Technical Report
CMU/SEI-95-TR-021
ESC-TR-95-021

Quality Attributes

Mario Barbacci

Mark H. Klein

Thomas A. Longstaff

Charles B. Weinstock

December 1995

Technical Report
CMU/SEI-95-TR-021
ESC-TR-95-021
December 1995

Quality Attributes

Mario Barbacci
Thomas H. Longstaff
Mark H. Klein
Charles B. Weinstock

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc., 800 Vinial Street, Suite C201, Pittsburgh, PA 15212. Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page. The URL is http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / Attn: BRR / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: (703) 767-8274 or toll-free in the U.S. — 1-800 225-3842).

Contents

List of Figures

Quality Attributes

Abstract: Computer systems are used in many critical applications where a failure can have serious consequences (loss of lives or property). Developing systematic ways to relate the software quality attributes of a system to the system's architecture provides a sound basis for making objective decisions about design trade-offs and enables engineers to make reasonably accurate predictions about a system's attributes that are free from bias and hidden assumptions. The ultimate goal is the ability to quantitatively evaluate and trade off multiple software quality attributes to arrive at a better overall system. The purpose of this report is to take a small step in the direction of developing a unifying approach for reasoning about multiple software quality attributes. In this report, we define software quality, introduce a generic taxonomy of attributes, discuss the connections between the attributes, and discuss future work leading to an attribute-based methodology for evaluating software architectures.

1 Introduction

Computer systems are used in many critical applications where a failure can have serious consequences (loss of lives or property). Critical applications have the following characteristics:

- The applications have long life cycles (decades rather than years) and require evolutionary upgrades.
- The applications require continuous or nearly non-stop operation.
- The applications require interaction with hardware devices.
- The applications assign paramount importance to quality attributes such as timeliness, reliability, safety, interoperability, etc.

Developing systematic ways to relate the software quality attributes of a system to the system's architecture provides a sound basis for making objective decisions about design trade-offs and enables engineers to make reasonably accurate predictions about a system's attributes that are free from bias and hidden assumptions. The ultimate goal is the ability to quantitatively evaluate and trade off multiple software quality attributes to arrive at a better overall system.

The purpose of this report is to take a small step in the direction of developing a unifying approach for reasoning about multiple software quality attributes. This report examines the following four software quality attributes: performance, dependability, security, and safety. Each attribute has matured (or is maturing) within its own community, each with their own vernacular and point of view. We propose a generic taxonomy for describing each attribute and attempt to use this taxonomy to

- describe how each community thinks about its respective attribute
- highlight some of the important methods used by each community

- draw out the connections between the attributes
- suggest a direction for developing an attribute-based methodology for evaluating software architectures.

Section 2 defines software quality and introduces the generic taxonomy. The four sections that follow cover each of the four attributes:

- Section 3 Performance
- Section 4 Dependability
- Section 5 Security
- Section 6 Safety

In these sections the following conventions are used in the text:

- **bold** - indicates that a term is defined in the glossary starting on page 47.
- *italics* - indicates that a term is shown in the figure illustrating the taxonomy.
- ***bold italics*** - indicates that a term is both shown in the figure illustrating the taxonomy and defined in the glossary.

Section 7 discusses the connections between the four attributes by highlighting the relationships between attributes and their approaches and makes several recommendations.

Section 8 discusses future work leading to an attribute-based methodology for evaluating software architectures.

2 Software Quality Attributes

Developers of critical systems are responsible for identifying the requirements of the application, developing software that implements the requirements, and for allocating appropriate resources (processors and communication networks). It is not enough to merely satisfy functional requirements. Critical systems in general must satisfy security, safety, dependability, performance, and other, similar requirements as well.

Software **quality** is the degree to which software possesses a desired combination of attributes (e.g., reliability, interoperability) [IEEE 1061].

2.1 How Various Communities Have Addressed Quality Attributes

There are different schools/opinions/traditions concerning the properties of critical systems and the best methods to develop them:

- performance — from the tradition of hard real-time systems and capacity planning
- dependability — from the tradition of ultra-reliable, fault-tolerant systems
- security — from the traditions of the government, banking and academic communities
- safety — from the tradition of hazard analysis and system safety engineering

Systems often fail to meet **user** needs (i.e., lack quality) when designers narrowly focus on meeting some requirements without considering the impact on other requirements or by taking them into account too late in the development process. For example, it might not be possible to meet dependability and performance requirements simultaneously:

- Replicating communication and computation to achieve dependability might conflict with performance requirements (e.g., not enough time).
- Co-locating critical processes to achieve performance might conflict with dependability requirements (e.g., single point of failure).

This is not a new problem and software developers have been trying to deal with it for a long time, as illustrated by Boehm:

> Finally, we concluded that calculating and understanding the value of a single overall metric for software quality may be more trouble than it is worth. The major problem is that many of the individual characteristics of quality are in conflict; added efficiency is often purchased at the price of portability, accuracy, understandability, and maintainability; added accuracy often conflicts with portability via dependence on word size; conciseness and conflict with legibility. Users generally find it difficult to quantify their preferences in such conflict situations [Boehm 78].

2.2 Software Quality Attribute Trade-offs

Designers need to analyze trade-offs between multiple conflicting attributes to satisfy user requirements. The ultimate goal is the ability to quantitatively evaluate and trade off multiple quality attributes to arrive at a better overall system. We should not look for a single, universal metric, but rather for quantification of individual attributes and for trade-off between these different metrics, starting with a description of the software architecture.

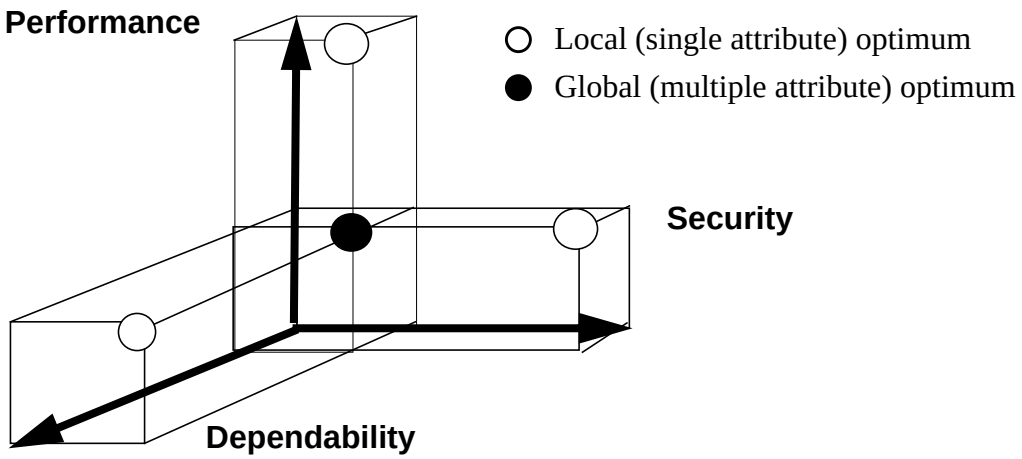

Figure 2-1: Software Quality Attribute Trade-offs

2.3 Generic Taxonomy for Quality Attributes

Attributes will be thought of as properties of the service delivered by the system to its users. The **service** delivered by a system is its behavior as it is perceived by its user(s); a **user** is another system (physical or human) which interacts with the former [Laprie 92]. We think of the service as being initiated by some **event**, which is a stimulus to the system signaling the need for the service. The stimulus can originate either within the system or external to the system.

For each quality attribute (performance, dependability, security and safety) we use a taxonomy (see Figure 2-2) that identifies:

Concerns — the parameters by which the attributes of a system are judged, specified and measured. Requirements are expressed in terms of concerns.

Attribute-specific factors —properties of the system (such as policies and mechanisms built into the system) and its environment that have an impact on the concerns. Depending on the attribute, the attribute-specific factors are internal or external properties affecting the concerns. Factors might not be independent and might have cause/effect relationships. Factors and their relationships would be included in the system's architecture:

- ***Performance factors*** — the aspects of the system that contribute to performance. These include the demands from the environment and the system responses to these demands.
- ***Dependability impairments*** — the aspects of the system that contribute to (lack of) dependability. There is a causal chain between faults inside the system and failures observed in the environment. Faults cause errors; an error is a system state that might lead to failure if not corrected.
- ***Security factors*** — the aspects of the system that contribute to security. These include system/environment interface features and internal features such as kernelization.
- ***Safety impairments*** — the aspects of the system that contribute to (lack of) safety. Hazards are conditions or system states that can lead to a mishap or accident. Mishaps are unplanned events with undesirable consequences.

Methods — how we address the concerns: analysis and synthesis processes during the development of the system, and procedures and training for users and operators. Methods can be for analysis and/or synthesis, procedures and/or training, or procedures used at development or execution time.

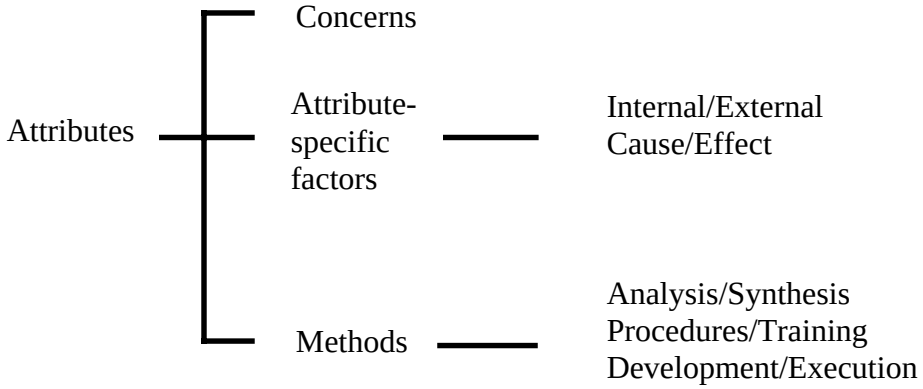

Figure 2-2: Generic Taxonomy for Quality Attributes

3 Performance

3.1 Overview

3.1.1 Definition

3.1.1.1 IEEE 610.12 Definition

"Performance" has many connotations. The definition given in the IEEE Standard Glossary of Software Engineering Terminology [IEEE-610.12] is: "**Performance**. The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage." This definition is too broad for our purposes.

3.1.1.2 Smith's Definition

Performance as a software quality attribute refers to the timeliness aspects of how software systems behave. We adopt a slight generalization of Smith's definition of performance: **"**Performance refers to responsiveness: either the time required to respond to specific events or the number of events processed in a given interval of time" [Smith 93, p. 720]. Performance is that attribute of a computer system that characterizes the timeliness of the service delivered by the system.

3.1.1.3 Performance vs. Speed

A misconception about performance is that it equates to speed—that is, the notion that poor performance can be salvaged simply by using more powerful processors or communication links with higher bandwidth. Faster might be better, but for many systems faster is not sufficient to achieve timeliness. This is particularly true of real-time systems. As noted by Stankovic [Stankovic 88], the objective of "fast computing" would be to minimize the average response time for some group of services, whereas the objective of real-time computing is to meet individual timing requirements of each service. Moreover, hardware mechanisms such as caching, pipelining and multithreading, which can reduce average response time, can make worst-case response times unpredictable.

"Predictability, not speed, is the foremost goal in real-time-system design" [Stankovic 88]. In general, performance engineering is concerned with predictable performance—whether it is worst-case or average-case performance. Execution speed is only one factor.

3.1.2 Taxonomy

3.1.2.1 Abstract Performance Model

The performance of a system stems from the nature of the resources used to fulfill demands and how shared resources are allocated when the multiple demands must be carried out on the same resources. This type of problem is known as a scheduling problem and has been studied for years. See, for example, Conway [Conway 67].

Conway [Conway 67, p. 6] says a scheduling problem can described by four types of information:

1. jobs and operations to be processed
2. number and types of machines
3. disciplines that restrict the manner in which assignments can be made
4. the criteria by which the schedule will be evaluated

From a modeling point of view, Smith [Smith 93, p. 723] describes five types of data needed for constructing and evaluating software performance engineering models:

- Performance requirements - quantitative requirements defined in terms of events of interest and timing constraints for responding to each event.
- Behavior patterns and intensity - the number of event streams[1] and the worst-case and steady-state arrival rates for each event stream
- Software descriptions - the software operations executed in response to events.
- Execution environment - the hardware devices and software services needed to carry out the aforementioned software operations.
- Resource usage estimates - resource requirements for carrying software operations such as processor execution time, I/O demands or memory requirements.

The points of view of Conway and Smith differ somewhat; nevertheless, both points of view call out

- ***performance concerns***, such as criteria for evaluating the schedule, and timing constraints for responding to events
- ***performance factors***, such as
  - behavior patterns and intensity, resource usage, software descriptions, and jobs and operations, which characterize system *demand*
  - execution environment and numbers and types of machines, which characterize the ***system***
- ***performance methods***, such as *synthesis* and *analysis* that draw upon *queuing theory*, *scheduling theory*, and f*ormal methods* that are used to understand the relationship between the factors and the concerns.

This is reflected in the taxonomy shown in Figure 3-1.

1. An **event stream** is a sequence of events from the same source— for example, a sequence of interrupts from a given sensor.

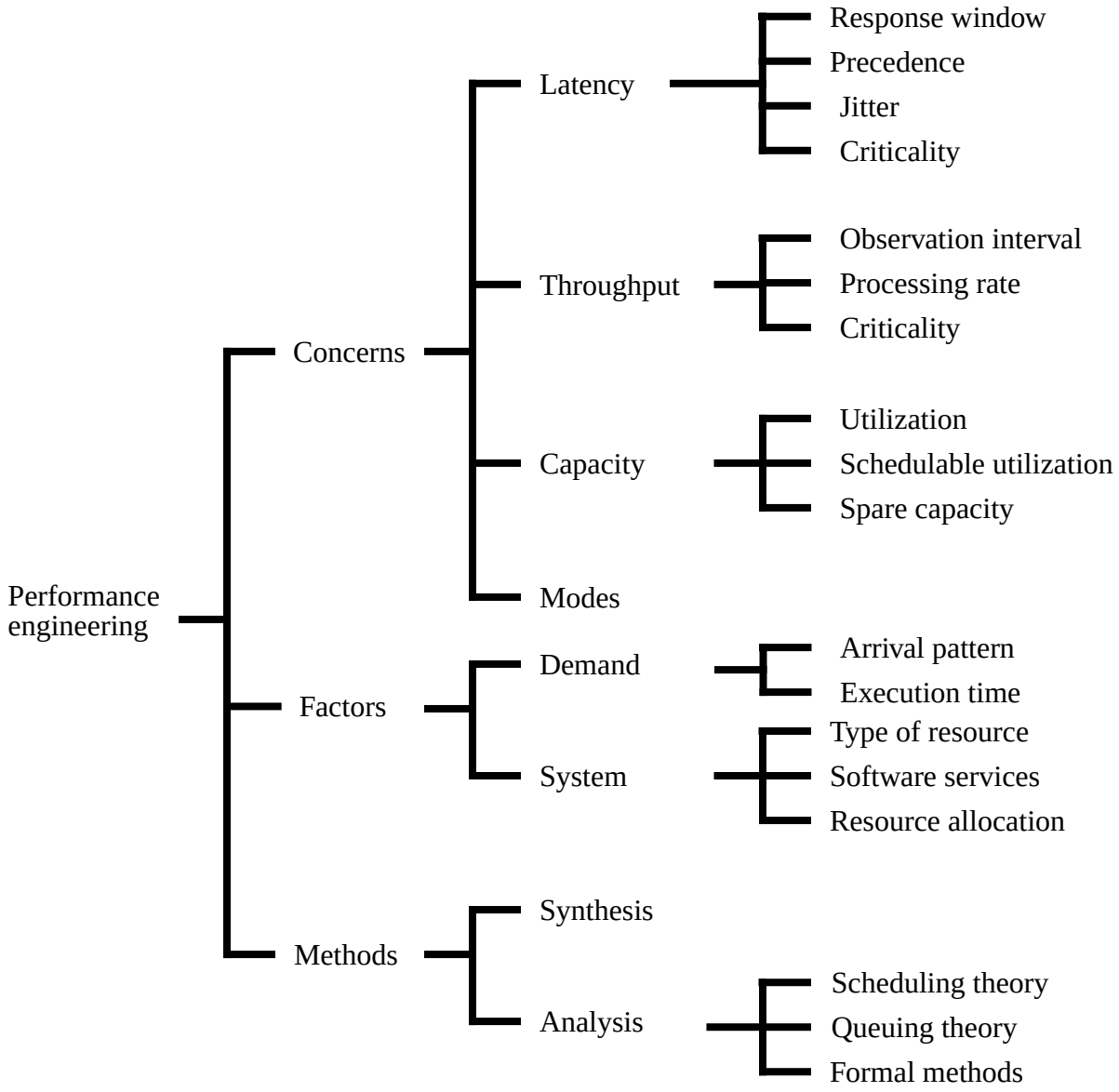

Figure 3-1: Performance Taxonomy

3.2 Concerns

The performance concerns (or requirements) used to specify and assess the performance of the system are

- ***latency*** - How long does it take to respond to a specific event?
- ***throughput*** - How many events can be responded to over a given interval of time?

- ***capacity*** - How much demand can be placed on the system while continuing to meet latency and throughput requirements?
- ***modes*** - How can the demand and resources change over time? What happens when system capacity is exceeded and not all events can be responded to in a timely manner?

3.2.1 Latency

Latency refers to a time interval during which the **response** to an event must be executed. The time interval defines a ***response window*** given by a starting time (minimum latency) and an ending time (maximum latency). These can either be specified as absolute times (time of day, for example) or offsets from an event which occurred at some specified time. The ending time is also known as a deadline. Latency sub-concerns include: ***precedence*** (a specification for a partial or total ordering of event responses), ***jitter*** (the variation in the time a computed result is output to the external environment from cycle to cycle), and ***criticality*** (the importance of the function to the system).

3.2.2 Throughput

Throughput refers to the number of event responses that have been completed over a given ***observation interval*** [Lazowska 84, p. 41]. This definition suggests that it is not sufficient to just specify a ***processing rate***, but that one or more observation intervals should also be specified. For example, a system that can process 120 transactions every hour might not guarantee that 2 transactions will be processed every minute. Perhaps no transactions are processed during the first 30 minutes and all of the transactions are processed during the remaining 30 minutes.

Criticality is also a sub-concern of throughput.

3.2.3 Capacity

Capacity is a measure of the amount of work a system can perform. Capacity is usually defined in terms of throughput, and has several possible meanings [Jain 91, p. 39]:

> The maximum achievable throughput under ideal workload conditions. That is, the maximum number of events per unit time that can be achieved if you could pick the theoretically ideal set of events. For networks this is called bandwidth, which is usually expressed in megabits per second.

However, often there is also a response time requirement that accompanies the throughput requirement (as mentioned above). Therefore, a more practical definition is the following:

> The maximum achievable throughput without violating specified **latency requirements**. Jain refers to as usable capacity [Jain 91].

For real-time systems, throughput is not as important as predictably meeting latency requirements. While we can still consider looking at the maximum achievable throughput while continuing to meet all hard deadlines, another useful metric is schedulable utilization.

Utilization is the percentage of time a resource is busy. ***Schedulable utilization***, then, is the maximum utilization achievable by a system while still meeting timing requirements. Sha [Sha 90] refers to this as schedulability, one of the fundamental measures of merit for real-time systems.

Since capacity is a measure of the amount of work a system can perform, ***spare capacity***, then, is then a measure of the unused capacity.

3.2.4 Modes

It is not uncommon for systems to have different sets of requirements for different phases of execution. For example, an avionics system could have different requirements for the take-off phase than for the cruising phase. We refer to these different phases as modes. A mode can be characterized by the state of the demand being placed on the system and the state of the system (that is, the configuration of resources used to satisfy the demand).

Two commonly encountered modes are **reduced capacity** and **overload**. A system might have to operate with reduced capacity if resources cease to function properly. A system might have to sacrifice timing requirements of less important events during periods of overload.

3.3 Factors Affecting Performance

Performance is a function of the demand placed on the system, the types of resources used by the system, and how the system allocates those resources. Performance factors represent the important aspects of the system and its environment that influence the performance concerns. There are environment performance factors (demand) and system performance factors.

- *demand* - How many events streams are there? What are the arrival rates associated with each event stream? What is the resource usage associated with responding to each event?
- *system* - What are the properties of the scheduler used to allocated resources, the properties of the software operations that comprise the responses to events and the relationships between responses?

3.3.1 Demand

Demand is a characterization of how much of a resource is needed. Demand can be thought of in terms of how much utilization a specific event requires. However, it is useful to think of demand in terms of

- arrival pattern for each event stream and
- execution time requirements for responding to each event

The *arrival pattern* and *execution time* are important since these are two pieces of information that can be used by scheduling theory and/or queuing theory for predicting latency and/or throughput. The arrival pattern is either periodic or aperiodic.

- ***Periodic*** arrivals occur repeatedly at regular intervals of time.
- ***Aperiodic*** arrivals occur repeatedly at irregular time intervals. The frequency of arrival can be bounded by a minimum separation (also known as sporadic) or can be completed random [Lehoczky 94, pp. 1011-1012].

For execution times, the worst-case and best-case execution times can be used to help define boundary-case behavior. Queuing theoretic techniques specify execution times using probability distribution functions.

3.3.2 System

Resources comprise a system and are needed to carry out event responses. We think of the system in terms of

- types of resources
- software services for managing resources
- resource allocation

Common resource types are: CPU, memory, I/O device, backplane bus, network, and data object. Associated with each *type of resource* there are software services for managing the use of the resource and resource allocation policies. It is beyond the scope of this paper to discuss all of these resource types. We will focus on operating systems services and CPU scheduling.

A primary factor that influences the concerns of performance is the *software services* that are provided to allocate resources and to provide an interface to resources. These software services are usually provided by the operating system. For this discussion we will focus our attention on real-time operating systems, since real-time operating systems are explicitly concerned with time and thus serve to highlight some of the important issues.

Stankovic groups real-time operating systems into three categories [Stankovic 94]:

- small, fast proprietary kernels
- real-time extensions of commercial operating systems
- research-oriented operating systems

Some of the important factors of an OS discussed by Stankovic that can affect performance are: context switch times; interrupt latency; time during which interrupts are disabled; use of virtual memory; bounds on the execution of system calls; precision of timer facilities; support for predictable communication; scheduling overhead; non-preemptible sections and FIFO queues. Other important OS factors [Klein 93, p. 7-4] are: priority of the OS service; implicit use of shared resources; and limited representations of application or system parameters such as insufficient number of priority levels or insufficient precision in time representation.

The *resource allocation* policy (that is, scheduling algorithm) used to resolve contention for shared resources has a primary influence on the performance of a system. Scheduling algorithms can be distinguished by whether the schedule is constructed off-line or on-line [Lehoczky 94].

Off-line scheduling requires complete knowledge of all events and their responses. This knowledge is used to construct a time-line in advance of program execution that lays out the order in which all event responses will be executed. This type of scheduling strategy is known as a cyclic executive [Locke 92]. This strategy can be very efficient and simple for cases in which there are a small number of periodic events with periods that are close to harmonic. In these cases predicting the performance of a system is very straightforward since it has been completely predetermined. However, as systems became more complex it was realized that cyclic executives were relatively inflexible in the face of the inevitable modifications made to systems.

In on-line scheduling, decisions are made at run-time and thus they tend to be more flexible than off-line algorithms. Static-priority algorithms (e.g., rate monotonic and deadline monotonic scheduling algorithms) assign a priority to the response to the event; the response uses that priority for responding to every event in the event stream. Dynamic-priority algorithms (e.g., earliest deadline first, least laxity first, best-effort scheduling) allow event responses to change the priority for every invocation and during a single response.

3.4 Methods

Methods to achieve performance include the following:

Synthesis—methods used to synthesize (such as real-time design methodologies) a system or Smith's software performance engineering philosophy as discussed in [Smith 90].

Analysis—techniques used to analyze system performance such as queuing analysis and scheduling analysis.

3.4.1 Synthesis

Smith [Smith 90, p. 14] advocates a philosophy of software performance engineering intended to augment rather than supplant other software engineering methodologies. The goal is to carry out the fundamental engineering steps of understanding, creating, representing and evaluating, but to complement these steps with an explicit attention paid to performance. This manifests itself in developing models to represent the performance of the system early, and continuous evaluation of the performance of the system as it evolves.

3.4.2 Analysis

Performance analysis methods seem to have grown out of two separate schools of thought, queueing theory and scheduling theory.

Queueing theory — Queuing theory can be used to model systems as one or more service facilities that perform services for a stream of arriving customers. Each arrival stream of customers is described using a stochastic process. Each service facility comprises a server and a queue for waiting customers. Service times for customers are also described using stochastic processes. Queuing analysis is mostly concerned with average case aggregate behaviors

—which is appropriate for performance capacity planning and management information systems, for example. However, when worst-case behavior is of interest, scheduling analysis might be more appropriate.

Scheduling theory — Classical scheduling theory has its roots in job-shop scheduling [Audsley 95, p. 176]. Many of the results of scheduling are either directly applicable to performance analysis of real-time systems or offer valuable intuition. Many of these are summarized in [Stankovic 95]. Many of the analysis techniques relevant to static priority preemptive scheduling are discussed in [Lehoczky 94].

The analysis techniques that are applicable to real-time systems offer conditions under which specific events will meet their timing constraints. These techniques are predicated on knowing the conditions under which worst-case event responses will occur. Two typical types of analysis are based on

- computing utilization bounds
- computing response times

Utilization bounds are used to guarantee timing requirements by computing the utilization of the system and then comparing it to a theoretically-derived bound. Given the right preconditions, when utilization is kept under the specified bound, timing requirements are guaranteed to be met.

Other results allow one to calculate the worst-case response times for specified events. This worst-case response time can then be compared to the deadline to determine if latency requirements will be satisfied.

The use of *formal methods* involves developing a formal specification of the desired temporal behavior of a system, developing a formal specification of a design or implementation of some or all of the system, and finally, conducting a formal verification that the system satisfies the desired behavior.

Typically, these methods involve the use of formal mathematical notations for describing the characteristics of a system and then use inference techniques to deduce system properties. Various forms of timed logic systems [Jahanian 86] or timed process algebras are used, for example.

4 Dependability

4.1 Overview

4.1.1 Definition

Unlike the other properties discussed in this report, the dependability community has been able to reach a consensus on terminology. This agreed upon terminology is codified by Laprie [Laprie 92]. A subsequent draft revision[1] forms the basis for much of this section.

4.1.1.1 IFIP WG10.4 Definition [Laprie 92]

Dependability is that property of a computer system such that reliance can justifiably be placed on the service it delivers. Dependability has several attributes, including

- ***availability*** — readiness for usage
- ***reliability*** — continuity of service
- ***safety*** — non-occurrence of catastrophic consequences on the environment
- ***confidentiality*** — non-occurrence of unauthorized disclosure of information
- ***integrity*** — non-occurrence of improper alterations of information
- ***maintainability*** — aptitude to undergo repairs and evolution

Notice that the last three attributes correspond to the safety and security areas being discussed in other sections of this report. In [Laprie 92] confidentiality and integrity are grouped under the rubric “security.” In a later draft [Laprie 94] the two aspects of security are called out as above.

4.1.2 Taxonomy

Figure 4-1 shows a dependability tree. In addition to the attributes of dependability, it shows the means to achieving dependability, and the impairments to achieving dependability.

4.2 Concerns

The concerns of dependability are the parameters by which the dependability of a system are judged. A dependability-centric view of the world subsumes the usual *attributes* of reliability, availability, safety, and security (confidentiality and integrity). Depending on the particular application of interest, different attributes are emphasized.

4.2.1 Availability

The ***availability*** of a system is a measure of its readiness for usage. Availability is always a concern when considering a system’s dependability, though to varying degrees, depending upon the application.

1. [Laprie 94] J.C. Laprie (ed.) *Dependability: Basic Concepts and Terminology, Revision* (Draft), 1994.

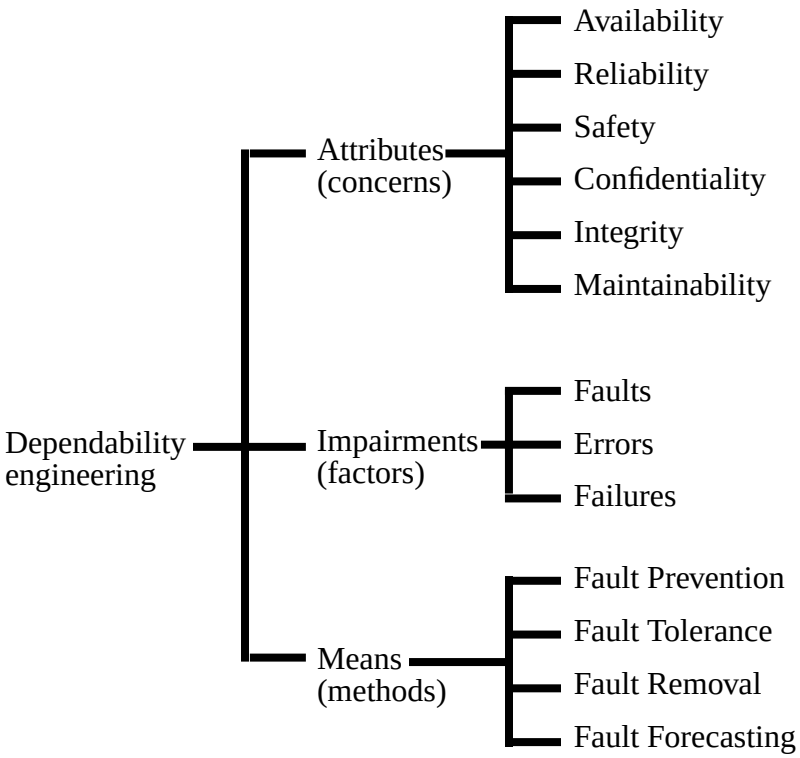

Figure 4-1: Dependability Tree [Laprie 92]

Availability is measured as the limit of the probability that the system is functioning correctly at time t, as t approaches infinity. This is the steady-state availability of the system. It may be calculated [Trivedi 82] as:

$$\alpha = \frac{MTTF}{MTTF + MTTR}$$

where MTTF is the mean time to failure, and MTTR is the mean time to repair.

4.2.2 Reliability

The ***reliability*** of a system is a measure of the ability of a system to keep operating over time. Depending on the system, long-term reliability may not be a concern. For instance, consider an auto-land system. The availability requirement of this system is high—it must be available when called upon to land the plane. On the other hand, the reliability requirement is somewhat low in that it does not have to remain operational for long periods of time.

The reliability of a system is typically measured as its mean time to failure (MTTF), the expected life of the system.

4.2.3 Maintainability

The ***maintainability*** of a system is its aptitude to undergo repair and evolution. It is less precisely measured than the previous two concerns. MTTR is a quantitative measure of maintainability, but it does not tell the whole story. For instance, repair philosophy should be taken into account. Some systems are maintained by the user, others by the manufacturer. Some are maintained by both (e.g., the machine diagnoses a board failure, sends a message to the manufacturer who sends a replacement board to the user with installation instructions.) There is a cost vs. MTTR trade-off which comes into play. For instance, built-in diagnostics can reduce the MTTR at the possible cost of extra memory, run-time, or development time.

4.2.4 Safety

From a dependability point of view, ***safety*** is defined to be the absence of catastrophic consequences on the environment. Leveson [Leveson 95] defines it as freedom from accidents and loss. This leads to a binary measure of safety: a system is either safe or it is not safe.

Safety is treated separately elsewhere in this report.

4.2.5 Confidentiality

Confidentiality is the non-occurrence of unauthorized disclosure of information. It is treated separately, in the “Security” section of this report (see Section 5 on page 25).

4.2.6 Integrity

Integrity is the non-occurrence of the improper alteration of information. Along with confidentiality, this subject is treated separately in Section 5 on page 25.

4.3 Impairments to Dependability

The ***impairments to dependability*** include the fault, error, and failure properties of the hardware and software of which the system is comprised, as shown in Figure 4-1.

4.3.1 Failures

As previously stated, a system fails when its behavior differs from that which was intended. Notice that we define ***failure*** with respect to intent, and not with respect to the specification. If the intent of the system behavior ends up differing from the specification of the behavior we have a specification fault.

There are many different ways in which a system can fail. As shown in Figure 4-2, the so-called “failure modes” of a system may be loosely grouped into three categories; *domain failures*, *perception by the users*, and *consequences on the environment*.

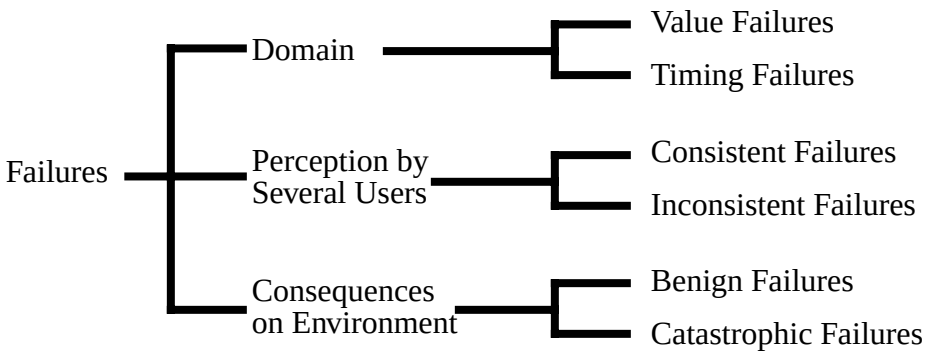

Figure 4-2: The Failure Classes [Laprie 94]

Domain failures include both ***value failures*** and ***timing failures***. A value failure occurs when an improper value is computed, one inconsistent with the proper execution of the system. Timing failures occur when the system delivers its service either too early or too late.

An extreme form of a timing failure is the **halting failure**—the system no longer delivers any service to the user. It is difficult to distinguish a very late timing failure from a halting failure. A system whose failures can be made to be only halting failures is called a **fail-stop** system [Schlichting 83]. The fail-stop assumption can lead to simplifications in dependable system design. Another special case of the halting failure which lead to simplification is one in which a failed system no longer generates any outputs. This is termed a **fail-silent** system.

There are two types of perception failures. A failure can be either consistent, or inconsistent. In the case of a ***consistent failure***, all system users have the same perception of a failure. In the case of an ***inconsistent failure***, some system users may have perceptions of the failure which differ from each other. These sorts of failures are called **Byzantine failures** [Lamport 82] and are the hardest failures to detect.

Finally, we can grade failures by their consequences on the environment. Although extremely difficult to measure, failures can be classified in the range ***benign*** to ***catastrophic***. A system which can only fail in a benign manner is termed **fail-safe**.

4.3.2 Errors

An ***error*** is a system state that is liable to lead to a failure if not corrected. Whether or not it will lead to a failure is a function of three major factors:

1. the redundancy (either designed in or inherent) in the system
2. the system activity (the error may go away before it causes damage)
3. what the user deems acceptable behavior. For instance, in data transmission there is the notion of “acceptable error rate”

4.3.3 Faults

A ***fault*** is the adjudged or hypothesized cause of an error. As shown in Figure 4-3, they can be classified along five main axes: phenomenological *cause, nature, phase of creation,* system *boundary,* and *persistence*.

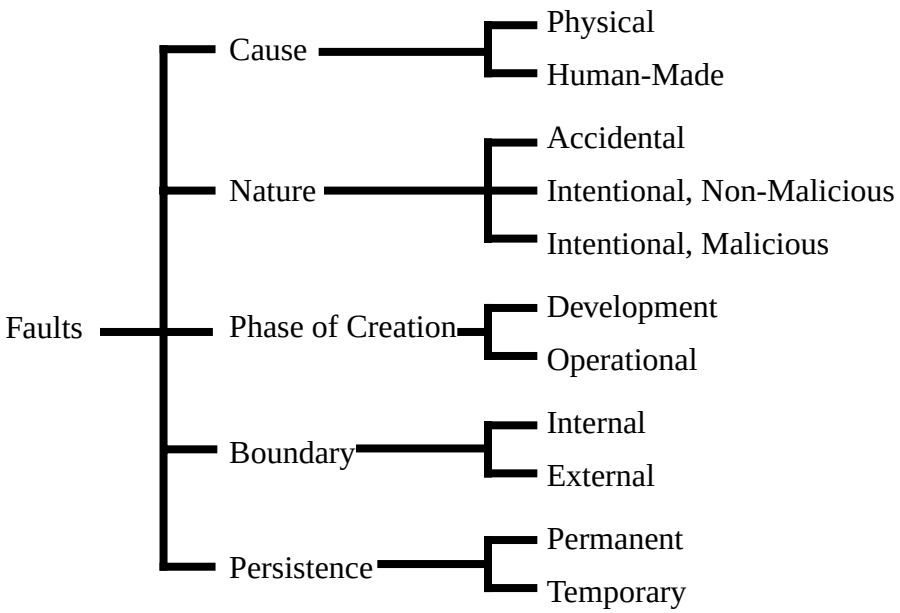

Figure 4-3: Fault Classes [Laprie 94]

Physical faults are those faults which occur because of adverse physical phenomena (e.g., lightning.) ***Human-made faults*** result from human imperfection and may be the result of many factors, singly or in cooperation, including poor design, inadequate manufacture, or misuse.

Accidental faults appear to be or are created by chance. ***Intentional faults*** are created deliberately, with or without malicious intent.

Faults can be created at *development* time, or while the system is running (*operational*).

Faults can be ***internal faults***, which are those parts of the internal state of the system which, when invoked, will produce an error. Alternatively, faults can be induced externally, for instance, via radiation.

Finally, faults can be ***permanent*** or ***temporary*** in which case the fault disappears over time. Temporary faults which result from the physical environment (i.e., temporary external faults) are often termed **transient faults**. Those which result from internal faults are often termed **intermittent faults**.

As shown in [Laprie 94], the cross product of the above would result in 48 different fault classes. However, many of these aren't meaningful. The number of important combinations is 15. These 15 can be loosely grouped into five more general classes; physical faults, design faults, interaction faults, malicious logic faults, and intrusions. See [Laprie 94].

4.3.4 Relationship Between Impairments

4.3.4.1 Fault Pathology

In the above model, faults produce errors which lead to failures. A fault that has not yet produced an error is **dormant**. A fault which produces an error is called **active**. An error may be **latent** or detected. An error may disappear before it is detected, or before it leads to a failure. Errors typically propagate, creating other errors. Active faults cannot be observed, only errors can. A failure occurs when an error affects the service being delivered to the user.

A system is typically built up of components. The failure of a component of a system may or may not result in the failure of the system. If the user does not see the service delivered by the failed component directly, no failure (with respect to the user) has occurred. A failure has only occurred with respect to the system which uses the component.

4.3.4.2 Another View of Faults, Errors and Failures

Some find the dichotomy just given—faults, failures, and errors—to be confusing. Heimerdinger and Weinstock [Heimerdinger 92] have proposed the elimination of the term "error" as a way of making things more understandable. In their view, failure has the same meaning as previously given. However, their alternate view of fault is to consider them failures in other systems that interact with the system under consideration—either a subsystem internal to the system under consideration, a component of the system under consideration, or an external system that interacts with the system under consideration (e.g., the environment.) Every fault is a failure from some point of view. A fault can lead to other faults, or to a failure, or neither.

But what of errors? As defined above, errors are a passive concept associated with incorrect values in the system state. However, it is extremely difficult to develop unambiguous criteria for differentiating between faults and errors. Many researchers refer to value faults, which are also clearly erroneous values. The connection between error and failure is even more difficult to describe.

However, the reality of the situation is that the fault-error-failure terminology is so well entrenched that, as much as we'd like not to, we will use that view in the rest of this document.

4.4 Methods

As shown in Figure 4-1, there are four major ways to achieve dependability: we can prevent faults from happening in the first place, we can tolerate their presence in the operational system, and we can remove them from the operational system once they have appeared. In addition, the figure shows, we can forecast how dependable the system is and use the information gleaned to improve it.

In this section, we will concentrate on the last three of these means, as **fault prevention** comes under the more general heading of good software engineering practice.

4.4.1 Fault Tolerance

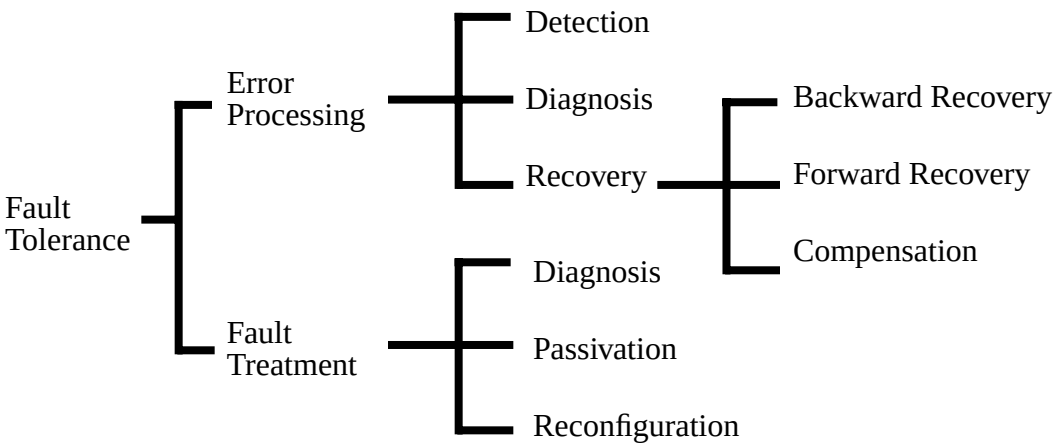

Figure 4-4: Fault Tolerance [Laprie 94]

Fault-tolerant systems attempt to detect and correct latent errors before they become effective. The dependability tree for ***fault tolerance*** is shown in Figure 4-4. The two major means for fault tolerance include ***error processing*** and ***fault treatment***.

Error processing is aimed at removing errors, if possible, before the occurrence of a failure. Fault treatment is aimed at preventing previously-activated faults from being re-activated.

Error processing involves *detecting* that the error exists, *diagnosing* the damage that an error causes, and *recovering* from the error by substituting an error-free state for the erroneous state. Errors can be recovered from via backward recovery, forward recovery, or compensation.

Backward recovery replaces the erroneous state with some previous state known to be error-free (e.g., via checkpoints or recovery blocks.) *Forward recovery* repairs the system state by finding a new one from which the system can continue operation. Exception handling is one method of forward recovery. *Compensation* uses redundancy to mask the error and allow transformation (perhaps via reconfiguration) to an error-free state. Compensation is achieved by modular redundancy—independent computations are voted upon and a final result is se-

lected by majority voting. Majority voting might be supplemented with other algorithms to mask complex, Byzantine faults. Modular redundancy requires independence among component failures. This is a reasonable assumption for physical faults but questionable for software design faults (e.g., N-version programming).

Fault treatment steps include *fault diagnosis* and *fault passivation* (removal and reconfiguration). Fault treatment is aimed at preventing faults from being activated again.

- Fault diagnosis consists of determining the cause(s) of error(s) in terms of both location and nature.
- Fault passivation consists of removing the component(s) identified as being faulty from further execution. If the system is no longer capable of delivering the same service as before, a reconfiguration may take place.

4.4.2 Fault Removal

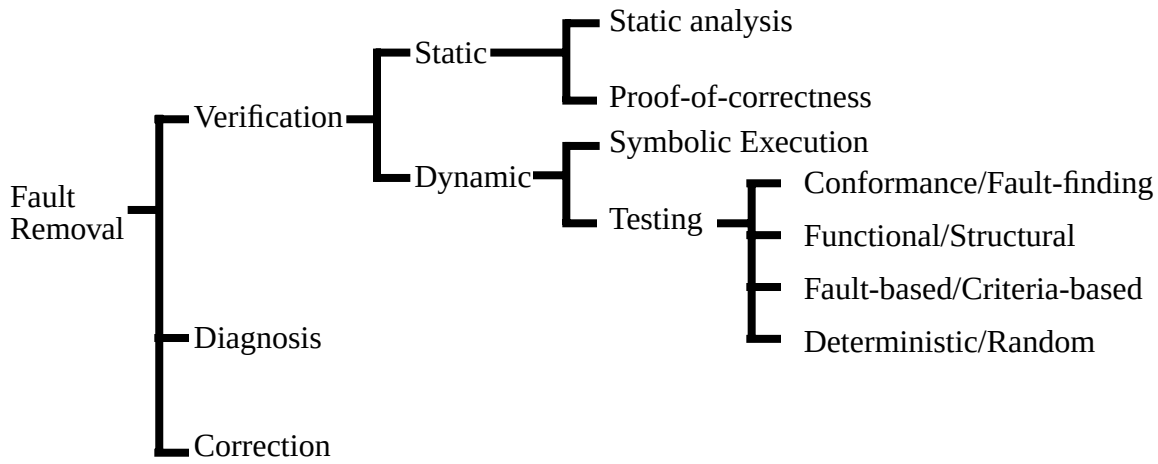

Figure 4-5: Fault Removal [Laprie 94]

As shown in Figure 4-4, ***fault removal*** is composed of three steps: *verification*, *diagnosis*, and *correction*. The steps are performed in that order: after it has been determined that the system does not match its specifications through verification the problem is diagnosed and, hopefully, corrected. The system must then be verified again to ensure that the correction succeeded.

Static verification involves checking the system without actually running it. Formal verification [Craigen 87] is one form of static verification. Code inspections or walk-throughs [Myers 79] is another.

Dynamic verification involves checking the system while it is executing. The most common form of dynamic verification is *testing*. Exhaustive testing is typically impractical. *Conformance testing* checks whether the system satisfies its specification. *Fault-finding testing* attempts to locate faults in the system. *Functional testing* (otherwise known as blackbox testing) tests that the system functions correctly without regard to implementation. *Structural testing* (otherwise known as whitebox testing) attempts to achieve path coverage to ensure that the system is

implemented correctly. *Fault-based testing* is aimed at revealing specific classes of faults. *Criteria-based testing* attempts to satisfy a goal such as boundary value checking. Finally, the generation of test inputs may be *deterministic* or *random*.

The above viewpoints may be combined. For example, the combination of fault-finding, structural, and fault-based testing is called mutation testing [DeMillo 78] when applied to software.

4.4.3 Fault Forecasting

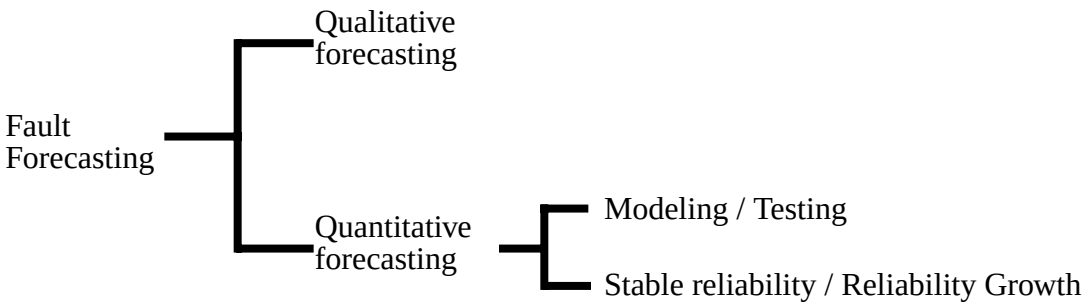

Figure 4-6: Fault Forecasting [Laprie 94]

As shown in Figure 4-4, ***fault forecasting*** can be qualitative or quantitative. *Qualitative forecasting* is aimed at identifying, classifying and ordering the failure modes, or at identifying the event combinations leading to undesired events. *Quantitative forecasting* is aimed at evaluating, in probabilistic terms, some of the measures of dependability.

There are two main approaches to quantitative fault forecasting which are aimed at deriving probabilistic estimates of the dependability of the system. These are *modeling* and *testing*. The approaches towards modeling a system differ based on whether the system is considered to be stable (that is, the systems level of reliability is “unchanging”) or in reliability growth (that is, the reliability of the system is improving over time as faults are discovered and removed.)

Evaluation of a system in *stable reliability* involves constructing a model of the system and then processing the model. *Reliability growth* models [Laprie 90] are aimed at performing reliability predictions from data relative to past system failures.

5 Security

5.1 Overview

5.1.1 Context of the Security Attribute

The definition of a security attribute depends on the context in which the attribute is addressed. Historically, there have been three main areas which have addressed security: government and military applications; banking and finance; and academic and scientific applications. In each of these cases, different aspects of security were stressed, and the definition of individual security attributes depended upon the stressed security aspects.

5.1.1.1 Government and Military

For government and military applications, the disclosure of information was the primary risk that was to be averted at all costs. To achieve this, applications and operating systems were developed to address the separation of data and processes through hardware and software designs that mimicked the existing system of classified documents. The standards culminated in the Orange Book - DoD 5200.28.STD and its related interpretations (collectively known as the Rainbow Series). These documents contained a model, architecture, and method of evaluation and rating for secure computing.

5.1.1.2 Banking and Fiance

In banking, finance, and business-related computing, the security emphasis is on the protection of assets. While disclosure is an important risk, the far greater risk is the unauthorized modification of information. Protecting the integrity of information produces trust from the customers, and thus confidence in the institution responsible for maintaining these data and processes. Unlike the DoD, there is no single standard that addresses these concerns, and in each case the integrity of the systems and applications are embodied in the detailed requirements of the systems to be developed or procured. Due to a lack of standardization in the definition of these requirements, the resulting effectiveness in terms of implemented security attributes varies widely.

5.1.1.3 Academic and Scientific

For academic and scientific computing, the main security emphasis is on protection from unauthorized use of resources. This stems from the time when computers and computing time was very expensive and a critical resources to research and scientific applications. This emphasis has led to the standards that exist in system administration and intrusion detection on large shared networks such as the Internet.

In the following sections, the definition and taxonomy for the security attribute will be attempted in a generic context. One that would apply to any of the above situations. Where appropriate, the relevant standards for each context are identified.

5.1.2 Definition

A general definition of security is provided in Appendix F of the National Research Council's report, "Computers at Risk":

1. Freedom from danger; safety.
2. Protection of system data against disclosure, modification, or destruction. Protection of computer systems themselves. Safeguards can be both technical and administrative.
3. The property that a particular security policy is enforced, with some degree of assurance.
4. Often used in a restricted sense to signify confidentiality, particularly in the case of multilevel security.

In the case of the security attribute, the second and third definitions apply. The main elements of the taxonomy, then, are the protection from disclosure (confidentiality), modification (integrity), and destruction (availability). Each of these elements must be addressed in the context of an overall security policy. This security policy sets the context for how to establish requirements and evaluate the effectiveness of each of these general categories of security. It is security policy that distinguishes between the environments, as was specified in the introduction.

The existing models have thus far focused primarily on the security policy that stresses confidentiality above all else, which leads to the fourth definition from *Computers at Risk*, as well as the treatment of security in other software attribute papers such as [Rushby 94].

5.1.3 Taxonomy

Most existing security taxonomies are based on a risk analysis of a specific environment; that risk analysis is then used as a framework to describe either the security faults or protection mechanisms in the system. As an example, the taxonomy described in [Aslam 95] is centered around security faults discovered in the UNIX operating system. This taxonomy decomposes coding faults into units that cover typical mistakes during the engineering of software. The difficulty is that this type of taxonomy does little to suggest how to handle security requirements or trade off the engineering methodologies for other quality attributes. In [Rushby 94], security is balanced with other quality attributes, but the definition and coverage of the security attribute is restricted to confidentiality.

To bring in other aspects of security and compare them to other quality attributes, the concerns of security are broken down into the three basic categories of confidentiality, integrity, and availability. From these concerns, the security factors at the boundary of the systems (the interface or environment), and the internal factors can be identified. Once the concerns and factors are identified, the current broad approaches for synthesis and analysis are identified.

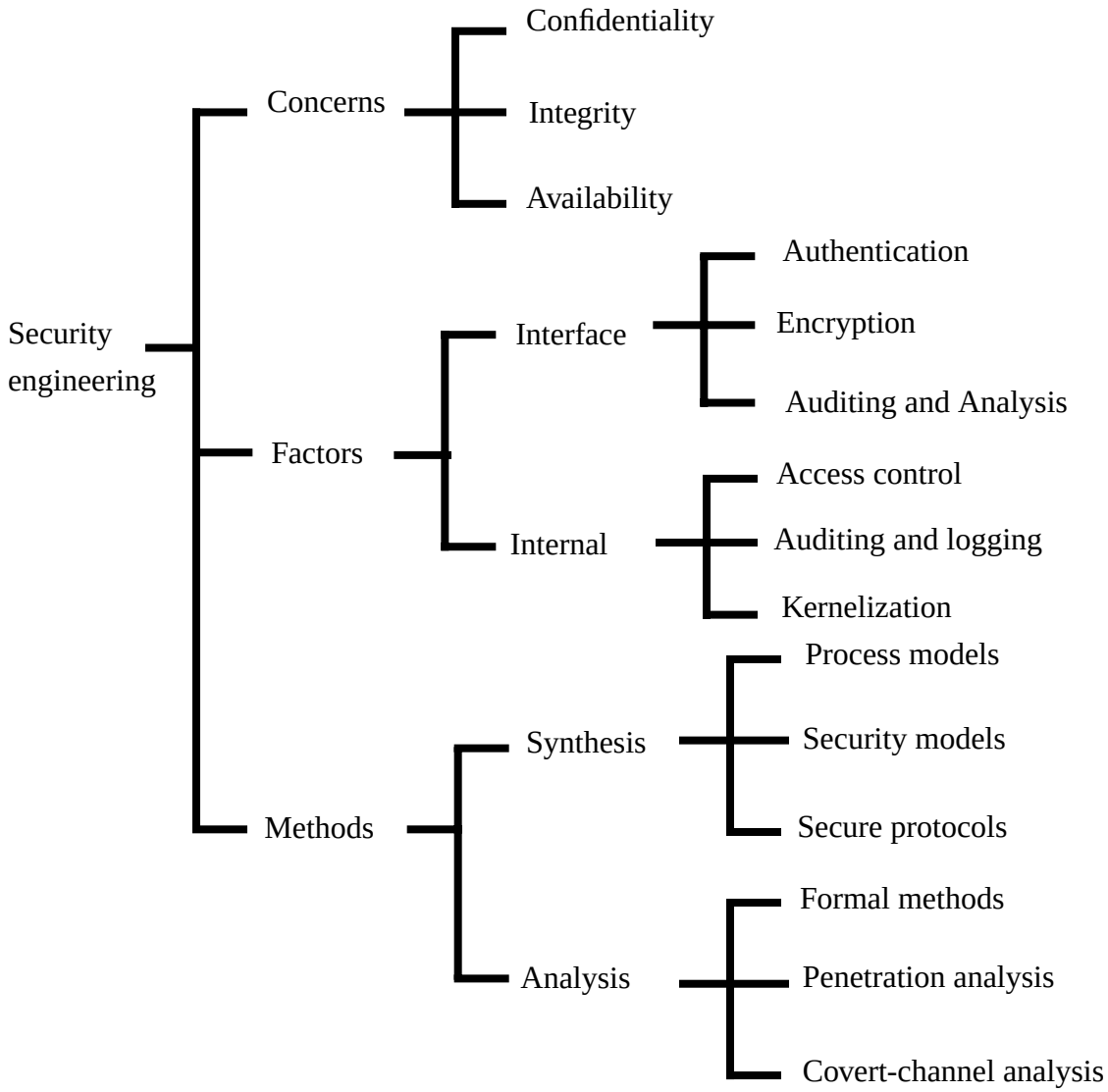

Figure 5-1: Security Taxonomy

5.2 Concerns

The security concerns for any given environment (based on a security policy) can be categorized into three basic types:

- ***Confidentiality*** is the requirement that data and processes be protected from unauthorized disclosure.
- ***Integrity*** is the requirement that data and process be protected from unauthorized modification.
- ***Availability*** is the requirement that data and processes be protected from denial of service to authorized users.

5.2.1 Confidentiality

Confidentiality is the property that data be unaccessible to unauthorized users. Usually, this requirement is specified in terms of a security policy, which in turn places requirements on the design and implementation of a system. For example, in a military environment it may be necessary to process both secret and confidential data on a single system. The confidentiality requirement, then, is that access to the secret information be restricted to only those users with the appropriate clearance.This requirement has a strong impact on the design of the file system, access control, process control, authentication, and administration of the resulting system.

A fault concerning confidentiality results in the unauthorized disclosure of information or process control.This fault can occur in the normal operation of the system by a fault in the implementation or in the interface through inadequate design specification.

The strength of confidentiality in a system is usually measured in the resources required to disclose information in a system. For a communication system, this may be stated as the time it would take an adversary with the resources of a foreign power to read a communication copied during transit (this measure is often used for data encryption). An internal measure of confidentiality may be to restrict the bandwidth of covert channels[1] to a given number of bits per second.

5.2.2 Integrity

Integrity is the property that the data be resistant to unauthorized modification. Like confidentiality, this must be in relation to a security policy that defines which data should be modified by whom so that there is a clear definition of unauthorized modification. One example of this is that the password file on UNIX systems should be modified only by the root user. A requirement associated with integrity is often specified as a file access requirement. For operating systems or database systems, this is specified as write access to a file. In more general terms, the integrity requirement may be used for either data or processes to specify how modifications are made to data or how control is passed to processes.

An integrity fault results in unauthorized modification or destruction of data or processes in a system. In the case of a “trusted” system, a loss of integrity may also lead to a loss of trust in all down-stream or dependent data or processes in a system. In this way, loss of integrity may be propagated through the dependencies associated with the original information modified without authorization. For example, if the underlying operating system of a financial computer is modified, it may cause all data processed by this system to be modified without authorization in violation of policy.

1. Covert channels are communication of information through data paths not explicitly specified during the design, such as locking IO devices or controlling the number of processes started.

Integrity is usually measured by the time and resources it would take an adversary to modify data or processes without authorization. These measures are often subjective or dependent on the average time to guess a specific integrity checksum. When cryptographic methods are used to guarantee the integrity of a system, the metrics are very similar to those for confidentiality as described above. In addition, however, integrity measures are often associated with mean time to failure in software systems, as these failures are equivalent to unauthorized modification.

5.2.3 Availability

Availability is the property that the resources that should be available to authorized user actually are available. This property is closely associated with availability in other quality attribute domains (i.e., safety and dependability), but is usually defined in terms of the amount of time it would take an active intruder to cause a denial of service. A fault associated with availability is a denial of service attack. Unlike the other quality attributes, a fault associated with availability in the security attribute is a denial of service caused by an adversary rather than a random fault of hardware or software.

As in dependability, availability is usually measured proportional to mean time to failure (see the definition in the section on Dependability). During the requirement definition or design of a system, availability is required for security critical aspects of any given system. For example, it is usually required that the auditing and alarm systems in a secure operating system be available whenever it is possible to start processes on a system.

5.3 Security Factors

The factors regarding security are grouped according to whether they are associated with the interface to a system, or are internal to the operation of a system. Both sets of factors are commonly known as security features.

5.3.1 Interface

The interface factors are those security features that are available for the user or between systems. The main types of interface features are

- *authentication* services
- *encryption* services
- *auditing and analysis* services

Authentication services are those that perform the mapping between the user's identity within the system or application and the person or system accessing the system. This service is essential for many of the concerns in security, as most of the internal security decisions rely on correctly identifying and authenticating the user or system. There are many types of authentication, including password, bio-metric, third-party, and capability-based.

Encryption services are data or control protection between the internal system or application and the user accessing the interface. These may take place on a link between systems where the isolation of the intermediate transfer mechanism cannot be assured. Encryption services may also serve to verify the integrity of information through the interface by using cryptographically strong checksum information. Encryption services are often employed in protocols between system components or across communication links.

Auditing and analysis services are used primarily for the security administrator of a system to detect unauthorized activities or to discover the history of some access or transaction. These security services often serve as an alarm to alert an outside user of a policy violation detected on some internal component.

5.3.2 Internal

Internally, security factors take a variety of design strategies. There are no generally accepted principles for the internal security factors, but three common areas for security factors are in the access control system, a secure kernel, and in auditing and logging.

Access control refers to all access to internal objects. These include data and processes and access by both internal objects and external users. Data access is usually accomplished through a file system abstraction; the types of access control depend on how objects and data are described in the security policy. One common model for access control is the access matrix (as embodied in the Bell-LaPadula model described in DoD 5200.28). Access to processes usually centers around ownership of the process, but in some secure systems the process is treated as a data object with the same set of access control restrictions as is provided by the file system abstraction.

Kernelization is the abstraction of all security-related functionality to a small (and hopefully provably secure) kernel with a strictly defined interface for the rest of the system. This is the preferred design for DoD secure systems as it abstracts and contains all of the critical security functionality to a small subset of the overall system.

The *auditing and logging* security features are often used as add-on security features to applications and operating systems that were not originally designed with strong security in mind. These internal factors assure that any action taken internally can be logged and audited so that in the event of a security violation the actions may be attributed to the base cause. These features are also used in conjunction with access control and kernelization to provide tracing ability in secure systems.

5.4 Methods

Methods to achieve security include the following:

Synthesis—Methods used to synthesize a secure system include process models such as the Trusted System Design Methodology (TSDM94) or the Trusted Capability Maturity Model (TCMM95), security models such as the Mach Security Kernel, and secure protocols such as Kerberos.

Analysis—Techniques used to analyze system security include formal method, penetration analysis, and covert-channel analysis.

5.4.1 Synthesis

Process models. The most common technique for developing a secure computing system as regulated by DoD 5200.28 is to use a process model that involves formal design, integration, and testing. Two recent additions to this description are the TSDM and the TCMM.

Security models. Another method of synthesis is to modify an existing security model for design and implementation to suit another application or system. The Mach Security Kernel is a kernelized model and reference implementation that is often used as a basis to synthesize new systems taking into account security requirements. The reuse of other security components such as auditing or intrusion detection tools is another method of synthesizing a complex system from base components.

Secure protocols. For other distributed applications, a standard security protocol may be used to build security functionality on existing or new applications. The Kerberos family of protocols uses a third-party authenticating mechanism and well defined interface to address security concerns within systems and applications.

5.4.2 Analysis

Formal Methods. For highly secure systems, formal analysis of the design and specification of the system is used to verify that the design of the system meets the requirements and specification of the security policy.

Penetration Analysis. For most systems that address security, penetration analysis is performed during the testing phases of the system. This employs standard attack scenarios to determine if the system is resilient to these attacks. This analysis has the drawback of not addressing attacks unknown at the time of the test.

Covert-Channel Analysis. Covert-channel analysis is usually performed on multi-level secure systems as specified in DoD 5200.28 to determine the bandwidth of any secondary data channel that is identified in the system.

6 Safety

6.1 Overview

6.1.1 Definition

As previously stated, dependability is that property of a computer system such that reliance can justifiably be placed in the service it delivers [Laprie 94].

Paraphrasing this definition, we can define safety as that property of a computer system such that reliance can justifiably be placed in the absence of accidents.

- Dependability is concerned with the occurrence of failures, defined in terms of internal consequences (services are not provided).
- Safety is concerned with the occurrence of accidents or ***mishaps***, defined in terms of external consequences (accidents happen).

The difference of intents—"good things (services) must happen" vs. "bad things (accidents) must not happen"—gives rise to the following paradox: If the services are specified incorrectly, a system can be dependable but unsafe; conversely, it is possible for a system to be safe but undependable.

- A system might be dependable but unsafe — for example, an avionics systems that continues to operate under adverse conditions yet directs the aircraft into a collision course.
- A system might be safe but undependable — for example, a railroad signaling system that always fails-stops.

6.1.2 Taxonomy

The taxonomy for the safety attributes defines conditions of the system (hazards) that might lead to undesirable consequences (mishaps); methods normally used to identify hazards, evaluate the consequences of a hazard, and eliminate or reduce the possibility of mishaps; and indicators of safety in the aggregate (system, environment, users and operators).

6.2 Concerns

Perrow [Perrow 84] identifies two properties of critical systems that can serve as indicators of system safety: ***interaction complexity*** and ***coupling strength***.

6.2.1 Interaction Complexity

Interaction complexity ranges from linear to complex and is the extent to which the behavior of one component can affect the behavior of other components. **Linear interactions** are those in expected and familiar production or maintenance sequence, and those that are quite visible

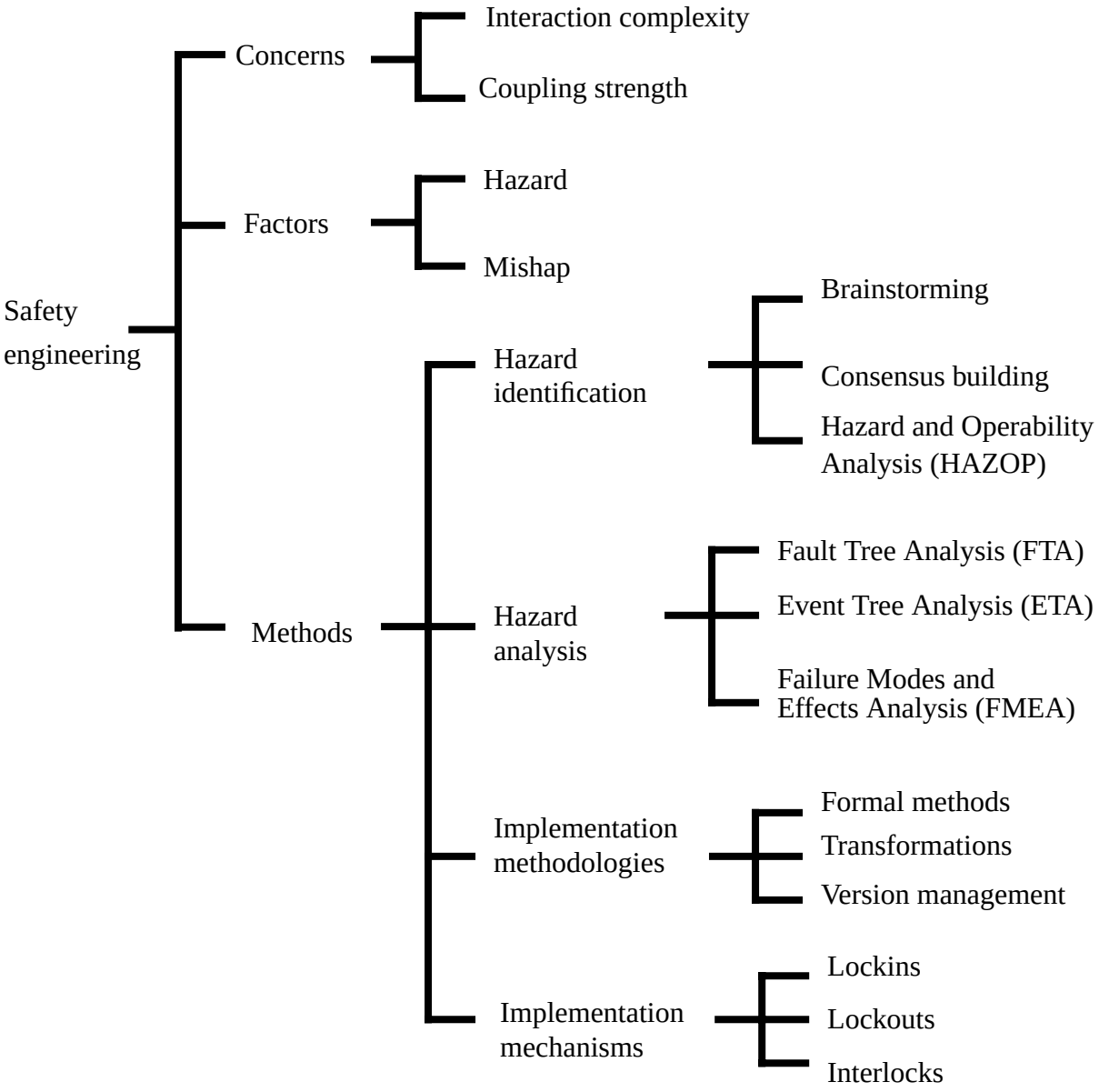

Figure 6-1: Safety Taxonomy

even if unplanned. **Complex interactions** are those of unfamiliar sequences, or unplanned and unexpected sequences, and either not visible or not immediately comprehensible. [Perrow 84, Table 3.1] suggests the following indicators of interaction complexity:

Indicators of complex interactions include

- proximity—physical (components) or logical (steps)
- common-mode connections
- interconnected subsystems

- limited isolation or substitution of failed components
- unfamiliar or unintended feedback loops
- multiple and interacting control parameters
- indirect or inferential information sources
- limited understanding of some processes

Indicators of linear interactions include

- segregation between components or steps
- dedicated connections
- segregated subsystems
- easy isolation and substitutions
- few feedback loops
- single purpose, segregated controls
- direct, on-line information
- extensive understanding

6.2.2 Coupling Strength

Coupling strength ranges from **loose coupling** to **tight coupling** and is the extent to which there is flexibility in the system to allow for unplanned events. **Tightly coupled** systems have more time-dependent processes: they cannot wait or stand by until attended to; the sequences are more invariant and the overall design allows for very limited alternatives in the way to do the job; they have “unifinality”—one unique way to reach the goal. **Loosely coupled** processes can be delayed or put in standby; sequences can be modified and the system restructured to do different jobs or the same job in different ways; they have “equifinality”—many ways to reach the goal. [Perrow 84, Table 3.2] suggests the following indicators of coupling strength:

Indicators of tight coupling include

- delays in process not possible
- invariant sequences
- only one method to achieve goal
- little slack [in resources] possible
- buffers and redundancies are designed-in, deliberate
- substitutions [of resources] limited and designed-in

Indicators of loose coupling include

- processing delays possible
- order of sequences can be changed
- alternative methods available

- slack in resources possible
- buffers and redundancies fortuitously available
- substitutions fortuitously available

6.2.3 Advantages and Disadvantages

There are advantages and disadvantages of the extremes of interaction and coupling [Rushby 93].

Complex interactions can be undesirable because interactions and their consequences can be hard to understand, predict, or even enumerate. In general, hazard analysis demands few linear and known interactions to facilitate analysis.

Tight coupling can be undesirable because the system can be hard to adapt to changing situations. Safety mechanisms demand loose coupling to prevent cascading of failures and to allow reconfigurations and intervention by operators.

Nevertheless, complex interactions and tight coupling are often desirable to promote performance (shared address space), dependability (N-modular redundancy, transactions), or security (authentication protocols, firewalls).

Finally, the degree of interaction and coupling could be inherent to the application or problem domain. Smarter design or experience might reduce them, but often we do not have many choices.

6.3 Factors

Hazards are conditions (i.e., state of the controlled system) that can lead to a mishap.

Mishaps are unplanned events that result in death, injury, illness, damage or loss of property, or harm to the environment.

The occurrence or non-occurrence of a mishap may depend on conditions beyond the control of the system thus, in safety engineering, attention is focused on preventing hazards rather that preventing mishaps directly.

6.4 Methods

The safety engineering approach consists of

- hazard identification and analysis processes
- implementation methodologies and mechanisms

Hazard identification and *hazard analysis* are performed at several different stages of the design lifecycle (e.g., preliminary, subsystem, system, operational hazard analysis).

The objective of the *implementation methodologies* and mechanisms is to avoid the introduction of errors during the development process or to detect and correct errors during operation.

Notions from system safety engineering can be applied to software — the basic idea is to focus on consequences that must be avoided rather than on the requirements of the system itself. Techniques proposed to conduct hazard identification and analysis in software-intensive systems are derived from well-known techniques used in industry (e.g., chemical process). However, software-specific hazard identification and analysis techniques are not well established and lack adequate integrated tools.

6.4.1 Hazard Identification

Hazard identification attempts to develop a list of possible system hazards before the system is built. This can be expensive and time consuming and must be performed by application domain experts.

- *Brainstorming* — experts generate list of possible system hazards until some threshold (e.g., time to identify new hazards) is reached.
- *Consensus techniques* — facilitated iteration among experts with specific responsibilities and well defined goals. Example techniques are Delphi and Joint Application Design.
- ***Hazard and Operability Analysis*** *(HAZOP)* — evaluates a representation of a system and its operational procedures to determine if humans or environment will be exposed to hazards and the possible measures that might be employed to prevent the mishap. The procedure is to search the representation, element by element, for every conceivable deviation from its normal operation, followed by group discussions of causes and consequences.

6.4.2 Hazard Analysis

Following the identification of a hazard, the hazard analysis process consists of the following risk mitigation steps:

1. Categorize hazard on a scale from catastrophic to negligible. Catastrophic hazards have the potential to lead to extremely serious consequences. Negligible hazards have no significant consequences.
2. Determine how or whether that hazard might arise using backward reasoning from the hazard (*what could possibly cause this situation?*) or forward reasoning from the hypothesized failure (*what could happen if this failure occurs?*)
3. Remove hazards with unacceptable risk through re-specification, redesign, incorporating safety features or warning devices, or instituting special operating and training procedures.

Common techniques for hazard analysis include ***Fault Tree Analysis*** (FTA), ***Event Tree Analysis*** (ETA) and ***Failure Modes and Effects Analysis*** (FMEA).

Fault Tree Analysis (FTA) is a technique that was first applied in the 1960s to minimize the risk of inadvertent launch of a Minuteman missile. The hazard to be analyzed is the root of the tree.

- Necessary preconditions for the hazard are describing at the next level in the tree, using AND or OR relationships to link subnodes
- Subnodes are expanded in similar fashion until all nodes describe events of calculable probability or are incapable of further analysis for some reason.

Software Fault Tree Analysis (SFTA) is an adaptation to software of a safety engineering analysis methodology. The goal of SFTA is to show that the logic contained in the software design will not cause mishaps, and to determine conditions that could lead to the software contributing to a mishap. The process is as follows:

1. Use hazard analysis to identify a possible condition for a mishap.
2. Assume that software has caused the condition.
3. Work backwards to determine the set of possible causes (including environment, hardware, and operator) for the condition to occur.

SFTA differs from conventional software inspection techniques in that it forces the analysis to examine the program from a different perspective than that used in development.

Event Tree Analysis (ETA) reverses the order followed in FTA. Starting with some initiating (desirable or undesirable) event, a tree is developed showing all possible (desirable and undesirable) consequences. This technique requires judicious selection of the initiating events to keep the cost and time required for within reason.

Failure Modes and Effects Analysis (FMEA) attempts to anticipate potential failures so that the sources of these failures can be eliminated.

An FMEA table identifies, for each component failure

- the frequency of occurrence (rare to common)
- the severity of the effect (minor to very serious)
- the chances of detection before deployment (certain to impossible)

The product of all three elements is a “risk priority number” which can be used to determine how effort should be spent during development.

FMEA uses both Event Tree Analysis (to determine the effects of a component failure) and Fault Tree Analysis (to determine the cause of a component failure) iteratively.

Frequency of occurrence, severity of failure, and chances of detection are simple integer values (e.g., 1 to 10) and are assigned based on knowledge and experience of the developers.

An extension of FMEA is Failure Modes, Effects, and Criticality Analysis (FMECA)—it uses a more formal criticality analysis to rank the results than just the result of multiplying the three factors.

6.4.3 Implementation Methodologies

The objective is to avoid introducing errors during the development process and, if unavoidable, detect and correct them during operation.

Different implementation methodologies are applicable during the development phases.

Requirements — Specification, analysis and validation using notations with various degrees of formality (e.g., Petri-nets, state machines, Statecharts). Requirements expressed in natural languages are often ambiguous or incomplete. The first step must be to represent the requirements in a notation that is not ambiguous and can be analyzed, that is, generate a specification of the behavior of the system (and validate it with the customer!).

- The specification must be analyzed for inconsistencies and incompleteness, although the latter depends on the expertise of the analysts rather than the specification technique used.
- Some specifications can be “executed” and the behavior of the system can thus be simulated; other specifications can be validated through symbolic execution to predict behavior given some initial state and a set of inputs

Design — Using *formal methods* —i.e., a formal design notation—and proving that it satisfies the specification or derive the design by transformation of the specification. There are a number of design notations; however, errors can be introduced that can only be detected by comparing to the specification. This proof can be hard, and the use of two different notations makes tracing requirements to design more difficult.

Implementation — Strict *version management* to retain confidence that the source code that is analyzed is the code used to build the system and formally verified source code translators (and hardware). Successive *transformations* of the specification reduces the introduction of errors but there are now many more representations of the system—making the requirements tracking even more difficult.

6.4.4 Implementation Mechanisms

Traditional *implementation mechanisms* employed in safety-engineering include

- ***lockins*** — lock the system into safe states
- ***lockouts*** — lock the system out of hazardous states
- ***interlocks*** — prescribe or disallow specific sequences of events

There are software analogs to these mechanisms but they must be implemented carefully — a “design for safety” must keep the system always in a safe state, even if the service is not available (i.e., the system is “unreliable”).

For example, monitoring a set of variables that must be between certain limits for safe operation. The naive approach might declare the variables to be “OK” by default and then do a linear scan to see if one of them is off-bounds in order to trigger a safety shutdown. If the scan is interrupted or stalled, certain variables that should be tripped might not be examined, and the safety shutdown might not occur when it should.

7 Relationships Between Attributes

Each of the attributes examined has evolved within its own community. This has resulted in inconsistencies between the various points of view.

7.1 Dependability Vis-a-vis Safety

The dependability tradition tries to capture all system properties (e.g., security, safety) in terms of dependability concerns—i.e., defining failure as “not meeting requirements.” It can be argued that this is too narrow because requirements could be wrong or incomplete and might well be the source of undesired consequences. A system could allow breaches in security or safety and still be called “dependable.”

The safety engineering approach explicitly considers the system context. This is important because software considered on its own might not reveal the potential for mishaps or accidents. For example, a particular software error may cause a mishap or accident only if there is a simultaneous human and/or hardware failure. Alternatively, it may require an environment failure to cause the software fault to manifest itself.

For example [Rushby 93], a mishap in an air traffic control system is a mid-air collision. A mid-air collision depends on a number of factors:

- the planes must be too close
- the pilots are unaware of that fact or
- the pilots are aware but
- fail to take effective evading action
- are unable to take effective evading action
- etc.

The air traffic control system cannot be responsible for the state of alertness or skill of the pilots; all it can do is attempt to ensure that the planes do not get too close together in the first place.

Thus, the hazard (i.e., erroneous system state that leads to an accident) that must be controlled by the air traffic control system is, say, “planes getting closer than two miles horizontally, or 1,000 feet vertically of each other.”

7.2 Precedence of Approaches

Safe software is always secure and reliable — Neumann [Neumann 86] presents a hierarchy of reliability, safety, and security. Security depends on reliability (an attribute of dependability) and safety depends on security, hence, also reliability.

- A secure system might need to be reliable because a failure might compromise the system's security (e.g., assumptions about atomicity of actions might be violated when a component fails).
- The safety critical components of a system need to be secure to prevent accidental or intentional alteration of code or data that were analyzed and shown to be safe.
- Finally, safety depends on reliability when the system requires the software to be operational to prevent mishaps.

Enhancing reliability is desirable, and perhaps necessary, but it is not sufficient to ensure safety. As noted in [Rushby 93], the relationships are more complex than a strict hierarchy:

- Fault tolerant-techniques can detect security violations — Virus detected through N-version programming, intrusions detected automatically as latent errors, and denial detected as omission or crash failures.
- Fault containment can enhance safety by ensuring that the consequences of a fault do not spread and contaminate other components of a system.
- Security techniques can provide fault containment through memory protection, control of communications, and process walls.
- A security kernel can enforce safety using runtime lockin mechanisms for "secure" states and interlocks to enforce some order of activities. Kernelization and system interlocks are primarily mechanisms for avoiding certain kinds of failure and do very little to ensure normal service.
  - A kernel can achieve influence over higher levels of the system only through the facilities it does *not* provide — if a kernel provides no mechanism for achieving certain behaviors, and if no other mechanisms are available, then no layers above the kernel can achieve those behaviors.
  - The kinds of behaviors that can be controlled in this way are primarily those concerning communication, or the lack thereof. Thus, kernelization can be used to ensure that certain processes are isolated from each other, or that only certain inter-process communication paths are available, or that certain sequencing constraints are satisfied.
  - Kernelization can be effective in avoiding certain faults of commission (doing what is not allowed) but not faults of omission (failing to do what is required)—that is, a security kernel cannot ensure that the processes correctly perform the tasks required of them.

7.3 Applicability of Approaches

The methods and mind set associated with each of the attributes examined in this report have evolved from separate schools of thought. Yet there appear to be common underpinnings that can serve as a basis for a more unified approach for designing critical systems. For example:

- Safety and dependability are concerned with detecting error states (errors in dependability and hazards in safety) and preventing error states from causing undesirable behavior (failures in dependability and mishaps in safety).
- Security and performance are concerned with resource management (protection of resources in security and timely use of resources in performance.)

The previous section offered examples of the applicability of methods usually associated with one attribute to other attributes.

The applicability of methods developed for one attribute to another attribute suggests that differences between attributes might be as much a matter of sociology as technology. Nevertheless, there are circumstances for which an attribute-specific mind set might be appropriate. Examples include the following:

- The dependability approach is more attractive in circumstances for which there is no safe alternative to normal service—a service *must* be provided (e.g., air traffic control).
- The safety approach is more attractive where there are specific undesired events — an accident *must* be prevented (e.g., nuclear power plant).
- The security approach is more attractive when dealing with faults of commission rather than omission — service *must not* be denied, information *must not* be disclosed.

This is not to suggest that other attributes could be ignored. Regardless of what approach is chosen, we still need a coordinated methodology to look at all of these attributes together, in the context of a specific design. In the next chapter we sketch a plan of activities that would lead to an attribute-based methodology for evaluating the design of an artifact—more specifically, for evaluating a software architecture with respect to these attributes.

8 Quality Attributes and Software Architecture

A (software) system architecture must describe the system's components, their connections and their interactions, and the nature of the interactions between the system and its environment. Evaluating a system design before it is built is good engineering practice. A technique that allows the assessment of a candidate architecture before the system is built has great value.

The architecture should include the factors of interest for each attribute. Factors shared by more than one attribute highlight properties of the architecture that influence multiple attribute concerns and provide the basis for trade-offs between the attributes. A mature software engineering practice would allow a designer to predict these concerns through changes to the factors found in the architecture, before the system is built.

We intend to continue our work by exploring the relationships between quality attributes and software architectures. All the attributes examined in this report seem to share classes of factors. There are events (generated internally or coming from the environment) to which the system responds by changing its state. These state changes have future effects on the behavior of the system (causing internal events or responses to the environment). The "environment" of a system is an enclosing "system," and this definition applies recursively, up and down the hierarchy. For example varying arrival patterns (events) cause system overload (state) that lead to jitter (event); faults (events) cause errors (state) that lead to failure (events); hazards (events) cause safety errors that lead to mishaps (events); intrusions (events) cause security errors that lead to security breaches (events). Additional classes of factors to consider include the policies and mechanisms used for process creation, allocation, address space sharing, connection, communication method, interaction style, synchronization, and composition of actions.

Architecture patterns are the building blocks of a software architecture. Examples of patterns include pipes-and-filters, clients-and-servers, token rings, blackboards, etc. The architecture of a complex system is likely to include instances of more than one of these patterns, composed in arbitrary ways. Collections of architecture patterns should be evaluated in terms of quality factors and concerns, in anticipation of their use. That is, it is conceivable that architecture patterns could be "pre-scored" to gain a sense of their relative suitability to meet quality requirements should they be used in a system.

In addition to evaluating individual patterns, it is necessary to evaluate compositions of patterns that might be used in an architecture. Identifying patterns that do not "compose" well (i.e., the result is difficult to analyze or the quality factors of the result are in conflict with each other) should steer a designer away from "difficult" architectures, towards architectures made of well behaved compositions of patterns.

In the end, it is likely that we will need both quantitative and qualitative techniques for evaluating patterns and architectures. Promising quantitative techniques include the various modeling and analysis techniques, including formal methods mentioned in this report. An example of a qualitative technique is being demonstrated in a related effort at the SEI. The Software Architecture Analysis Method (SAAM) [Clements 95, Kazman 95] illustrates software architecture evaluations using “scenarios” (postulated set of uses or transformations of the system). Scenarios are rough, qualitative evaluations of an architecture; scenarios are necessary but not sufficient to predict and control quality attributes and have to be supplemented with other evaluation techniques (e.g., queuing models, schedulability analysis). Architecture evaluations using scenarios should be enriched by including questions about quality indicators in the scenarios.

Appendix A Glossary

accidental faults — faults created by chance.

active fault — a fault which has produced an error.

aperiodic — an arrival pattern that occurs repeatedly at irregular time intervals. The frequency of arrival can be bounded by a minimum separation (also known as sporadic) or can be completely random.

attribute specific factors — properties of the system (such as policies and mechanisms built into the system) and its environment that have an impact on the concerns

availability — a measure of a system's readiness for use.

benign failure — a failure that has no bad consequences on the environment.

Byzantine failure — a failure in which system users have differing perceptions of the failure.

capacity — a measure of the amount of work a system can perform.

catastrophic failure — a failure that has bad consequences on the environment it operates in.

complex interactions — those of unfamiliar sequences, or unplanned and unexpected sequences, and either not visible or not immediately comprehensible.

component coupling — the extent to which there is flexibility in the system to allow for unplanned events. Component coupling ranges from tight (q.v.) to loose (q.v.)

confidentiality — the non-occurrence of the unauthorized disclosure of information.

consistent failure — a failure in which all system users have the same perception of the failure.

criticality — the importance of the function to the system.

dependability — that property of a computer system such that reliance can justifiably be placed on the service it delivers.

dependability impairments — the aspects of the system that contribute to dependability.

dormant fault — a fault that has not yet produced an error.

error — a system state that is liable to lead to a failure if not corrected.

event — a stimulus to the system signaling the need for the service.

event stream — a sequence of events from the same source—for example, a sequence of interrupts from a given sensor.

Event Tree Analysis (ETA) — a technique similar to Fault Tree Analysis. Starting with some initiating (desirable or undesirable) event, a tree is developed showing all possible (desirable and undesirable) consequences.

fail-safe — a system which can only fail in a benign manner.

fail-silent — a system which no longer generates any outputs.

fail-stop — a system whose failures can all be made into halting failures.

failure — the behavior of a system differing from that which was intended.

Failure Modes and Effects Analysis (FMEA) — a technique similar to Event Tree Analysis (ETA). Starting with potential component failures, identifying its consequences. and assigning a “risk priority number” which can be used to determine how effort should be spent during development.

Failure Modes, Effects, and Criticality Analysis (FMECA) — an extension of Failure Modes Effects Analysis (FMEA) that uses a more formal criticality analysis.

fault — the adjudged or hypothesized cause of an error.

fault avoidance — see *fault prevention*.

fault forecasting — techniques for predicting the reliability of a system over time.

fault prevention — design and management practices which have the effect of reducing the number of faults that arise in a system.

fault removal — techniques (e.g., testing) involving the diagnosis and removal of faults in a fielded system.

fault tolerance — runtime measures to deal with the inevitable faults that will appear in a system.

Fault Tree Analysis (FTA) — a technique to identify possible causes of a hazard. The hazard to be analyzed is the root of the tree and each necessary preconditions for the hazard or condition above are described at the next level in the tree, using AND or OR relationships to link subnodes, recursively

halting failure — a special case of timing failure wherein the system no longer delivers any service to the user.

hazard — a condition (i.e., state of the controlled system) that can lead to a mishap.

Hazard and Operability Analysis (HAZOP) — evaluates a representation of a system and its operational procedures to determine possible deviations from design intent, their causes, and their effects.

human-made faults — those resulting from human imperfection.

impairments to dependability — those aspects of the system that contribute to how the system (mis)behaves from a dependability point of view.

inconsistent failure — see *Byzantine failure*.

integrity — the non-occurrence of the improper alteration of information.

intentional faults — faults created deliberately, with or without malicious intent.

interaction complexity — the extent to which the behavior of one component can affect the behavior of other components. Interaction complexity ranges from linear (q.v.) to complex (q.v.).

interlocks — implementation techniques that prescribe or disallow specific sequences of events.

intermittent faults — a temporary fault resulting from an internal fault.

internal faults — those which are part of the internal state of the system.

jitter — the variation in the time a computed result is output to the external environment from cycle to cycle

latency — the length of time it takes to respond to an event.

latency requirement — time interval during which the response to an event must be executed.

latent error — an error which as not yet been detected.

linear interactions — interactions that are in expected and familiar production or maintenance sequence, and those that are quite visible even if unplanned.

lockins — implementation techniques that lock the system into safe states.

lockouts — implementation techniques that lock the system out of hazardous states

loose coupling — characterizes systems in which processes can be delayed or put in standby; sequences can be modified and the system restructured to do different jobs or the same job in different ways; they have “equifinality”—many ways to reach the goal.

maintainability — the aptitude of a system to undergo repair and evolution.

methods — how concerns are addressed: analysis and synthesis processes during the development of the system, procedures and training for users and operators.

mishaps — unplanned events that result in death, injury, illness, damage or loss of property, or environment harm.

mode — state of a system characterized by the state of the demand being placed on the system and the configuration of resources used to satisfy the demand.

observation interval — time interval over which a system is observed in order to compute measures such as throughput.

performance — responsiveness of the system—either the time required to respond to specific events or the number of events processed in a given interval of time.

performance concerns — the parameters by which the performance attributes of a system are judged, specified, and measured.

performance factors — the aspects of the system that contribute to performance.

periodic — an arrival pattern that occurs repeatedly at regular intervals of time.

permanent fault — a fault which, once it appears, is always there.

physical faults — a fault that occurs because of adverse physical phenomena.

precedence requirement — a specification for a partial or total ordering of event responses.

processing rate — number of event response processed per unit time.

quality — the degree to which software possesses a desired combination of attributes (e.g., reliability, interoperability) [IEEE 1061].

reliability — a measure of the rate of failure in the system that renders the system unusable. A measure of the ability of a system to keep operating over time.

response — the computation work performed by the system as a consequence of an event.

response window — a period of time during which the response to an event must execute; defined by a starting time and ending time.

safety — a measure of the absence of unsafe software conditions. The absence of catastrophic consequences to the environment.

safety indicators — the aspects of the system that contribute to safety.

schedulable utilization — the maximum utilization achievable by a system while still meeting timing requirements.

security factors — the aspects of the system that contribute to security.

service — a system's behavior as it is perceived by its user(s).

Software Fault Tree Analysis (SFTA) — an adaptation to software of a safety engineering analysis methodology. The goal of SFTA is to show that the logic contained in the software design will not cause mishaps, and to determine conditions that could lead to the software contributing to a mishap.

spare capacity — a measure of the unused capacity.

temporary fault — a fault which disappears over time.

throughput — the number of event responses that have been completed over a given observation interval.

tight coupling — characterizes systems that have more time-dependent processes: they cannot wait or stand by until attended to; the sequences are more invariant and the overall design allows for very limited alternatives in the way to do the job; they have "unifinality"—one unique way to reach the goal.

timing failure — a service delivered too early or too late.

transient fault — a temporary fault arising from the physical environment.

user of a system — another system (physical or human) which interacts with the former.

utilization — the percentage of time a resource is busy.

value failure — the improper computation of a value.

Bibliography

Anderson 85 — Anderson, T. Ch. 1, “Fault Tolerant Computing.” *Resilient Computing Systems*, London: Collins Professional and Technical Books, 1985.

Andrew 91 — Andrews, G.R. “Paradigms for Process Interaction in Distributed Programs.” *ACM Computing Surveys 23*, 1 (March 1991): 49-90.

Aslam 95 — Aslam, T. *A Taxonomy of Security Faults in the UNIX Operating System* (Master’s Thesis). West Lafayette, In: Purdue University, Department of Computer Science, 1985.

Audsley 95 — Audsley, N. C. et al. “Fixed Priority Pre-Emptive Scheduling: An Historical Perspective.” *Real-Time Systems 8*, 2-3 (March-May 1995): 173-198.

Avizienis 86 — Avizienis, A. and Laprie, J.C. “Dependable Computing: From Concepts to Design Diversity.” *Proceedings of the IEEE 74*, 5 (May 1986): 629-638.

Bacon 93 — Bacon, Jean. *Concurrent Systems—An Integrated Approach to Operating Systems, Database, and Distributed Systems*. Reading, Ma.: Addison-Wesley, 1993.

Barbacci 93 — Barbacci, M.R.; Weinstock, C.B.; Doubleday, D.L.; Gardner, M.J.; and Lichota, R.W. “Durra: A Structure Description Language for Developing Distributed Applications.” *Software Engineering Journal 8*, 2 (March 1993): 83-94.

Bell 71 — Bell, C.G. and Newell, A. *Computer Structures: Readings and Examples*. New York: McGraw-Hill, 1971.

Bell 76 — Bell, D.E. and La Padula, L.J. *Secure Computer Systems: Unified Exposition and Multics Interpretation* (Technical Report ESD-TR-75-306). Bedford, Ma.: MITRE Corporation, 1976.

Bishop 91 — Bishop, R. and Lehman, M. “A View of Software Quality,” 1/1-3. *IEEE Colloquium on Designing Quality into Software Based Systems*, London, UK, October 14, 1991. London: IEEE, 1991.

Boehm 78 — Boehm, B. et al. *Characteristics of Software Quality*. New York: American Elsevier, 1978.

Boehm 88 — Boehm, B. “A Spiral Model of Software Development and Enhancement.” *Computer 21*, 5 (May 1988): 61-75.

Butler 93 — Butler R.W. and Finelli, G.B. “The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software.” *IEEE Transactions on Software Engineering 19*, 1 (January 1993): 3-13

Christian 91 Christian, F. “Understanding Fault-Tolerant Distributed Systems.” *Communications of the ACM 34*, 2 (February 1991): 56-78.

Clapp 92 Clapp, J.A. and Stanten, S.F. *A Guide to Total Software Quality Control, Technical* (Technical Report RL-TR-92-316). Bedford, Ma.: MITRE Corporation, 1992.

Clements 95 Clements, P., Bass, L., Kazman, R., Abowd, G. “Predicting Software Quality by Architecture-Level Evaluation,” 485-497. *Proceedings of the 5th International Conference on Software Quality*, Austin, Texas, October 23-26, 1995. Milwaukee, Wi.: American Society for Quality Control, 1995.

Conway 67 Conway, R. W., Maxwell, W. L., and Miller, L. W. *Theory of Scheduling*. Reading, Ma.: Addison Wesley Publishing Company, 1967.

Hennel 91 Hennel, M.A. “Testing for the Achievement of Software Reliability.” *Software Reliability and Safety*, B. Littlewood and D. Miller (eds.). New York: Elsevier Applied Science, 1991.

IEEE-610.12 IEEE Std 610.12-1990. *IEEE Standard Glossary of Software Engineering Terminology.* New York: Institute of Electrical and Electronics Engineers, 1990.

IEEE-1061 IEEE Standard 1061-1992*. Standard for a Software Quality Metrics Methodology*. New York: Institute of Electrical and Electronics Engineers, 1992.

ISO-9126 International Organization for Standardization. *Information Technology - Software Product Evaluation - Quality Characteristics And Guidelines For Their Use*. Geneve, Switzerland: International Organization For Standardization, 1991.

Heimerdinger 92 Heimerdinger, W. L. *A Conceptual Framework for System Fault Tolerance* (CMU/SEI-92-TR-33, ADA264375). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1992.

Jahanian 86 Jahanian, F. and Mok, A. “Safety Analysis of Timing Properties in Real-Time Systems.” *IEEE Transactions on Software Engineering 12*, 9 (September 1986): 890-904

Jain 91 Jain, R. *The Art of Computer Systems Performance Analysis*. New York: Wiley, 1991.

Kazman 95 Kazman, R., Bass, L., Abowd, G., and Clements, P. "An Architectural Analysis Case Study: Internet Information Systems," 148-165. *Proceedings of the 1st International Workshop on Architectures for Software Systems*, Seattle, Washington, April, 1995 (Technical Report CMU-CS-95-151). Pittsburgh, Pa.: School of Computer Science, Carnegie Mellon University, 1995.

Klein 93 Klein, M.H., Ralya, T., Pollack, B., Obenza, R. and Harbour, Michael Gonzalez. *A Practitioners' Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time System*s. Boston: Kluwer Academic Publishers, 1993.

Laprie 92 *Dependable Computing and Fault-Tolerant Systems. Vol. 5, Dependability: Basic Concepts and Terminology in English, French, German, Italian, and Japanese.* Laprie, J.C. (ed.). New York: Springer-Verlag, 1992.

Lazowska 84 Lazowska, E. D, Zahorjan, J., Graham, G. S., and Sevik, K. C. *Quantitative System Performance: Computer System Analysis Using Queuing Network Models*. Englewood Cliffs, N.J.: Prentice-Hall, 1984

Lehoczky 94 Lehoczky, J.P. "Real-Time Resource Management Techniques," 1011-1020. *Encyclopedia of Software Engineering***,** Marciniak, J.J (ed.). New York: J. Wiley, 1994.

Leveson 83 Leveson, N.G. and Harvey, P.R. "Software Fault Tree Analysis." *Journal of Systems and Software 3*, 2 (June 1983): 173-181.

Leveson 86 Leveson, N.G. "Software Safety: Why, What, and How." *ACM Computing Surveys 18*, 2 (June 1986): 125-163.

Leveson 95 Leveson, Nancy G. *Safeware: System Safety and Computers*. Reading, Ma.: Addison-Wesley, 1995.

Littlewood 90 Littlewood, B. Ch.6, "Modelling Growth in Software Reliability." *Software Reliability Handbook*, Paul Rook (ed.). New York: Elsevier Applied Science, 1990.

Locke 92 Locke, C.D. "Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives." *The Journal of Real-Time Systems 4*, 1 (March 1992): 37-53.

MOD 91 UK Ministry of Defence, *Interim Defense Standard 00-56: Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment*, April 1991.

Musa 90 Musa, J.; Iannino, A.; and Okumoto, K. *Software Reliability: Measurements, Prediction, Application*. New York: McGraw-Hill, 1990.

Neumann 86 Neumann, P.G. "On Hierarchical Design of Computer Systems for Critical Applications." *IEEE Transactions on Software Engineering 12*, 9 (September 1986): 905-920.

Perrow 84 Perrow, C. *Normal Accidents: Living with High Risk Technologies*, Basic Books, New York: 1984.

Picciotto 92 J. Picciotto and J. Epstein. “A Comparison of Trusted X Security Policies, Architectures, and Interoperability,” 142-152*. Proceedings. Eighth Annual Computer Security Applications Conference*, San Antonio, Texas, Nov. 30–Dec 4, 1992. Los Alamitos, Ca.: IEEE Computer Society Press, 1992.

Place 93 Place, P.R.H. and Kang, K.C. *Safety-Critical Software: Status Report and Annotated Bibliography* (CMU/SEI-92-TR-5, ADA266993). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1993.

Rushby 93 Rushby, J. *Critical System Properties: Survey and Taxonomy* (Technical Report CSL-93-01). Menlo Park, Ca.: Computer Science Laboratory, SRI International, 1993.

Schlichting 83 Schlichting, R.D. and Schneider, F.B. “Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems.” *ACM Transactions on Computing Systems 1*, 3 (Aug. 1983): 222-238.

Sha 90 Sha, L. and Goodenough, J. B. *Real-Time Scheduling Theory and Ada*, *IEEE Computer 23*, 4 (April 1990), 53-62.

Smith 90 Smith, C. U. *Performance Engineering of Software Systems*. Reading, Ma.: Addison-Wesley, 1990.

Smith 93 Smith, C. U. and Williams, L. G. “Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives.” *IEEE Transactions on Software Engineering 19*, 7 (July 1993).

Stankovic 88 Stankovic, J. A. “Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems.” *IEEE Computer 21*, 10 (Oct. 1988): 10-19.

Stankovic 94 Stankovic, J. A. “Real-Time Operating Systems.” pp. 1009-1010, *Encyclopedia of Software Engineerin*g**,** Marciniak, J.J. (ed.). New York: J. Wiley, 1994.

Stankovic 95 Stankovic, J.A., et al. “Implications of Classical Scheduling Results for Real-Time Systems.” *IEEE Computer 28*, 6 (June 1995): 16-25.

Stotts 88 Stotts, P. David. Chapter 18, “A Comparative Survey of Concurrent Programming Languages,” 419-435. *Concurrent Programming*, Gehani and McGettrick (eds.) Reading, Ma.: Addison-Wesley, 1988.

Trivedi 82 Trivedi, K. S. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. Englewood Cliffs, N.J.: Prentice-Hall, 1982.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS
None

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

3. DISTRIBUTION/AVAILABILITY OF REPORT
Approved for Public Release
Distribution Unlimited

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
CMU/SEI-95-TR-021

5. MONITORING ORGANIZATION REPORT NUMBER(S)
ECS-TR-95-021

6a. NAME OF PERFORMING ORGANIZATION
Software Engineering Institute

6b. OFFICE SYMBOL (if applicable)
SEI

7a. NAME OF MONITORING ORGANIZATION
SEI Joint Program Office

6c. ADDRESS (city, state, and zip code)
Carnegie Mellon University
Pittsburgh PA 15213

7b. ADDRESS (city, state, and zip code)
HQ ESC/ENS
5 Eglin Street
Hanscom AFB, MA 01731-2116

8a. NAME OFFUNDING/SPONSORING ORGANIZATION
SEI Joint Program Office

8b. OFFICE SYMBOL (if applicable)
ESC/ENS

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F19628-95-C-0003

8c. ADDRESS (city, state, and zip code))
Carnegie Mellon University
Pittsburgh PA 15213

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO	PROJECT NO.	TASK NO	WORK UNIT NO.
63756E	N/A	N/A	N/A

11. TITLE (Include Security Classification)
Quality Attributes

12. PERSONAL AUTHOR(S)
Mario Barbacci, Mark H. Klein, Thomas A. Longstaff, Charles B. Weinstock

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (year, month, day)	15. PAGE COUNT
Final	FROM TO	December 1995	56

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (continue on reverse of necessary and identify by block number)
software architecture, software dependability, software performance, software quality attributes, software safety, software security

19. ABSTRACT (continue on reverse if necessary and identify by block number)

Computer systems are used in many critical applications where a failure can have serious consequences (loss of lives or property). Developing systematic ways to relate the software quality attributes of a system to the system's architecture provides a sound basis for making objective decisions about design trade-offs and enables engineers to make reasonably accurate predictions about a system's attributes that are free from bias and hidden assumptions. The ultimate goal is the ability to quantitatively evaluate and trade off multiple software quality attributes to arrive at a better overall system. The purpose of this report is to take a small step in the direction of developing a unifying approach for reasoning about multiple software quality attributes. In this report, we define software quality, introduce a generic taxonomy of attributes, discuss the connections between the attributes, and discuss future work leading to an attribute-based methodology for evaluating software architectures.

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
UNCLASSIFIED/UNLIMITED ■ SAME AS RPT □ DTIC USERS ■

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified, Unlimited Distribution

22a. NAME OF RESPONSIBLE INDIVIDUAL
Thomas R. Miller, Lt Col, USAF

22b. TELEPHONE NUMBER (include area code)
(412) 268-7631

22c. OFFICE SYMBOL
ESC/ENS (SEI)

ABSTRACT — continued from page one, block 19