– WHERE DO WE STAND?

Advances in integration, performance, and interconnection, are the elements of a virtuous spiral that led to the current state-of-the-art in computer and communication systems. However, two factors at least would tend to turn this virtuous spiral into a deadly one without dependability techniques: a) a decreasing natural robustness (due to, e.g., an increased sensitivity of hardware to environmental perturbations), and b) the inevitability of residual faults that goes along with the increased complexity of both hardware and software.

The interest in **fault tolerance** grows accordingly to our dependence in computer and communication systems. There exists a vast body of results, from error detecting and correcting codes to distributed algorithms for consensus in the presence of faults. Fault tolerance has been implemented in many systems; figure A lists some existing systems, either high availability systems or safety-critical systems.

		Tolerance to design faults
High availability platforms	Stratus [1]	—
	Tandem SeverNet [2]	elusive software design faults
	IBM S/390 cluster [3]	—
	Sun cluster [4]	—
Safety-critical systems	SACEM Subway speed control [5]	hardware design faults and compiler faults
	ELEKTRA Railway signalling system [6]	hardware and software design faults
	Airbus Flight Control System [7]	software design faults
	Boeing 777 Flight Control System [8]	hardware design faults and compiler faults

Figure A - Examples of fault tolerant systems

The progressive mastering of physical faults has enabled a dramatic improvement of computing systems dependability: the overall mean time to failure or unavailability has decreased by two orders of magnitude over the last two decades. As a consequence, design and human-made interaction faults currently dominate as sources of failure, as exemplified by figure B. Tolerance of those two classes of faults is still an active research domain, even if approaches have been devised and implemented, especially for software design faults.

	Computer systems (e.g., transaction processing [9], electronic switching [10])		Larger, controlled systems (e.g., commercial airplanes [11], telephone network [12])	
	Rank	Proportion of failures	Rank	Proportion of failures
Physical internal faults	3	~ 10 %	2	15 - 20 %
Physical external faults	3	~ 10 %	2	15 - 20 %
Human-made interaction faults	2	~ 20 %	1	40 - -50 %
Design faults	1	~ 60 %	2	15 - 20 %

Figure B - Proportions of failures due to accidental or non-malicious deliberate faults
(consequences and outage durations are highly application-dependent)

Security has benefited from advances in cryptography (e.g., public key schemes) and in the policies aimed at controlling information flow. Although there are much less in the way of published statistics about malicious faults than there is about accidental faults, a recent survey by Ernst & Young estimated that on the average two-third of the 1200 companies surveyed in 32 countries suffered from at least one fraud per year, and that 84% of the frauds were perpetrated by employees. System security policies are the current bottleneck, be it due to the breaches induced by the inevitable residual design faults in their implementation mechanisms, or by the necessary laxity without which systems would not be operable. Fault tolerance is thus needed, for protection from both malicious logics and intrusions.

The cost of verification and validation of a computing system is at least half of its development cost, and can go as high as three quarters for highly critical systems. The dominance in these costs of **fault removal** explains the importance of research in verification: the density of faults created during software development lies in the range of 10 to 300 faults/kLoC (thousand lines of executable code), down to 0.01 to 10 faults/kLoC after fault removal. The latter are residual faults, persisting during operation, and such high densities explain the importance of failures induced by design faults in large software systems.

Significant advances have taken place in both static and dynamic verification. Figure C lists various current model checking and theorem proving tools. The current bottleneck lies in the applicability to large-scale systems, and thus to scalability from critical embedded systems to service infrastructures.

Model checking	Theorem proving
Design/CPN (http://www.daimi.aau.dk/designCPN/) FDR (http://www.formal.demon.co.uk/FDR2.html) HyTech (http://www.eecs.berkeley.edu/~tah/HyTech) KRONOS (http://www-verimag.imag.fr/TEMPORISE/kronos/) NuSMV (http://sra.itc.it/tools/nusmv/index.html) SPIN (http://netlib.bell-labs.com/netlib/spin) UPPAAL (http://www.docs.uu.se/docs/rtmv/uppaal/)	Coq Proof Assistant (http://coq.inria.fr) HOL (http://lal.cs.byu.edu/lal/hol-desc.html) Isabelle (http://www.cl.cam.ac.uk/Research/HVG/Isabelle/) PVS (http://www.csl.sri.com/pvs.html)

Figure C - Examples of theorem prooving and model-checking tools

The difficulty, or even the impossibility, of removing all faults from a system, leads naturally to **fault forecasting**. Powerful probabilistic approaches have been developed, widely used as far as physical operational faults are concerned. Evaluating dependability with respect to software design faults is still a controversial topic, especially for highly-critical systems. Figure D lists various current software tools for stable dependability evaluation. Tools for reliability growth evaluation are described in the *Handbook of software reliability engineering* edited by M. Lyu (McGraw-Hill, 1996). Only recently has the use of probabilistic evaluation of security started to gain acceptance.

Stable dependability	
Tool name	Type of models
SHARPE (http://www.ee.duke.edu/~chirel/research1)	Fault trees and Markov chains
SPNP (http://www.ee.duke.edu/~chirel/research1)	Stochastic Petri nets and Markov chains
SURF-2 (http://www.laas.fr/laasef/index.htm)	Stochastic Petri nets and Markov chains
GreatSPN (http://www.di.unito.it/~greatspn/)	Stochastic Petri nets and Markov chains
Ultra-SAN (http://www.crhc.uiuc.edu/PERFORM/UltraSAN)	Stochastic Activity Networks
DSPNexpress (http://www4.cs.uni-dortmund.de/~Lindemann/software/DSPNexpress/mainE.html)	Deterministic and stochastic Petri nets

Figure D - Examples of software tools for dependability evaluation

Complementarity of fault removal and fault forecasting is yet more evident for fault tolerant systems for which verification and validation must, in addition to functional properties, address the ability of those systems to deal with faults and errors. Such verifications involve fault injection. Figure E lists various current fault-injection tools.

Hardware, pin level, fault injection	Software implemented fault injection	Simulators with fault injection capability
RIFLE (http://eden.dei.uc.pt:80/~henrique/Informatica/Rifle.html)	FERRARI (http://www.cerc.utexas.edu/~jaa/ftc/fault-injection.html) Xception (http://eden.dei.uc.pt:80/~henrique/Informatica/Xception.html) MAFALDA (http://www.laas.fr/laasve/index.htm)	DEPEND (http://www.crhc.uiuc.edu/DEPEND/depend.html) MEFISTO (http://www.laas.fr/laasve/index.htm) VERIFY (http://www3.informatik.uni-erlangen.de/Projects/verify.html)

Figure E - Examples of fault-injection tools

Confining within acceptable limits development costs of systems of ever-growing complexity necessitates development approaches involving pre-existing components, either acquired (COTS, commercial off-the-shelf software, or OSS, open source software) or re-used. The dependability of such components may be inadequate or simply unknown, which leads to the necessity of characterizing their dependability before using them in a development, or even to improve their dependability via wrapping them.

Improving the state of the art in dependable computing is crucial for the very future of Sciences and Technologies of Information and Communication. Computer failures cost to the society tens of billions of dollars each year, and the cumulative cost of canceled software developments is of the same order of magnitude (http://standishgroup.com/visitor/chaos.htm). Significant actions have been undertaken, such as the *High Confidence Software and Systems* component of the Federal Research Programme on Technologies of Information, or the *Cross-Programme Action on Dependability* of the European Information Society Technology Programme.

References

1. Wilson, "The STRATUS Computer System", in *Resilient Computing Systems,* (T. Anderson, Ed.), pp. 208-231, London, UK, Collins, 1985.
2. W.E. Baker, R.W. Horst, D.P. Sonnier, W.J. Watson, "A flexible ServerNet-based fault-tolerant architecture", in *Proc. 25th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-25)*, Pasadena, June 1995, pp. 2-11.
3. "Competitive analysis of reliability, availability, serviceability and cluster features and functions", D.H. Brown Associates, 1998.
4. N.S. Bowen, J. Antognini, R.D. Regan, N.C. Matsakis, "Availability in parallel systems: automatic process restart", *IBM Systems Journal,* vol. 36, no. 2, 1997, pp. 284-300.
5. C. Hennebert, G. Guiho, "SACEM: a fault-tolerant system for train speed control", *Proc. 23rd IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-23),* Toulouse, France, June 1993, pp. 624-628.
6. H. Kantz and C. Koza, "The ELEKTRA Railway Signalling-System: Field Experience with an Actively Replicated System with Diversity," *Proc. 25th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-25),* Pasadena, June 1995, pp. 453-458.
7. D. Briere, P. Traverse, "Airbus A320/A330/A340 electrical flight controls — a family of fault-tolerant systems", *Proc. 23rd IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-23),* Toulouse, France, June 1993, pp. 616-623.
8. Y.C. Yeh, "Dependability of the 777 primary flight control system", in *Dependable Computing for Critical Applications 5*, R.K. Iyer, M. Morganti, W.K. Fuchs, V. Gligor, eds, IEEE Computer Society Press, 1997, pp. 3-17.
9. J. Gray, "A census of Tandem system availability between 1985 and 1990", *IEEE Trans. on Reliability*, vol. 39, no. 4, Oct. 1990, pp. 409-418.
10. R. Cramp, M.A. Vouk, W. Jones, "On operational availability of a large software-based telecommunications system", *Proc. 3rd Int. Symp. on Software Reliability Engineering (ISSRE'92)*, Research Triangle Park, North Carolina, Oct. 1992, pp. 358-366.
11. B. Ruegger, "Human error in the cockpit", Swiss Reinsurance Company, 1990.
12. D.R. Kuhn, "Sources of failure in the public switched telephone network", *IEEE Computer*, vol. 30, no. 4, April 1997, pp. 31-36.