Critical System Properties: Survey and Taxonomy[1]

Original version published in *Reliability Engineering and System Safety*, Vol. 43, No. 2, pp. 189–219, 1994

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

Technical Report CSL-93-01, May 1993
Revised February 1994

[1]This work was supported by the National Aeronautics and Space Administration Langley Research Center and the US Naval Research Laboratory under contract NAS1-18969 and by the US Naval Research Laboratory under contract N00014-92-C-2177.

Abstract

Computer systems are increasingly employed in circumstances where their failure (or even their correct operation, if they are built to flawed requirements) can have serious consequences.

There is a surprising diversity of opinion concerning the properties that such "critical systems" should possess, and the best methods to develop them. The *dependability* approach grew out of the tradition of ultra-reliable and fault-tolerant systems, while the *safety* approach grew out of the tradition of hazard analysis and system safety engineering. Yet another tradition is found in the *security* community, and there are further specialized approaches in the tradition of *real-time* systems. In this report, I examine the critical properties considered in each approach, and the techniques that have been developed to specify them and to ensure their satisfaction.

Since systems are now being constructed that must satisfy several of these critical system properties simultaneously, there is particular interest in the extent to which techniques from one tradition support or conflict with those of another, and in whether certain critical system properties are fundamentally compatible or incompatible with each other. As a step toward improved understanding of these issues, I suggest a taxonomy, based on Perrow's analysis[1], that considers the complexity of component interactions and tightness of coupling as primary factors.

[1]C. Perrow. *Normal Accidents: Living with High Risk Technologies.* Basic Books, New York, NY, 1984.

Contents

Chapter 1

Introduction

In this report, we ask what is meant by a "critical system," examine some of the properties required of these systems, attempt to classify those properties in a systematic taxonomy, and consider the extent to which different "critical properties" and their associated techniques are compatible with each other.

There is a surprising amount of disagreement concerning the definition of a critical system, and the best methods to develop such systems. One view, which we can call the *dependability* approach, grew out of the tradition of ultra-reliable and fault-tolerant systems; another, which we can refer to as the *safety* approach, grew out of the tradition of hazard analysis and system safety engineering. Yet another tradition is found in the *security* community, and there are further specialized approaches in the tradition of *real-time* systems. There is also disagreement on the relationships between the different properties that may be required of critical systems—for example, the relationship between safety and security is much debated.

In addition to differing views regarding overall philosophy and the relationships among properties, quite different mechanisms and means of specification and assurance are often employed by the various traditions that have been concerned with critical systems. For example, the security community has developed the idea of kernelized systems, whereas the nuclear industry employs protection systems.

Systems are now being constructed that must satisfy several of these critical system properties simultaneously, so there is particular interest in the extent to which techniques from one tradition support or conflict with those of another, and in whether certain critical system properties are fundamentally compatible or incompatible with each other.

To answer these questions, we must begin by understanding the motivations and techniques of the various traditions and approaches that have contributed to the development of critical systems. Accordingly, Chapter 2 presents a survey of the main features of each of the four traditions considered, concentrating on the critical properties addressed and the mechanisms employed to safeguard them. Since formal methods are increasingly considered an essential component in the development of critical systems, Chapter 3 examines the specification and assurance techniques that have been proposed and used in the four traditions.

Finally, Chapter 4 attempts a synthesis of the information presented, and proposes a taxonomy of critical system properties. It is hoped that this will provide some guidance in determining which properties and approaches are likely to be compatible, and capable of being achieved simultaneously.

It is not the purpose of this report to suggest new approaches to the various topics described, nor even to offer a critique of existing approaches; rather, I hope that a single document describing some of the main themes and traditions will provide a foundation for future work that draws on elements of several different approaches in order to allow the specification, development, and assurance for systems that must simultaneously satisfy several different notions of "criticality."

Chapter 2

Traditions and Approaches

We consider four traditions in critical systems: dependability, system safety engineering, security, and real time.

2.1 The Dependability Approach

The concept of dependability was introduced by Jean-Claude Laprie [Lap85] as a contribution to an effort by IFIP Working Group 10.4 (Dependability and Fault Tolerance) to establish a standard framework and terminology for discussing reliable and fault-tolerant systems. This effort culminated in a book describing the basic concepts and terminology in five languages [Lap91]. The term "dependability" is used to escape the specialized technical meanings that have become attached to terms such as "reliability," and in order to have a term for a general approach that can embrace, subsume, and unify many issues and techniques that have generally been considered separately—such as fault tolerance, reliability, correctness, safety, survivability, and security.

In this framework, a *dependable system* is one for which reliance may justifiably be placed on certain aspects of the quality of service that it delivers [Lap85]. The quality of a service includes both its correctness (i.e., conformity with requirements, specifications, and expectations) and the continuity of its delivery.

A departure from the service required of a system constitutes a *failure*—so that a dependable system can also be described as one that does not fail. Not all failures are equally serious, however. A *benign* failure is one where the consequences are on the same order as the benefits provided by normal system operation; a *catastrophic* failure is one whose consequences are incommensurably greater than the benefit of normal system operation [Lap91, Glossary].

Failures are attributed to underlying causes called *faults*. Faults can include mistakes in design and implementation (i.e., "bugs," which are all comprehended in the term "design fault"), component failures, improper operation, and environmental anomalies (e.g., electromagnetic perturbations). Not all faults produce immediate failures, however. Failure is a property of the external behavior of a system—which is itself a manifestation of its internal states and state transitions. Suppose a system progresses through a sequence of internal states $s_1, s_2, \ldots, s_n$ and that its external behavior conforms to its service specification throughout the transitions from s_1 to s_{n-1} but that on entering state s_n the external behavior departs from that required. Is it reasonable to attribute the failure to state s_n? Clearly not, since there could have been something wrong with an earlier state s_j, say, that, while it did not produce an immediate failure, led inevitably to the sequence of transitions culminating in the failure at s_n. In this case, a fault is said to have been *activated* in state s_j, which then contains a *latent error* that persists through the states that follow and becomes *effective* in state s_n when it affects the service delivered and failure occurs. In general, an *error* is that part of the system state that has been damaged by the fault and (if uncorrected) can lead to failure. Fault-tolerant systems attempt to detect and correct latent errors before they become effective.[1]

Fault tolerance has its roots in hardware systems, where random component failures are intrinsic to the physical characteristics of the devices employed. The first step in fault tolerance is *error detection*: latent errors must be detected before they become effective. This can be done by "reasonableness" and internal consistency checks,[2] or by comparison with redundant computations. Once an error has been detected, the next step is *error recovery*, in which the erroneous state is replaced by an acceptable valid state. The replacement state may be constructed by repairing the erroneous state, or it may be a copy of some prior state that is believed to predate the activation of the fault. The first of these methods is called *forward* error recovery, and the second is *backward* error recovery. Exception handling [Cri89] is a principal means for organizing forward error recovery in software; checkpoints and "recovery blocks" [Ran75] provide a means for organizing backward error recovery.

An alternative to error recovery is *fault masking* (also called *error compensation*). Classically, this is achieved by modular redundancy, in which several components perform each computation independently and the final result is selected by majority voting. To mask certain types of fault, more complex Byzantine resilient algorithms [LSP82] may be required in addition to majority voting. To be effective, modular redundancy requires statistical independence (or very nearly so) among component failures. This is reasonable for physical faults in the underlying hardware; more questionable is its application to software design faults in the form of "N-version programming" (also called software diversity) [Avi85, KL86].

[1]It is normally desirable to detect and fix latent errors as soon as possible: even though a particular error may pose little immediate danger, it may cause the system to be much more susceptible to failure should a second fault be activated.

[2]Often referred to as "built-in self-test," or *BIST*.

Error recovery actions may attempt to achieve the same purpose as the original, but faulty, function by different means (e.g., using a different component or algorithm), or they may perform some different, and presumably less desirable, but still acceptable function. Similarly, the fault masking technique may revert to some alternative behavior when failures have exhausted its redundancy (for example, an airplane control surface may lock in the neutral position when there is no clear majority among its control signals). The response required of the system to a series of failures is generally described by sequences such as "fail-op, fail-op, fail-safe," meaning that the first two failures must leave the system fully operational, but the third is required only to leave it in a safe state.

Fault tolerance can be provided at many different levels in a system hierarchy, and in many different ways, so some organizing principles are needed. Cristian [Cri91] presents an excellent overview of design issues and choices in fault-tolerant systems. Some of the main concepts are the "depends" relation between components, and the distinction between the standard and failure semantics of a component. One component *depends* on another if its correctness is contingent on that of the other; *failure semantics* specify the behaviors that a component may exhibit when it fails to provide its standard (i.e., correct) behavior—that is, they specify what a failed component may do. There are various classifications of failure semantics: an *omission* failure occurs when a component fails to respond to an input; a *timing* failure occurs when a correct response is delivered, but outside the real-time interval required; *response* failures occur when a component either delivers an incorrect value or performs an incorrect state change. A *crash* failure occurs when a component suffers an omission failure and thereafter performs no actions until it is restarted; a *symmetric* fault is one that is observed consistently by nonfaulty components; an *arbitrary* (or *Byzantine*) failure is one that is totally unconstrained (in particular, it may be asymmetric and can display different symptoms to different observers). Failure semantics can be ordered by inclusion: for example, a system that satisfies crash-failure semantics also satisfies (trivially) the requirements for arbitrary failure semantics. Thus, crash semantics are a *stronger* failure property than arbitrary failure semantics. Faults can be classified according to the semantics of the failures they may induce; whereas we speak of failure *semantics*, it is normal to speak of fault *modes*. The "difficulty" of a fault mode is inversely related to the strength of the corresponding failure semantics; that is, faults that lead to arbitrary failures are in some sense more difficult than those that lead to crash failures (primarily because error detection is so much more difficult).

In developing a hierarchical fault-tolerant system, the designer must consider the failure semantics to be provided at each level, together with those provided by the components on which it depends. It is generally straightforward to "pass through" failure semantics from one layer to another, and generally more difficult and expensive to "improve" on them. For example, it requires a sophisticated architecture to provide crash-failure semantics for a layer that depends on components that can exhibit arbitrary failures.

Although failure semantics are usually far removed from the standard or desired semantics for the component concerned, it is possible to consider intermediate semantics that correspond to various forms of degraded, rather than totally broken, behavior; thus, notions of *graceful degradation* can be incorporated into this framework for fault tolerance [HW91].

In addition to their semantics, it is necessary to also consider the *stochastic* properties of failures: that is, how often failures (of each type) may be expected to occur. A design is usually constrained by requirements to provide reasonably rare and reasonably clean failure semantics at the application level (e.g., "mask any two failures and crash fail on the third"), and by physically determined failure properties of the lowest-level components. Careful design choices are required to balance fault tolerance and performance objectives; in particular, it is not always advisable to provide very strong failure semantics low down in the hierarchy, because mechanisms higher up the hierarchy may be able to mask low-level faults inexpensively [SRC84].

Fault-tolerant systems that cover many different fault modes may provide a different recovery mechanism for each, thereby promoting complexity—itself a significant source of design faults. One advantage of designing to very weak fault assumptions is that all less difficult fault modes are then included automatically. In particular, by designing to arbitrary failure semantics, we are assured of tolerating all possible faults (up to some number). Such uniformity and economy of mechanism may be bought at a price, however: it generally requires more redundancy to tolerate the more difficult fault modes, so that treating all faults in the manner required for the worst case may reduce the number of faults that can be tolerated. Thus, for a given level of redundancy, the designer must tradeoff the difficulty against the number of faults that can be tolerated. For example, a quad-redundant Byzantine fault-tolerant system can withstand a single fault of *any* kind, whereas a differently organized quad-redundant system can withstand as many as three crash faults, but no other kind; the Byzantine fault-tolerant system will fail if two crash faults arrive, and the other will fail under a single Byzantine fault. An interesting new line of investigation is the development of uniform algorithms and architectures that are effective with respect to *hybrid* fault models; that is, they are able to tolerate faults of several different kinds, and the trading of difficulty against number of faults tolerated is performed at run time, with respect to the faults that have actually arrived. Thus, a quad-redundant hybrid fault-tolerant system should be able to withstand either a single Byzantine fault, or a symmetric fault *and* a crash fault, or as many as three simultaneous crash faults. Unfortunately, the published algorithm for interactive consistency under hybrid faults [TP88] is incorrect. However, it can be repaired [LR93a, LR93b] and holds promise as the basis for a valuable new line of development.

Electronic devices are often afflicted by *transient* faults, in which a cosmic ray or electromagnetic disturbance temporarily disrupts operations but then goes away, leaving the device operating normally, but with possibly corrupted state data.[3] Experimental data indicate that transient faults are many times more common than permanent faults, so inexpensive and effective techniques for *transient recovery* are very valuable. In redundant systems, a transient fault is often manifested as a loss of coordination among the redundant components, and attractive recovery mechanisms can be based on the idea of *self-stabilization*, in which global coordination is restored through the local actions of individual components—rather in the way that certain physical processes automatically recover to a stable state in spite of small disturbances.

Self-stabilization was introduced by Dijkstra in 1974 [Dij74], but did not become widely appreciated until Lamport described it as "Dijkstra's most brilliant work... almost completely unknown... a milestone in work on fault tolerance" [Lam84]. One of the most attractive features of self-stabilization is that it provides a uniform mechanism for recovering from a variety of transient faults. Schneider [Sch93] and Arora and Gouda [AG93] provide good introductions to this topic.

Even in the absence of faults, maintaining coordination among the components of a distributed system poses a number of interesting challenges. If different components can simultaneously undertake activities that access global data, we need to be sure that the different activities do not interfere with each other—as they could if one changed data that another was using. The usual way to deal with this problem is to encapsulate activities within *transactions* and to run a distributed *concurrency control* algorithm that arranges matters so that the observed behavior is as if transactions ran in some serial order. In addition to concurrency control, transactions usually provide *failure atomicity*, which means that if a transaction fails, then any actions it may have performed (such as changing global data, or sending a message to some other transaction) are undone, so that the end result is as if the transaction had never been run. Of course, such repudiation of previous actions may cause other transactions to roll back also, which makes these *distributed commit* algorithms rather complicated, and their consequences potentially drastic. In addition, global data may be subject to *integrity constraints* that require certain properties of the data to be held invariant. Rather than rely on individual transactions to police these invariants, we may prefer to have the transaction management system fail any transaction that violates them. For example, a banking system might disallow any transaction that overdraws an account.

[3]Program code is usually held in ROM and is not vulnerable to this threat. Imporoper sequencing is possible if the cosmic ray hits internal registers, but such afflictions can often be overcome by a watchdog timer interrupt that forces a reset.

For efficiency, the components of a distributed system may cache some global data locally, and for reliability the system may maintain several copies of global data that are voted in some way. *Cache coherence* and *replica consistency* algorithms are needed to maintain consistency among such replicated data.

We use the single term *coordination* to cover to all these issues of concurrency control, consistency, integrity, and so on in distributed systems. The degree of coordination that can be maintained generally depends on the fault modes that can occur. For example, strong notions of replica consistency require that all components are able to communicate with each other, and require all activity to cease if the system divides into mutually isolated partitions. It is therefore necessary to adopt weaker notions of consistency in systems where partitioning is possible (e.g., if the system stays unpartitioned for a sufficient length of time, then consistency of replicated data will *eventually* be achieved).

No useful system stands alone: it must interact with (and exist within) an environment—and that environment may be a source of faults. Thus, dependable systems must usually provide fault tolerance with respect to external faults (or, more precisely, for externally induced errors). More controversial, however, is the use of fault-tolerance techniques to overcome errors due to faults in the design or implementation of the system itself (i.e., *design faults*). One such approach is "design diversity" (or "dissimilarity"), generally organized in the form of N-Version software [AL86, Avi85]. The idea is to use two or more independently developed software versions in conjunction with comparison or voting to avoid system failures due to design faults in individual software versions (this is discussed in more detail in Section 3.2). The alternative to tolerating design faults is to develop systems that are free of them; this approach is referred to as (design) *fault exclusion*.[4] Under the heading of fault exclusion fall those techniques that attempt, either directly by construction or indirectly by analysis, testing, and subsequent correction, to produce systems free from design faults. Systematic design methodologies, especially those with a strong mathematical foundation, aim to eliminate faults at their source; validation and systematic testing strategies, together with formal and informal verification, aim to discover and eliminate faults during the development process.

While endorsing the methods of fault exclusion, the dependability tradition recognizes that they may be imperfectly effective and advocates design fault tolerance to a greater degree than other traditions. While few would argue against fault tolerance as a way of providing protection against residual design faults, it is difficult to assess the reliability (with respect to design faults) achieved in this way unless we can estimate the correlation between failures in different versions. A simplistic analysis assumes that design faults in dissimilar components arise independently

[4]What I have called fault exclusion is generally termed fault *avoidance*; I prefer the former term since it has a more active and positive connotation.

and produce uncorrelated failures. This assumption has been found to be incorrect in several experiments [ECK$^+$91, KL86], and has also been questioned in theoretical studies [EL85], which show that even independent faults can produce correlated failures—though the correlation can be negative (i.e., beneficial) [LM89]. We consider assurance in some detail in Section 3.2, but will note here that it is not only software fault tolerance whose contribution to reliability is hard to quantify: there is no objective way to assign a reliability figure to software on the basis of fault-exclusion methods either.

The suggestion that system properties such as availability, reliability, safety, and security should be regarded as attributes of dependability does not meet with universal approval. In the dependability approach, safety, for example, is viewed as the absence of catastrophic failures, while security is seen as dependability with respect to "prevention of unauthorized access and/or handling of information" [Lap91, Glossary]. An objection to this interpretation of safety is that the notion that catastrophic failure (and hence safety) is relative to the benefit of normal system operation runs counter to other usages of the term "safety"—which are concerned only with the gravity of external consequences. This objection can be overcome by substituting more standard definitions (e.g., safety is the avoidance of unplanned events that result in death, injury, illness, damage to or loss of property, or environmental harm),[5] but other objections remain.

One of these concerns the basic definition of failure, which is interpreted with respect to the system service requirements. Even the revised definitions of safety, security, and so on, have the form "a departure from system service requirements that results in . . . (some qualifying phrase)." Thus, the possibility that a system might operate correctly, yet be unsafe (i.e., that the requirements on the system service might be wrong), is simply not admitted: it is assumed that the service requirements accurately capture all the "real" requirements. This goes contrary to the evidence claimed by Leveson [Lev91] and Perrow [Per84], that most accidents are due to inadequate design foresight and requirements specification, including incomplete or wrong assumptions about the behavior or operation of the controlled system,[6] and unanticipated states of the controlled system and its environment. This objection, too, can be overcome by recognizing that some requirements may be real but not written down (and perhaps recognized only with hindsight)—what matters is that there is some way to identify when a system has "gone wrong."

Finally, the mechanisms and techniques most associated with the dependability approach tend to focus on reliability and fault tolerance and place less stress on several alternative methods that have been found useful in practice. For example,

[5] [Lap91, page 4] gives a definition of safety different from the one in the glossary of that volume: "avoidance of catastrophic consequences on the environment."

[6] Safety concerns generally arise in real-time control systems, where the computer system is managing some physical system called the "controlled system" or "plant."

concern that the system service requirements might admit the possibility of unsafe operation is likely, under the dependability approach, to focus on methods for improving the quality of requirements elicitation and specification,[7] whereas methods that focus directly on safety might be more effective (for that specific purpose). These points can be seen most sharply by considering the very different interpretations of the safety engineering approach, which we consider next.

2.2 The Safety Engineering Approach

To safety engineers, reliability is not the same as safety, nor is a reliable system necessarily a safe one. Reliability is concerned with the incidence of *failures*; safety is concerned with the occurrence of accidents or *mishaps*—which are defined as unplanned events[8] that result in death, injury, illness, damage to or loss of property, or environmental harm. Whereas system failures are defined in terms of system services, safety is defined in terms of external consequences. If the required system services are specified incorrectly, then a system may be unsafe, though perfectly reliable. Conversely, it is feasible for a system to be safe, but unreliable. Enhancing the reliability of software components, though desirable and perhaps necessary, is not sufficient to ensure that they will not contribute to a mishap.

Leveson [Lev86, Lev91] has discussed the issue of "software safety" at length and proposed that some of the techniques of system safety engineering should be adapted and applied to software. The basic idea is to focus on the consequences that must be avoided rather than on the requirements of the system itself (since those might be the very source of undesired consequences). Next, because the occurrence or nonoccurrence of a mishap may depend on circumstances beyond the control of the system under consideration, attention is focused on preventing *hazards*, which are conditions (i.e., states of the controlled system) that can lead to a mishap, rather than preventing mishaps directly. For example, the mishaps for an air traffic control system certainly include mid-air collisions. But the occurrence of a mid-air collision depends on a number of factors: the planes must be too close, and the pilots must not be aware of that fact, or must fail—or be unable—to take effective evading action, and so on. 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 planes do not get too close together in the first place. Thus, the hazard that must

[7]And on correcting and expanding the requirements definition when experience, in design, testing, or operation, reveals inadequacies [PC86].

[8]The caveat "unplanned" is required because safety is often considered in the context of weapons systems, which are *designed* to cause destruction and death—and so we have to distinguish between planned destruction (of enemy assets) and that which is unplanned (e.g., of the launch vehicle, or during storage and transportation).

be controlled by the air traffic control system is planes getting closer than, say, two miles horizontally, or 1,000 feet vertically of each other.[9]

Some of the other terms used in system safety engineering include *damage*, which is a measure of the loss in a mishap. The *severity* of a hazard is an assessment of the worst possible damage that could result, while *danger* is the probability of the hazard leading to a mishap. *Risk* is the combination of hazard severity and danger. A principal tool in system safety engineering is *hazard analysis*; this is a rather large topic (see for example [MOD91b]), but the basic ideas are fairly straightforward. First, potential hazards are identified (e.g., planes getting too close) and categorized according to severity. The categories can range from "catastrophic," meaning that the hazard has the potential to lead to extremely serious consequences, down to "negligible," which denotes that the hazard has no significant consequences. Then a systematic exploration is performed to determine how or whether that hazard might arise. This can be done by reasoning backwards from the hazard ("what could possibly cause this situation to come about?"), or forwards from hypothesized failures ("what if the altitude transponder fails?"). Hazards that are found to have unacceptable risk must be dealt with by, for example, respecification or redesign of the system, incorporation of safety features or warning devices, or by instituting special operating and training procedures (in declining order of preference [MOD91b, section 4.3.1]).

For example, if the mishap concerned is destruction by fire, then the primary hazards are availability of combustible material, an ignition source, and a supply of oxygen. If at all possible, the preferred treatments are to eliminate or reduce these hazards by, for example, use of nonflammable materials, elimination of spark-generating electrical machinery, and reduction in oxygen content (e.g., substitution of air for pure oxygen during ground operations after the Apollo 1 fire). If hazard elimination is impossible or judged only partially effective, then addition of a fire suppression system and of warning devices may be considered. The effectiveness and reliability of these systems then becomes a safety issue, and new hazards may need to be considered (e.g., inadvertent activation of the fire suppression system).

Hazard analysis is performed at several different stages in the design lifecycle (e.g., preliminary, subsystem, system, and operational hazard analysis), and there are a number of supporting methodologies (e.g., hazard and operability studies, or HAZOPS, fault tree analysis, or FTA, and failure modes and effects analysis,

[9]Issues such as alertness and skill of the pilots are hazards that should be controlled, to the extent possible, by the larger system—that is, the air transportation system—of which the air traffic control system is a part. Regulations concerning pilot training and testing, and standards for good user-interface design, can eliminate or reduce certain sources of pilot error. But the central point is that the air traffic control system cannot control these hazards; its responsibility is to eliminate or control those hazards that are reasonably considered within its purview.

or FMEA[10]). Leveson has advocated application of these techniques to software. In particular, "Software Fault Tree Analysis" (SFTA) [LH83] is an adaptation to software of a technique that was developed and first applied in the late 1960s to minimize the risk of inadvertent launch of a Minuteman missile.

The goal of SFTA is to show that the logic contained in the software design will not cause mishaps, and to determine environmental conditions that could lead to the software contributing to a mishap. The basic procedure is to suppose that the software has caused a condition which the hazard analysis has determined could lead to a mishap, and then to work backward to determine the set of possible causes for the condition to occur. The root of the fault tree is the hazard to be analyzed, and necessary preconditions are described at the next level of the tree with either an AND or an OR relationship. Each subnode is expanded in a similar fashion until all leaves describe events of calculable probability or are incapable of further analysis for some reason. Cha, Leveson, and Shimeall [CLS88] present a tutorial example of SFTA in which a subtle timing error is revealed in an Ada program for a traffic-light controller.

An experimental application of SFTA to the flight and telemetry control system of a spacecraft is described by Leveson and Harvey [LH83]. They report that the analysis of a program consisting of over 1,250 lines of Intel 8080 assembly code took two days and discovered a failure scenario that could have resulted in the destruction of the spacecraft. Conventional testing performed by an independent group prior to SFTA had failed to discover the problem revealed by SFTA. Leveson attributes the success of SFTA in finding errors undiscovered by other techniques to the fact that it forces the analyst to examine the program from a different perspective than that used in development; she likens it to vacuuming a rug in two directions: conventional approaches "brush the pile" in one direction, SFTA in the other, so that between them they do a better job than either used alone.

There is some similarity between SFTA and formal techniques for deriving the placement of exception checks [BC81, Cri82]. These techniques use predicate transformer semantics to work backwards from the postcondition and forwards from the initial condition; the derived conditions should agree at the point where they meet—if not, an exception should be signaled in order to control the failure that would otherwise result.

An advantage of the safety engineering approach is that it explicitly considers the system context. This is important, because software considered on its own might not reveal the potential for mishaps. For example, a particular software error may cause a mishap only if there is a simultaneous human and/or hardware failure. Alternatively, it may require an environmental failure to cause the software fault to manifest itself. For this reason, safety and similar properties are said to

[10]FMEA is often extended to FMECA—failure modes, effects and *criticality* analysis—which explicitly considers the criticality of the consequences of component or subsystem failures.

be “emergent,” meaning they are manifested only by the system as a whole, and are not to be found in microcosm within its components. This is in accord with the history of system failures, which are usually the result of multiple faults and unexpected subsystem interactions [Per84].

We have described the methodological approach of system safety engineering, but have not yet said anything about specific implementation mechanisms. Among the mechanisms employed in safety-critical systems are those for “lockins,” “lockouts,” and “interlocks.” *Lockin* and *lockout* mechanisms are intended to lock the system into safe states and out of hazardous states, respectively. An *interlock* mechanism is concerned with sequencing; it might require that an event A may not occur until another event B has taken place, or that A may not occur while event C is active, and so on. An example is a switch that breaks the circuit when an access door to high-voltage equipment is opened.

These mechanisms are normally conceived as physical ones, but software analogs exist. For example (this is taken from Leveson [Lev91]), a program for the protection system of a nuclear reactor must test several plant variables and mark as “tripped” those that are beyond their set points. The protection system will then shut down the reactor if certain combinations of variables are tripped. The naïve way to program the comparisons of the plant variables against their set points would start with all variables untripped, and then examine each variable in turn and set it to tripped if it exceeds its set point. This is hazardous: if the program gets interrupted or stalled for some reason, certain variables that should be tripped might not be examined, and the protection might fail to shut down the plant when it should (this assumes a watchdog timer to break the processor out of loops and stalls). A “design for safety” approach, on the other hand, would use the idea of a lockin and seek to maintain the system state in a “safe” configuration at all times. Observing that the system is in a safe state when all variables are tripped, such an approach would lead to a design that starts with all variables tripped, and would then examine each variable in turn and set it to untripped only if it is below its set point. Premature exit from such a program would render the plant unreliable, but not unsafe.

The safety engineering approach can be applied to properties other than safety. For example, security can clearly be approached in the same way, where mishaps are interpreted as unauthorized disclosure of information. (I understand such an approach is actually used in the case of cryptographic devices. The “security scenario” considers threats, vulnerabilities, safeguards, countermeasures.)

A characteristic that distinguishes the safety engineering approach from the dependability approach is that safety engineering focuses directly on the elimination of undesired events, whereas the dependability approach is rather more concerned with providing the expected service—and avoids catastrophic failure only as a side effect. At the risk of reducing the approaches to caricature, we could say that dependability tries to maximize the extent to which the system works well, while safety engineering tries to minimize the extent to which it can fail badly. Each approach seems the most natural in the application areas to which it is traditionally applied. For example, the dependability approach seems very natural in circumstances for which there is no safe alternative to normal service (aircraft flight control is the quintessential example), whereas the safety engineering approach is clearly attractive where there are specific undesired events (e.g., inadvertent release of weapons).

2.3 The Secure Systems Approach

Secure systems are those that can be trusted to keep secrets and safeguard privacy. Traditionally the main concern has been to prevent unauthorized *disclosure* of information; secondary concerns have been to protect the *integrity* of information, and to prevent *denial of service* to those entitled to receive it. Various policies and models (i.e., specifications) have been proposed for these attributes of security. Nondisclosure policies are mainly concerned with *mandatory* security in which information is given a *classification* drawn from some partially ordered set, individuals have *clearances* drawn from the same set, and the goal is to ensure that information is not disclosed to individuals unless their clearances are greater than or equal to the classification of the information concerned. The threat to be countered is that untrusted programs operating on behalf of highly cleared users may somehow leak highly classified information to users without the necessary clearance. The threat includes not only direct transmission or copying of information, but also *covert channels* whereby information is conveyed indirectly, generally by modulating the behavior of shared resources.

The more formal specifications of security policies are called security *models*. These usually comprise two components: a *system* component and a *security* component. The system component defines what is meant by “computing system” in the context of the model, while the security component defines what “security” means for that system model. Originally, it seemed that one security model (that of Bell and La Padula [BL76]) captured most of what was required. Bell and La Padula’s is an *access-control* model: it is assumed that the underlying hardware can control “read” and “write” access to data. Bell and La Padula then require that classifications are assigned to processes and data (the model is actually couched in terms of “subjects” and “objects”) and that the access-control mechanisms are set up so that processes can only read data classified at or below their own level (this is called the *simple security property*), and can only write data classified at or above their own level (this is called the *-*property*[11]).

Because security concerns arise in many different contexts, many additional models beyond that of Bell and La Padula have subsequently been developed. These

[11]Pronounced “star-property.”

can differ from each other in their system components (e.g., sequential systems, distributed systems, databases, or expert systems), and in their security components (e.g., whether or not covert channels, or the problems of inference and aggregation, are considered). Millen and Cerniglia [MC83] give a good overview of many of these concerns, although it predates much recent work.

The identification of "security" with mandatory access control, which was prevalent in the early days (and perpetuated to some extent by the "Orange Book" criteria that govern evaluation of secure systems [DoD85]), is increasingly being replaced, or at least augmented, by application-specific interpretations. These can include concepts such as "roles" [LHM84] (so that, for example, a user with the role "security officer," has privileges different from those of other users, and orthogonal to those conferred by clearance level), "two-man" rules (so that two independent users must authorize certain critical operations), and procedural constraints (so that a message must be reviewed by a "release officer" before being transmitted over a network). Further variations arise in "guards," where information must essentially flow the "wrong way,"[12] and in dynamic policies, such as the "Chinese Wall" [BN89], where the data that may be accessed depends on what has been accessed before (e.g., an accountant user can access financial data of either of two competing companies, but not both).

Integrity has received rather less attention than disclosure, although it is arguably more important in some applications. The simplest notions treat integrity as the dual of security in an access-control formulation [Bib75]; that is, integrity levels are assigned to processes and to data, and processes are allowed to read data only of equal or higher integrity level, and to write data only of equal or lower integrity level.

Clark and Wilson [CW87] consider integrity from the perspective of commercial applications, and focus on two key elements: *well-formed transactions*, and *separation of duties*. The former requires that important data cannot be modified by arbitrary actions, but only by certain procedures (i.e., well-formed transactions) that have been certified to preserve its integrity in some suitably defined sense. Activation of such a procedure requires human authorization; the person giving the authorization is then accountable for that action. In order to avoid fraud and minimize faults, separation of duties requires that different individuals authorize the different procedures that constitute a larger action. For example, the person who authorizes purchase of an item should not be the same as the person who selects its supplier. Clark and Wilson claimed that mechanisms developed for mandatory

[12]Many older computer systems cannot adequately segregate data of different classifications, so everything is treated the same as the most highly classified data managed by the system. When data is to be transmitted from this "system high" environment to a more lowly cleared destination, it is necessary to check that the data concerned is of the correct classification, and has not been contaminated by its proximity to more highly classified material. A *guard* is an automatic (or semiautomatic) device that performs the necessary checking and transfer.

security are inadequate to enforce their model of commercial integrity—a claim that has generated much debate [BK85, Kar88, KR89, Lee88, Lip82].

Denial of service has received even less attention in the literature, than integrity, the main contributions being those of Gligor [Gli84, YG90] and Millen [Mil92]. One process can deny service to another by monopolizing some resource. Crude attacks, such as a task that refuses to relinquish the CPU or that acquires all available disk space, can be overcome using quotas. More subtle attacks exploit logical interdependences among tasks, and the coordination mechanisms described earlier. For example, a task that holds a write-lock (one of the implementation mechanisms for concurrency control) to a file can deny service to another task that also needs to write that file. Situations where a low-priority task delays a high-priority one in this way can arise innocently in real-time systems (where they are called "priority inversions" [DS92]), and can be overcome by allowing the low-priority task to complete its execution ("priority inheritance" is one of the mechanisms for arranging this). Obviously, this approach will be ineffective if the denial of service is malicious (or if the delaying task is broken); in this case the delaying task must be aborted and its transaction rolled back.

Clearly, a delicate balance must be struck between failure to prevent denial of service, and over-zealous aborting of tasks that simply overrun their allocation without impeding other tasks. Gligor and Millen therefore specify denial of service polices in terms of "user agreements" and finite (or maximum) waiting time. Such policies can be enforced by a trusted operating system component, rather similar to the "kernels" used to ensure nondisclosure, and which are described next.

Kernelization is a unique feature of the secure systems approach; the desired critical property—in this case, the absence of unauthorized disclosure of information—is ensured by a single mechanism, called a security kernel. This approach had its origins in the Anderson Panel recommendations of 1972 [And72], which introduced the concept of a *reference monitor*: a single isolated mechanism that would mediate all accesses to data in order to enforce the given security policy. A reference monitor is required to be:

Correct: it must correctly enforce the chosen security policy. It should be sufficiently small and simple that it can be subject to analysis and tests whose completeness is assured.

Complete: it must mediate *all* accesses between subjects and objects. It must not be possible to bypass it.

Tamper-Proof: it must protect itself from unauthorized modification. This property is also called *isolation*.

The Anderson panel identified the idea of a *security kernel* as a means of realizing a reference monitor on conventional hardware. A security kernel may be considered as a stripped-down operating system that manages the protection facilities provided by the hardware and contains only those functions necessary to achieve the three requirements listed above. The rest of the system, and that should include most of the operating system as well as all user code, is constrained to operate in the protected environment maintained by the kernel—and may therefore be completely untrusted (with respect to security).

Ideally, a security kernel should contain only the code necessary to achieve the three requirements listed earlier. In practice, however, certain other operating system functions usually need to be brought inside the kernel interface in order to achieve acceptable performance on a conventional hardware base. Conversely, it is rarely possible to bring all the security-critical functions inside the kernel, and so *trusted processes* are introduced. These operate outside the security kernel, but are not subject to the same constraints on their behavior as ordinary untrusted processes. In time, the combination of a security kernel and nonkernel trusted processes came to be known as a *trusted computing base* (TCB), and this is the concept that is described in the Department of Defense Trusted Computer System Evaluation Criteria [DoD85].

There is some similarity between the mechanisms of security and safety: a security kernel is essentially a run-time lockin mechanism for "secure" states. A kernel can also provide interlocks—for example, to enforce a requirement that certain processes are executed in a certain order (e.g., "ready, aim, fire")—although it cannot ensure that the processes correctly perform the tasks required of them. Herbert and Needham [HN81] show how other techniques from security (in this case, capabilities) can be used to enforce similar properties across a network.

Whereas fault tolerance is primarily a mechanism to ensure normal, or acceptably degraded, service despite the occurrence of faults, kernelization and the various forms of system interlocks are primarily mechanisms for avoiding certain kinds of failure, and do very little to ensure normal service. The interesting question is, what kinds of failure can they avoid? Rushby [Rus89] argues that kernelization can be effective in avoiding certain faults of commission (doing what is not required), but not faults of omission (failing to do what is required). A kernel can achieve influence over higher levels of the system only through the facilities it does *not* provide; if a kernel provides no mechanisms for achieving certain behaviors, and if no other mechanisms are available, then no layers above the kernel can achieve those behaviors.[13] 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 interprocess communication paths are available, or that certain sequencing

[13]It might seem that a kernel could enforce certain "positive" properties by providing the *only* way to achieve certain behaviors, but this seems a second-order effect, the first-order effect being the denial of other means to achieve the behavior.

constraints are satisfied. Rushby [Rus89] gives a formal characterization of these properties.

In addition to safety, security techniques also have some application to dependability and fault tolerance. A major issue in fault tolerance is *fault containment*: ensuring that the consequences of a fault do not spread and contaminate other components of the system [Add91, Hel86].[14] Clearly, this is a function that computer-security techniques can handle very well: simple memory protection and control of communications can do much to limit fault propagation. More sophisticated fault containment requirements extend the need for protection across processor boundaries, so that a faulty processor cannot write another's memory [HD92].

Although security has clear applications to fault tolerance, there has been relatively little consideration of the converse—namely, use of fault tolerance in the context of security. The exceptions include the work of Neumann [Neu86], who advocates a holistic approach, Dobson and Randell [DR86], who argue for broad application of fault-tolerance techniques to security and, more concretely, that of Fray, Deswarte, and Powell [FDP86], who propose breaking data into pieces that are scattered to different locations[15] as a means both for fault tolerance and to reduce the losses if an intrusion occurs, and also that of Joseph and Avižienis [JA88], who advocate virus detection using the mechanisms of N-version programming. Intrusion detection [Den87], which has received much attention of late, can also, perhaps, be regarded as a step towards a fault-tolerant approach to security: an intrusion can be considered an error[16] that is to be detected automatically by the intrusion-detection system, but with recovery delegated to human operators.

Denial of service is related to (un)availability (a dependability property): the most gross denial of service is that which deliberately crashes or otherwise renders unavailable the service concerned. Fault-tolerant mechanisms could possibly help prevent such denials of service, and fault containment (a safety concern) also seems relevant. Other facets of denial of service seem related to real-time properties, which are considered next.

2.4 The Real-Time Systems Approach

A real-time system is one whose correctness depends not only on values of its outputs, but also on the times at which they are produced. Generally, a real-time system executes a collection of tasks subject to both *deadline* and *jitter* constraints:

[14]The consequences of the absence of fault containment are well-illustrated by the Phobos spacecraft [Che89, Coo90].

[15]This is how the disk technology known as RAID actually works [PGK88].

[16]It probably indicates a failure of the primary security mechanisms (unless it is an "insider" attack), but need not be considered a failure at the level of the larger system until the confidentiality or integrity of data is breached.

once activated (either by a timer, if it is a periodic task, or by some external event if it is aperiodic), a task is required to produce its outputs before its deadline, and with low variability (jitter) from one activation to another. Tight constraints on jitter arise in feedback control, where the stability of the transfer function may depend on consistent timing in the feedback loop, and in some systems dependent on I/O devices that require precise timing between inputs and outputs [Loc92]. In some systems it can be important not to produce results too early, so that deadlines are often best treated as intervals, rather than points, in time. *Hard* real time refers to circumstances where a missed deadline is potentially catastrophic; for *soft* real-time constraints, there is still some (diminished) utility in the results of a task that has missed its deadline [SR90]. Thus, to be considered correct or useful, real-time systems must deliver results within specified time intervals, either without exception (hard real time), or with high probability (soft real time) [HLCM92].

There are two main issues in the development of real-time systems: the derivation of the timing constraints, and the construction of a system structure (particularly a scheduling regimen) that guarantees to satisfy those constraints. Traditionally, the second of these—especially scheduling theory—has received the most attention, but it can be argued that the first is the more fundamental.

A real-time system usually begins with some qualitative timing requirements, such as constraints on the simultaneity and ordering of events. For example, "cars and trains must not be on the crossing at the same time," or "the valve must be closed before the hose is disconnected" (examples such as these could derive from hazard analysis performed in the system safety engineering tradition). Sometimes, quantitative constraints are present, too: for example, "close the valve, wait 10 seconds, and then check the pressure" or "the control loop must execute at a frequency of 40 Hz, with no more than 2 msec jitter on sensor sampling or actuator output."[17]

In either case, detailed timing requirements emerge as the top-level requirements are elaborated into designs. For example, one way to avoid cars and trains occupying a crossing at the same time is to lower a gate across the road shortly before a train arrives on the crossing. We can calculate how long it takes to lower the gate, and the minimum time that can elapse between a train passing a sensor and reaching the crossing, and from that we can deduce timing constraints on the gate controller. The specification problem at this level is to describe the combination of functional and timing constraints that the gate controller is to satisfy; the verification problem is to prove that this specification satisfies the requirement that trains and cars are not present on the crossing simultaneously. Recently, there has been much interest in developing logics to support these kinds of specification and verification problems [AH89, JM86, Koy90, Ost90].

[17]It can be argued that these are really derived requirements, following from qualitative higher-level requirements or constraints (e.g., "it takes a few seconds for the pressure sensor to stabilize," or "here are the equations for the control laws").

Once a detailed functional and timing specification is available, the next job is to implement it. The issues here are, firstly, how to organize a system implementation so that its timing behavior is predictable and, secondly, how to use that organization to guarantee the particular timing behavior required. The largest body of work in real-time systems concerns these issues, particularly the behavior of preemptive scheduling algorithms. These topics can be rather arcane, but it is important to understand their consequences for system structure, and their failure modes (i.e., their behavior under overload).

There are two basic ways to organize a real-time system: a cyclic executive, or a preemptive one. In the cyclic executive, a fixed schedule of tasks is executed cyclically at a fixed rate. Tasks that need to be executed at a faster rate are allocated multiple slots in the task schedule. This means that all task iteration rates are some multiple of the iteration rate of the basic cycle (which can lead to tasks being scheduled rather more frequently than necessary, thereby wasting CPU resources, or rather more slowly, thereby possibly compromising basic timing constraints, or reducing performance). Even tasks that need to be executed aperiodically (e.g., only when there is keyboard input) are still scheduled periodically (to poll for input and process it if present). The maximum execution time of each task is calculated and sufficient time is allocated within the schedule to allow each task to run to completion. This static allocation of CPU resources allows other resources (e.g., communications bandwidth, I/O devices) to be statically scheduled, also. These fixed allocations mean that the time slots of tasks that finish early cannot generally be allocated to other tasks (usually, such time is used to run self-checking diagnostic tasks).

One disadvantage of cyclic executives is that sufficient time must be allocated to each task to allow it to run to completion in the worst case. This can lead to very low CPU utilization, and to low iteration rates. This problem is exacerbated with modern CPU and communications architectures that use pipelines, caches, and contention protocols: the average case performance of these systems is considerably improved at the price of a very long tail on the worst-case probability distribution. Cyclic executives must be designed for this very pessimistic worst case.

The other main disadvantage of the cyclic executives is the fragility and complexity they impose on their application tasks. Suppose, for example, that one task requires 100 msec to perform its functions but is only scheduled every 400 msec, while another takes only 5 msec but must be scheduled every 50 msec. The only way to accommodate these conflicting requirements is to break the longer task into several smaller fragments, thereby complicating its design and the vulnerability of its intermediate state data to interference by (possibly erroneous) intervening tasks. Requirements for low jitter on output may also cause a single task to be broken into two: because the execution time of the basic task may not be accurately predictable (it may be data dependent), the task is broken into one part that does the calculation and leaves its result in a buffer, and another part that relies on the accuracy of the cyclic scheduler to invoke it at exactly the right time to transfer the data from the buffer to its actuator, or other time-critical destination.

The fragility of this organization becomes apparent when it is necessary to insert a new task, or to change the iteration rate of an existing task. All the static allocations of resources must be recalculated and adjusted, taking into account the interdependencies of tasks that are really components of some larger task. So difficult can these adjustments become that, rather than adjust the whole schedule to accommodate new tasks, it is common to squeeze them into the tail end of existing tasks of the correct iteration rate that do not use all their time slot. Thus, the cyclic executive can impose high software engineering costs when maintenance and other lifecycle expenses are considered for its application tasks.

An advantage of the cyclic executives is their complete determinism. This allows accurate prediction of their timing behavior and makes for simple and efficient implementations. The downside to their determinism is their fragility when timing assumptions are violated: there is no good solution if a task overruns its allocation. Allowing the task to complete and slipping the rest of the schedule is extremely hazardous to all other timing-dependent behavior; aborting the task risks creating an inconsistent state that can also have unpredictable downstream consequences. Locke [Loc92], from whom most of this discussion is derived, observes that "virtually every practical system will encounter frame overruns at some point during its lifetime, frequently under unanticipated high load stress (i.e., the time when correct system execution is most critical)." In an early version of SIFT [W^+78], for example, data errors could lead to the voting tasks taking longer than anticipated, thereby causing them to miss their deadlines and leading good channels to disagree on fault status ([PB85, p. 16] describes the sensitivity of the voting time to errors, but the failure scenario is not documented).

The major alternative to the cyclic executive is the priority-driven preemptive executive. Here, each task has a fixed priority and the executive always runs the highest-priority task that is ready for execution. If a high-priority task becomes ready (e.g., because of a timer or external interrupt) while a lower-priority task is running, the lower-priority task is interrupted and the high-priority task is allowed to run. Note that this organization requires relatively expensive context switching to save and later restore the state of an interrupted task (since tasks run to completion under cyclic executives, context switching there can be simple and fast). The challenge with priority-driven executives is to allocate priorities to tasks in such a way that overall system behavior is predictable and all deadlines are satisfied. Originally, a number of plausible and ad-hoc schemes were tried (such as allocating priorities on the basis of "importance"), but the field is now dominated by the rate monotonic scheduling (RMS) scheme of Liu and Layland [LL73]. Under RMS, priorities are simply allocated on the basis of iteration rate (the highest priorities going to the tasks with the highest rates[18]). The deadline for each task is taken to be the end of the period in which it is scheduled for execution, and it can be shown that all tasks will meet their deadlines as long as the utilization of the processor does not exceed 69%.[19] Furthermore, it can be shown that if any priority scheme allows a particular task set to meet its deadlines, then RMS will.

The early theory of RMS made many simplifying assumptions—for example, that the context-switch time was zero, and that tasks were independent (so that no synchronization was required). More recent treatments have lifted most of these restrictions [SLR86,SRL90]. In particular, the priority inheritance and priority ceiling protocols [SRL90] have overcome the problems of "priority inversions." Inversions arise primarily when tasks synchronize on shared resources [DS92]; for example, a high-priority task may be blocked by a low-priority task that has locked a resource needed by the high-priority task (the high-priority task cannot preempt the low-priority task because it will not be able to access the locked resource until the low-priority task releases it), but a medium-priority task that does not need the locked resource will be able to preempt the low-priority task, thereby delaying the time when the high-priority task will be able to execute. Priority inheritance overcomes this problem by causing the low-priority task to inherit the priorities of the tasks that it is blocking. In the example, the low-priority task will temporarily execute with the priority of the blocked high-priority task, thereby avoiding preemption by the medium-priority task. Mechanisms such as this also seem necessary to avoid denial of service in secure systems that employ priority-based scheduling, even in the absence of hard real-time constraints.

The main advantage of priority-based executives is that they greatly simplify the software engineering of the application tasks. There is no need to break long tasks into several shorter ones[20] (long tasks will be preempted by tasks of higher iteration rates, and the executive will look after the problem of saving and restoring their states), and no difficulty in adding new tasks, nor changing the iteration rates of existing tasks. Furthermore, there is no need for all tasks to execute at some multiple of the basic iteration cycle: tasks can execute at their most appropriate rates. It can be argued that by not requiring tasks to execute with excessive frequency, simply to match a multiple of the basic cycle, priority-based executives recover more than enough time, relative to cyclic executives, to compensate for their more expensive context switches.

Priority-based executives behave much more gracefully than cyclic executives when a task runs longer than expected: as long as total CPU utilization does not

[18]Another scheme is "earliest deadline first"; this is claimed to have some advantages over RMS.

[19]69% (the natural logarithm of 2) is a worst-case figure; the actual bound depends on the periodic relationship of the tasks and is generally in the range of 88% to 98%.

[20]That is not quite true: it can be necessary to break tasks into two and/or fiddle with priorities in order to minimize jitter (only the highest-priority tasks execute without jitter under priority-based scheduling) or to avoid producing results too soon.

exceed the limit for the task set concerned, all other tasks will continue to meet their deadlines. Furthermore, schedulability analyses allow prediction of which tasks will miss their deadlines under overload scenarios, allowing preplanned responses to be developed.[21]

It is a topic of much debate whether schedules for critical systems should be static, based on the cyclic executive model, or dynamic and based on a priority-driven executive with rate monotonic scheduling. Proponents of static schedules point to *Richards' anomalies* [Man67, Ric60] (in which the early completion of one task can cause another to be late), priority inversions, and other difficulties in dynamic scheduling as indications that the predictability required for hard real-time systems is best achieved by static scheduling. The contrary point of view argues that the external environment does not always behave predictably, and that a real-time system must adjust its behavior in order to adapt to changed circumstances [BW91, Loc92, SR90]. The two poles of this debate reflect different kinds of concerns and systems: static scheduling has traditionally been used for predictable environments, such as flight control, where there is a fixed schedule of activities to be performed; dynamic scheduling is more natural for unpredictable environments, such as target engagement, where it must be possible to accommodate the number of threats actually present. Even so, there are those who advocate static allocation even for the latter kind of systems [XP91].

Beyond the dynamic scheduling of priority-driven executives lies an even more adaptive approach exemplified by the Alpha system [Jen92]. The model here assumes a very uncertain and dynamic environment, and correspondingly many and changing goals that the system must strive to achieve. Scheduling of tasks in Alpha is not performed according to deadlines or priorities, but according to a model of "benefit accrual." Alpha tasks have a *benefit function* associated with them, that indicates the overall benefit $b(t)$ of completing the task at time t. In the case of a point-defense system, for example, there is a period following sighting of an incoming missile when "later is better" for firing the point-defense system's own missile (firing later allows more accurate targeting and more fuel for maneuver). Thus the benefit function will show a curve rising with time (see Figure 2.1). At some point, however, the incoming missile will be too close for a successful intercept to be likely, so the benefit function shows a steep decline, becoming negative if the defensive missile is fired when it has no chance of intercepting the attacker. Alpha attempts to schedule activities in such a way that maximum overall benefit accrues. Its scheduling is "best-effort," in that it seeks to do the best it can, given the resources available, and the demands placed upon it. This approach is advocated for

[21] Basically, it is the lowest-priority (i.e., lowest-frequency) tasks that will miss their deadlines; it is possible to adjust priorities in certain circumstances so that important, but low-frequency, tasks can be given higher priorities [SLR86]. Similar adjustments to priorities can be made in order to improve jitter [LSD89].

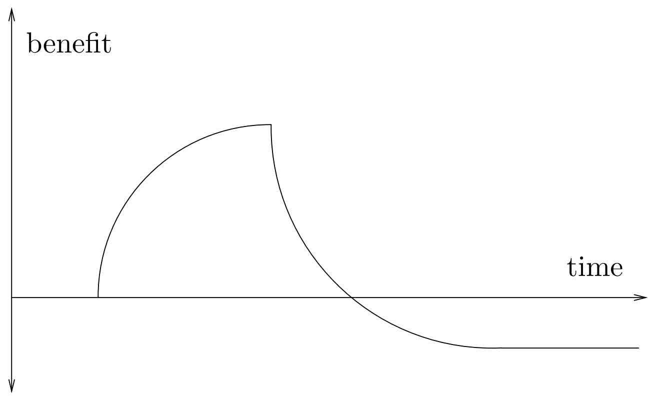

Figure 2.1: A Benefit Function

use in “soft” real-time environments, rather than in those “hard” environments that demand absolute predictability.

In part, the concern for absolute predictability (and hence the preference for cyclic executives) in certain real-time systems seems to stem from the conviction that any missed deadline is potentially catastrophic. Yet deadlines are generally derived requirements, and there is often some flexibility in the values chosen. It seems that a closer integration between the derivation of timing constraints and the properties of execution mechanisms could allow better informed engineering decisions. In particular, use of formal methods in the derivation of timing constraints might allow more relaxed deadline and jitter requirements: the greater precision of formal methods would allow these requirements to be calculated more exactly, whereas current informal methods tend to err on the conservative side, and thereby impose more stringent deadlines than may really be necessary. Such formally derived deadlines would also allow more reliable design changes in response to scheduling conflicts or changed requirements, since the rationale for the existing design would be explicitly recorded.

Real-time systems are often required to be fault tolerant, and this is generally organized as fault masking based on modular redundancy (although, see [AK83] for a method based on backwards recovery) in which all calculations are performed by N identical computer systems and the results are submitted to some form of averaging or voting. The delays due to voting and distributed agreement algorithms do not unduly complicate the scheduling of cyclic executives, but can lead to anomalies in priority-driven schemes, since the delays and task timings may not be the same in all channels, possibly causing their schedules to diverge. McElvany Hugue and Stotts consider scheduling problems in fault-tolerant systems [HS91].

Another topic of debate is whether the redundant channels of fault-tolerant real-time systems should operate synchronously or asynchronously. Numerous practical difficulties attend the asynchronous approach [IRM84,Mac88] (see [Rus91, Chapter 1] for a summary), but it is often used in practice because the synchronous approach (which is overwhelmingly preferred by researchers) requires rather sophisticated Byzantine-resilient algorithms for clock synchronization [LMS85], distribution of sensor samples [LSP82], fault diagnosis [Wal90], group membership [Cri88,KGR89], and other basic tasks [Cri91].

An important error-recovery mechanism for real-time systems is the watchdog timer. In its simplest application, the timer is set on entry to the main loop, timed to go off slightly after control should have returned to the same point. If control does indeed return, the timer is reset and the process repeats. If control does not return for any reason, the watchdog timer will generate an unmaskable interrupt that can force control to an error handler, or even trigger a system reset. More sophisticated fault-tolerance mechanisms have been proposed for circumstances where task timings can be very variable, yet guaranteed deadlines must be met. These techniques trade time against quality or accuracy of results. In Campbell's scheme [CHB79,WHCC80,LC86], a task is allowed to run until some fixed time before its deadline. If it has not produced a result by then, it is aborted and some (presumably) less-desirable task is run that is guaranteed to produce an acceptable result in the remaining time. An alternative scheme first computes the acceptable result in guaranteed time, then uses any remaining time to improve it in some way (e.g., accuracy) [CLL90].

Chapter 3

Formal Models and Assurance

We have so far mainly considered critical system properties in terms of the mechanisms used to achieve them. Additional perspectives on critical system properties can be obtained by considering techniques used in their formal specification, and the different methods of assurance employed.

3.1 Formal Specification Techniques

In attempting to understand system properties and their relation to each other and to mechanisms for satisfying them, it can be valuable to consider formal characterizations that have been proposed. Dependability, safety, and real-time properties tend to be application specific, and so there have been relatively few attempts to provide generalized formulations of these properties. Certain security properties, however, have been formalized in rather general ways.

The first security models assumed a system composed of active *subjects* (programs operating on behalf of users) and passive *objects* (repositories of information); subjects can read or write objects according to restrictions imposed by an access control mechanism that indicates whether a particular subject is allowed to reference a particular object in a particular manner. The Bell and La Padula model [BL76] considered the situation in which subjects and objects are given clearances and classifications, respectively, drawn from some partially ordered set of security classes. Security is then identified with the requirements that the access control mechanism: (a) allow a subject read access to an object only if the clearance of the subject is greater than or equal to (in the partial order) the classification of the object (this is called the *simple security property*), and (b) allow write access only if the classification of the object is greater than or equal to the clearance of the subject (this is called the $*$-*property*).

The Bell and La Padula model suffers from several deficiencies. In the first place, it allows "covert channels" in which information is conveyed from a highly classified object to a more lowly cleared subject (contrary to any reasonable interpretation of security) without violating the requirements of the model. Covert channels are present in the "Multics Interpretation" that Bell and La Padula used to demonstrate the application of their model.[1] The second class of problems in the Bell and La Padula model follows from the fact that it imposes no semantic characterizations on the interpretations of "read" and "write." It is easy to violate the intent of the model by interpreting these perversely [McL85].

These weaknesses of the Bell and La Padula model are overcome in the *noninterference* formulation of mandatory security for sequential systems [GM82]. The idea underlying noninterference is that the behavior perceived by a lowly cleared user should be independent of the actions of highly cleared users. The model is formalized in terms of a state machine: inputs from lowly and highly cleared users are interleaved arbitrarily; lowly classified users can observe the outputs produced in response to their inputs and must observe the same input/output behavior no matter what inputs are present from highly cleared users.

A useful class of security policies that are distinct from the multilevel policies (i.e., those based on a partially ordered set of security classes) comprises those that describe the "wiring diagrams" of systems composed of otherwise isolated subsystems. For example, the "red/black" separation required in end-to-end cryptographic devices can be described in this way.[2] Rushby [Rus81] called these *channel-control* policies. Boebert and Kain have argued persuasively [BK85] that a variation on channel control called "type enforcement" can be used to solve many vexing security problems. The first satisfactory formal treatment of these polices was given by Haigh and Young [HY87].

Important components of noninterference formulations of security are the "unwinding" theorems that establish conditions on the behavior of individual actions sufficient to ensure security of the system [GM84]. These unwinding theorems provide the basis of practical methods for formally verifying that an implementation satisfies a noninterference security policy. Rushby [Rus92b] argues that Haigh and Young's unwinding theorem for channel control policies is incorrect; he proposes a slightly different treatment called "intransitive noninterference" for which he derives and formally verifies a modified unwinding theorem.

[1]Examples of such channels are described by Taylor [Tay84] and by Millen and Cerniglia [MC83]; they exploit operations that allow the classifications of objects and clearances of subjects to be reassigned.

[2]Plaintext messages and headers arrive at the "red side"; messages are sent through an encryption device and headers are sent through a "bypass" to a "black side," where the two parts are reassembled for transmission over unsecured communications links. An important security requirement here is that the only "wires" connecting the red and black sides should be those via the encryption device and the bypass: there must be no way for plaintext messages to go directly from the red to black sides.

Extension of noninterference to the distributed or parallel case has proved tricky because the nondeterminism of a parallel system means that the lowly cleared user may see different behavior on different "runs" independently of the behavior of the highly cleared user. We have to say that the total *variety* of possible behaviors perceived by the lowly cleared user is unchanged by actions of the highly cleared user. There have been many attempts to formulate this notion in an effective way (in particular, in a way that is preserved under composition) [JT88, McC87, McC88, Mil90, WJ90], and the large number of different formulations indicates that this goal is proving elusive.[3]

Most of these security models for distributed systems describe the behavior of such systems in terms of their *traces*. Several variations on the precise definition are possible, but a trace is essentially a time-ordered sequence giving the history of values passed over the communications channels of the system.[4] The *behavior* of a system is then the set of traces that it can generate; a *specification* is likewise a set of traces. A *property* is a predicate on traces.[5]

Since every predicate defines a set (at least in typed logics), and vice versa, specifications and properties are formally equivalent. However, there seems to be a useful distinction between properties, which we think of as being defined for traces considered in isolation, and other kinds of specifications that require consideration of the whole set of traces concerned. Consider, for example, the specification "average response time shall be less than one minute." If we interpret this as meaning the average over each individual run[6] of the system, then it is a property: given the trace for any run, we will be able to tell whether or not it satisfies the response rate requirement. But if the average is interpreted over all runs, then we cannot tell whether a given trace satisfies the requirement without knowing something more about the complete set of traces satisfying the specification: a given trace may have an average response time (within the trace) of ten minutes, but may still satisfy the specification if the specification admits other traces with very fast responses, so that the *overall* average is under one minute.

The distinction we are seeking between properties and other kinds of specifications seems difficult to capture precisely. It may be that "nonproperty" specifications are related to the logician's notion of an *impredicative* definition, or it may simply

[3]None of these definitions consider the probability distribution over the variety of possible behaviors and therefore admit covert timing channels. Probabilistic models are needed to rule these out [Gra91, McL90].

[4]An *event* (c, v) corresponds to the value v being sent over channel c; a trace is then a sequence of such events recorded in their order of occurrence. Simultaneous events are recorded in some arbitrary order.

[5]Behaviors, specifications, and properties should be *prefix closed*: if a given trace satisfies a specification, for example, so should all its prefixes.

[6]Or rather, each run longer than some minimum duration (to avoid special start-up cases).

be that they are higher-order properties.[7]) One reason this distinction seems important is that security appears to be a "nonproperty" specification in the sense used here, and this may help explain the apparent difficulty in finding an attractive, composable, definition of security for distributed systems.

A classical distinction in system modeling is that between safety and liveness properties.[8] A safety property stipulates that specific "bad things" do not happen during execution, while a liveness property stipulates that certain "good things" do happen (eventually). Formally, a safety property S is one such that for all traces $\tau \notin S$, there is a prefix α of τ such that $\forall \beta : \alpha \circ \beta \notin S$ (where $\circ$ denotes concatenation of sequences). In other words, a safety property is one such that if a trace "goes bad," there is an identifiable point at which it happens [AS85]. A liveness property L is one such that $\forall \alpha \exists \beta : \alpha \circ \beta \in L$. In other words, a liveness property is one such that every trace can be extended to satisfy the property. Distinguishing between safety and liveness properties is useful because they require different mechanisms and assurance techniques. It has been shown that any system property can be expressed as the conjunction of a safety property and a liveness property [AS85] (incidentally, we have formally verified Schneider's proof of this theorem [Sch87]).

Results of Abadi and Lamport [AL93] show that "well-behaved" properties are preserved under composition (that is to say, if you have two systems possessing the property concerned, connecting them together will yield a larger system satisfying the property). "Well-behaved" in this context means satisfying the premises of Abadi and Lamport's Theorem 2. These premises are deeply technical; the development leading up to the statement of the theorem is 40 pages long, and involves introduction of concepts such as "stuttering-equivalence," "μ-abstractness," "realizability," "μ-strategy," "μ-realizability," "μ-receptivity," and "constrains-at-most μ." However, a rough idea of what is going on can be conveyed without too much technicality. The basic goal is to compose systems satisfying properties of the form $E_i \supset M_i$ where the E_i's are assumptions about the behavior of the environment, the M_i's are the behaviors of the components, and $\supset$ denotes implication. One of the conditions necessary for systems and properties to compose nicely is that the E_i's should be safety properties.

Thus, one approach to finding attractive, composable specifications of security (or else explaining why such specifications cannot be found) might be to seek formulations of security as a property (recall the standard definitions of security for

[7]Recent work by John McLean [McL94] resolves this issue: he shows that concepts such as "average response time" and security are properties of sets of traces (i.e., they are second-order properties).

[8]Note that here "safety" is a technical term, having no special connection to the concerns of systems safety engineering.

distributed systems seem to be "nonproperty" specifications) that, additionally, satisfy Lamport and Abadi's general conditions for composability.[9]

There have been some attempts to specify fault tolerance in a general manner. These specifications also use a trace model and can be divided into two classes. The *calculational* class treats the activation of a fault as the result of an operation performed (or input provided) by the environment. A trace of the form $\alpha \circ \beta$ will be transformed into one of the form $\alpha \circ e \circ \beta'$, where e is the fault-activation event that causes the subsequent behavior of the system to change from β to β'. The system can be considered fault tolerant with respect to the class of faults represented by e, if the trace $\alpha \circ e \circ \beta'$ satisfies the specification for the system (or some acceptably degraded version of the specification). This approach is called calculational because it can require the value β' to be calculated by considering the effects of e.

The alternative approach, which we will call the (failure) *specification* approach, regards a system as the composition of several subsystems, each of which has a standard specification and one or more failure specifications. The requirement is for the system to deliver its standard specification if all its components do, and to satisfy one of its failure specifications if some of its components depart from their standard specifications. Whereas in the calculational approach we must calculate the behavior of the system subsequent to the activation of a fault in order to see whether the system specification is satisfied, in the specification approach we simply compose standard and failure specifications for the system components.

The specification approach is closest in spirit to Cristian's exposition of the principles of fault-tolerant design [Cri91]. It is articulated by Nordahl [Nor92], who states that it can also be seen in the work of Mancini and Pappalardo [MP88]. Schepers' [Sch92] approach is also in this vein. Nordahl cites the work of Peleska [Pel91] and the state-machine technique [Sch90] as examples of the calculational approach. Formal techniques for specifying graceful degradation [HW91] are essentially the same as the specification approach to specifying fault tolerance.

A connection can be seen between a strong form of the calculational approach (strong in that all faults must be masked) and the noninterference formulations of security. In the case of security, the arrival of "high" inputs must produce no effect on the behavior seen by "low" users; for fault tolerance, it is the arrival of inputs corresponding to the activation of a fault that should have no effect on the behavior seen by users. Weber [Web89] was the first to recognize this similarity.

Unfortunately, the parallel noted by Weber is neither as close nor as useful as might be hoped: if it were, we could use the techniques of security to build fault-

[9]Again, the recent work of McLean [McL94] goes a long way to resolving and explaining this issue. He shows that security is a property of sets of traces that is not closed, in general, under subsetting. Composition (and, especially, feedback) usually reduces the set of traces that a system may exhibit, and the reduced subset may fail to satisfy a security property that was satisifed by the original set.

tolerant systems. The reason the parallel breaks down is that operations invoked by "high" users in the security context are assumed to satisfy the standard specification of the system, whereas those corresponding to a fault activation may be much less constrained.[10]

Although it does not seem that noninterference specifications extend usefully from security to general fault tolerance, it may be that they do extend to the important property of fault *containment*: for example, once a fault-activation operation has flipped a bit in the address space of some process, we want to be sure that this cannot cause the CPU (which might, perhaps, interpret the damaged word as an address) to corrupt the state of some other process. The requirement for such fault containment can be discerned in certain formal specifications of fault masking [Rus92a], and further exploration of these connections would be useful.

Self-stabilizing formulations of fault tolerance can be described by a variation on the calculational approach. For example, Arora and Gouda [AG93] use states and state predicates, rather than traces, to characterize system behavior. A state predicate is said to be *closed* with respect to a set of operations *ops* if execution of any operation in *ops* in a state where the predicate holds results in a state where it also holds. A predicate T *converges* to the predicate S under the operations *ops* if every computation involving only operations from the set *ops* starting in a state where T holds eventually reaches a state where S holds. A system is said to be self-stabilizing for a set of "standard" operations *std*, fault-activation operations F, and specification S, if there exists a state predicate T such that:

1. S is closed with respect to the operations *std*,
2. Execution of an operation from F in a state satisfying S results in a state satisfying T,
3. T is closed with respect to the set $\mathit{std} \cup F$ of both standard and fault-activation operations, and
4. T converges to S under the operations in *std*.

In other words, the system normally satisfies the invariant S; a fault activation can knock it into a state satisfying T, but further fault activations will leave T invariant; and if no fault activations arrive for some time, the system will eventually return to a state satisfying S.

An interesting feature of this definition is that it is essentially identical to a formulation of safety proposed by Leveson [Lev84]. In her formulation, S corresponds

[10]For example, a "high" operation—such as one that flips a bit in a word of memory—is constrained by the memory-management and protection features of the hardware (setting these features up appropriately is most of what secure system design is all about), whereas a similar fault-activation operation may flip a bit *anywhere* in memory (or even in the memory-management unit).

to "safe" states, and T to "unsafe" states.[11] There is, however, a difference in the interpretation intended for Arora and Gouda's self-stabilizing model, and that intended for Leveson's safety model, in that Leveson's notion of "state" definitely comprehends the entire system (i.e., including the physical plant), whereas Arora and Gouda seem to include just the state variables of the computer system.

We have seen that there are some connections between the standard formal treatments of security and some specialized interpretations of safety, between security and fault tolerance, and also between the self-stabilization formulation of fault tolerance and safety. Another attempt to establish formal connections and distinctions between security and safety has been proposed by Burns, Dobson, and McDermid [BMD92]. Their model uses somewhat different characterizations of the terms "security" and "safety" than those considered before, and it is not clear in what sense their model is formal (i.e., it does not appear to provide any deductive apparatus). Basically, their idea is that a safety violation is something that does immediate harm, whereas a security violation is one that creates conditions that allow harm to be done later.

Although fault tolerance, safety, and security must often be considered in application-specific terms, we have described some formal specifications for these concepts that do seem to have broad application. This does not seem to be possible with real-time properties: whereas one can give a general specification of what it means to be, say, a secure computer system, real-time properties are essentially application-specific and generalized specifications make little sense. For this reason, formal treatments of real-time properties have focused on concepts and notations for expressing a wide range of such properties.

Time can be used to express notions of simultaneity, mutual exclusion, and sequencing as well as the durations of, and between, events. Those properties that concern temporal *ordering* rather than duration are conveniently specified using *temporal logics* [Eme90]. These are modal logics (usually specializations of a standard one called S4) where the modalities, $\Box$ and $\Diamond$, range over a temporal order. $\Box A$ ("henceforth" A) says that the temporal assertion A is forever true, and $\Diamond A$ ("eventually" A) asserts that A eventually becomes true. Pnueli [Pnu77] was the first to recognize that these logics could be used to reason about distributed computations and, in particular, about liveness properties. These techniques were developed and popularized by Lamport [Lam83b, Lam89] and others.

There are two families of temporal logics: linear time and branching time [Lam83a]. Both families have their adherents, and both have led to effective specification techniques. It is usually necessary to embellish the basic logics of either family with additional operators in order to achieve a comfortable degree of

[11]Leveson's model ranks the unsafe states according to "risk" and allows states from which no totally safe recovery is possible, but these details can easily be incorporated into the self-stabilizing model.

expressiveness; examples include “next state,” “until,” and backwards-time operators. Interval logics are temporal logics specialized for reasoning over intervals of activity [SMSV83]. The Temporal Logic of Actions (TLA) [Lam91] is a temporal logic in which the modal operators are generalized from states to pairs of states (actions); it achieves considerable expressiveness with very little mechanism.

Many modal logics have what is called the “finite model property,” which renders them decidable. The models of temporal logic are essentially finite-state machines; conversely a finite-state machine is a potential model for a temporal logic formula. This observation gives rise to “model checking”: the goal is to check whether a finite-state machine describing a system implementation satisfies a desired property specified as a temporal logic formula. This process is equivalent to testing whether the specified machine is a model for the specified formula. Because temporal logic can be quite expressive, and because model checking is decidable, this technique offers a completely automatic means for verifying certain properties of certain systems [CES86]. Very clever model-checking algorithms allow finite-state machines with large numbers of states to be checked in reasonable time [BCM$^+$90]. Model checking is not a replacement for conventional theorem proving in support of verification (it is applicable to only certain properties and implementations), but it can be a very valuable adjunct.

Despite their name, temporal logics do not provide ways to reason about “time” in a direct or quantitative (i.e., “real-time”) sense. Several extensions to temporal logic have been proposed for reasoning about real-time properties. The simplest extension is Metric Temporal Logic (MTL) [Koy90] where bounded versions of the temporal operators are introduced. Thus $\Box_{>3}A$ asserts that A is true in every state that is more than 3 time units in the future, and $\Diamond_{\leq 3}A$ asserts that A will eventually hold within 3 time units. The constraint that an acknowledgment should be sent within 6 units of receiving a message can be expressed in MTL as

$$\Box(\mathit{Rcv} \supset \Diamond_{\leq 6}\mathit{Ack}),$$

where Rcv indicates “message received” and Ack “acknowledgment sent” (and $\supset$ is the symbol for “implies”).

The *explicit-clock* temporal logics such as Real-Time Temporal Logic (RTTL) [Ost90] use a special variable T to indicate the value of time at any state and use first-order quantification over that variable to make real-time assertions. The previous example could be expressed in an explicit clock logic as

$$\forall x \Box((\mathit{Rcv} \wedge x = T) \supset \Diamond(\mathit{Ack} \wedge T \leq x + 6))$$

(where $\forall$ means “for all,” and $\wedge$ means “and”). The idea here is that x takes a “snapshot” of the clock in a state at which Rcv is true, so that the right hand side of the implication can say that there must eventually be a state in which Ack is true and the clock has advanced at most 6 units.

Mixing first-order quantification (i.e., $\forall$ and $\exists$) with the temporal modalities can lead to complexity. Timed Propositional Temporal Logic (TPTL) [AH89] is an extension to temporal logic that provides just enough mechanism to take "snapshots" of the (implicit) clock, without adding the full power of first-order quantification. In TPTL, the temporal modalities bind a variable that "freezes" the current value of time, which can then appear in inequalities—so that our example then appears as:

$$\Box x.Rcv \supset \Diamond y.Ack \wedge y \leq x + 6.$$

Alur and Henzinger [AH91] survey these and many other real-time variants of temporal logic; they identify six dimensions of choice that must be made in creating such a logic, and describe current knowledge concerning expressiveness and decidability for various choices. It seems that no single logic is uniformly superior to the others (for example, there are properties that are much more difficult to specify in MTL than TPTL, and also vice-versa). Their survey does not include interval temporal logics, which have also been extended to real-time systems [CHR92,MS87].

Classical logics can also be used to reason about time. Jahanian and Mok's Real-Time Logic (RTL) [JM86] was one of the earliest attempts; it uses an occurrence operator $@(i, e)$ to denote the time of the i'th occurrence of event e, so that our example would appear as

$$\forall i : @(i, Rcv) + 6 \geq @(i, Ack).$$

A graphical framework for state machine specifications called "modecharts" has been based on RTL [HLCM92], and a verifier allows certain real-time properties to be proved of such specifications [Stu90]. However, the examples that have been subjected to mechanically-supported analysis in this way are relatively straightforward. Most of the really hard examples of real-time reasoning (for example, those concerning clock-synchronization algorithms [LMS85,RvH93,Sha92]) have used "brute-force" encodings of time within classical first- or higher-order logic.

3.2 Assurance Methods

An important attribute of critical systems is that they must not only satisfy their critical properties, they must be seen to do so. Thus, extremely rigorous methods of assurance are typically employed for such systems. The different traditions that have considered critical system properties differ in the extent to which quantifiable measures are employed in the statement of assurance requirements. Some understanding of quantitative—that is, probabilistic—assessment of system properties is helpful, even when considering traditions (such as security) that have generally shunned such approaches: appreciation of the numbers reinforces awareness of the responsibility carried by the assurance methods used. For this reason, we treat probabilistic assessment at some length (the discussion is derived from [Rus93]).

System failures can be *random* or *systematic*; the former are due to latent manufacturing defects, wear-out and other effects of aging, environmental stress (e.g., single-event upsets caused by cosmic rays), and other degradation mechanisms that afflict hardware components, while the latter (which are sometimes called generic faults) are due to faults in the specification, design, or construction of the system. Random failures are naturally measured in probabilistic terms; the probability of random failure in a system can be estimated by sufficiently extensive and realistic testing, or (for suitably simple systems) it can be calculated from historical reliability data for its component devices and other known factors, such as environmental conditions.

Systematic failures are not random: faults in specification, design, or construction will cause the system to fail under specific combinations of system state and input values, and the failure is *certain* whenever those combinations arise. But although systematic failures occur in specific circumstances, occurrences of those circumstances are associated with a random process, namely, the sequence over time of inputs to the system. Thus, the manifestations of systematic failures behave as stochastic processes and can be treated probabilistically: to talk about a piece of software having a failure rate of less than, say, 10^{-9} per hour, is to say that the probability of encountering a sequence of inputs that will cause it to exhibit a systematic failure is less than 10^{-9} per hour. Note that this probabilistic measure applies whether we are talking about system reliability or a system property such as safety; what changes is the definition of failure. For reliability, a failure is a departure from required or expected behavior—whereas for safety, failure is any behavior that constitutes a hazard to continued safe operation. By similar interpretations, security, and departure from required real-time behavior, can also be measured in probabilistic terms.

Where critical system properties differ from ordinary reliability requirements is in the extremely small probabilities of failure that can be tolerated. Highly reliable systems may be required to achieve failure rates in the range of 10^{-3} to 10^{-6} per hour, whereas requirements for critical system properties often stipulate failure rates in the range of 10^{-7} to 10^{-12} per hour. We will speak of required failure rates of 10^{-7} to 10^{-12} per hour as the "ultra-critical" range, and will talk of such systems as "ultra-critical." Bear in mind that these probabilities generally refer only to the incidence of critical failures, and not to the general reliability of the systems concerned.

The change in acceptable failure rates between reliable and ultra-critical systems has such a profound impact that it goes beyond a difference of degree and becomes a difference in kind—the reason being that it is generally impossible to directly validate failure rates as low as those stipulated for critical system properties.

There are two ways to estimate the failure rate of a system: one is to measure it directly in a test environment, and the other is to calculate it from the known or measured failure rates of its components plus knowledge of its design or structure (the second method is often accomplished using Markov models).

The direct measurement approach faces two difficulties: first is the question of how accurately the test environment reproduces the circumstances that will be encountered in operation; second is the large number of tests required. If we are looking for very rare failures, it will be necessary to subject the system to "all up" tests in a highly realistic test environment. Furthermore, it will clearly be necessary to subject the system to very large numbers of tests (just how large a number is a topic we will come to shortly)—and if we are dealing with a reactive system, then a test input is not a single event, but a whole trajectory of inputs that drives the system through many states.[12] Furthermore, if we are dealing with a component of a larger system, then it will also be necessary to conduct tests under conditions of single and multiple failures of components that interact with the system under test. Obviously, it is very expensive to set up and run such a test environment, and very time consuming to generate the large and complex sets of test inputs required.

So how many tests will be required? Using both classical and Bayesian probabilistic approaches, it can be shown that if we want a median time to failure of n hours, then we need to see approximately n hours of failure-free operation under test [LS93].[13] So if we are concerned with a critical property with a required failure rate of 10^{-9} per hour, we will need to see 10^9 failure-free hours of operation under test. And 10^9 hours is a little over 114,000 years! [14]

Since empirical quantification of software failure rates is infeasible in the ultra-critical region, we might consider calculating the overall failure rate from those of smaller components of the software. To be feasible, this approach must require relatively modest reliabilities of the components (otherwise we cannot measure them); the components must fail independently, or nearly so (otherwise we do not achieve the multiplicative effect required to deliver ultra-critical quality from components of lesser dependability); and the interrelationships among the components must be simple (otherwise we cannot use reliability of the components to calculate that of

[12]The key issue here is the extent to which the system accumulates state; systems that reinitialize themselves periodically can be tested using shorter trajectories than those that must run for long periods. For example, the clock-drift error that led to failure of Patriot missiles [GAO92] required many hours of continuous operation to manifest itself in a way that was externally detectable.

[13]The Bayesian analysis shows that if we bring no prior belief to the problem, then following n hours of failure-free operation, there is a 50:50 chance that a further n hours will elapse before the first failure.

[14]Butler and Finelli [BF93] present a similar analysis and conclusion (see also Hamlet [Ham92]). Parnas, van Schouwen, and Kwan [PvSK90] use a slightly different model. They are concerned with estimating *trustworthiness*—the probability that software contains no potentially catastrophic flaws—but again the broad conclusion is the same.

the whole). Ordinary software structures do not have this last property: the components communicate freely and share state, so one failure can corrupt the operation of other components [PvSK90]. However, specialized fault-tolerant system structures have been proposed that seek to avoid these difficulties.

One such approach is N-Version programming, which was mentioned in Section 2.1. The idea here is to use two or more independently developed software versions in conjunction with comparison or voting to avoid system failures due to systematic failures in individual software versions. For this technique to be effective, failures of the separate software versions must be independent, or very nearly so. The difficulty is that since independence cannot be assumed (experiments indicate that coincident failures of different versions are not negligible [ECK+91,KL86], and theoretical studies suggest that independent faults can produce correlated failures [EL85, LM89]), the probability of coincident failures must be measured. But for this design approach to be effective, the incidence of coincident failures must be in the ultra-critical region—and then we are again faced with the infeasibility of experimental quantification of extremely rare events [BF93]. For these reasons, the degree of protection provided by software diversity "is generally not measurable" and N-Version software does not provide a means for achieving safety-critical requirements, but "is generally applied as a means of providing additional protection after verification objectives... have been met" [RTCA92, Subsection 2.3.2].

If we cannot validate ultra-critical software by direct measurement of its failure rate, and we cannot make substantiated predictions about N-version or other combinations of less-dependable software components, there seems no alternative but to base certification of critical systems at least partly on other factors, such as analysis of the design and construction of the software, examination of the lifecycle processes used in its development, operational experience gained with similar systems, and perhaps the qualifications of its developers.

We might hope that if these "subjective" factors gave us a reasonable prior expectation of high quality, then a comparatively modest run of failure-free tests would be sufficient to confirm its suitability for ultra-critical applications. Unfortunately, a Bayesian analysis shows that feasible time on test cannot confirm failure rates in the ultra-critical region, unless our prior belief is already that the system is in the ultra-critical region [LS93]. In other words, the requirement for ultra-criticality is so many orders of magnitude removed from the failure rates that can be determined empirically in feasible time on test, that essentially *all* our assurance of has to come from subjective factors such as examination of the lifecycle processes of its development, and review and analysis of the software itself. Of course, extensive testing is still required, but it is perhaps best seen as serving to validate the assumptions that underpin the software design, and to corroborate the broad argument for its correctness, rather than as a validation of claims for ultra-critical reliability.

The goals of the very disciplined lifecycle processes required by almost all standards and guidelines for critical software are to minimize the opportunities for introduction of faults into a design, and to maximize the likelihood and timeliness of detection and removal of those faults that do creep in. The means for achieving these goals are structured development methods, extensive documentation tracing all requirements and design decisions, and careful reviews, analyses, and tests. The more critical a piece of software, the more stringent will be the application of these means of control and assurance.

So now we need to ask what methods are effective for quality control and assurance in critical software development, and what part formal methods should play. It is a rather startling fact that very little documented evidence attests to the efficacy of the various methods for software quality control and assurance when applied to critical software. Several studies indicate significant reductions in "errors per KSLOC" (i.e., programming faults per thousand lines of source code), compared with industry averages, when certain software engineering methods or techniques are employed. For example, the Cleanroom approach has been shown to reduce the density of faults discovered in operation from an industry average of about 3 per KSLOC to fractions of one. However, it is not easy to relate the density of faults in code to the incidence of critical failures in operation, and it could be that some techniques are good at reducing the total number of faults, but are not specially effective on those that cause critical failures. Thus, although there is evidence that various methods are effective in quality *control* (i.e., in preventing, detecting, and eliminating faults during the development process), there seems little objective evidence to correlate these successes with any quantifiable level of quality *assurance*, especially for failure densities at the critical level.

Formal methods are advocated by all the traditions concerned with critical systems, and are explicitly allowed as "alternative means of compliance" for safety-critical aircraft systems [RTCA92], and are required for certain hazardous military systems [MOD91a], and for some secure systems [DoD85]. But it is important to recognize that although formal methods (and some other systematic development and verification methodologies) have a rational basis and offer solid evidence for "correctness" of some aspects of a system's design and implementation, there is no *evidence* that systems built using these techniques will achieve failure rates in the ultra-critical range, and no basis whatever for attaching a reliability number (especially 1) to software on the basis of its development processes. Assurance derived from control and evaluation of development processes is *necessarily* subjective.

This rather chastening conclusion is, essentially, the burden of most standards, guidelines, and criteria for critical systems, for example:

> "...it is not feasible to assess the number or kinds of software errors, if any, that may remain after the completion of system design, development, and test" [FAA88, paragraph 7.i].

"Development of software to a given level does not imply the assignment of a reliability level for that software" [RTCA92, Subsection 2.2.3].

See also [MOD91b, paragraph 6.6 and Annex F].

These conclusions do not vitiate the value of formal methods: rigorously executed, they can guarantee with mathematical certainty that a model of some aspect of the system possesses certain modeled properties. The focus of subjective assurance can therefore shift to other issues (e.g., the fidelity of the modeling employed, the relevance of the verified properties), and to those aspects of the system that have not been formally modeled. These may still present formidable challenges, but should be less than those with which we began.

The available evidence indicates that very few serious faults are introduced (or remain undetected) in the later stages of the development lifecycle under the very disciplined processes used for critical systems; instead, it points to the early lifecycle and to faults in requirements specification as the primary source of catastrophic failures. For example, Lutz [Lut93] reports faults detected during integration and system testing of the Voyager and Galileo spacecraft. Of these faults, 197, were characterized as having potentially significant or catastrophic effects (with respect to the spacecraft's missions). Only 3 of the faults found were coding errors; the other 194 were due to problems in the specifications of functions and interfaces and many of them concerned areas of intrinsic technical difficulty, rather than simply failing to follow requirements to the letter. If formal methods or any other techniques are to make major contributions to critical systems, then it seems that they should concentrate on the early lifecycle and on the hardest aspects of a design.

Unfortunately, standards and guidelines for critical systems do not always leave sufficient freedom to determine where techniques such as formal methods might be most effectively applied. In secure systems, for example, there are clearly vulnerabilities at several levels in the implementation hierarchy that supports a secure system. The lowest level of the kernel might manage the underlying hardware protection facilities incorrectly, so that certain interrupts, say, leave the system inadequately protected; or the higher levels of the kernel might manage the access controls incorrectly, allowing a SECRET process, say, access to TOP SECRET information. The criteria that govern the evaluation of secure systems [DoD85] are very specific about the assurance techniques that must be used to demonstrate the absence of vulnerabilities at various levels. At the A1 level of assurance, for example, formal methods are required to demonstrate the absence of covert channels, and "specification to code correspondence" must be demonstrated at the TCB interface, but there are no specific assurance requirements levied on the lowest level of the kernel where the critical hardware protection features are manipulated.

One could speculate that hazard analysis would reveal that a different allocation of priorities could be more effective. But as far as I know, hazard analysis and the techniques of system safety engineering are not employed in the design of secure systems. Similarly, with few exceptions, the dependability approach is not applied to secure systems either. Again, one can speculate that some of the techniques of fault tolerance (for example, FMEA or built-in self-test) are highly appropriate to secure systems and should be required.[15]

A serious difficulty arises when the conventional approach to hierarchical verification in formal methods [Hoa72] is applied to properties such as security, where faults of commission must be excluded. The difficulty is that verification typically establishes only that each layer in the hierarchy is sufficient to implement the layer above it; it does not establish necessity (technically, an implication—not an equivalence—is established). Thus, there is nothing to stop a lower level from doing *more* than is desired (e.g., copying files to undesired locations) and thereby compromising the desired policy. It seems that security properties and others that share this characteristic must be verified directly at the implementation level. Leveson [Lev91] makes the same point about safety, and it is possible that Jacob's observation [Jac89] that certain refinement techniques do not work for security properties has similar origins.

A notion, which derives from certain nonmonotonic logics used in AI, called the *closed-world assumption* seems relevant here. Under a closed-world assumption, properties not explicitly stated to be true are assumed to be false. Moriconi and Qian [MQ92] apply this idea to the refinement of what they call "software architectures." Whereas conventional software specification techniques are concerned with function and behavior, architectural specifications are concerned with the *structure* of software—that is, with the components from which it is to be constructed, and the relationships among them. Among the relationships considered by Moriconi and Qian are control and data flow; the latter can be refined into various forms of message passing, or use of shared variables. The closed-world assumption means that if a relationship is not specified between certain components, then it should not exist, and may not be created in subsequent refinements of the architecture. Moriconi and Qian prove that a number of standard refinements preserve an appropriate notion of correctness under this closed-world assumption. Thus, it seems that properties such as security and safety, which are sensitive to "doing more" than is required, and which can be subverted by unexpected interactions and communications, may best be served by Moriconi and Qian's notion of architectural refinement rather than (or in addition to) traditional notions of hierarchical refinement of function and behavior.

[15]An example concerning the release authorization for certain weapons is instructive. The authorization required many independent cryptographic code sequences, leading to the conclusion that the probability of inadvertent authorization was less than 2^{-429}. However, the comparisons on the received code sequences were programmed in such a way that all would succeed independently of the data if the accumulator flag bit had a stuck-at-0 fault [Unk90].

Chapter 4

Discussion and Taxonomy

We have considered critical systems from a number of perspectives and have examined representative properties that such systems may be required to maintain or enforce, together with techniques for specifying those properties, mechanisms for enforcing them, and methods for providing assurance that critical system goals have been met. In this section, we attempt to bring these threads together and suggest a tentative basis for a taxonomy that may help organize thinking on these topics. The motivation for proposing a taxonomy is that modern systems are often required to satisfy two or more critical system properties simultaneously: for example, command and control systems may be required to be secure, fault tolerant, and real time. Accordingly, the main attribute we desire of our taxonomy is that it should help identify those combinations of critical system properties that are "compatible" with each other, and also those that are "incompatible."

There are many criteria on which such a taxonomy might be founded: for example, we could consider the system structures employed, or the formal specification techniques, or validation methods used. However, classifications based on such attributes tend to reflect and reinforce traditional divisions based on the separate evolutions of techniques in the dependability, safety, security, and real-time fields.

A rather more productive approach seems to be one that focuses on very general attributes of the different critical system properties: for example, are some properties more easily realized on a single, centralized system than on a decentralized one, and are some properties best served by rich possibilities for interaction while others favor limited interaction? Consideration of classifications along these lines reveals that degree of "coordination" and related attributes seem the most significant. In fixing on particular attributes, I have chosen to follow the lead of Charles Perrow, who, in his book "Normal Accidents," based his analysis on two attributes that he named "interaction" and "coupling" [Per84, chapter 3]. Perrow's work was focused on safety, and on the structure of the human organization and physical plant found in large systems (such as an oil refinery), but the attributes that he identified seem significant for other critical properties, and for computer systems, too.

Interaction, which can range from "linear" to "complex," refers to the extent to which the behavior of one component in a system can affect the behavior of other components. In a simple, linear system, components affect only those others that are functionally "downstream" of them; in a more complex system, a single component may participate in many different sequences of interactions with many other components. For example, the fuel on board an airplane is functionally upstream of the engines, but its distribution among the various tanks also affects (i.e., has an interaction with) the airplane's weight and balance.[1] In computer systems, the notion of "component" must include both physical and abstract entities; for example, the abstract entity "database" is a component, as are its processes and data, and also the devices that provide execution and storage. Computer systems that maintain global notions of coordination and consistency (e.g., distributed databases) are considered to have complex interactions, since activities in different locations interact with each other.

Coupling, which can range from "loose" to "tight," refers to the extent to which there is metaphorical "slack" or "flexibility" in the system. Coupling is not an independent notion; we really have to ask "coupling of *what*?" For the preliminary analysis being undertaken here, however, we can tolerate the imprecision of the unqualified term, and supply more specificity when needed. Loosely coupled systems are usually less time constrained than tightly coupled ones, can tolerate things being done in different sequences than those expected, and may be adaptable to different purposes or to operate under different assumptions than those originally considered. For example, craft industries are usually loosely coupled, whereas production lines with just-in-time inventory control are tightly coupled. Viewed as a computer system, the telephone switching network may be considered loosely coupled, since there are usually multiple ways to route calls, whereas most hard-real-time control systems are tightly coupled, since they depend on everything behaving as expected.

In Perrow's analysis, systems with complex interactions (intended or unintended) can promote accidents because interactions (and therefore possible behaviors) are hard to understand, predict, or even enumerate. An important element in some accidents is that component faults can open up unexpected paths for interaction, leading to unforeseen consequences. For example, shrapnel from a disintegrating airplane engine may sever hydraulic lines in the wing, possibly causing the slats to retract, and thereby generating an unexpected interaction between engine and wing. Tight coupling is also considered to contribute to accidents because it leaves less room for maneuver when things start to depart from their planned course. On

[1]When a component that participates in multiple interactions fails, it may produce consequences in several subsystems; this is an example of a "common mode" failure.

the other hand, complex interactions and tight coupling are generally introduced to promote efficiency, and contribute positive value when things are going well.

Returning to computer systems, it is my contention that some critical system properties demand limited interaction and/or loose coupling, while others are associated with the opposite requirements. Critical system properties are *compatible* if they are associated with similar requirements for interaction and coupling; otherwise, they are incompatible and can only be combined with difficulty and compromise.

The most fundamental security concern—that of nondisclosure—is a property that *demands* few interactions. In its strictest form, security is best achieved by independent systems, each allocated to a particular security classification, so that there can be no interactions among the different classifications. In its less strict (and more useful) forms, security allows interaction among different security classifications, but only in specified and strictly controlled ways. Security, more than any other property, extends its concern to tenuous and subtle forms of interaction: specifically, those that provide covert channels for information flow. Elimination or control of such interactions is usually best achieved by "virtualizing" the resources of the system: dividing them into independent virtual resources that can be allocated to separate security classifications. These allocations are usually based on fixed quotas, since otherwise one classification can signal to another by manipulating its own resource consumption. Fixed quotas reduce or eliminate interaction among processes at different security classifications, but produce a tightly coupled, inflexible, system: a surge of activity at one security classification cannot be accommodated by borrowing resources from other classifications, since this would provide a covert channel.

In summary, security requires an environment with few and controlled interactions, and leads to tightly coupled systems. We can expect the mechanisms developed for security to be of value for other system properties that require few interactions; conversely we can expect security to be incompatible with system properties that require complex interactions, or flexible resource allocations.[2]

Real time provides an example of a system property that, in at least some of its manifestations, exactly fulfills these requirements for being incompatible with the strictest interpretations of security. Because they usually arise in control applications, real-time systems generally participate in complex interactions with the controlled plant and its environment. Thus, the controlled plant could possibly be used as a channel for covert information flow between processes of different secu-

[2]Observe that inflexible resource allocations are a consequence of considering the subtle interactions that constitute covert channels; if these channels are considered unimportant in some applications, then their parent interactions can be ignored. In this case, security techniques that control just the direct interactions can be used, and these do not incur the full penalty of inflexible resource allocation.

rity classifications [DRC86, page 6-9]. Less arcane opportunities for covert channels arise in the more dynamic kinds of real-time system through the complex interactions created by dynamic scheduling and preemption.

Conversely, the inflexible resource allocations required for strict security conflict with the needs of real-time systems—especially those of the more dynamic kind, which seek to optimize allocation of system resources at a global level. To take an extreme case, a highly urgent process that accesses highly classified data might be denied maximum use of the processor because such preemption could be used as a signaling channel to a more lowly classified process.

Notice that the spectrum of approaches to real-time systems represents an explicit trading of complexity of interactions against tightness of coupling. Cyclic executives maintain very linear interactions among tasks, so that overall behavior and timing is very predictable. These systems are very tightly coupled, however; we saw earlier that they are inflexible with regard to iteration rate, can accommodate new tasks or changed task durations only with difficulty, and that the consequences of a missed deadline can propagate to all subsequent tasks in the frame. Priority-driven executives increase the complexity of interactions among the tasks (in that their order of execution is no longer fixed) and (their detractors claim) lessen predictability of overall behavior in order to reduce coupling: these systems are much more flexible in their response to transient overload and occasional missed deadlines than cyclic executives. Fully dynamic systems such as Alpha optimize flexibility of response at the expense of essentially unpredictable task execution sequences and consequently greater possibilities for interaction.

The conflicts between security and real time are most sharp when the most flexible and dynamic forms of real-time resource allocation and the strictest notions of security (no covert channels), are considered. If we are prepared to ease one or both of these requirements, then compatible compromises can be found. For example, a study on security for the Alpha real-time system [CG93] considers an approach in which "timeliness" is incorporated into the security policy, where it can be *explicitly* traded against covert channel requirements. On the other hand, if we consider less dynamic types of real-time system, such as the cyclic executives and priority-driven schemes, then resource requirements can be predicted in advance and accommodated within the fixed resource allocations of a strict secure system.[3]

[3]For efficiency, the simple kinds of real-time system usually execute tasks within a single address space and manipulate data in shared buffers. These techniques are inimical to security, which requires separate, protected address spaces for tasks of different security classifications, and explicit copying of data. However, there is nothing *intrinsically* incompatible between these approaches: the unsecure techniques used in cyclic executives are expedient, not essential. Security mechanisms (particularly expensive task-switching and interrupt handling) contribute to overhead, and may therefore reduce performance, but they are not in fundamental conflict with the linear interactions of cyclic executives.

Since Perrow's analysis was focused on accidents, classification of safety-critical systems with respect to interaction and coupling is quite straightforward: in general, safety-critical systems benefit by having few, linear, and known interactions, and from loose coupling. Linear interactions facilitate comprehension of the behavior of the system, and thereby contribute to its predictability. A chief concern in safety-critical systems is the possibility of unexpected interactions arising as a result of component failures. Hazard analysis and related techniques can be seen as systematic methods for anticipating these (and other) interactions, and one of the goals of safe system design is to minimize possibilities for unwanted interaction (notably what are called "unintended effects"). Since security is also closely associated with simple and strictly controlled interactions, it follows that security should be largely compatible with safety, and that techniques from security should find application in safety-critical systems (and vice-versa).

Perrow favors loose coupling for safety because it provides scope for human operators to intervene and rescue a system that is malfunctioning; loose coupling allows the possibility of intervention at several points, and allows time for a response to be developed. There is a tendency, deplored by many observers, for modern computer systems to exclude such opportunities for intervention (e.g., the lack of manual overrides and of direct modes of control in some fly-by-wire systems[4]). In the case of computer control systems, the usual goal is to prevent the system getting into a hazardous state in the first place; should it do so, however, then Leveson's model of a safe system as one that will eventually return to a safe state (which, as we have seen, is essentially the self-stabilization model of fault tolerance), depends on there being sufficiently loose coupling between the hazardous state and catastrophe that there is time to perform the correction.

As we saw in the case of real-time systems, simplicity of internal interactions must often be traded against looseness of coupling. When this is so, simplicity of interaction seems likely to have the greater benefit for safety in most cases. This argues for using the simpler types of real-time control system in safety-critical applications, but the specific circumstances of each application must obviously be taken into account.

Mechanisms for dependability and fault tolerance often add new interactions. For example, reliable distributed processing and communication usually requires end-to-end acknowledgments—so that what might have been considered a one-way interaction (from sender to recipient), becomes two-way. Because security necessitates limited interactions, even such simple fault-tolerant mechanisms as acknowledgments can cause difficulties: in particular, they provide a covert channel if the recipient is more highly classified than the sender. If fault tolerance is required as well as security, the usual solution is to interpose trusted components to shield the

[4]An unstable fighter airplane is too tightly coupled to be flown without computer control, but this is less obviously true of passenger airplanes.

untrusted ones from direct exposure to acknowledgments and error handling. Unfortunately, since the appropriate response to errors and negative acknowledgments is specific to the application, this approach leads to some of the applications code migrating into the trusted components.

Difficulties of this kind are seen very clearly in distributed applications that must maintain global consistency across replicated and distributed data. Even in the absence of fault tolerance, the usual consistency requirement of single-copy serializability is very hard to maintain in the presence of security constraints. The reason is that locking of file or database records creates complex interactions among clients of the database system. If covert channels are to be avoided, it seems necessary either to relinquish strong notions of consistency, or to accept that some subjects may be denied service for arbitrary periods, or that they may be denied access to the most current data [MG90]. Alternatively, we can revisit the assumption that security can be achieved by breaking operations into separate transactions that each operate at a single security level, and can extend the notion of serializability to multilevel transactions [CM92].

Fault tolerance can compound these difficulties and can require further tradeoffs, since backward recovery of one process can trigger a domino effect, causing other processes to roll back also. Spurious activation of such fault-tolerant mechanisms provides a clear possibility for denial of service, as well as opportunities for covert channels. Integrity constraints lead to further difficulties: if we know that an airplane can hold 20 tons of cargo and that only 15 tons of unclassified material are on board, then a rejected request to add another ton of unclassified cargo suggests there must be some classified material aboard. Inference and covert channel possibilities such as these indicate that mechanisms intended to promote dependability can create complex interactions that are inimical to security.

We have just seen that mechanisms for dependability and fault tolerance may increase the complexity of interactions within a system, and can therefore be incompatible with security and other properties that favor simple interactions. Since we know that safety is such a property, this seems to suggest the disturbing conclusion that dependability and fault-tolerant mechanisms are inimical to safety. This need not be so, however: security is sensitive to interactions of a very subtle kind (those that can be exploited by conniving agents to yield covert channels), whereas safety is affected by rather more substantial interactions. The subtle interactions created by consistency and integrity mechanisms may be of little negative consequence to safety, and the conceptual simplification provided by the assurance of consistency and integrity is likely to be a positive safety benefit—as long as the mechanisms continue to work. However, global consistency and integrity mechanisms tightly couple the components of a system, so that if they fail—perhaps because some fundamental assumption (such as absence of network partitioning) is violated—then safe operation or recovery may be difficult or impossible. It seems that if dependability mechanisms that tighten coupling are used in safety-critical systems, then strong assurances must be provided for the assumptions on which correct operation of the mechanisms depend. On the other hand, a loosely-coupled system is not automatically safe: there must be some mechanism that, with high assurance, will restore the system to a safe state if it errs (loose coupling may just provide the time to do this).

Another example of the interactions introduced by dependability mechanisms, and of the need for care when fault tolerance is used in the service of safety, is found in deep-space probes.[5] Safe operation of the instruments and spacecraft resources depends on satisfaction of numerous sequencing constraints, such as "the time separation between powering on the S-band transmitter and powering on the X-band transmitter shall be either less than 30 seconds or greater than 6 minutes" [LW92]. The command schedules that operate the spacecraft are scrutinized to ensure that they satisfy these constraints. However, there are also fault-protection schedules that are activated when certain anomalies are detected. These place the spacecraft instruments and resources in "safe modes," and configure the antennas and radios to receive commands from earth. If the fault-protection schedule is designed to use the X-band system, unfortunate timing could lead to it powering on the X-band system inside the forbidden 30 second to 6 minute window after the S-band system was powered on. For safe operation, it is necessary to consider all possible interactions between the regular and fault-protection schedules [LW92].

The recognition that mechanisms for dependability and fault tolerance can increase complexity of interactions does not render all of dependability incompatible with those properties, such as security, safety, and hard real time, that generally require limited interactions. Dependability and fault tolerance comprise a wide range of techniques and mechanisms; the analysis suggested here can help identify those that are most compatible with the other properties considered. Among fault-tolerant mechanisms, fault masking seems the one most compatible with limited interactions.

Rather than looking only at how different techniques can coexist, we can also consider whether they can be positively reinforcing. One possibility along these lines concerns fault tolerance and security. Secure systems are usually tightly coupled in the sense that any failures of the mechanisms for protection are assumed to lead directly to compromise of sensitive information. To loosen this coupling, we can look to methods from fault tolerance, which suggest using monitoring techniques to detect potential or actual intrusions (i.e., error detection) and forward recovery techniques (e.g., apprehending the intruder, or changing plans based on compromised material) to minimize damage following an intrusion.[6]

[5]For such missions, safety is equated with preservation of the spacecraft.

[6]Notice also, that as well as applying ideas from fault tolerance, what we are also doing here is widening our conception of the system to encompass the human organization in which it operates.

Interactions	Coupling	
	Loose	Tight
Linear	Weak[†] security Safety	Strict[†] security Fault masking Static real time
Complex	Dynamic real time Fault tolerance	Global coordination

[†] Strict security requires absence of covert channels.

Table 4.1: Critical System Properties versus Interactions and Coupling

In summary, dependability and fault tolerance encompass a wide range of mechanisms. None seem likely to reduce complexity of interactions; some may loosen coupling, others may tighten it. As with real-time systems, selection of dependability and fault-tolerant mechanisms needs to be performed with care if they are to be combined with security or safety requirements. Combining them with real-time mechanisms should generally be more straightforward, though there is a possibility that the complexity of interactions produced by the combination could lead to ill-understood behavior.

Collecting all these points of discussion together, we can crudely classify various critical system properties and techniques according to their associated degrees of interaction and coupling; a tabular taxonomy constructed in this way is shown in Table 4.1.

Our classification is clearly not founded on strictly scientific principles (e.g., the assertion that safety is associated with linear interactions and loose coupling is not falsifiable in the sense required by Karl Popper's interpretations of the scientific method). Nonetheless, we find that it has some explanatory and predictive value. We have already described how it can help anticipate those attributes of the dependability, safety, security, and real-time approaches that are potentially incompatible with each other, as well as those that are compatible. In this respect, it seems more widely applicable than other classifications, such as our earlier one based on "positive" and "negative" properties [Rus89]: that classification could identify candidates for kernelization, but shed no other light on potentially compatible or incompatible properties.

4.1 Conclusion

In my opinion, the best notion of "critical system" is the one that derives from the system safety engineering tradition: *a critical (computer) system is one whose mal-*

function could lead to unacceptable consequences. The "unacceptable consequences" depend on context and could include loss of life, damage to the environment, or disclosure of sensitive information. The determination whether a system is critical should be made by hazard analysis. In many contexts, it will be appropriate to assign degrees or levels of criticality. For example, the guidelines for computers on board aircraft, DO-178B [RTCA92], consider five levels from A (most critical) to E.[7]

The development of critical systems should draw on all the intellectual and technical innovations that can contribute to their quality. Many of the fields that give rise to critical systems have developed their own individual approaches, seemingly in isolation; it is hoped that this report has highlighted some of the main contributions of the different fields and approaches and may contribute to some cross fertilization.

However, we do not expect, and do not advocate, wholesale adoption by one field of techniques from another. The approaches we identified have been durable precisely because they have captured and investigated important classes of systems, problems, and techniques. It is when a system must satisfy simultaneously properties that have traditionally been considered separately that awareness of all the available techniques becomes vital. We have proposed a taxonomy, based on Perrow's analysis [Per84], that considers the complexity of component interactions and tightness of coupling as primary factors in determining the extent to which different properties and mechanisms may be expected to be compatible with each other or not.

In future work, we hope to explore the possibility of placing some of this analysis on a more formal foundation, and to work out some representative examples in concrete detail.

Acknowledgements

My understanding of these topics has benefitted from numerous discussions with my colleagues Jack Goldberg, Ira Greenberg, Peter Neumann, Sam Owre, and Shankar, and from comments by Connie Heitmeyer of the US Naval Research Laboratory. Tom Anderson (University of Newcastle) and Bev Littlewood (City University) provided helpful comments on an earlier version that I have tried to address in this one.

[7]Level A is "software whose anomalous behavior... would lead to a failure of system function resulting in a catastrophic failure condition for the aircraft" [RTCA92, section 2.2.2]. Catastrophic failure conditions are "those which would prevent continued safe flight and landing" [FAA88]. Other fields have similar provisions.

Bibliography

[Add91] Edward Addy. A case study on isolation of safety-critical software. In *COMPASS '91 (Proceedings of the Sixth Annual Conference on Computer Assurance)*, pages 75–83, Gaithersburg, MD, June 1991. IEEE Washington Section.

[AG93] Anish Arora and Mohamed Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, November 1993.

[AH89] R. Alur and T. A. Henzinger. A really temporal logic. In *30th IEEE Symposium on Foundations of Computer Science*, pages 164–169, 1989.

[AH91] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In de Bakker et al. [dBHdRR91], pages 74—106.

[AK83] T. Anderson and J. C. Knight. A framework for software fault tolerance in real-time systems. *IEEE Transactions on Software Engineering*, SE-9(3):355–364, May 1983.

[AL86] A. Avižienis and J. C. Laprie. Dependable computing: From concepts to design diversity. *Proceedings of the IEEE*, 74(5):629–638, May 1986.

[AL93] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.

[And72] J. P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, US Air Force, October 1972. (Two volumes).

[AS85] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[Avi85] Algirdas Avižienis. The N-Version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, December 1985.

[BC81] Eike Best and Flaviu Cristian. Systematic detection of exception occurrences. *Science of Computer Programming*, 1(1):115–144, 1981.

[BCM$^+$90] J. R. Burch, E. M. Clarke, K. L McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 2^{20} states and beyond. In *5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, PA, June 1990. IEEE Computer Society.

[BF93] Ricky W. Butler and George B. Finelli. The infeasibility of experimental quantification of life-critical software reliability. *IEEE Transactions on Software Engineering*, 19(1):3–12, January 1993.

[Bib75] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR 3153, Mitre Corporation, Bedford, MA, June 1975.

[BK85] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings 8th DoD/NBS Computer Security Initiative Conference*, pages 18–27, Gaithersburg, MD, September 1985.

[BL76] D. E. Bell and L. J. La Padula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Mitre Corporation, Bedford, MA, March 1976.

[BMD92] A. Burns, J. McDermid, and J. Dobson. On the meaning of safety and security. *Computer Journal*, 35(1):3–15, February 1992.

[BN89] David F. C. Brewer and Michael J. Nash. The Chinese Wall security policy. In *Proceedings of the Symposium on Security and Privacy*, pages 204–219, Oakland, CA, May 1989. IEEE Computer Society.

[BW91] A. Burns and A. J. Wellings. Criticality and utility in the next generation. *Real-Time Systems*, 3(4):351–354, December 1991. (Correspondence).

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CG93] Raymond C. Clark and Ira B. Greenberg. *Supporting Real-Time Processing in Multilevel Secure Systems.* Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Submitted for publication.

[CHB79] R. H. Campbell, K. H. Horton, and G. G. Belford. Simulations of a fault-tolerant deadline mechanism. In *Fault Tolerant Computing Symposium 9*, pages 95–101, Madison, WI, June 1979. IEEE Computer Society.

[Che89] Mikhail Chernyshov. Post-mortem on failure. *Nature*, 339:9, May 4, 1989.

[CHR92] Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1992.

[CLL90] Jen-Yao Chung, Jane W. S. Liu, and Kwei-Jay Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transactions on Computers*, 39(9):1156–1174, September 1990.

[CLS88] Stephen S. Cha, Nancy G. Leveson, and Timothy J. Shimeall. Safety verification in Murphy using fault tree analysis. In *10th International Conference on Software Engineering*, pages 377–386, Singapore, April 1988. IEEE Computer Society.

[CM92] Oliver Costich and John McDermott. A multilevel transaction problem for multilevel secure database systems and its solution for the replicated architecture. In *Proceedings of the Symposium on Research in Security and Privacy*, pages 192–203, Oakland, CA, May 1992. IEEE Computer Society.

[Coo90] Henry S. F. Cooper Jr. Annals of space (the planetary community)—part 1: Phobos. *New Yorker*, pages 50–84, June 11, 1990.

[Cri82] Flaviu Cristian. Robust data types. *Acta Informatica*, 17:365–397, 1982.

[Cri88] Flaviu Cristian. Agreeing who is present and who is absent in a synchronous distributed system. In *Fault Tolerant Computing Symposium 18*, pages 206–211, Tokyo, Japan, June 1988. IEEE Computer Society.

[Cri89] Flaviu Cristian. Exception handling. In T. Anderson, editor, *Dependability of Resilient Computers*. Blackwell Scientific Publications, 1989.

[Cri91] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[CW87] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the Symposium on Security and Privacy*, pages 184–194, Oakland, CA, April 1987. IEEE Computer Society.

[dBHdRR91] J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors. *Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, REX Workshop, Mook, The Netherlands, June 1991. Springer-Verlag.

[Den87] D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2):222–232, February 1987.

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[DoD85] *Department of Defense Trusted Computer System Evaluation Criteria.* Department of Defense, December 1985. DOD 5200.28-STD (supersedes CSC-STD-001-83).

[DR86] J. E. Dobson and B. Randell. Building reliable secure computing systems out of unreliable unsecure components. In *Proceedings of the Symposium on Security and Privacy*, pages 187–193, Oakland, CA, April 1986. IEEE Computer Society.

[DRC86] *Distributed Systems Technology Assessment for SDI.* Dynamics Research Corporation, Wilmington, MA, September 1986. Final Report to Rome Air Development Center, TEMS Task 0029, Contract F19628-84-D-0016.

[DS92] Sadegh Davari and Lui Sha. Sources of unbounded priority inversions in real-time systems and a comparative study of possible solutions. *ACM Operating Systems Review*, 26(2):110–120, April 1992.

[ECK$^+$91] Dave E. Eckhardt, Alper K. Caglayan, John C. Knight, Larry D. Lee, David F. McAllister, Mladen A. Vouk, and John P. J. Kelly. An experimental evaluation of software redundancy as a strategy for improving reliability. *IEEE Transactions on Software Engineering*, 17(7):692–702, July 1991.

[EL85] Dave E. Eckhardt, Jr. and Larry D. Lee. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Transactions on Software Engineering*, SE-11(12):1511–1517, December 1985.

[Eme90] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 16, pages 995–1072. Elsevier and MIT press, Amsterdam, The Netherlands, and Cambridge, MA, 1990.

[FAA88] *System Design and Analysis*. Federal Aviation Administration, June 21, 1988. Advisory Circular 25.1309-1A.

[FDP86] Jean-Michael Fray, Yves Deswarte, and David Powell. Intrusion-tolerance using fine-grain fragmentation-scattering. In *Proceedings of the Symposium on Security and Privacy*, pages 194–201, Oakland, CA, April 1986. IEEE Computer Society.

[GAO92] *Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia*. United States General Accounting Office, Washington, DC, February 1992. GAO/IMTEC-92-26.

[Gli84] Virgil D. Gligor. A note on denial-of-service in operating systems. *IEEE Transactions on Software Engineering*, SE-10(3):320–324, May 1984.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the Symposium on Security and Privacy*, pages 11–20, Oakland, CA, April 1982. IEEE Computer Society.

[GM84] J. A. Goguen and J. Meseguer. Inference control and unwinding. In *Proceedings of the Symposium on Security and Privacy*, pages 75–86, Oakland, CA, April 1984. IEEE Computer Society.

[Gra91] James W. Gray, III. On information flow security models. In *Proceedings of the Computer Security Foundations Workshop IV*, pages 55–60, Franconia, NH, June 1991. IEEE Computer Society.

[Ham92] Dick Hamlet. Are we testing for true reliability? *IEEE Software*, 9(4):21–27, July 1992.

[HD92] Kenneth Hoyme and Kevin Driscoll. SAFEbus™. In *AIAA/IEEE Digital Avionics Systems Conference*, 1992.

[Hel86] K. A. Helps. Some verification tools and methods for airborne safety-critical software. *IEE/BCS Software Engineering Journal*, 1(6):248–253, November 1986.

[HLCM92] C. L. Heitmeyer, B. G. Labaw, P. C. Clements, and A. K. Mok. Engineering CASE tools to support formal methods for real-time software development. In *Fifth International Workshop on Computer-Aided Software Engineering*, pages 110–113, Montreal, Quebec, Canada, July 1992. IEEE Computer Society.

[HN81] A. J. Herbert and R. M. Needham. Sequencing computation steps in a network. In *8th ACM Symposium on Operating System Principles*, pages 59–63, Asilomar, CA, December 1981. (ACM *Operating Systems Review*, Vol. 15, No. 5).

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[HS91] M. C. McElvany Hugue and P. David Stotts. Guaranteed task deadlines for fault-tolerant workloads with conditional branches. *Real-Time Systems*, 3(3):275–305, September 1991.

[HW91] Maurice P. Herlihy and Jeannette M. Wing. Specifying graceful degradation. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):93–104, January 1991.

[HY87] J. Thomas Haigh and William D. Young. Extending the noninterference version of MLS for SAT. *IEEE Transactions on Software Engineering*, SE-13(2):141–150, February 1987.

[IFI92] *3rd IFIP Working Conference on Dependable Computing for Critical Applications*, Mondello, Sicily, Italy, September 1992. IFIP WG 10.4. Preprint proceedings.

[IRM84] Stephen D. Ishmael, Victoria A. Regenie, and Dale A. Mackall. Design implications from AFTI/F16 flight test. NASA Technical Memorandum 86026, NASA Ames Research Center, Dryden Flight Research Facility, Edwards, CA, 1984.

[JA88] Mark E. Joseph and Algirdas Avižienis. A fault tolerance approach to computer viruses. In *Proceedings of the Symposium on Security and Privacy*, pages 52–58, Oakland, CA, April 1988. IEEE Computer Society.

[Jac89] Jeremy Jacob. On the derivation of secure components. In *Proceedings of the Symposium on Security and Privacy*, pages 242–247, Oakland, CA, May 1989. IEEE Computer Society.

[Jen92] E. Douglas Jensen. *Asynchronous Decentralized Realtime Computer Systems*. NATO Advanced Study Institute on Real-Time Computing, Sint Maarten, October 1992.

[JM86] Farnam Jahanian and Aloysius Ka-Lau Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

[JT88] Dale M. Johnson and F. Javier Thayer. Security and the composition of machines. In *Proceedings of the Computer Security Foundations Workshop*, pages 72–89, Franconia, NH, June 1988. Mitre Corporation.

[Kar88] Paul A. Karger. Implementing commercial data integrity with secure capabilities. In *Proceedings of the Symposium on Research in Security and Privacy*, pages 130–139, Oakland, CA, April 1988. IEEE Computer Society.

[KGR89] H. Kopetz, G. Grünsteidl, and J. Reisinger. Fault-tolerant membership service in a synchronous distributed real-time system. In A. Avižienis and J. C. Laprie, editors, *Dependable Computing for Critical Applications*, pages 411–429, Santa Barbara, CA, August 1989. Volume 4 of *Dependable Computing and Fault-Tolerant Systems*, Springer-Verlag, Wien, Austria.

[KL86] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, November 1990.

[KR89] Stuart W. Katzke and Zell G. Ruthberg, editors. *Report of the Invitational Workshop on Integrity Policy in Computer Information Systems (WIPCIS)*, Gaithersburg, MD, January 1989. National Institute of Standards and Technology. NIST Special Publication 500-160.

[Lam83a] L. Lamport. Sometime is sometimes not never. In *10th ACM Symposium on Principles of Programming Languages*, pages 174–185, Austin, TX, January 1983.

[Lam83b] L. Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83*, pages 657–668, Paris, 1983. North-Holland.

[Lam84] Leslie Lamport. Solved problems, unsolved problems and non-problems in concurrency. In *3rd ACM Symposium on Principles of Distributed Computing*, pages 1–11, Vancouver, B.C., Canada, August 1984. Association for Computing Machinery.

[Lam89] Leslie Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[Lam91] Leslie Lamport. The temporal logic of actions. Technical Report 79, DEC Systems Research Center, Palo Alto, CA, December 1991. To Appear in ACM Transactions on Programming Languages and Systems.

[Lap85] J. C. Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *Fault Tolerant Computing Symposium 15*, pages 2–11, Ann Arbor, MI, June 1985. IEEE Computer Society.

[Lap91] J. C. Laprie, editor. *Dependability: Basic Concepts and Terminology in English, French, German, Italian and Japanese*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, Wien, Austria, February 1991.

[LC86] Arthur L. Liestman and Roy H. Campbell. A fault-tolerant scheduling problem. *IEEE Transactions on Software Engineering*, SE-12(11):1089–1095, November 1986.

[Lee88] Theodore M. P. Lee. Using mandatory integrity to enforce "commercial" security. In *Proceedings of the Symposium on Research in Security and Privacy*, pages 140–146, Oakland, CA, April 1988. IEEE Computer Society.

[Lev84] N. G. Leveson. Software safety in computer controlled systems. *IEEE Computer*, 17(2):48–55, February 1984.

[Lev86] Nancy G. Leveson. Software safety: Why, what and how. *ACM Computing Surveys*, 18(2):125–163, June 1986.

[Lev91] Nancy G. Leveson. Software safety in embedded computer systems. *Communications of the ACM*, 34(2):34–46, February 1991.

[LH83] N. G. Leveson and P. R. Harvey. Analyzing software safety. *IEEE Transactions on Software Engineering*, SE-9(5):569–579, September 1983.

[LHM84] C. E. Landwehr, C. L. Heitmeyer, and J. McLean. A security model for military message systems. *ACM Transactions on Computer Systems*, 2(3):198–222, August 1984.

[Lip82] Steven B. Lipner. Non-discretionary controls for commercial applications. In *Proceedings of the Symposium on Security and Privacy*, pages 2–10, Oakland, CA, April 1982. IEEE Computer Society.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[LM89] B. Littlewood and D. R. Miller. Conceptual modeling of coincident failures in multiversion software. *IEEE Transactions on Software Engineering*, 15(12):1596–1614, December 1989.

[LMS85] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[Loc92] C. Douglass Locke. Software architecture for hard real-time applications: Cyclic executives vs. priority executives. *Real-Time Systems*, 4(1):37–53, March 1992.

[LR93a] Patrick Lincoln and John Rushby. Formal verification of an algorithm for interactive consistency under a hybrid fault model. In Costas Courcoubetis, editor, *Computer Aided Verification, CAV '93*, pages 292–304, Elounda, Greece, June/July 1993. Volume 697 of *Lecture Notes in Computer Science*, Springer-Verlag.

[LR93b] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault Tolerant Computing Symposium 23*, pages 402–411, Toulouse, France, June 1993. IEEE Computer Society.

[LS93] Bev Littlewood and Lorenzo Strigini. Validation of ultrahigh dependability for software-based systems. *Communications of the ACM*, pages 69–80, November 1993.

[LSD89] John P. Lehoczky, Lui Sha, and Ye Ding. The rate monotonic scheduling algorithm—exact characterization and average case behavior. In *Real Time Systems Symposium*, pages 166–171, Santa Monica, CA, December 1989. IEEE Computer Society.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[Lut93] Robyn R. Lutz. Analyzing software requirements errors in safety-critical embedded systems. In *IEEE International Symposium on Requirements Engineering*, pages 126–133, San Diego, CA, January 1993.

[LW92] Robyn R. Lutz and Johnny S. K. Wong. Detecting unsafe error recovery schedules. *IEEE Transactions on Software Engineering*, 18(8):749–760, August 1992.

[Mac88] Dale A. Mackall. Development and flight test experiences with a flight-crucial digital control system. NASA Technical Paper 2857, NASA Ames Research Center, Dryden Flight Research Facility, Edwards, CA, 1988.

[Man67] G. K. Manacher. Production and stabilization of real-time task schedules. *Journal of the ACM*, 14(3):439–465, July 1967.

[MC83] J. K. Millen and C. M. Cerniglia. Computer security models. Working Paper WP25068, Mitre Corporation, Bedford, MA, September 1983.

[McC87] Daryl McCullough. Specifications for multi-level security and a hook-up property. In *Proceedings of the Symposium on Security and Privacy*, pages 161–166, Oakland, CA, April 1987. IEEE Computer Society.

[McC88] Daryl McCullough. Noninterference and composability of security properties. In *Proceedings of the Symposium on Security and Privacy*, pages 177–186, Oakland, CA, April 1988. IEEE Computer Society.

[McL85] John McLean. A comment on the "basic security theorem" of Bell and La Padula. *Information Processing Letters*, 20:67–70, 1985.

[McL90] John McLean. Security models and information flow. In *Proceedings of the Symposium on Research in Security and Privacy*, pages 180–187, Oakland, CA, May 1990. IEEE Computer Society.

[McL94] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of the Symposium on Research in Security and Privacy*, Oakland, CA, May 1994. IEEE Computer Society. To appear.

[MG90] William T. Maimone and Ira B. Greenberg. Single-level multiversion schedulers for multilevel secure database systems. In *Proceedings of the Sixth Annual Computer Security Applications Conference*, pages 137–147, Tucson, AZ, December 1990. IEEE Computer Society.

[Mil90] Jonathan K. Millen. Hookup security for synchronous machines. In *Proceedings of the Computer Security Foundations Workshop III*, pages 84–90, Franconia, NH, June 1990. IEEE Computer Society.

[Mil92] Jonathan K. Millen. A resource allocation model for denial of service. In *Proceedings of the Symposium on Research in Security and Privacy*, pages 137–147, Oakland, CA, May 1992. IEEE Computer Society.

[MOD91a] *Interim Defence Standard 00-55: The procurement of safety critical software in defence equipment.* UK Ministry of Defence, April 1991. Part 1, Issue 1: Requirements; Part 2, Issue 1: Guidance.

[MOD91b] *Interim Defence Standard 00-56: Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment.* UK Ministry of Defence, April 1991.

[MP88] Luigi V. Mancini and Giuseppe Pappalardo. Towards a theory of replicated processing. In M. Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 175–192, Warwick, England, September 1988. Volume 331 of *Lecture Notes in Computer Science*, Springer-Verlag.

[MQ92] Mark Moriconi and Xiaolei Qian. Closed-world refinement of software architectures. Technical Report SRI-CSL-92-10, Computer Science Laboratory, SRI International, Menlo Park, CA, October 1992.

[MS87] P. M. Melliar-Smith. Extending interval logic to real time systems. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, pages 224–242, Altrincham, UK, April 1987. Volume 398 of *Lecture Notes in Computer Science*, Springer-Verlag.

[Neu86] P. G. Neumann. On hierarchical design of computer systems for critical applications. *IEEE Transactions on Software Engineering*, SE-12(9), September 1986. Reprinted in Rein Turn (ed.), Advances in Computer System Security, Volume 3, Artech House, 1988.

[Nor92] Jens Nordahl. Design for dependability. In *3rd IFIP Working Conference on Dependable Computing for Critical Applications* [IFI92], pages 29–38.

[Ost90] Jonathan S. Ostroff. A logic for real-time discrete event processes. *IEEE Control Systems Magazine*, 10(4):95–102, June 1990.

[PB85] Daniel L. Palumbo and Ricky W. Butler. Measurement of SIFT operating system overhead. NASA Technical Memorandum 86322, NASA Langley Research Center, Hampton, VA, April 1985.

[PC86] D. L. Parnas and P. C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2), February 1986. Correction: Aug 86, page 874.

[Pel91] Jan Peleska. Design and verification of fault tolerant systems with CSP. *Distributed Computing*, 5(2):95–106, 1991.

[Per84] Charles Perrow. *Normal Accidents: Living with High Risk Technologies.* Basic Books, New York, NY, 1984.

[PGK88] D. A. Patterson, G. A. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference Proceedings*, pages 109–116, Chicago, IL, June 1988.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, November 1977. ACM.

[PvSK90] David L. Parnas, A. John van Schouwen, and Shu Po Kwan. Evaluation of safety-critical software. *Communications of the ACM*, 33(6):636–648, June 1990.

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[Ric60] P. Richards. Timing properties of multiprocessor systems. Technical Report TDB60-27, Tech. Operations Inc., Burlington, MA, August 1960.

[RTCA92] *DO-178B: Software Considerations in Airborne Systems and Equipment Certification.* Requirements and Technical Concepts for Aviation, Washington, DC, December 1992. This document is known as EUROCAE ED-12B in Europe.

[Rus81] John Rushby. The design and verification of secure systems. In *8th ACM Symposium on Operating System Principles*, pages 12–21, Asilomar, CA, December 1981. (ACM *Operating Systems Review*, Vol. 15, No. 5).

[Rus89] John Rushby. Kernels for safety? In T. Anderson, editor, *Safe and Secure Computing Systems*, chapter 13, pages 210–220. Blackwell Scientific Publications, 1989. (Proceedings of a Symposium held in Glasgow, October 1986).

[Rus91] John Rushby. Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems. Technical Report SRI-CSL-91-3, Computer Science Laboratory, SRI International, Menlo Park, CA, January 1991. Also available as NASA Contractor Report 4384, July 1991.

[Rus92a] John Rushby. Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems. In Vytopil [Vyt92], pages 237–257.

[Rus92b] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report SRI-CSL-92-02, Computer Science Laboratory, SRI International, Menlo Park, CA, October 1992.

[Rus93] John Rushby. *Formal Methods and Digital Systems Validation for Airborne Systems*. Federal Aviation Administration Technical Center, Atlantic City, NJ, 1993. Forthcoming chapter for "Digital Systems Validation Handbook," DOT/FAA/CT-88/10.

[RvH93] John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, January 1993.

[Sch87] Fred B. Schneider. Decomposing properties into safety and liveness using predicate logic. Technical Report 87-874, Department of Computer Science, Cornell University, Ithaca, NY, October 1987.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[Sch92] Henk Schepers. Tracing fault tolerance. In *3rd IFIP Working Conference on Dependable Computing for Critical Applications* [IFI92], pages 39–48.

[Sch93] Marco Schneider. Self stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.

[Sha92] Natarajan Shankar. Mechanical verification of a generalized protocol for Byzantine fault-tolerant clock synchronization. In Vytopil [Vyt92], pages 217–236.

[SIG91] *SIGSOFT '91: Software for Critical Systems*, New Orleans, LA, December 1991. Published as ACM SIGSOFT Engineering Notes, Volume 16, Number 5.

[SLR86] Lui Sha, John P. Lehoczky, and Ragunathan Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In *Real Time Systems Symposium*, pages 181–191, New Orleans, LA, December 1986. IEEE Computer Society.

[SMSV83] R. L. Schwartz, P. M. Melliar-Smith, and F. Vogt. An interval logic for higher-level temporal reasoning. In *ACM Symposium on Principles of Distributed Computing*, pages 173–186, August 1983.

[SR90] John A. Stankovic and Krithi Ramamritham. What is predictability for real-time systems? *Real-Time Systems*, 2(4):247–254, November 1990. (Editorial).

[SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[Stu90] Douglas A. Stuart. Implementing a verifier for real-time systems. In *Real Time Systems Symposium*, pages 62–71, Lake Buena Vista, FL, December 1990. IEEE Computer Society.

[Tay84] T. Taylor. Comparison paper between the Bell and La Padula model and the SRI model. In *Proceedings of the Symposium on Security and Privacy*, pages 195–202, Oakland, CA, April 1984. IEEE Computer Society.

[TP88] Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *7th Symposium on Reliable Distributed Systems*, pages 93–100, Columbus, OH, October 1988. IEEE Computer Society.

[Unk90] Unknown. The unique signal concept. In *Proceedings of the First Las Cruces Workshop on Software Safety*, page 3rd from end, Las Cruces, NM, October 1990. Department of Computer Science, New Mexico State University. (Untitled, unnumbered, copies of viewgraphs).

[Vyt92] J. Vytopil, editor. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, Nijmegen, The Netherlands, January 1992. Springer-Verlag.

[W$^+$78] John H. Wensley et al. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.

[Wal90] Chris J. Walter. Identifying the cause of detected errors. In *Fault Tolerant Computing Symposium 20*, pages 48–55, Newcastle upon Tyne, UK, June 1990. IEEE Computer Society.

[Web89] D. G. Weber. Formal specification of fault-tolerance and its relation to computer security. In *Proceedings of the Fifth International Workshop on Software Specification and Design*, pages 273–277, Pittsburgh, PA, May 1989. Published as ACM SIGSOFT Engineering Notes, Volume 14, Number 3.

[WHCC80] A. Y. Wei, K. Hiraishi, R. Cheng, and R. H. Campbell. Application of the fault-tolerant deadline mechanism to a satellite on-board computer system. In *Fault Tolerant Computing Symposium 10*, pages 107–109, Kyoto, Japan, June 1980. IEEE Computer Society.

[WJ90] J. Todd Wittbold and Dale M. Johnson. Information flow in nondeterministic systems. In *Proceedings of the Symposium on Research in Security and Privacy*, pages 144–161, Oakland, CA, May 1990. IEEE Computer Society.

[XP91] Jia Xu and David Lorge Parnas. On satisfying timing constraints in hard-real-time systems. In SIG [SIG91], pages 132–146.

[YG90] Che-Fn Yu and Virgil D. Gligor. A specification and verification method for the prevention of denial of service. *IEEE Transactions on Software Engineering*, 16(6):581–592, June 1990.