

28

Fault-Tolerant Avionics

Ellis F. Hitt

Battelle

Dennis Mulcare

*Science Applications
International Co.*

- 28.1 [Introduction](#)
Motivation • Definitional Framework • Dependability
• Fault Tolerance Options • Flight Systems
Evolution • Design Approach
 - 28.2 [System Level Fault Tolerance](#)
General Mechanization • Redundancy
Options • Architectural Categories • Integrated Mission
Avionics • System Self Tests
 - 28.3 [Hardware-Implemented Fault Tolerance
\(Fault-Tolerant Hardware Design Principles\)](#)
Voter Comparators • Watchdog Timers
 - 28.4 [Software-Implemented Fault Tolerance—State
Consistency](#)
Error Detection • Damage Confinement and
Assessment • Error Recovery • Fault
Treatment • Distributed Fault Tolerance
 - 28.5 [Software Fault Tolerance](#)
Multiversion Software • Recovery Blocks • Trade-Offs
 - 28.6 [Summary](#)
Design Analyses • Safety • Validation • Conclusion
- [References](#)
[Further Information](#)

28.1 Introduction

Fault-tolerant designs are required to ensure safe operation of digital avionics systems performing flight-critical functions. This chapter discusses the motivation for fault-tolerant designs, and the many different design practices that are evolving to implement a fault-tolerant system. The designer needs to make sure the fault tolerance requirements are fully defined to select the design concept to be implemented from the alternatives available. The requirements for a fault-tolerant system include performance, dependability, and the methods of assuring that the design, when implemented, meets all requirements. The requirements must be documented in a specification of the intended behavior of a system, specifying the tolerances imposed on the various outputs from the system [Anderson and Lee, 1981].

Development of the design proceeds in parallel with the development of the methods of assurance to validate that the design meets all requirements including the fault tolerance. The chapter concludes with references to further reading in this developing field.

A fault-tolerant system provides continuous, safe operation in the presence of faults. A fault-tolerant avionics system is a critical element of flight-critical architectures which include the fault-tolerant computing system (hardware, software, and timing), sensors and their interfaces, actuators, elements, and data communication among the distributed elements. The fault-tolerant avionics system ensures integrity

of output data used to control the flight of the aircraft, whether operated by the pilot or autopilot. A fault-tolerant system must detect errors caused by faults, assess the damage caused by the fault, recover from the error, and isolate the fault. It is generally not economical to design and build a system that is capable of tolerating all possible faults in the universe. The faults the system is to be designed to tolerate must be defined based on analysis of requirements including the probability of each fault occurring, and the impact of not tolerating the fault.

A user of a system may observe an error in its operation which is the result of a fault being triggered by an event. Stated another way, *a fault is the cause of an error, and an error is the cause of a failure*. A mistake made in designing or constructing a system can introduce a fault into the design of the system, either because of an inappropriate selection of components or because of inappropriate (or missing) interactions between components. On the other hand, if the design of a system is considered to be correct, then an erroneous transition can occur only because of a failure of one of the components of the system. Design faults require more powerful fault-tolerance techniques than those needed to cope with component faults. Design faults are unpredictable; their manifestation is unexpected and they generate unanticipated errors. In contrast, component faults can often be predicted, their manifestation is expected, and they produce errors which can be anticipated [Anderson and Lee, 1981].

In a non-fault-tolerant system, diagnosis is required to determine the cause of the fault that was observed as an error. Faults in avionics systems are of many types. They generally can be classified as hardware, software, or timing related. Faults can be introduced into a system during any phase of its life cycle including requirements definition, design, production, or operation.

In the 1960s, designers strived to achieve highly reliable safe systems by avoiding faults, or masking faults. The Apollo guidance and control system employed proven, highly reliable components and triple modular redundancy (TMR) with voting to select the correct output. Improvements in hardware reliability, and our greater knowledge of faults and events which trigger them, has led to improved design methods for fault-tolerant systems which are affordable.

In any fault-tolerant system, the range of potential fault conditions that must be accommodated is quite large; enumerating all such possibilities is a vital yet formidable task in validating the system's airworthiness, or its readiness for deployment. The resultant need to handle each such fault condition prompts attention to the various assurance-oriented activities that contribute to certification system airworthiness.

28.1.1 Motivation

Safety is of primary importance to the economic success of the aviation system. The designer of avionics systems must assure that the system provides the required levels of safety to passengers, aircrew, and maintenance personnel. Fault-tolerant systems are essential with the trend to increasingly complex digital systems.

Many factors necessitate fault tolerance in systems that perform functions that must be sustained without significant interruption. In avionics systems, such functions are often critical to continued safe flight or to the satisfactory conduct of a mission; hence the terms flight-critical and mission-critical. The first compelling reality is that physical components are non-ideal, i.e., they are inescapably disposed to physical deterioration or failure. Clearly then, components inherently possess a finite useful life, which varies with individual instances of the same component type. At some stage, then, any physical component will exhibit an abrupt failure or excessive deterioration such that a fault may be detected at some level of system operation.

The second contributing factor to physical faults is the non-ideal environment in which an avionics system operates. Local vibrations, humidity, temperature cycling, electrical power transients, electromagnetic interference, etc. tend to induce stress on the physical component which may cause abrupt failure or gradual deterioration. The result may be a transient or a permanent variation in output, depending on the nature and severity of the stress. The degree of induced deterioration encountered may profoundly influence the useful life of a component. Fortunately, design measures can be taken to reduce susceptibility to the various environmental effects. Accordingly, a rather comprehensive development approach is

needed for system dependability, albeit fault tolerance is the most visible aspect because it drives the organization and logic of the system architecture.

The major factor necessitating fault tolerance is design faults. Tolerance of design faults in hardware and software and the overall data flow is required to achieve the integrity needed for flight-critical systems. Reliance on the hardware chip producing correct output when there is no physical failure has been shown to be risky, as demonstrated by the design error discovered in the floating point unit of a high-performance microprocessor in wide use. Because of the difficulty in eliminating all design faults, dissimilar redundancy is used to produce outputs which should be identical even though computed by dissimilar computers. Use of dissimilar redundancy is one approach to tolerating common-mode failures. A common-mode failure (CMF) occurs when copies of a redundant system suffer faults nearly simultaneously, generally due to a single cause [Lala, 1994].

28.1.2 Definitional Framework

A digital avionics system is a “hard real-time” system producing time-critical outputs that are used to control the flight of an aircraft. These critical outputs must be dependable. A dependable system is both reliable and safe. Reliability has many definitions, and is often expressed as the probability of not failing. Another definition of reliability is the probability of producing a “correct” output [Vaidya and Pradhan, 1993]. Safety has been defined as the probability that the system output is either correct, or the error in the output is detectable [Vaidya and Pradhan, 1993]. Correctness has been defined as the requirement that the output of all channels agree bit-for-bit under no-fault conditions [Lala, 1994]. Another design approach, approximate consensus, considers a system to be correct if the outputs agree within some threshold. Both approaches are in use.

Hardware component faults are often classified by extent, value, and duration [Avizienis, 1976]. Extent applies to whether the errors generated by the fault are localized or distributed; value indicates whether the fault generates fixed or varying erroneous values; duration refers to whether the fault is transient or permanent. Several studies have shown that permanent faults cause only a small fraction of all detected errors, as compared with transient faults [Sosnowski, 1994]. A recurring transient fault is often referred to as intermittent [Anderson and Lee, 1981]. Figure 28.1 depicts these classifications in the tree of faults.

Origin faults may result from a physical failure within a hardware component of a system, or may result from human-made faults. System boundary internal faults are those parts of the system’s state which, when invoked by the computation activity, will produce an error, while external faults result from system interference caused by its physical environment, or from system interaction with its human environment. Origin faults classified by the time phase of creation include design faults resulting from

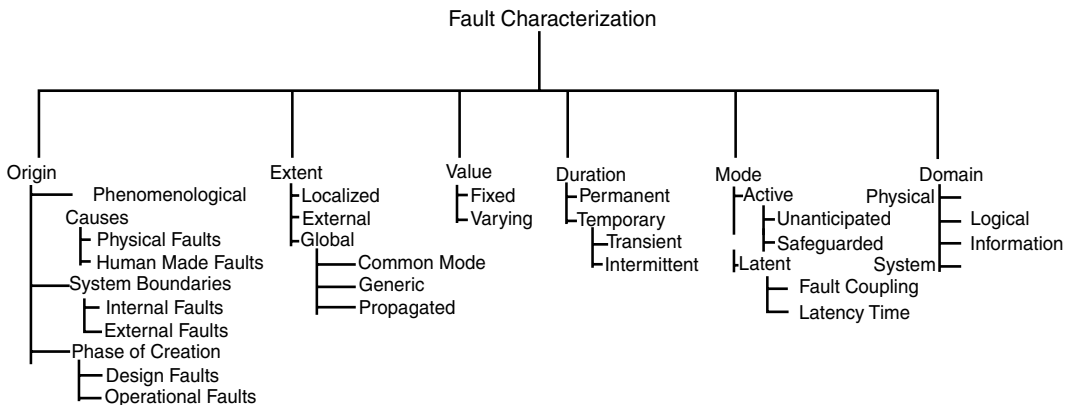


FIGURE 28.1 Fault classification.

imperfections that arise during the development of the system (from requirements specification to implementation), subsequent modifications, the establishment of procedures for operating or maintaining the system, or operational faults which appear during the system operation [Lala and Harper, 1994].

A fault is in the active mode if it yields an **erroneous state**, either in hardware or software, i.e., a state that differs from normal expectations under extant circumstances. Alternatively, a fault is described as latent when it is not yielding an erroneous state. Measures for error detection that can be used in a fault-tolerant system fall into the following broad classification [Anderson and Lee, 1981]:

1. Replications checks
2. Timing checks
3. Reversal checks
4. Coding checks
5. Reasonableness checks
6. Structural checks
7. Diagnostic checks.

These checks are discussed in Section 28.4.

During the development process, it is constructive to maintain a perspective regarding the fault attributes of Domain and Value in [Figure 28.1](#). Basically, Domain refers to the universe of layering of fault abstractions that permit design issues to be addressed with a minimum of distraction. Value simply refers to whether the erroneous state remains fixed, or whether it indeterminately fluctuates. While proficient designers tend to select the proper abstractions to facilitate particular development activities, these associated fault domains should be explicitly noted:

- Physical: elemental PHYSICAL FAILURES of hardware components — *underlying short, open, ground faults*
- Logical: manifested LOGICAL FAULTS per device behavior — *stuck-at-one, stuck-at-zero, inverted*
- Informational: exhibited ERROR STATES in interpreted results — *incorrect value, sign change, parity error*
- System: resultant SYSTEM FAILURE provides unacceptable service — *system crash, deadlock, hardover.*

These fault domains constitute levels of design responsibilities and commitments as well as loci of fault tolerance actions per se. Thus, fault treatment and, in part, fault containment, are most appropriately addressed in the physical fault domain. Similarly, hardware fault detection and assessment are most readily managed in the logical fault domain, where the fixed or fluctuating nature of the erroneous value(s) refines the associated fault identification mechanism(s). Lastly, error recovery and perhaps some fault containment are necessarily addressed in the informational fault domain, and service continuation in the system fault domain.

For safety-critical applications, physical hardware faults no longer pose the major threat to dependability. The dominant threat is now common-mode failures. Common-mode failures (CMFs) result from faults that affect more than one fault containment region at the same time, generally due to a common cause. Fault avoidance, fault removal through test and evaluation or via fault insertion, and fault tolerance implemented using exception handlers, program checkpointing, and restart are approaches used in tolerating CMFs [Lala, 1994]. [Table 28.1](#) presents a classification of common-mode faults with the X indicating the possible combinations of faults that must be considered which are not intentional faults.

Physical, internal, and operational faults can be tolerated by using hardware redundancy. All other faults can affect multiple fault-containment regions simultaneously. Four sources of common-mode failures need to be considered:

1. Transient (External) Faults which are the result of temporary interference to the system from its physical environment such as lightning, High Intensity Radio Frequencies (HIRF), heat, etc.;
2. Permanent (External) Faults which are the result of permanent system interference caused by its operational environment such as heat, sand, salt water, dust, vibration, shock, etc.;

TABLE 28.1 Classification of Common-Mode Faults

Phenomenological Cause		System Boundary		Phase of Creation		Duration		Common Mode Fault Label
Physical	Human Made	Internal	External	Design	Operational	Permanent	Temporary	
X			X		X		X	Transient (External) CMF
X			X		X	X		Permanent (External) CMF
	X	X		X			X	Intermittent (Design) CMF
	X	X		X		X		Permanent (Design) CMF
	X		X		X		X	Interaction CMF

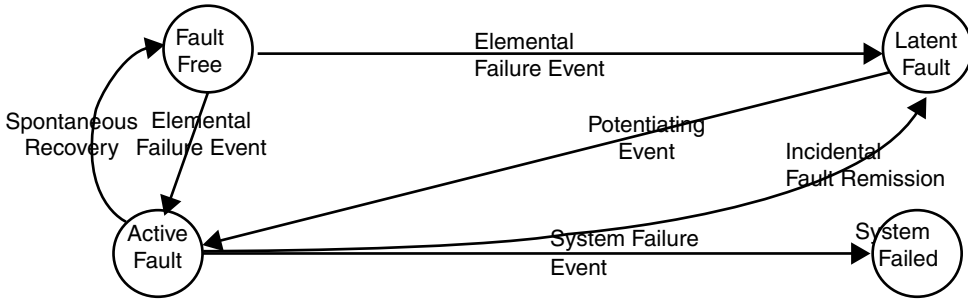


FIGURE 28.2 Hardware states (no corrective action).

- Intermittent (Design) Faults which are introduced due to imperfections in the requirements specifications, detailed design, implementation of design, and other phases leading up to the operation of the system;
- Permanent (Design) Faults are introduced during the same phases as intermittent faults, but manifest themselves permanently [Lala, 1994].

An **elemental physical failure**, which is an event resulting in component malfunction, produces a physical fault. These definitions are reflected in [Figure 28.2](#), a state transition diagram that portrays four fault status conditions and associated events in the absence of fault tolerance. Here, for example, a Latent Fault Condition transitions to an Active Fault Condition due to a *Potentiating* Event. Such an event might be a functional mode change that caused the fault region to be exercised in a revealing way. Following the incidence of a sufficiently severe active fault from which spontaneous recovery is not forthcoming, a **system failure event** occurs wherein expected functionality can no longer be sustained. If the effects of a particular active fault are not too debilitating, a system may continue to function with some degradation in services. Fault tolerance can of course forestall both the onset of system failure and the expectation of degraded services.

The Spontaneous Recovery Event in [Figure 28.2](#) indicates that faults can sometimes be *transient* in nature when a fault vanishes without purposeful intervention. This phenomenon can occur after an external disturbance subsides or an intermittent physical aberration ceases. A somewhat similar occurrence is provided through the Incidental Fault Remission Event in [Figure 28.2](#). Here, the fault does not

TABLE 28.2 Delineation of Fault Conditions

Recovered Mode	Latent Mode	Active Mode
Spontaneous Recovery following Disruption or Marginal Physical Fault Recovery	—	Erroneous State Induced by Transient Disturbance or Passing Manifestation of Marginal Fault
—	Hard Physical Fault Remission or Hard Physical Fault Latency	Passing Manifestation of Hard Fault or Persistent Manifestation of Hard Fault

vanish, but rather spontaneously reverts from an active to a latent mode due to the cessation or removal of fault excitation circumstances. Table 28.2 complements Figure 28.2 in clarifying these fault categories. Although these transient fault modes are thought to account for a large proportion of faults occurring in deployed systems, such faults may nonetheless persist long enough to appear as permanent faults to the system. In many cases then, explicit features must be incorporated into a system to ensure the timely recovery from faults that may induce improper or unsafe system operation.

Three classes of faults are of particular concern because their effects tend to be global regarding extent, where global implies impact on redundant components present in fault-tolerant systems. A **common mode-fault** is one in which the occurrence of a single physical fault at a one particular point in a system can cause coincident debilitation of all similar redundant components. This phenomenon is possible where there is a lack of protected redundancy, and the consequence would be a massive failure event. A **generic fault** is a development fault that is replicated across similar redundant components such that its activation yields a massive failure event like that due to a common mode fault. A **propagated fault** is one wherein the effects of a single fault spread out to yield a compound erroneous state. Such fault symptoms are possible when there is a lack of fault containment features. During system development, particular care must be exercised to safeguard against these classes of global faults, for they can defeat fault-tolerance provisions in a single event. Considerable design assessment is therefore needed, along with redundancy provisions, to ensure system dependability.

28.1.3 Dependability

Dependability is an encompassing property that enables and justifies reliance upon the services of a system. Hence, dependability is a broad, qualitative term that embodies the aggregate nonfunctional attributes, or “ilities,” sought in an ideal system, especially one whose continued safe performance is critical. Thus, attributes like safety, reliability, availability, and maintainability, which are quantified using conditional probability formulas, can be conveniently grouped as elements of dependability. As a practical matter, however, dependability usually demands the incorporation of fault tolerance into a system to realize quantitative reliability or availability levels. Fault tolerance, moreover, may be need to achieve maintainability requirements, as in the case of on-line maintenance provisions.

For completeness, sake, it should be noted as depicted in Figure 28.3 that the attainment of dependability relies on fault avoidance and fault alleviation, as well as on fault tolerance.

This figure is complemented by Table 28.3, which emphasizes the development activities, such as analyzing fault possibilities during design to minimize the number and extent of potential fault cases. This activity is related to the criteria of containment in Figure 28.3 in that the analysis should ensure that both the number and propagation of fault cases are contained. The overall notion here is to minimize the number of fault possibilities, to reduce the prospects of their occurrences, and to ensure the safe handling of those that do happen in deployed systems.

TABLE 28.3 Ensuring Dependability

	Physical Faults	Development Faults
Fault Avoidance	Minimize by <i>Analysis</i>	Prevent by <i>Rigor Development Errors</i>
Fault Alleviation	<i>Selectivity the Incidence of Faults</i>	Remove by <i>Verification</i>
Fault Tolerance	<i>Ensure by Redundancy</i>	<i>Testing</i>

Note: Both physical and development fault handling may be present but any deficiency revealed is a development defect.

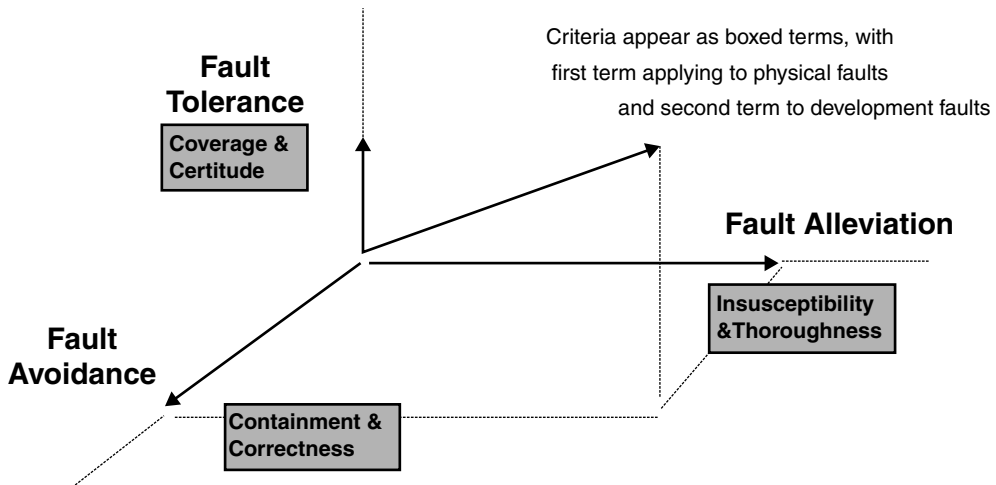


FIGURE 28.3 Dependability.

28.1.4 Fault Tolerance Options

System reliability requirements derive from the function criticality level and maximum exposure time. A **flight-critical** function is one whose loss might result in the loss of the aircraft itself, and possibly the persons on-board as well. In the latter case, the system is termed **safety-critical**. Here, a distinction can be made between a civil transport, where flight-critical implies safety-critical, and a combat aircraft. The latter admits to the possibility of the crew ejecting from an unflyable aircraft, so its system reliability requirements may be lower. A **mission-critical** function is one whose loss would result in the compromising or aborting of an associated mission. For avionics systems, a higher cost usually associates with the loss of an aircraft than with the abort of a mission (an antimissile mission to repel a nuclear weapon could be an exception). Thus, a full-time flight-critical system would normally pose much more demanding reliability requirements than a *flight-phase* mission-critical system. Next, the system reliability requirements coupled with the marginal reliabilities of system components determine the level of redundancy to ensure fault survivability, i.e., the minimum number of faults that must survive. To ensure meeting reliability requirements then, the evolution of a fault-tolerant design must be based on an interplay between design configuration commitments and substantiating analyses.

For civil transport aircraft, the level of redundancy for a flight-phase critical function like automatic all-weather landing is typically **single fail-operational**, meaning that the system should remain operable after any potential single elemental failure. Alternatively, a full-time critical function like fly-by-wire primary flight controls is typically **double fail-operational**. It should be noted that a function that is not flight-critical itself can have failure modes that threaten safety of flight. In the case of active controls to alleviate structural loads due to gusts or maneuvering, the function would not be critical where the purpose is merely to reduce

structural fatigue effects. If the associated flight control surfaces have the authority during a hardover or runaway fault case to cause structural damage, then such a failure mode is safety-critical. In such cases, the failure modes must be designed to be **fail-passive**, which precludes any active mode failure effect like a hardover control surface. A failure mode that exhibits active behavior can still be **failsafe**, however, if the rate and severity of the active effects are well within the flight crews capability to manage safely.

28.1.5 Flight Systems Evolution

Beginning in the 1970s, NASA's F-8 digital fly-by-wire (DFBW) flight research program investigated the replacement of the mechanical primary flight control systems with electronic computers and electrical signal paths. The goal was to explore the implementation technology and establish the practicality of replacing mechanical linkages to the control surfaces with electrical links, thereby yielding significant weight and maintenance benefits. The F-8 DFBW architecture relied on bit-wise exact consensus of the outputs of redundant computers for fault detection and isolation [Lala et al., 1994].

The Boeing 747, Lockheed L-1011, and Douglas DC-10 utilized various implementations to provide the autoland functions which required a probability of failure of $<10^{-9}$ during the landing. The 747 used triply redundant analog computers. The L-1011 used digital computers in a dual-dual architecture, and the DC-10 used two identical channels, each consisting of dual-redundant fail-disconnect analog computers for each axis.

Since that time, the Airbus A-320 uses a full-time DFBW flight control system. It uses software design diversity to protect against common-mode failures. The Boeing 777 flight control computer architecture uses a 3 by 3 matrix of 9 processors of 3 different types. Multiversion software is also used.

28.1.6 Design Approach

It is virtually impossible to design a complex avionics system that will tolerate all possible faults. Faults can include both permanent and transient faults, hardware and software faults, and they can occur singularly or concurrently. Timing faults directly trace to the requirement of real-time response within a few milliseconds, and may be attributable to both hardware and software contributions to data latency, as well as incorrect data.

The implementation of fault tolerance entails increased system overhead, complexity, and validation challenges. The overhead, which is essentially increased system resources and associated management activities, lies in added hardware, communications, and computational demands. The expanded complexity derives from more intricate connectivity and dependencies among both hardware and software elements; the greater number of system states that may be assumed; and appreciably more involved logic to manage the system. Validation challenges are posed by the need to identify, assess, and confirm the capability to sustain system functionality under a broad range of potential fault cases. Hence, the design approach taken for fault-tolerant avionics must attain a balance between the costs incurred in implementing fault tolerance and the degree of dependability realized.

Design approach encompasses both system concepts, development methodology, and fault tolerance elements. The system concepts derive largely from the basic fault-tolerance options introduced in Section 28.1.4, with emphasis on judicious combinations of features that are adapted to given application attributes. The development methodology reduces to mutually supportive assurance-driven methods that propagate consistency, enforce accountability, and exact high levels of certitude as to system dependability. Fault tolerance design elements tend to unify the design concepts and methods in the sense of providing an orderly pattern of system organization and evolution. These fault tolerance elements, which in general should appear in some form in any fault-tolerant system, are

- Error Detection — recognition of the incidence of a fault
- Damage Assessment — diagnosis of the locus of a fault
- Fault Containment — restriction of the scope of effects of a fault

- Error Recovery — restoration of a restartable error-free state
- Service Continuation — sustained delivery of system services
- Fault Treatment — repair of fault.

A fundamental design parameter that spans these elements and constrains their mechanization is that of the granularity of fault handling. Basically, the detection, isolation, and recovery from a fault should occur at the same level of modularity to achieve a balanced and coherent design. In general, it is not beneficial or justified to discriminate or contain a fault at a level lower than that of the associated fault-handling boundary. There may be exceptions, however, especially in the case of fault detection, where a finer degree of granularity may be employed to take advantage of built-in test features or to reduce fault latency.

Depending on the basis for its instigation, fault containment may involve the inhibition of damage propagation of a physical fault and/or the suppression of an erroneous computation. Physical fault containment has to be designed into the hardware, and software error-state containment has to be designed into the applications software. In most cases, an erroneous software state must be corrected because of applications program discrepancies introduced during the delay in detecting a fault. This error recovery may entail resetting certain data object values and backtracking a control flow path in an operable processor. At this point, the readiness of the underlying architecture, including the coordination of operable components, must be ensured by the infrastructure. Typically, this activity relies heavily on system management software for fault tolerance. Service continuation, then, begins with the establishment of a suitable applications state for program restart. In an avionics system, this sequence of fault tolerance activities must take place rather quickly because of real-time functional demands. Accordingly, an absolute time budget must be defined, with tolerances for worst-case performance, for responsive service continuation.

28.2 System Level Fault Tolerance

28.2.1 General Mechanization

As discussed in Section 28.1.2, system failure is the loss of system services or expected functionality. In the absence of fault tolerance, a system may fail after just a single crucial fault. This kind of system, which in effect is zero fail-operational, would be permissible for non-critical functions. [Figure 28.2](#), moreover, characterizes this kind of system in that continued service depends on spontaneous remission of an active fault or a fault whose consequences are not serious enough to yield a system failure.

Where the likelihood of continued service must be high, redundancy can be incorporated to ensure system operability in the presence of any permanent fault(s). Such fault-tolerant systems incorporate an additional fault status state, namely that of recovery, as shown in [Figure 28.4](#). Here, system failure occurs only after the exhaustion of spares or an unhandled severe fault. The aforementioned level of redundancy can render it extremely unlikely that the spares will be exhausted as a result of hardware faults alone. An unhandled fault could occur only as a consequence of a design error, like the commission of a generic error wherein the presence of a fault would not even be detected.

This section assumes a system-level perspective, and undertakes to examine fault-tolerant system architectures and examples thereof. Still, these examples embody and illuminate general system-level principles of fault tolerance. Particular prominence is directed toward flight control systems, for they have motivated and pioneered much of the fault tolerance technology as applied to digital flight systems. In the past, such systems have been functionally dedicated, thereby providing a considerable safeguard against malfunction due to extraneous causes. With the increasing prevalence of integrated avionics, however, the degree of function separation is irretrievably reduced. This is actually not altogether detrimental, more avionics functions than ever are critical, and a number of benefits accrue from integrated processing. Furthermore, the system developer can exercise prerogatives that afford safeguards against extraneous faults.

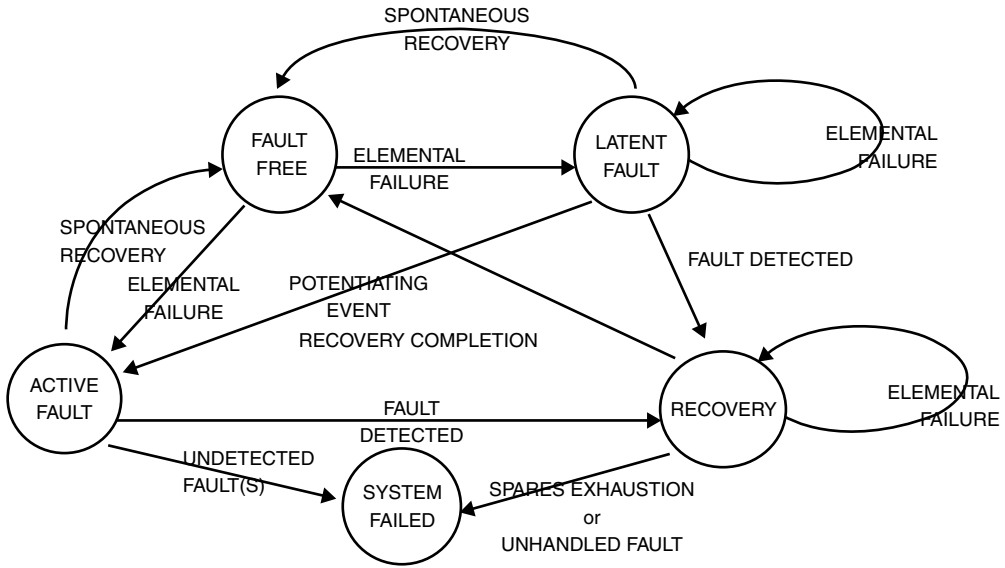


FIGURE 28.4 Hardware states (with corrective action).

28.2.2 Redundancy Options

Fault tolerance is usually based on some form of redundancy to extend system reliability through the invocation of alternative resources. The redundancy may be in hardware, software, time, or combinations thereof. There are three basic types of redundancy in hardware and software: static, dynamic, and hybrid. Static redundancy masks faults by taking a majority of the results from replicated tasks. Dynamic redundancy takes a two-step procedure for detection of, and recovery from faults. Hybrid redundancy is a combination of static and dynamic redundancy [Shin and Hagbae, 1994].

In general, much of this redundancy resides in additional hardware components. The addition of components reduces the mean-time-between-maintenance actions, because there are more electronics that can, and at some point will, fail. Since multiple, distinct faults can occur in fault-tolerant systems, there are many additional failure modes that have to be evaluated in establishing the airworthiness of the total system. Weight, power consumption, cooling, etc. are other penalties for component redundancy. Other forms of redundancy also present system management overhead demands, like computational capacity to perform software-implemented fault tolerance tasks. Like all design undertakings, the realization of fault tolerance presents trade-offs and the necessity for design optimization. Ultimately, a balanced, minimal, and validatable design must be sought that demonstrably provides the safeguards and fault survival margins appropriate to the subject application.

A broad range of redundancy implementation options exist to mechanize desired levels and types of fault tolerance. Figure 28.5 presents a taxonomy of redundancy options that may be invoked in appropriate combinations to yield an encompassing fault tolerance architecture.

This taxonomy indicates the broad range of redundancy possibilities that may be invoked in system design. Although most of these options are exemplified or described in later paragraphs, it may be noted briefly that redundancy is characterized by its category, mode, coordination, and composition aspects. Architectural commitments have to be made in each aspect. Thus, a classical system might, for example, employ fault masking using replicated hardware modules that operate synchronously. At a lower level, the associated data buses might use redundancy encoding for error detection and correction. To safeguard against a generic design error, a backup capability might be added using a dissimilar redundancy scheme.

Before considering system architectures per se, however, key elements of Figure 28.5 need to be described and exemplified. Certain of these redundancy implementation options are basic to formulating a fault-tolerant

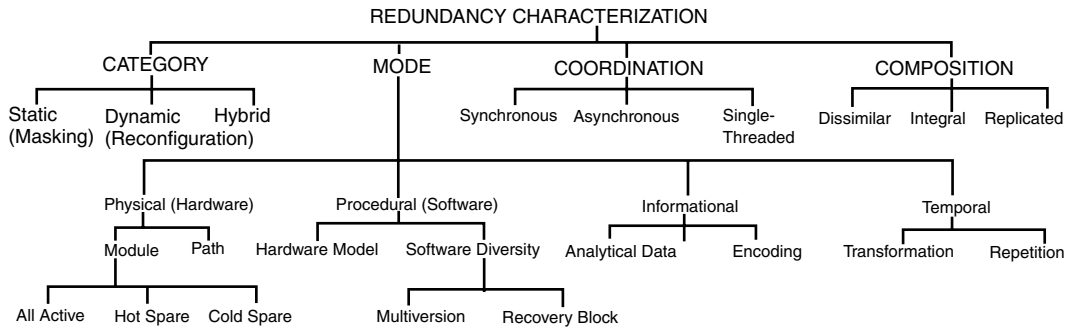


FIGURE 28.5 Redundancy classification.

TABLE 28.4 Fault Avoidance Techniques and Tools

Technique
Use of mature and formally verified components
Conformance to standards
Formal methods
Design automation
Integrated formal methods and VHDL design methodology
Simplifying abstractions
Performance common-mode failure avoidance
Software and hardware engineering practice
Design diversity

architecture, so their essential trade-offs need to be defined and explored. In particular, the elements of masking vs. reconfiguration, active vs. standby spares, and replicated vs. dissimilar redundancy are reviewed here.

No unifying theory has been developed that can treat CMFs the same way the Byzantine resilience (BR) treats random hardware or physical operational faults. Three techniques, fault-avoidance, fault-removal, and fault-tolerance are the tools available to design a system tolerant of CMFs. The most cost effective phase of the total design and development process for reducing the likelihood of CMFs is the earliest part of the program. Table 28.4 presents fault avoidance techniques and tools that are being used [Lala and Harper, 1994].

Common-mode fault removal techniques and tools include design reviews, simulation, testing, fault injection, and a rigorous quality control program. Common-mode fault tolerance requires error detection and recovery. It is necessary to corroborate the error information across redundant channels to ascertain which recovery mechanism (i.e., physical fault recovery, or common-mode failure recovery) to use. Recovery from CMF in real time requires that the state of the system be restored to a previously known correct point from which the computational activity can resume [Lala and Harper, 1994].

28.2.3 Architectural Categories

As indicated in Figure 28.5, the three categories of fault-tolerant architectures are masking, reconfiguration, and hybrid.

28.2.3.1 Fault Masking

The masking approach is classical per von Neumann's triple modular redundancy (TMR) concept, which has been generalized for arbitrary levels of redundancy. The TMR concept centers on a voter that, within a spares exhaustion constraint, precludes a faulted signal from proceeding along a signal path. The approach is passive in that no reconfiguration is required to prevent the propagation of an erroneous state or to isolate a fault.

Modular avionics systems consisting of multiple identical modules and a voter require a trade-off of reliability and safety. A “module” is not constrained to be a hardware module; a module represents an entity capable of producing an output. When safety is considered along with reliability, the module design affects both safety and reliability. It is usually expected that reliability and safety should improve with added redundancy. If a module has built-in error detection capability, it is possible to increase both reliability and safety with the addition of one module providing input to a voter. If no error detection capability exists at the module level, at least two additional modules are required to improve both reliability and safety. An error control arbitration strategy is the function implemented by the voter to decide what is the correct output, and when the errors in the module outputs are excessive so that the correct output cannot be determined, the voter may put out an unsafe signal. Reliability and safety of an n-module safe modular redundant (nSMR) depend on the individual module reliability and on the particular arbitration strategy used. No single arbitration strategy is optimal for improving both reliability and safety. Reliability is defined as the probability the voter’s data output is correct and the voter does not assert the unsafe signal. Safety = Reliability plus the probability the voter asserts the unsafe signal. As system reliability and safety are interrelated, increasing system reliability may result in a decrease in system safety, and vice versa [Vaidya and Pradhan, 1993].

Voters that use bit-for-bit comparison have been employed when faults consist of arbitrary behavior on the part of failed components, even to the extreme of displaying seemingly intelligent malicious behavior [Lala and Harper, 1994]. Such faults have been called Byzantine faults. Requirements levied on an architecture tolerant of Byzantine faults (referred to as Byzantine-resilient [BR]) comprise a lower bound on the number of fault containment regions, their connectivity, their synchrony, and the utilization of certain simple information exchange protocols. No *a priori* assumptions about component behavior are required when using bit-for-bit comparison. The dominant contributor to failure of correctly designed BR system architecture is the common-mode failure.

Fault effects must be masked until recovery measures can be taken. A redundant system must be managed to continue correct operation in the presence of a fault. One approach is to partition the redundant elements into individual fault containment regions (FCRs). An FCR is a collection of components that operates correctly regardless of any arbitrary logical or electrical fault outside the region. A fault containment boundary requires the hardware components be provided with independent power and clock sources. Interfaces between FCRs must be electrically isolated. Tolerance to such physical damage as a weapons hit necessitates a physical separation of FCRs such as different avionics bays. In flight control systems, a channel may be a natural FCR. Fault effects manifested as erroneous data can propagate across FCR boundaries. Therefore, the system must provide error containment as well. This is done using voters at various points in the processing including voting on redundant inputs, voting the result of control law computations, and voting at the input to the actuator. Masking faults and errors provides correct operation of the system with the need for immediate damage assessment, fault isolation, and system reconfiguration [Lala and Harper, 1994].

28.2.3.2 Reconfiguration

Hardware interlocks provide the first level of defense prior to reconfiguration or the use of the remaining non-faulty channels. In a triplex or higher redundancy system, the majority of channels can disable the output of a failed channel. Prior to taking this action, the system will determine whether the failure is permanent or transient.

Once the system determines a fault is permanent or persistent, the next step is to ascertain what functions are required for the remainder of the mission and whether the system needs to invoke damage assessment, fault isolation, and reconfiguration of the remaining system assets. The designer of a system required for long-duration missions may undertake to implement a design having reconfiguration capability.

28.2.3.3 Hybrid Fault Tolerance

Hybrid fault tolerance uses hybrid redundancy, which is a combination of static and dynamic redundancy, i.e., masking, detection, and recovery that may involve reconfiguration. A system using hybrid redundancy will have N-active redundant modules, as well as spare (S) modules. A disagreement detector detects if

the output of any of the active modules is different from the voter output. If a module output disagrees with the voter, the switching circuit replaces the failed module with a spare. A hybrid (N,S) system cannot have more than $(N-1)/2$ failed modules at a time in the core, or the system will incorrectly switch out the good module when two out of three have failed.

28.2.3.4 Hybrid Fault Tolerance

Hybrid fault tolerance employs a combination of masking and reconfiguration, as noted in Section 28.2.3. The intent is to draw on strengths of both approaches to achieve superior fault tolerance. Masking precludes an erroneous state from affecting system operation and thus obviating the need for error recovery. Reconfiguration removes faulted inputs to the voter so that multiple faults cannot defeat the voter. Masking and reconfiguration actions are typically implemented in a voter-comparator mechanism, which is discussed in Section 28.3.1.

Figure 28.6 depicts a hybrid TMR arrangement with a standby spare channel to yield double fail-operational capability. Upon the first active channel failure, it is switched out of the voter-input configuration, and the standby channel is switched in. Upon a second channel failure, the discrepant input to the voter is switched out. Only two inputs remain then, so a succeeding (third) channel failure can be detected but not properly be identified by the voter per se. With a voter that selects the lower of two remaining signals, and hence precludes a hardover output, a persistent miscomparison results in a fail-passive loss of system function.

An alternative double-fail operational configuration would forego the standby channel switching and simply employ a quadruplex voter. This architecture is actually rather prevalent in dedicated flight-critical systems like fly-by-wire (FBW) flight control systems. This architecture still employs reconfiguration to remove faulty inputs to the voter.

The fault tolerance design elements described in Section 28.1.6 are reflected in the fault-tolerant architecture in Figure 28.6 by way of annotations. For example, error detection is provided by the comparators; damage assessment is then accomplished by the reconfiguration logic using the various comparator states. Fault containment and service continuation are both realized through the voter, which also obviates the need for error recovery. Last, fault treatment is accomplished by the faulty path switching prompted by the reconfiguration logic. Thus, this simple example illustrates at a high level how the various aspects of fault tolerance can be incorporated into an integrated design.

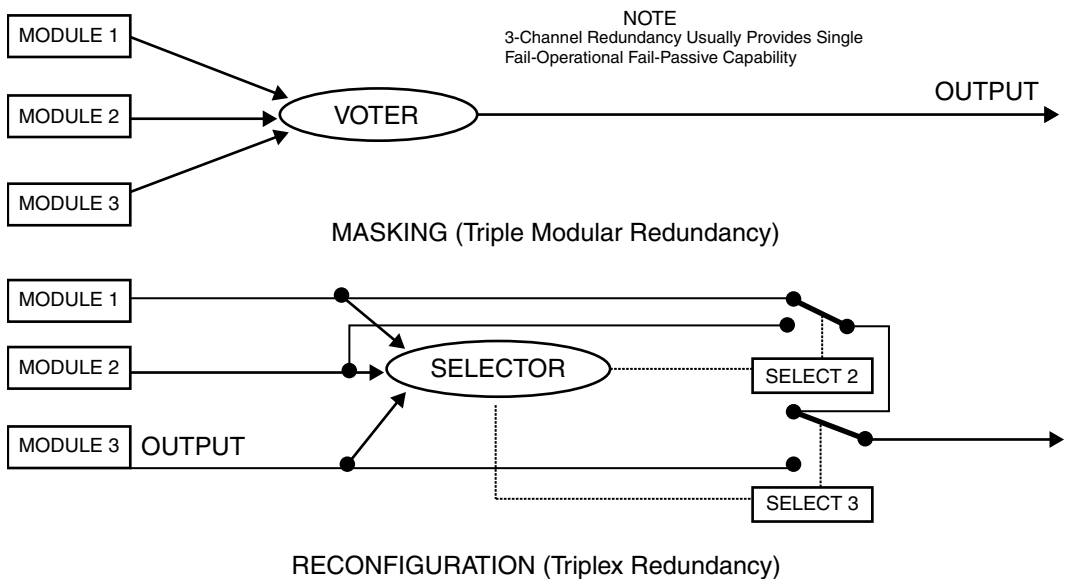


FIGURE 28.6 Masking vs. Reconfiguration.

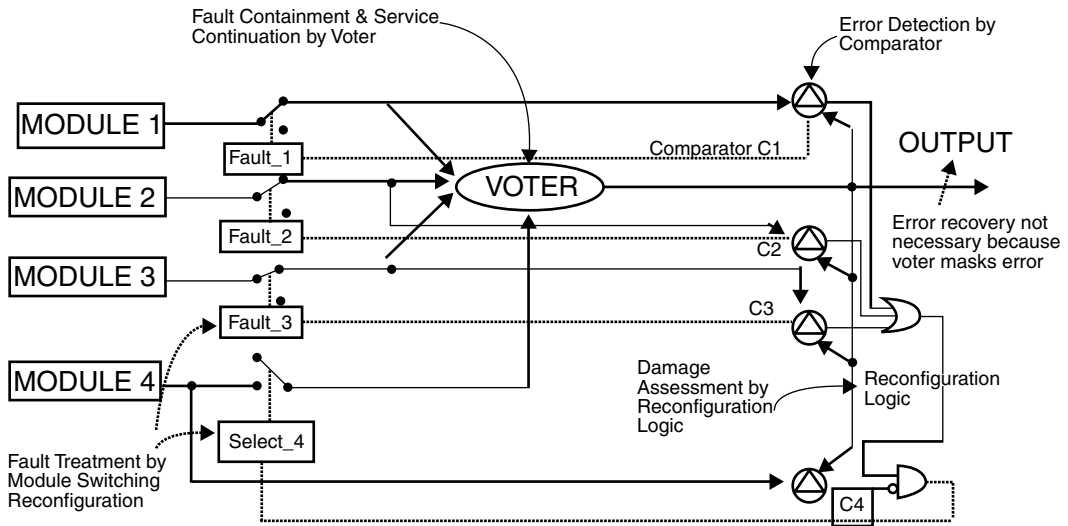


FIGURE 28.7 Hybrid TMR arrangement.

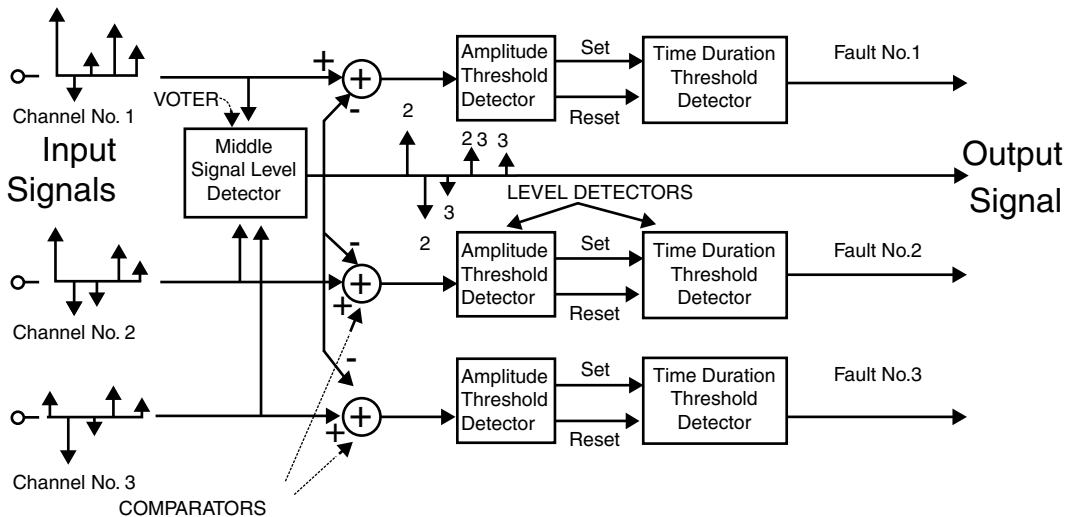


FIGURE 28.8 Triplex voter-comparator.

28.2.4 Integrated Mission Avionics

In military applications, redundant installations in some form will be made on opposite sides of the aircraft to avoid loss of functionality from battle damage to a single installation. Vulnerability to physical damage also exists in the integrated rack installations being used on commercial aircraft. Designers must take these vulnerabilities into account in the design of a fault-tolerant system.

28.2.5 System Self Tests

Avionics system reliability analyses are conditional on assumptions of system readiness at dispatch. For lower-criticality systems, certain reductions in redundancy may sometimes be tolerable at dispatch. For

full-time flight-critical systems, however, a fully operable system with all redundancy intact is generally assumed in a system reliability prediction. This assumption places appreciable demands on system preflight self test in terms of coverage and confidence values. Such a test is typically an end-to-end test that exercises all elements in a phased manner that would not be possible during flight. The fault tolerance provisions demand particular emphasis. For example, such testing deliberately seeks to force seldom-used comparator trips to ensure the absence of latent faults, like passive hardware failures. Analysis of associated testing schemes and their scope of coverage is necessarily an ongoing design analysis task during development. These schemes must also include appropriate logic interlocks to ensure safe execution of the preflight test, e.g., a weight-on-wheels interlock to preclude testing except on the ground. Fortunately, the programming of system self-tests can be accomplished in a relatively complete and high-fidelity manner.

Because of the discrete-time nature of digital systems, all capacity is not used for application functions. Hence, periodic self tests are possible for digital components like processors during flight. Also, the processors can periodically examine the health status of other system components. Such tests provide a self-monitoring that can reveal the presence of a discrepancy before error states are introduced or exceed detection thresholds. The lead time afforded by self tests can be substantial because steady flight may not simulate comparator trips due to low-amplitude signals. Moreover, the longer a fault remains latent, the greater the possibility that a second fault can occur. Hence, periodic self-tests can significantly enhance system reliability and safety by reducing exposure to coincident multiple fault manifestations.

Self-monitoring may be employed at still lower levels, but there is a trade-off as to the granularity of fault detection. This trade-off keys on fault detection/recovery response and on the level of fault containment selected. In general, fault containment delineates the granularity of fault detection unless recovery response times dictate faster fault detection that is best achieved at lower levels.

28.3 Hardware-Implemented Fault Tolerance (Fault-Tolerant Hardware Design Principles)

28.3.1 Voter Comparators

Voter comparators are very widely used in fault-tolerant avionics systems, and they are generally vital to the integrity and safety of the associated systems. Because of the crucial role of voter comparators, special care must be exercised in their development. These dynamic system elements, which can be implemented in software as well as hardware, are not as simple as they might seem. In particular, device integrity and threshold parameter settings can be problematic.

Certain basic elements and concerns apply over the range of voter-comparator variants. A conceptual view of a triplex voter-comparator is depicted in [Figure 28.7](#). The voter here is taken to be a middle signal selector, which means that the intermediate level of three inputs is selected as the output. The voter section precedes the comparators because the output of the voter is an input to each comparator. Basically, the voter output is considered the standard of correctness, and any input signal that persists in varying too much from the standard is adjudged to be erroneous.

In [Figure 28.7](#), the respective inputs to each of the signal paths is an amplitude-modulated pulse train, as is normal in digital processing. Each iteration of the voter is a separate selection, so each voter output is apt to derive from any input path. This is seen in [Figure 28.8](#), where the output pulse train components are numbered per the input path selected at each point in time. At each increment of time, the voter output is applied to each of the comparators, and the difference with each input signal is fed to a corresponding amplitude threshold detector. The amplitude threshold is set so that accumulated tolerances are not apt to trip the detector. As shown here, the amplitude detector issues a set output when an excessive difference is first observed. When the difference falls back within the threshold, a reset output is issued.

Because transient effects may produce short-term amplitude detector trips, a timing threshold is applied to the output of each amplitude detector. Typically, a given number of consecutive out-of-tolerance amplitude threshold trips are necessary to declare a faulty signal. Hence, a time duration threshold detector begins a count whenever a set signal is received, and in the absence of further inputs, increments the count for each sample interval thereafter. If a given cycle count is exceeded, an erroneous state is declared and a fault logic signal is set for the affected channels. Otherwise, the count is returned to zero when a reset signal is received.

The setting of both the timing and amplitude thresholds is of crucial importance because of the trade-off between nuisance fault logic trips and slow response to actual faults. Nuisance trips erode user confidence in a system, their unwarranted trips can potentially cause resource depletion. On the other hand, a belated fault response may permit an unsafe condition or catastrophic event to occur. The allowable time to recover from a given type of fault, which is application-dependent, is the key to setting the thresholds properly. The degree of channel synchronization and data skewing also affect the threshold settings, because they must accommodate any looseness. The trade-off can become quite problematic where fast fault recovery is required.

Because the integrity and functionality of the system is at stake, the detailed design of a voter comparator must be subject to careful assessment at all stages of development. In the case of a hardware-implemented device, its fault detection aspects must be thoroughly examined. Passive failures in circuitry that is not normally used are the main concern. Built-in test, self-monitoring, or fail-hard symptoms are customary approaches to device integrity. In the case of software-implemented voter comparators, their dependability can be reinforced through formal proof methods and in-service verified code.

28.3.2 Watchdog Timers

Watchdog timers can be used to catch both hardware and software wandering into undesirable states [Lala and Harper, 1994]. Timing checks are a form of assertion checking. This kind of check is useful because many software and hardware errors are manifested in excessive time taken for some operation. In synchronous data flow architectures, data are to arrive at a specific time. Data transmission errors of this type can be detected using a timer.

28.4 Software-Implemented Fault Tolerance—State Consistency

Software performs a critical role in digital systems. The term “software implemented fault tolerance” as used in this chapter is used in the broader sense indicating the role software plays in the implementation of fault tolerance, and not as a reference to the SRI International project performed for NASA in the late 1970s and referred to as SIFT.

28.4.1 Error Detection

Software plays a major role in error detection. Error detection at the system level should be based on the specification of system behavior. The outputs of the system should be checked to assure that the outputs conform to the specification. These checks should be independent of the system. Since they are implemented in software, the checks require access to the information to be checked, and therefore may have the potential of corrupting that information. Hence, the independence between a system and its check cannot be absolute. The provision of ideal checks for error detection is rarely practical, and most systems employ checks for acceptability [Anderson and Lee, 1981].

Deciding where to employ system error detection is not a straightforward matter. Early checks should not be regarded as substitute for last-moment checks. An early check will of necessity be based on a knowledge of the internal workings of the system and hence will lack independence from the system. An early check could detect an error at the earliest possible stages and hence minimize the spread of damage.

A last moment check ensures that none of the output of the system remains unchecked. Therefore, both last-moment and early checks should be provided in a system [Anderson and Lee, 1981].

In order to detect software faults, it is necessary that the redundant versions of the software be independent of each other, that is, of diverse design [Avizenis and Kelley, 1982] (See Section 28.5).

28.4.1.1 Replication Checks

If design faults are expected, replication must be provided using versions of the system with different designs. Replication checks compare the two sets of results produced as outputs of the replicated modules. The replication check raises an error flag and initiates the start of other processes to determine which component or channel is in error [Anderson and Lee, 1981].

28.4.1.2 Timing Checks

Timing checks are used to reveal the presence of faults in a system, but not their absence [Anderson and Lee, 1981] In synchronous hard real-time systems, messages containing data are transmitted over data buses at a specific schedule. Failure to receive a message at the scheduled time is an error. The error could be caused by faults in a sensor, data bus, etc. In this case, if the data were critical, a method of tolerating the fault may be to use a forward state extrapolation.

28.4.1.3 Reversal Check (Analytical Redundancy)

A reversal check takes the outputs from a system and calculates what the inputs should have been to produce that output. The calculated inputs can then be compared with the actual inputs to check whether there is an error. Systems providing mathematical functions often lend themselves to reversal checks [Anderson and Lee, 1981].

Analytic redundancy using either of two general error detection methods, multiple model (MM) or generalized likelihood ratio (GLR), is a form of reversal check. Both methods make use of a model of the system represented by Kalman filters. The MM attempts to calculate a measure of how well each of the Kalman filters is tracking by looking at the prediction errors. Real systems possess nonlinearity and the model assumes a linear system. The issue is whether the tracking error from the extended Kalman filter corresponds to the linearized model “closest to” the true, nonlinear system and is markedly smaller than the errors from the filters based on “more distant” models. Actuator and sensor failures can be modeled in different ways using this methodology [Willisky, 1980].

The Generalized Likelihood Ratio (GLR) uses a formulation similar to that for MM, but different enough that the structure of the solution is quite different. The starting point for GLR is a model describing normal operation of the observed signals or of the system from which they come. Since GLR is directly aimed at detecting abrupt changes, its applications are restricted to problems involving such changes, such as failure detection. GLR, in contrast to MM, requires a single Kalman filter. Any detectable failure will exhibit a systematic deviation between what is observed and what is predicted to be observed. If the effect of the parametric failure is “close enough” to that of the additive one, the system will work.

Underlying both the GLR and MM methods is the issue of using system redundancy to generate comparison signals that can be used for the detection of failures. The fundamental idea involved in finding comparison signals is to use system redundancy, i.e., known relationships among measured variables to generate signals that are small under normal operation and which display predictable patterns when particular anomalies develop. All failure detection is based on analytical relationships between sensed variables, including voting methods, which assume that sensors measure precisely the same variable. The trade-off using analytical relationships is that we can reduce hardware redundancy and maintain the same level of fail-operability. In addition, analytical redundancy allows extracting more information from the data, permitting detection of subtle changes in system component characteristics. On the other hand, the use of this information can cause problems if there are large uncertainties in the parameters specifying the analytical relationships [Willisky, 1980].

The second part of a failure detection algorithm is the decision rule that uses the available comparison signals to make decisions on the interruption of normal operation by the declaration of failures. One advantage of these methods is that the decision rule — maximize and compare to a threshold — are simple, while the main disadvantage is that the rule does not explicitly reflect the desired trade-offs. The Bayesian Sequential Decision approach, in which an algorithm for the calculation of the approximate Bayes decision, has exactly the opposite properties, i.e., it allows for a direct incorporation of performance trade-offs, but is extremely complex. The Bayes Sequential Decision Problem is to choose a stopping rule and terminal decision rule to minimize the total expected cost, and the expected cost that is accrued before stopping [Willisky, 1980].

28.4.1.4 Coding Checks

Coding checks are based on redundancy in the representation of an object in use in a system. Within an object, redundant data are maintained in some fixed relationship with the (nonredundant) data representing the value of the object. Parity checks are a well-known example of a coding check, as are error detection and correction codes such as the Hamming, cyclic redundancy check, and arithmetic codes [Anderson and Lee, 1981].

28.4.1.5 Reasonableness Checks

These checks are based on knowledge of the minimum and maximum values of input data, as well as the limits on rate of change of input data. These checks are based on knowledge of the physical operation of sensors, and employ models of this operation.

28.4.1.6 Structural Checks

Two forms of checks can be applied to the data structures in a computing system. Checks on the semantic integrity of the data will be concerned with the consistency of the information contained in a data structure. Checks on the structural integrity will be concerned with whether the structure itself is consistent. For example, external data from subsystems is transmitted from digital data buses such as MIL-STD-1553, ARINC 429 or ARINC 629. The contents of a message (number of words in the message, contents of each word) from a subsystem are stored and the incoming data checked for consistency.

28.4.1.7 Diagnostic Checks

Diagnostic checks create inputs to the hardware elements of a system, which should produce a known output. These checks are rarely used as the primary error detection measure. They are normally run at startup, and may be initiated by an operator as part of a built-in test. They may also run continuously in a background mode when the processor might be idle. Diagnostic checks are also run to isolate certain faults.

28.4.2 Damage Confinement and Assessment

When an error has been discovered, it is necessary to determine the extent of the damage done by the fault before error recovery can be accomplished. Assessing the extent of the damage is usually related to the structure of the system. Assuming timely detection of errors, the assessment of damage is usually determined to be limited to the current computation or process. The state is assumed consistent on entry. An error detection test is performed before exiting the current computation. Any errors detected are assumed to be caused by faults in the current computation.

28.4.3 Error Recovery

After the extent of the damage has been determined, it is important to restore the system to a consistent state. There are two primary approaches — backward and forward error recovery. In backward error recovery, the system is returned to a previous consistent state. The current computation can then be

retried with existing components (retry)* with alternate components (reconfigure), or it can be ignored (skip frame).** The use of backward recovery implies the ability to save and restore the state. Backward error recovery is independent of damage assessment.

Forward error recovery attempts to continue the current computation by restoring the system to a consistent state, compensating for the inconsistencies found in the current state. Forward error recovery implies detailed knowledge of the extent of the damage done, and a strategy for repairing the inconsistencies. Forward error recovery is more difficult to implement than backward error recovery [Hitt et al., 1984].

28.4.4 Fault Treatment

Once the system has recovered from an error, it may be desirable to isolate and/or correct the component that caused the error. Fault treatment is not always necessary because of the transient nature of some faults or because the detection and recovery procedures are sufficient to cope with other recurring errors. For permanent faults, fault treatment becomes important because the masking of permanent faults reduces the ability of the system to deal with subsequent faults. Some fault-tolerant software techniques attempt to isolate faults to the current computation by timely error detection. Having isolated the fault, fault treatment can be done by reconfiguring the computation to use alternate forms of the computation to allow for continued service. (This can be done serially, as in recovery blocks, or in parallel, as in N-Version programming.) The assumption is that the damage due to faults is properly encapsulated to the current computation and that error detection itself is faultless (i.e., detects all errors and causes none of its own) [Hitt et al., 1984].

28.4.5 Distributed Fault Tolerance

Multiprocessing architectures consisting of computing resources interconnected by external data buses should be designed as a distributed fault-tolerant system. The computing resources may be installed in an enclosure using a parallel backplane bus to implement multiprocessing within the enclosure. Each enclosure can be considered a virtual node in the overall network. A network operating system, coupled with the data buses and their protocol, completes the fault-tolerant distributed system. The architecture can be asynchronous, loosely synchronous, or tightly synchronous. Maintaining consistency of data across redundant channels of asynchronous systems is difficult [Papadopoulos, 1985].

28.5 Software Fault Tolerance

Software faults, considered design faults, may be created during the requirements development, specification creation, software architecture design, code creation, and code integration. While many faults may be found and removed during system integration and testing, it is virtually impossible to eliminate all possible software design faults. Consequently, software fault tolerance is used. [Table 28.5](#) lists the major fault-tolerant software techniques in use today.

The two main methods that have been used to provide software fault tolerance are N-version software and recovery blocks.

28.5.1 Multiversion Software

Multiversion software is any fault-tolerant software technique in which two or more alternate versions are implemented, executed, and the results compared using some form of a decision algorithm. The goal is to develop these alternate versions such that software faults that may exist in one version are not

*This is only useful for transient timing on hardware faults.

**For example, in a real-time system no processing for the current computation is accomplished in the current frame, sometimes called “skip frame.”

TABLE 28.5 Categorization of Fault-Tolerant Software Techniques

Multiversion Software
N-Version Program
Cranfield Algorithm for Fault-Tolerance (CRAFT) Food Taster
Distinct and Dissimilar Software
Recovery Blocks
Deadline Mechanism
Dissimilar Backup Software
Exception Handlers
Hardened Kernel
Robust Data Structures and Audit Routines
Run Time Assertions ^a
Hybrid Multiversion Software and Recovery Block Techniques
Tandem
Consensus Recovery Blocks

^a Not a complete fault-tolerant software technique as it only detects errors.

Source: From Hitt, E. et al., Study of Fault-Tolerant Software Technology, NASA CR 172385.

contained in the other version(s) and the decision algorithm determines the correct value from among the alternate versions. Whatever means are used to produce the alternate versions, the common goal is to have distinct versions of software such that the probability of faults occurring simultaneously is small and that faults are distinguishable when the results of executing the multiversions are compared with each other.

The comparison function executes as a decision algorithm once it has received results from each version. The decision algorithm selects an answer or signals that it cannot determine an answer. This decision algorithm and the development of the alternate versions constitute the primary error detection method. Damage assessment assumes the damage is limited to the encapsulation of the individual software versions. Faulted software components are masked so that faults are confined within the module in which they occur. Fault recovery of the faulted component may or may not be attempted.

N-versions of a program are independently created by N-software engineering teams working from a (usually) common specification. Each version executes independently of the other versions. Each version must have access to an identical set of input values and the outputs are compared by an executive which selects the result used. The choice of an exact or inexact voting check algorithm is influenced by the criticality of the function and the timing associated with the voting.

28.5.2 Recovery Blocks

The second major technique shown in Table 28.5 is the recovery block and its subcategories — deadline mechanism and dissimilar backup software. The recovery block technique recognizes the probability that residual faults may exist in software. Rather than develop independent redundant modules, this technique relies on the use of a software module which executes an acceptance test on the output of a primary module. The acceptance test raises an exception if the state of the system is not acceptable. The next step is to assess the damage and recover from the fault. Given that a design fault in the primary module could have caused arbitrary damage to the system state, and that the exact time at which errors were generated cannot be identified, the most suitable prior state for restoration is the state that existed just before the primary module was entered [Anderson and Lee, 1981].

28.5.3 Trade-Offs

Coverage of a large number of faults has an associated overhead in terms of redundancy, and the processing associated with the error detection. The designer may use modeling and simulation to determine the amount of redundancy required to implement the fault tolerance vs. the probability and impact of the different types of faults. If a fault has minimal or no impact on safety, or mission completion,

investing in redundancy to handle that fault may not be effective, even if the probability of the fault occurring is significant.

28.6 Summary

Fault-tolerant systems must be used whenever a failure can result in loss of life, or loss of a high-value asset. Physical failures of hardware are decreasing whereas design faults are virtually impossible to completely eliminate.

28.6.1 Design Analyses

In applying fault-tolerance to a complex system, there is a danger that the new mechanisms may introduce additional sources of failure due to design and implementation errors. It is important, therefore, that the new mechanisms be introduced in a way that preserves the integrity of a design, with minimum added complexity. The designer must use modeling and simulation tools to assure that the design accomplishes the needed fault tolerance.

Certain design principles have been developed to simplify the process of making design decisions. Encapsulation and hierarchy offer ways to achieve simplicity and generality in the realization of particular fault-tolerance functions. Encapsulation provides:

- Organization of data and programs as uniform objects, with rigorous control of object interaction.
- Organization of sets of alternate program versions into fault-tolerant program modules (e.g. recovery blocks and N-version program sets).
- Organization of consistent sets of recovery points for multiple processes.
- Organization of communications among distributed processes as atomic (indivisible) actions.
- Organization of operating system functions into recoverable modules.

Examples of the hierarchy principle used to enhance reliability of fault-tolerance functions include:

- Organization of all software, both application and system type, into layers, with unidirectional dependencies among layers.
- Integration of service functions and fault-tolerance functions at each level.
- Use of nested recovery blocks to provide hierarchical recovery capability.
- Organization of operating system functions so that only a minimal set at the lowest level (a “kernel”) needs be exempted from fault tolerance.
- Integration of global and local data and control in distributed processors.

That portion of the operating system kernel that provides the basic mechanisms the rest of the system uses to achieve fault-tolerance should be “trusted.” This kernel should be of limited complexity so that all possible paths can be tested to assure correct operation under all logic and data conditions. This kernel need not be fault tolerant if the foregoing can be assured.

28.6.2 Safety

Safety is defined in terms of hazards and risks. A hazard is a condition, or set of conditions that can produce an accident under the right circumstances. The level of risk associated with the hazard depends on the probability that the hazard will occur, the probability of an accident taking place if the hazard does occur, and the potential consequence of the accident [Williams, 1992].

28.6.3 Validation

Validation is the process by which systems are shown through operation to satisfy the specifications. The validation process begins when the specification is complete. The difficulty of developing a precise specification that will never change has been recognized. This reality has resulted in an iterative

development and validation process. Validation requires developing test cases and executing these test cases on the hardware and software comprising the system to be delivered. The tests must cover 100% of the faults the system is designed to tolerate, and a very high percentage of possible design faults, whether hardware, software, or the interaction of the hardware and software during execution of all possible data and logical paths. Once the system has been validated, it can be put in operation. In order to minimize the need to validate a complete Operational Flight Program (OFP) every time it is modified, development methods attempt to decompose a large system into modules that are independently specified, implemented, and validated. Only those modules and their interfaces that are modified must be revalidated using this approach (see Chapter 29).

Rapid prototyping, simulation, and animation are all techniques that help validate the system. Formal methods are being used to develop and validate digital avionics systems. There are arguments both in favor of and against the use of formal methods [Rushby, 1993; Williams, 1992].

28.6.4 Conclusion

For safety-critical systems, fault tolerance must be used to tolerate design faults which are predominately software- and timing-related. It is not enough to eliminate almost all faults introduced in the later stages of a life cycle; assurance is needed that they have been eliminated, or are extremely improbable. Safety requirements for commercial aviation dictate that a failure causing loss of life must be extremely improbable, on the order of 10^{-9} per flight-hour. The designer of safety-critical fault-tolerant systems should keep current with new development in this field since both design and validation methods continue to advance in capability.

References

- Anderson, T. and Lee, P. A., 1981. *Fault Tolerance, Principles and Practices*, Prentice-Hall, London.
- Anderson, T., Ed., 1989. *Safe and Secure Computing Systems*, Blackwell Scientific, Oxford, U.K.
- Avizienis, A., 1976. Fault-Tolerant Systems, *IEEE Trans. Comput.*, C-25(12):1304-1312.
- Avizienis, A. and Kelly, J., 1982. Fault-Tolerant Multi-Version Software: Experimental Results of a Design Diversity Approach, UCLA Computer Science Department. Los Angeles, CA.
- Avizienis, A., Kopetz, H., and Laprie, J.C., Eds., 1987. *The Evolution of Fault-Tolerant Computing*, Springer-Verlag, New York.
- Best, D. W., McGahee, K. L., and Shultz, R. K.A., 1988. Fault Tolerant Avionics Multiprocessing System Architecture Supporting Concurrent Execution of Ada Tasks, Collins Government Avionics Division, AIAA 88-3908-CP.
- Gargaro, A.B. et al., 1990. Adapting Ada for Distribution and Fault Tolerance, in *Proc. 4th Int. Workshop Real-Time Ada Issues*, ACM.
- Gu, D., Rosenkrantz, D. J., and Ravi, S. S., 1994. Construction of Check Sets for Algorithm-Based Fault Tolerance, *IEEE Trans. Comput.*, 43(6): 641-650.
- Hitt, E., Webb, J., Goldberg, J., Levitt, K., Slivinski, T., Broglio, C., and Wild, C., 1984. *Study of Fault-Tolerant Software Technology*, NASA CR 172385, Langley Research Center, VA.
- Hudak, J., Suh, B.H., Siewiorek, D., and Segall, Z., 1993. Evaluation and Comparison of Fault Tolerant Software Techniques, *IEEE Trans. Reliability*.
- Lala, J. H. and Harper, R. E., 1994. Architectural Principles for Safety-Critical Real-Time Applications, *Proc. IEEE*, 82(1): 25-40.
- Lala, P. K., 1985. *Fault Tolerant & Fault Testable Hardware Design*, Prentice-Hall, London, ISBN 0-13-308248-2.
- Papadopoulos, G. M., 1985. Redundancy Management of Synchronous and Asynchronous Systems, Fault Tolerant Hardware/Software Architecture for Flight Critical Functions, AGARD-LS-143.
- Rushby, J., 1993. *Formal Methods and Digital Systems Validation for Airborne Systems*, NASA CR 4551, Langley Research Center, VA.

- Shin, K. G. and Parmeswaran R., 1994. Real-Time Computing: A New Discipline of Computer Science and Engineering, *Proc. IEEE*, 82(1): 6–24.
- Shin, K. G. and Hagbae, K., 1994. A Time Redundancy Approach to TMR Failures Using Fault-State Likelihoods, *IEEE Trans. Comput.*, 43(10): 1151–1162.
- Sosnowski, J., 1994. Transient Fault Tolerance in Digital Systems, *IEEE Micro*, 14(1): 24–35.
- Tomek, L., Mainkar, V., Geist, R. M., and Trivedi, K. S., 1994. Reliability Modeling of Life-Critical, Real-Time Systems, *Proc. IEEE*, 82(1): 108–121.
- Vaidya, N. H. and Pradhan, D. K., 1993. Fault-Tolerant Design Strategies for High Reliability and Safety, *IEEE Trans. Comput.*, 42(10): 1195–1206
- Williams, L. G., 1992. Formal Methods in the Development of Safety Critical Software Systems, UCRL-ID-109416, Lawrence Livermore National Laboratory.
- Willisky, A. S., 1980. Failure Detection in Dynamic Systems, *Fault Tolerance Design and Redundancy Management Techniques*, AGARD-LS-109.

Further Information

A good introduction to fault-tolerant design is presented in *Fault Tolerance, Principles and Practices*, by Tom Anderson and P.A. Lee. Hardware-specific design techniques are described by P.K. Lala in *Fault Tolerant & Fault Testable Hardware Design*.

Other excellent journals are the IEEE publications, *Computers*, *Micro*, *Software*, and *Transactions on Computers*, and *Software Engineering*.