

### 3 FAULTS, ERRORS, AND FAILURES

#### 3.1 FAULT CLASSIFICATION

- 3.1.1 Fault Activity
- 3.1.2 Fault Duration
- 3.1.3 Fault Perception
- 3.1.4 Fault Intent
- 3.1.5 Fault Count
- 3.1.6 Fault Cause (single fault)
- 3.1.7 Fault Cause (multiple faults)
- 3.1.8 Fault Extent
- 3.1.9 Fault Value
- 3.1.10 Fault Observability
- 3.1.11 Fault Coincidence
- 3.1.12 Fault Creation Phase
- 3.1.13 Fault Nature
- 3.1.14 Fault Source

#### 3.2 FAILURE CLASSIFICATION

- 3.2.1 Failure Type
- 3.2.2 Failure Hazards
- 3.2.3 Failure Risks
- 3.2.4 Failure Accountability
- 3.2.5 Failure Effect Control

#### 3.3 PROBLEMS AND EXERCISES

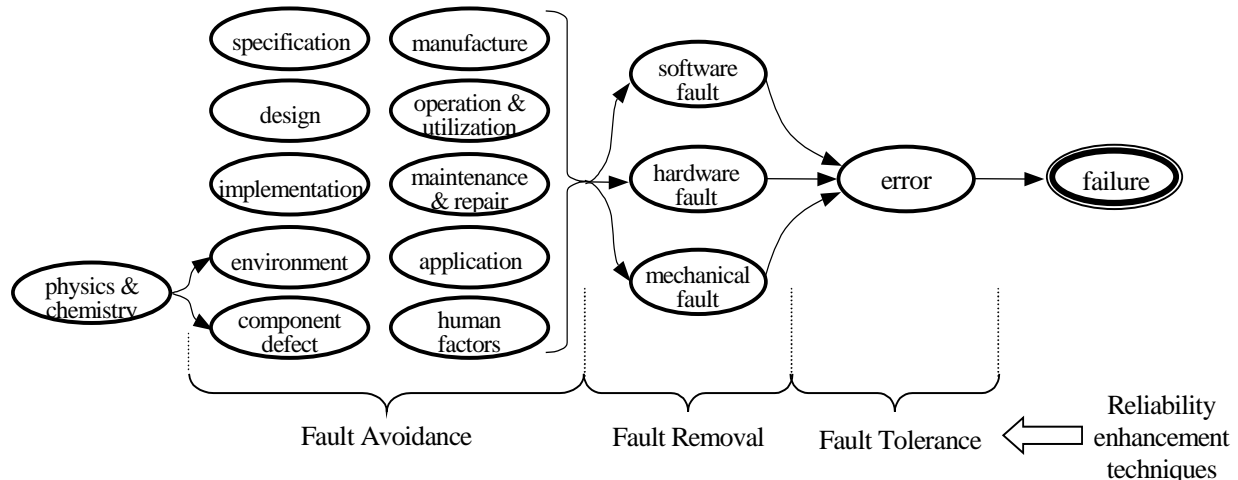
#### 3.4 REFERENCES AND BIBLIOGRAPHY

### 3     FAULTS, ERRORS, AND FAILURES

A reliable and dependable system has the ability to provide its intended, expected, and agreed upon functions, behavior, and operations, in a correct and timely manner. The inability of a system to provide this performance is referred to as a **failure**. A system's architects, developers, designers, assessors, and users, all need insight into this ability, or lack thereof. This is why the distinction is made between faults, errors, and failures. Many definitions of these terms exist, and they are sometimes used interchangeably.

All devices and other system resources (incl. human operators) have a state, value, operational mode, or condition. We speak of an **error**, if this deviates from the one that is correct or desired. The latter is based on specification, computation, observation, or theory. A system transitions to its "failed" state, when an error prevents it from delivering the required performance. I.e., a failure is the system-level manifestation of an error. The term "error" is often reserved for software as well as data items and structures.

Activities in any of the system's life cycle phases can lead to a defect, omission, incorrectness, or other flaw that enters into, or develops within that system. As illustrated in Figure 3.1, the life cycle covers system development and manufacture, through installation, operation, and maintenance. The flaw, whether hypothesized or actually identified, is the cause of an error. This cause is referred to as a **fault**. It can occur within *any* subsystem or component. This applies equally in the domains of software, electronic, mechanical, optical, chemical, and nuclear systems.



**Figure 3.1**     *Causal relationship between faults, errors, and failures*

A fault in the physical (non-software) domain requires a physical change, such as dielectric breakdown, a wire chafing through, or a flow-control valve seizing. Such changes can result from selection of components that are not suitable for either the system application, or the operating environment. Interference from that environment involves the accumulation, release, and transfer of unwanted energy to a vulnerable target in the absence of adequate barriers [Ciemens81]. The energy can be in the form of heat, shock or vibration, electromagnetic fields [Fuller95, Shooman93], electrical power, nuclear or particle irradiation, potential and kinetic energy, chemicals (incl. medication), etc. Other forms of environmental interference are caused by contaminants such as corrosive or conductive fluids or vapor, sand, or dust. Another source of

faults is the interaction with other systems, whether physical or via data. Human induced faults can be either inadvertent, or deliberate.

Faults in the software domain, often referred to as “bugs”, are related to the generation of executable code. These faults can assume a sheer unlimited number of forms, ranging from specification and design faults, coding faults, compiler flaws, logical errors, improper stack sizing, improper initialization of variables, etc. A plethora of cases is captured in [Risks]. They are generally dependent on data, data sequences, or timing [Storey96]. Though the timing of software tasks is normally implemented with hardware counters and clock oscillators, the initialization of these counters is usually done by software, as is timer interrupt handling. In addition, the associated operating system or task scheduler is also a fallible software item, as is timing-sensitive interaction between tasks. Any mutation in executable code is a *physical* fault. It can be caused by events such as memory corruption or a hard disk crash. Another source of software faults is the installation of hardware modifications without altering the software accordingly.

As outlined above, a failure is the result of an error, which in turn is caused by a fault [Laprie85, Avizienis86, Meissner89, Laprie90, IEC61508]. The reverse relationship, however, is conditional: a fault *may* cause one or more other faults or errors, and an error *may* cause failure. A fault must be activated to cause an error: the faulty system resource must be exercised. For example, irradiation or a supply voltage spike can corrupt the data at a certain computer memory location. This fault cannot cause an error until that location is read and the data used. A fault can also disappear without being present long enough to result in an error. The system may contain time constants due to mechanical or chemical process inertia, or due to analog or digital signal filtering. Sometimes the faulty state is (temporarily) the same as the correct state. It is possible that a fault or error has no perceptible effect on the system performance, so the system remains non-failed. Similarly, system performance may be degraded (perceptibly or not), but still be within specified limits. The system may include tolerance mechanisms that mask, or detect and correct errors. It must also be noted that an error typically occurs at some physical or logical distance from where the fault enters into, or resides within the system.

So far, the term “failure” has been used for a behavioral manifestation at the system boundary, with faults and errors reflecting the internal state of the system. However, each sub-system or component is *itself* a system that comprises interacting elements with their own performance requirements. Conversely, each system is a component of a larger system. The same phenomenon can be a fault at one level in the system hierarchy, and an error at the next level. How far the iterative decomposition should be carried out, depends on the system context and the reasons for investigating the causal relationships. The tracing must contribute to understanding. The “ultimate” fault is of importance in most commercial systems. It is the one that must be fixed to restore the system’s performance, or be changed to prevent recurrence, even though the underlying theoretical failure mechanisms may be intriguing.

For instance, when estimating the probability of system failure, it may suffice to go down to an integrated circuit’s behavior at pin level, ignoring the IC’s internal behavior at gate level. An accident investigation of a car’s drive-by-wire system might conclude that the *ultimate* cause is component damage due to electromagnetic interference. In other analyses, this same damage might be identified as merely the *immediate* cause. The ultimate cause could be traced to a flaw in the component selection or screening process, an omission in the environmental section of a system specification, etc. From the latter point of view, *any* failure can ultimately be construed as human-made. This is often not practical or insightful. It is important to make a distinction between physical or component faults, and human-made faults such as those related to

specification, design, human-machine interaction, etc. Contrary to physical faults, the latter tend to be unpredictable, the resulting errors unanticipated, and their manifestations unexpected [Anderson81]. Hence, they require different methods and techniques to prevent, remove, and tolerate them.

Faults, errors, and failures are commonly referred to as **impairments** to the reliability and dependability performance of a system [Laprie85, IFIP10.4, Prasad96]. Many types and classes of these impairments can occur during the life of a system. They must be identified, together with:

- How, and under which conditions they enter into the system
- The subsystem in which each of them occurs, or can occur
- If, and how they propagate through the system
- What effect they have at the system boundary.

There are important reasons to establish and evaluate these hierarchical, causal relationships, origins and effects [Avizienis87]. This enables qualitative and quantitative modeling and analysis of the reliability and dependability performance. It also serves as a basis for specification of goals and requirements for that performance. It is the essence of troubleshooting, whether *a priori* during system development, or *a posteriori* as part of maintenance or system improvement.

There are many methods and techniques for achieving and enhancing the aforementioned system performance. The identification of impairments gives insight into which particular **means** can, or should be applied at each level of the system hierarchy, or to each applicable life cycle activity. See Figure 3-1. The identification also supports the evaluation of the effectiveness of such means, whether they are proposed or actually implemented. Those means are typically categorized as:

- Fault avoidance,
- Fault removal, and
- Fault tolerance.

The purpose of **fault avoidance** is to prevent fault from entering into, or developing within the system. It usually is not possible to guarantee complete prevention. For most systems, fault avoidance is limited to reducing the probability of fault occurrence. Nonetheless, fault avoidance is powerful: a fault avoided, is one that need not be removed or tolerated with other, often costly, means. Fault avoidance techniques are applied to *all* life cycle phases of the system. This can actually precede the system inception phase, by carrying over experience and “lessons learned” from all sources: in-house, industry, customers, users, authorities, competition, etc. A major technique is the rigorous application of Systems Engineering [EIA632]. This includes the capture and management of requirements for the system’s functionality, interfaces, performance, and technology. Requirements must be consistent, complete, unambiguous, verifiable, and traceable. Other techniques range from the selection of inherently reliable components, the performance of analyses, adherence to standards, design for testability, through concern for human factors: the attention to “live-ware”. A supplementary approach is the use of “formal methods”: the use of mathematical techniques, with explicit requirements and assumptions, in the design and analysis of computer hardware and software [Rushby93]. It should be noted that no method guarantees unequivocal correctness.

Especially in complex systems, faults are present despite fault avoidance measures. **Fault removal** is a methodology that minimizes their presence by finding and removing them. This is the process of establishing and documenting whether system implementation, items, processes, and documentation, conform to specified requirements. This approach can take the form of requirements and design reviews, inspections, tests, audits, and other forms of verification

[Anderson81, DSVH89]. The (potential) faults exposed by this process are subsequently removed by altering the system design or the interaction with its environment. I.e., fault avoidance techniques are applied to the newly identified faults. This again is subject to verification. Fault removal does not refer to fixing a fault by means of repair actions.

**Fault tolerance** uses redundancy of system resources to provide alternate means of providing the system functionality and operation. This enables the system to sustain one or more faults or errors, in a way that is transparent to the operating environment and users. I.e., it prevents a fault from causing a failure, by masking the effects of errors resulting from that fault. This implies **fault containment**, as the fault is precluded from propagating and affecting system level performance. In general, fault tolerance can only handle a limited number of *specified*, non-simultaneous faults. Fault tolerance is typically used when a non-redundant system, combined with the above techniques, would pose unacceptable exposure to risks. This includes inaccessibility for repair actions, loss of revenue due to service interruption, damage to equipment, property, or the environment, and death, injury, or illness.

The avoidance and tolerance of faults are means to provide a system with reliable and dependable performance. This is complemented by fault removal, to avoid recurrence of identified faults. The latter means allows confidence to be placed on the ability of the system to provide that performance.

---

### 3.1 FAULT CLASSIFICATION AND TYPES

---

It is impractical to enumerate all possible variations of faults that can occur in a general system, or even in a particular system, other than the most simplistic. To get a handle on this, we must aggregate fault types into manageable classes or categories, each representing a major fault attribute. Table 3-1 shows a general fault classification based on commonly used attributes [Avizienis86, Laprie90, McElvany91]. This table also lists the fault types that each attribute distinguishes. They are described in the subsections below.

Fault Attribute	Fault Types
Activity	Dormant, Latent, Active
Duration	Permanent, Temporary, Intermittent
Perception	Symmetrical, Asymmetrical
Intent	Benign, Malicious, Byzantine
Count	Single, Multiple
Cause (single fault)	Random, Deterministic, Generic
Cause (multiple faults)	Independent, Correlated, Common-Mode
Extent	Local, Global, Distributed
Value	Fixed, Varying
Observability	Observable, Not Observable
Coincidence	Distinct, Coincident, Near-Coincident
Phase of creation	Development, Manufacture, Operation, Support
Nature	Accidental, Intentional
Source	Hardware, Software

**Table 3-1 General Fault Classification**

This classification must be done in the context of the architecture and design of the particular system, its application, as well as the operating and maintenance environment of interest. The same applies to the mapping of individual fault cases to fault classes. I.e., a particular fault case in a system may map to a different class (or classes), based on the operational mode or mission phase. Likewise, the same fault may map to different classes for different systems. Not all classes apply, or are relevant to all systems: the classification must be tailored. Conversely, the list of attribute classes may have to be expanded to reflect aspects of fault detection and tolerance.

---

### 3.1.1    Fault Activity

---

Fault activity indicates whether a fault is producing an error or failure, and if so, whether the latter is detected or observed. This combines the notions of fault propagation and detection. See Table 3-2.

	Not Propagating	Propagating
Not Detected	Dormant	Latent
Detected		Active

*Table 3-1    Fault activity classification*

A fault is **dormant**, if it is not (yet) causing an error. It remains in this “lurking” state until the faulty system element is exercised, utilized, or otherwise called upon. For example:

- A punctured spare tire, or a backup generator without fuel
- An control discrete that is stuck at the currently commanded state
- A corrupted location in Program Memory, corresponding to a subroutine or task that has not yet been executed
- A broken fire detection sensor that remains undetected until tested, or the malfunction becomes apparent when a fire occurs.

A dormant fault becomes **active** as soon as its manifestations are detected. This may be through direct observation at the faulty system resource, or be inferred from sub-system behavior and system performance. Detection does not imply that the faulty element has been identified and located. Many detection mechanisms need a certain amount of time to reliably determine that a fault has occurred, once its effects become observable to that detector. During this time the fault is **latent**. Not all faults have manifestations that are detectable by the system. Such faults are also called latent. Sometimes the terms latent and dormant are used interchangeably.

Multiple latent faults may accumulate undetected and compound over time until they eventually manifest themselves simultaneously. This can defeat the fault tolerant capability of the system, or of the operator. This makes it important to reduce the detection latency: the time between a fault becoming observable, and positive detection of the fault. The latency can be reduced by periodically performing a system integrity check with adequate coverage [McGough89].

---

### 3.1.2 Fault Duration

---

The **duration** attribute indicates how long a fault is present in a system. This implies that a timescale is applied to classify this fault persistence. A distinction is made between permanent faults, and temporary or transient faults. The threshold depends on the fault itself, as well as on the system in which it occurs.

A **permanent** or **hard fault** has *indefinite* duration. However, “indefinite” is a relative term. If truly indefinite, an irreversible physical change has taken place and a maintenance action is required to remove the fault. E.g., repair or replacement of a damaged component or faulty subsystem. The fault remains, even if its excitation disappears. Sometimes a seemingly hard fault can be removed with a system reset or cold start. Even if a fault is not permanent, it can still be perceived as such by the system’s fault detection mechanism or by the user. After a certain confirmation persistence has expired, it may no longer matter whether the fault disappears, or it cannot be reliably determined whether it has. Persistence thresholds are typically based on time constants or bandwidth of the system, or the maximum time allowed before an uncontrolled fault causes an unsafe condition. Sometimes the term “hard fault” is used if repeated use of the same input and initial conditions always result in the same incorrect response. However, this reflects determinism, rather than duration.

A **temporary** or **transient fault** is only present for a limited period of time. The functional ability of the faulty element recovers without corrective action being taken: the fault disappears spontaneously and no permanent damage is done [Anderson81]. Transient faults are also referred to as **soft faults**, **glitches**, **upsets**, or “**hiccups**”. They are the effect of a temporary internal or external condition or stimulus. The causes are often environmental, such as electromagnetic interference, electrical power drops or surges, irradiation, or mechanical shock. Some transients are caused by (temporary) conditions that were not, or not adequately, anticipated and covered during the specification and design of the system. Transient faults in digital computing systems typically occur an order of magnitude more often than hard faults [Siewiorek92]. The transient duration of a fault should not be confused with the transient *magnitude* of its excitation or effects.

A recurring transient fault is called **intermittent**. Such faults are often only present occasionally and inconsistently. I.e., they occur irregularly and infrequently with respect to the mission duration. In general, they are caused by hardware that is unstable, or only marginally stable for certain operating conditions. This can be due to design, manufacturing flaws, aging and wear, unexpected operating environment, etc. For example:

- the combination of vibration and a loose connection (e.g., chafed through wiring, poorly socketed component, bad solder joint, poor bonding inside IC).
- dirty or poorly seated connector (high contact resistance), causing reduced noise margin on logic levels.
- excessive drift of a clock oscillator’s frequency, causing occasional brief loss of synchronization between processing nodes.

Marginal software process timing, or processor throughput, can also cause intermittent faults, although software itself cannot fail intermittently. Intermittent faults are often hard to trouble shoot due to their seemingly elusive, inconsistent, and unpredictable occurrence. More complex systems include extensive logging of the fault circumstances (system state, mission phase, internal temperature and voltages, the value of control inputs, etc.) to help correlate the fault to its cause.

---

### 3.1.3     Fault Perception

---

Fault tolerant computing systems typically comprise multiple concurrent processing nodes. The nodes exchange data for the purpose of reaching consensus regarding input data to be selected, the computed output data, control modes that are to be activated, diagnosis of faulty nodes or other system resources, etc. **Fault perception** indicates the consistency between the observation of the fault by all non-failed subsystems. The fault is **symmetrical** if it is seen *identically* by those subsystems. Conversely, a fault is called **asymmetrical** if it produces different symptoms for different observers.

A standard example of asymmetry is a data transmitter that is fanned out to several receivers, creating a number of transmitter-receiver pairs. Individually, the transmitter and receivers may operate within their respective specifications, but combined this is not necessarily the case for all pairs. Asymmetry can also result from a computing resource failing in such a way that it sends conflicting versions of the same information to the other members of the computing resource population. Also see sections 3.1.4 and 3.1.14.

---

### 3.1.4     Fault Intent

---

The difficulty encountered in tolerating a particular fault is driven by the behavior of that fault and the resulting errors. There are two basic types of fault tolerance, both based on redundancy techniques:

- Active or dynamic redundancy, which uses duplication and comparison to perform fault diagnosis (i.e., detect and locate).
- Passive redundancy, where duplication is combined with a voting mechanism (redundant or not) to mask fault occurrence and avoid errors, often without attempting to locate or even detect faults.

The **fault intent** indicates whether this behavior is devious, doing seemingly anything it can to cause a system failure. In turn, this determines the type of fault tolerance required.

A **benign** fault can be detected by any non-faulty observer. It does not matter if this is a single observer, or a set of redundant observers that coordinate their fault detection. An example of such a fault is a processing task that takes too long to complete. This can be detected and tolerated with active redundancy.

Errors caused by a **malicious** fault may not be directly recognizable, and cause different observers to see different symptoms, if any. Such asymmetrical faults can confuse and even defeat the diagnostic capability of computing systems with standard, active redundancy. Worst case, multiple faulty resources can seemingly collude to escape detection and corrupt the system [Dolev83, Shin87, Barborak93, Lala94]. This may develop into:

- incrimination of healthy system resources,
- total disagreement between processing nodes (system crash), or
- agreement on the wrong conclusion; i.e., a system failure that is not contained and not annunciated by the system itself.

Such effects can also occur at circuit level [Nanya89].

When no *a priori* assumptions are made regarding the malicious characteristics of faulty components we speak of **Byzantine faults**. They exhibit unrestricted, arbitrary misbehavior (timing, value, state, etc.). This type of pernicious fault is called “Byzantine”, in analogy with the so-called Byzantine Generals Problem [Lamport82, Walter88]. The scenario is that a number of Byzantine Army divisions are besieging a town. The commanding generals (redundant processing nodes) have to reach consensus on whether to attack or to retreat (mode switching or computed parameter). Each general has a messenger (datalink) to send his own, local opinion to each of the other generals, as well as to relay the opinions received from those generals. The sending generals and messengers can be loyal or treacherous (conveying wrong, distorted, or inconsistent messages to different generals), and receiving generals can have interpretation problems. The provable, classical solution to the Byzantine problem requires  $3m+1$  nodes, where  $m$  is the number of (simultaneously) failing “generals” that must be tolerated. I.e., a minimum of 4 nodes! This is higher than the  $2m+1$  required in simple majority voting schemes. Multiple rounds of interactive voting are needed to reach global consensus and to identify a “malicious” node.

---

### 3.1.5 Fault Count

---

**Fault count** simply indicates the multiplicity of fault occurrence during the time period of interest. It is the prime ingredient of fault statistics such as the actuary and predicted rate of occurrence. It is also an important attribute for fault tolerant systems, as they can only tolerate a limited number (and types) of faults before exhausting their redundancy.

In the case of a **single fault**, only one fault occurs, has occurred, or is hypothesized to occur. This allows the particular fault to be treated in an isolated fashion, for the purpose of analysis or fault tolerance.

**Multiple** faults take the form of repeated occurrence of the *same* fault, or the occurrence of multiple, *different* faults. Related attributes are and whether they are invoked by the same cause, and whether they occur at the same time. See sections 3.1.7 and 3.1.11. Both of these attributes determine if the multiple faults can be treated as separate, unrelated single faults.

---

### 3.1.6 Fault Cause (single fault)

---

The **fault cause** indicates whether a fault is random or deterministic. By definition, **random faults** are non-deterministic. More precisely, the term “random” is used to model and label events as such<sup>1</sup>. In the end, every variation, failure, event, and uncertainty has an assignable root cause that can be identified if sufficient effort is expended [Evans99]. Occurrences of such faults have a statistical distribution. They are not correlated to one another, or to a particular causal event. See section 2.x.. A fault is **deterministic** if it happens with statistical certainty, anytime a particular trigger condition or event takes place. I.e., there is an “if-then” causal relationship.

Software cannot fail randomly. Software is time-invariant, and does not wear out. If software has a fault, then that fault is always present, whether observable or not. The related error is produced

---

<sup>1</sup> “Nothing in nature is random... A thing appears random only through the incompleteness of our knowledge”, Baruch Spinoza, Dutch philosopher 1632-1677

deterministically, each time it is triggered by a particular data value, data sequence, or timing with respect to other processes. If the faulty software is never executed, the fault will never produce an error or failure. So, it is difficult to apply hardware-oriented terms like failure rates to **software reliability** [Gordon91, Pfleeger92, Dunn86, Hecht86], nor is it quantifiable [Butler93]. What is generally meant by “reliable” software, is that each process generates “correct” internal states and outputs when supplied with valid inputs or sequences thereof, and that the response to invalid inputs is pre-defined. Software is “correct” if it operates as specified.

---

### 3.1.7     Fault Cause (multiple faults)

---

When multiple faults occur, it is important to know to what extent their causes are related. Two faults are **independent** if they have unrelated causes. The probability that one occurs is unaffected by the occurrence of the other, and vice versa. In other words, the conditional probability is equal to the unconditional probability, and the combined probability is equal to the product of the probabilities of the individual faults. See chapter 2. In contrast, **correlated faults** have a combined probability of occurrence that is significantly higher than that for the same number of unrelated random faults [Pullum99]. The combined probability of occurrence of dependent events is **not** equal to the product of the probabilities of the individual events. If system performance is based on independence, this must be assured in the actual implementation. This undesirable dependence can happen in two basic ways:

- One fault either directly causes an other one, or makes it more likely that the other occurs. Worst case, this can turn into a cascade or avalanche effect.
- Multiple faults are caused by the same event. These are called common-mode or common-cause faults.

**Common-mode** or **common-cause faults** occur if a *single* condition or event causes *multiple* system elements to fail. The invoked faults need not happen exactly at the same time, nor is it necessary that the affected components are identical or redundant. Examples of common-causes are:

- Common requirements.
- Common environment: vibration, pressure, cooling, irradiation, explosion or fire, contamination by fluids, uncontained rupture of rotating parts or pressure vessels, etc.
- Common (shared) circuitry: a processor and memory, time base or clock, synchronization mechanism, data bus, or power source.
- Common input data, shared by multiple processes.
- Common manufacturing process.
- Common operation: operating procedures, Human-Machine interface.
- Common maintenance and test procedures, or test equipment.

This type of fault can have serious consequences, if the same cause results in a fault *and* prevents detection of that fault. E.g., substrate fracture inside an integrated circuit that generates a sensor excitation signal and at the same time monitors that signal. Fault tolerant systems contain redundant resources to continue the system’s services and performance, despite the presence of a (specified) limited number of faults. This redundancy may be exhausted at once if a single cause affects several or all of those resources. E.g., if the rotor disk of a jet engine disintegrates and the shrapnel severs all of supply lines of the aircraft’s redundant hydraulic system [Hughes89].

A special type of common-cause fault is the **generic fault** [Yount85]. As the name suggests, it affects system resources that are *identical* or *similar*. Such faults are of particular concern in fault tolerant systems that use redundant resources that are identical, e.g., multiple computing nodes that execute the same software on identical hardware. A software flaw may cause all nodes to perform the same but wrong calculations at the same time. Despite these errors, the redundant nodes remain in agreement. This defeats all fault tolerance techniques that are based on the premise that a fault causes a *difference* between the redundant resources.

The general approach for protection against generic faults and their consequences is to ensure independence of faults by applying **design diversity**, also known as **dissimilarity**. This concept dates back nearly 175 years to the era of Charles Babbage's calculating machine [Lardner34]. In the software domain dissimilarity is usually referred to as **N-version** programming [Avizienis85, Voges88]. Dissimilarity is used where component replication to tolerate faults is not feasible or effective, i.e., when identical copies show identical faults (nearly) at the same time. This is the case with software, as it is not subject to physical faults due to noise, aging, or wear. Care must be taken not to compromise version-independence when fixing ambiguities or divergence between versions. The level of achievable version-independence is open for debate [Knight86, Knight90]. Dissimilarity provides little or no advantage in protecting against common-mode environmental faults. It also does not reduce the probability of simultaneous *random* faults.

Often, generic faults are referred to as *residual design errors* [Avizienis99]. However, their causes can be found in system life cycle phases other than design. For instance, all hardware components of a particular type can have the same manufacturing flaw (i.e., a "bad lot"). The system specification can have deficiencies that equally affect all systems designed in compliance with this specification. This can be a specification error, ambiguity, or omission. Applying dissimilarity at this level is usually difficult, as the redundant system elements are expected to provide the same (correct) functionality and behavior. Of course, when the generic flaw is in a common specification as this causes different implementations of the same fault.

Diversity can be applied at software, hardware component, system, and process level, e.g.:

- Different programming languages combined with identical or different processors
- Same software source code but different compilers and assemblers
- Object oriented programming vs. functional (structured) programming
- Fixed vs. floating point arithmetic
- Different coordinate systems, e.g., Cartesian vs. Polar
- Operating system based on a state machine vs. dynamic task scheduling
- Electromagnetic vs. hydraulic actuation (but no cross-power generation)
- Generation of aerodynamic control forces via control surface deflection vs. engine thrust vectoring
- Data bus with copper wire vs. optic fiber
- Different algorithms for the same function
- Integrated circuits with same form, fit, and function, but implemented with different technologies, cell libraries, or processes.

Obviously the application of dissimilarity has a significant impact on the cost of the system. The *non-recurring* development effort must be replicated (incl. costly software verification and validation), multiple manufacturing lines set up, qualification testing performed, service and training manuals written, etc. The *recurring* cost also increases: more component and subassembly part numbers must be tracked, different types of hardware or mechanical

components be procured and stocked (in smaller quantities than in a system with identical redundancy, and often with limited choice), configuration management expanded, etc.

---

### 3.1.8     Fault Extent

---

The **fault extent** describes the scope of a fault: how far it propagates through the system, and which areas of the system are effected. The boundary of this area is called the Fault Containment Region (FCR). Uncontained faults propagate *outside* the system boundary, to connected systems and users.

The fault extent is **local**, if the fault does not propagate beyond the subsystem or module that contains the faulty component. If the fault causes errors outside this subsystem, the fault extent is called **distributed**.

---

### 3.1.9     Fault Value

---

Each system component has an associated erroneous value, state, or status. The **fault value** indicates whether this changes with time upon fault occurrence.

A **fixed** fault value does not change. This is also referred to as a **stationary** or **frozen** value. How this value is reached depends on the component and particular fault in question. It can be a jump, drift (also known as “creep-away” or “slow-over”), monotonic slew, oscillatory or erratic move. The terminal value can be:

- an extreme value, e.g., stuck at logical level 1 or 0, the maximum position of an actuator (also known as a “hard-over”), the supply voltage or ground potential
- the component’s value at moment of the fault occurrence
- some predetermined value, due to the nature of the component, or forced by the system upon fault detection.
- an arbitrary value.

A **varying** erroneous value changes anywhere between extreme, fixed or arbitrary limits between extremes. The rate of change can be fixed or varying in sign and magnitude.

---

### 3.1.10     Fault Observability

---

A fault is **observable** if a symptom of its existence is available to a monitoring mechanism inside or outside the system. Some fault manifestations can be directly observed at the affected system component. Others can be inferred from behavior and performance at higher levels in the system hierarchy, i.e., from errors and failures. Fault observability is important, as it is the prerequisite for fault detection.

In theory, *any* fault can be made directly observable if enough effort is spent on devising a method. This is typically not practical from a technical and economical point of view, nor is it

necessary. In addition, observation mechanisms themselves are subject to faults and contribute to unreliability of the overall system.

For instance, a loose wire bond inside an integrated circuit is observable by visual inspection inside the component package. Obviously, providing such a level of detection outside the chip manufacturing process is unthinkable. Exercising the chip during periodic system test could surface the fault. In some system it is critical that component faults are detected timely. This can be done with redundancy techniques at component level or higher, or with special detectors.

---

### 3.1.11 Fault Coincidence

---

The **fault coincidence** attribute indicates the separation in time of fault occurrences. As a fault cannot coincide with itself, this attribute is reserved for multiple faults. These faults need not be identical, nor affect the same component(s). Temporal separation does not imply causal independence: common-cause faults need not happen at the same time, despite their correlation.

**Coincident** or **simultaneous faults** occur at exactly the same time. No assumptions are made regarding their duration being identical. **Distinct** or **separate faults** do not occur at the same time. However, their duration may still overlap. The distinction between “coincident” and “distinct” is not as clear as it may seem. Strictly speaking, truly simultaneous events do not exist. Even if they did exist, their exact simultaneity could never be determined with absolute certainty.

It makes more sense to consider **near-simultaneous** or **near-coincident** events [McGough83]. This “nearness” allows for a finite time window that can be (approximately) quantified, based on the type of faults and the system at hand. If multiple faults occur within the span of such a window, they are near enough to be treated as “coincident”.

Near-coincident faults must usually be considered for critical systems that are required to provide extremely high reliability over long periods of time. Such systems employ redundancy to achieve fault tolerance. The redundancy management often uses fault detection mechanisms that are based on pair-wise comparison of the redundant resources. These monitors typically apply a persistence time window to confirm the miscompare fault-condition. During this time window, the system is vulnerable to a consecutive fault that affects the same redundant set of monitored parameters. The faulty parameters may happen to be in agreement. This can deceive the fault detection in such a way that either no fault is detected, or the remaining “good” resources are incriminated. Either way, the faults continue to propagate and the fault tolerance is defeated.

---

### 3.1.12 Fault Creation Phase

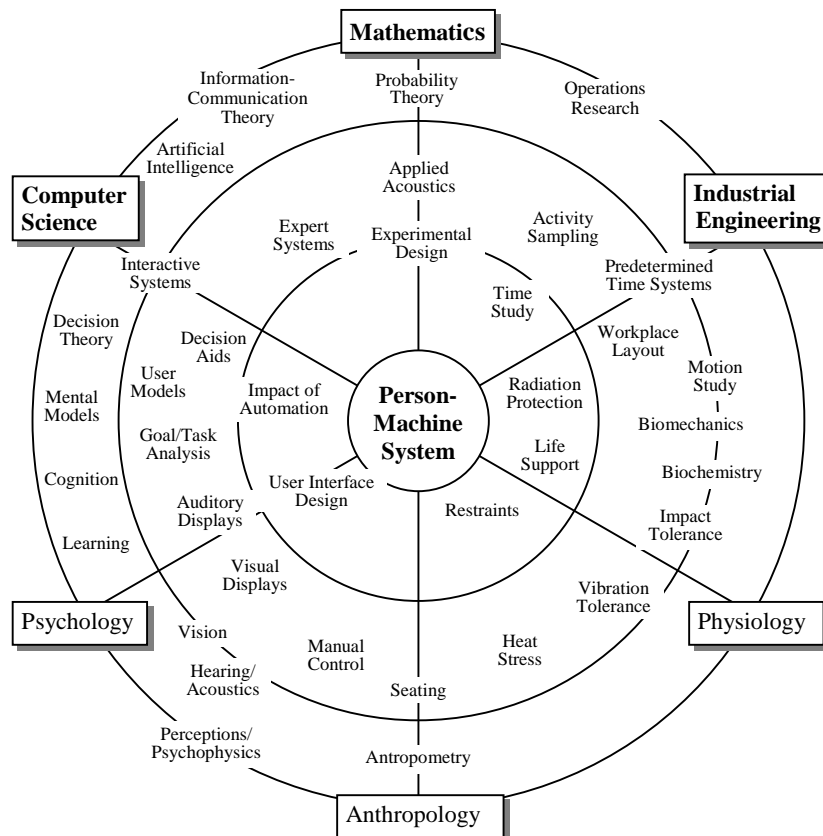
---

The life span of a system is divided into so-called life cycle *phases*. Faults can (and do) enter into, or develop within the system during any of these phases [Redmill97a]. Many life cycle models exist, e.g., [EIA632]. They typically distinguish phases and activities corresponding to those listed in the generic five-phase model below. The transition between the phases is generally marked by a program decision milestone. It indicates that all tasks of the preceding phase have been completed in a satisfactory manner. All phases influence the system reliability.

The system **concept and planning phase** of commercial systems typically begins with the identification of customer needs and expectations, and of market opportunity. System concepts are formulated. Plans are made for the execution of subsequent phases, and the required resources are identified and approved. Risks are identified and mitigated in the area of technology, program schedule, cost, etc.

The system **development phase** of system requirements are captured, analyzed, and managed. Based on this, several system architecture alternatives are evaluated for suitability and feasibility. The selected alternative is the basis for preliminary and detailed design of hardware, software, and mechanical subsystems. Often prototypes and simulations are used to validate concepts and designs.

The designs are verified against the requirements prior to implementation. The latter involves selecting components for the circuit design, as well as laying out circuit boards. Software implementation covers the application software and the operating system. The implemented subsystems are integrated and tested at subsystem and system level. The development phase is an iterative process, especially for complex systems. This phase usually includes the development of test equipment or tools that are used during the system development or manufacturing and support phases. Quality Assurance techniques must be applied throughout the development, as must design for testability.



**Figure 3-2 Relationship of Human Factors to Other Areas of Science**

**From: [Fogel63]**

The development phase must pay close attention to the Human-Machine Interface (HMI), the system will be installed, operated, and maintained by humans. This requires *physical* access to, and interaction with the system while it is installed in its operating environment. The interaction involves tools and test equipment, data entry and control devices, actuators, displays, aural and visual annunciators, etc. The operator or user also has *mental* interaction with the system by directing its functionality, interpreting its behavior, etc. Cognitive aspects of automation must be devised carefully, as it is prone to causing confusion about active operational modes and changes thereof: “how do I make it do ...”, “what is it doing?”, “why did it do that?”, “what is it going to do next?”, “why is it doing this again?” [Wiener88, Hughes95, Leveson97]. Human Factors engineering (HFE) involves many sciences and engineering disciplines, as illustrated in Figure 3-2 [Fogel63, Donchin95], and is especially important for especially in safety-critical systems [Redmill97b].

The **manufacturing phase** includes the procurement, fabrication and screening of components and tools. These are subsequently installed in subassemblies and combined into the system level assembly. If necessary, adjustments are made or calibration “tuning” is performed prior to testing the system. The latter may involve burn-in testing such as temperature cycling, to weed out “weak” systems. Early in the phase, production equipment is finalized and quality processes are established. The completed system is delivered to the customer or end-user in the field, often subject to an acceptance test.

During the **operation and support phase** the system is deployed: it is installed in its fixed or mobile location, and checked out. Subsequently it is utilized: it performs its mission, including interaction with users, operators, other systems and the environment. This may involve operating manuals and operator training. System and customer support covers training of operators and maintenance personnel as well as the performance of maintenance actions. Maintenance is performed to restore system performance via repair or replacement of faulty components, or installing modifications and upgrades. This involves procedures and documentation (manuals), as well as components. Replacement of parts is subject to installation faults similar to those that can occur during the original system manufacture. Upgrades and modifications (e.g., when “drop in” replacement parts can no longer be procured) are also subject to development faults. Experience with the fielded system can be fed back to any of the preceding phases to improve the system.

**Disposal** is the life-cycle phase in which the system is decommissioned, i.e., permanently taken out of service. The system’s ability to provide expected or required services is no longer an issue. However, parts of a retired system may end up being re-used in other systems. This applies to hardware and software components as well their requirements and design. In addition, a system may contain “hazardous waste material” causing its disposal to have environmental implications.

---

### 3.1.13 Fault Nature

---

Depending on the system, it maybe necessary to examine the **nature** of faults. This indicates whether a fault develops **accidentally** during one of the system’s life cycle phases, or as the result of an intentional act. **Intentional** faults result from intrusions into the system during any life cycle phase, for the explicit purpose of causing faults. Examples of such faults are benign or malicious computer viruses, and the destructive acts of vandals, terrorists, and otherwise disgruntled or hostile persons.

---

### 3.1.14 Fault Source

---

The **fault source** indicates whether a fault is a **hardware** fault or a **software** fault. As software executes on, and interacts with hardware, the underlying cause of a software error may actually be a hardware error (an incorrect or undesired state) or fault. Conversely, flawed software can invoke a hardware fault or error. It is difficult to foresee all such possible interactions and dependencies during the development of complex systems [Littlewood93]. Correction of a software problem may actually require a hardware change, and vice versa.

There are many types of software faults, e.g. [Littlewood92, Sullivan92]:

- computation of addresses that cause (attempted) access violations of memory or other hardware resources
- “hogging” (no timely release) of allocated hardware resources, to the exclusion of access by other software tasks or users
- incorrect pointer management or statement logic, causing wrong execution order or omission
- incorrect or missing variable initialization
- existence of a path to an undefined or unanticipated state
- stack overflows
- wrong algorithm used (but program runs without “crashing”)
- incomplete or incorrect implementation of a function
- timing or synchronization error (management of real-time resources)
- software-build error: the software must be compiled, linked to the correct version of libraries and other software modules, located, and loaded or installed. Faults can occur in each of these steps. These steps also involve tools that are software themselves and prone to faults like the developed application or executive software.

Software reliability issues are discussed in Chapter 4.

The classical distinction between hardware and software is less clear in “hardware-implemented software” or “hardware-near-software”. This refers to the embodiment of (complex) algorithmic functions into hardware devices such as complex Programmable Array Logic (PALs) or high-density Programmable Logic Devices (PLDs) such as large Field Programmable Gate Arrays (FPGAs) and Complex<sup>2</sup> PLDs (CPLDs), certain Application Specific Integrated Circuits (ASICs), and high gate-count processors [Wichman93]. In many cases this involves design and implementation via Hardware Description “programming” Languages (HDLs), and the use of software tools for design, synthesis, simulation, and test. I.e., a development process that is very similar to that of regular software. As with complex software, it may not be feasible to guarantee correctness of the device. Its development must be subject to same rigorous assurance measures as would have to be used for an equivalent software implementation.

---

## 3.2 FAILURE CLASSIFICATION

---

A **failure mode** is the manner in which an item *has* failed (observation), or *can* fail (conjecture). Simple devices may have only one or two failure modes. For instance, a relay can fail “open” or “closed”. More complex functions, or devices such as microprocessors, can have a vast number of failure modes. These may have to be lumped so they can be treated in a practical manner.

---

<sup>2</sup> A device’s gate count does not necessarily correlate to its level of complexity.

The failure modes affect the service and performance that is delivered by the failed item or by the system that contains it. This in turn impacts all users and user-systems that directly or indirectly depend on this service. In addition, system malfunctions may cause damage to people and objects that do not directly participate in the normal operation of the system: manufacturers and designers, bystanders, underwriters, nature, society at large, etc. The actual or perceived consequences are called the **failure effects** [ARP4761, Walter95].

Failures can be categorized according to attributes such as:

- Type
- Hazards
- Risks
- Failure accountability
- Failure effect controls

---

### 3.2.1 Failure Type

---

The **failure type** indicates the correctness and timeliness of the functionality delivered by the system, when faults and errors propagate to the system boundary:

- **Omission failure**: the system starts tasks too late, only performs them partially, or does not start the tasks at all. The latter is also known as a “system crash”. The system is called **fail-stop** if the system ceases all activities at its interfaces, independent of the continuation of internal activities. Fail-stop behavior is usually easy to detect.
- **Timing failure** is similar to omission failure, but expanded with starting tasks too early or *completing* too late.
- **Incorrect results**: computations in the system generate timing and value faults that propagate to the system boundary; the faults occur even if the system is supplied with correct input data.
- A system is called **fail-silent** if failures never produce incorrect results: the system either delivers correct performance or fails by omission.
- **Graceful degradation** occurs if a system continuous to provide its primary functions, and the loss of the remaining functionality is not hazardous. This situation may increase operator workload or required vigilance. This is also called **fail-soft**, and is often preferred for safety critical applications, when retention of some degraded is preferred over complete loss of function. Take for example a mechanical actuation system with two independently controlled motors whose outputs are velocity-summed. If one control channel fails, the drive system can still provide actuation without loss of torque, but at half speed. An other example is a control system that sheds certain automatic control or protection modes upon failure of particular sensor or computational resources, and annunciates this to the operators.

**Fail-Operational** behavior does not appear in the above listing, as it implies that the system is fault tolerant and continues to deliver correct performance, despite the presence of a certain predetermined number and type of faults in the system. I.e., there is no failure at system level. It should also be noted that absence of operation by no means implies safety, as is the case for heart pacemakers, and control systems for processes that are destructively unstable without computer control. Similarly, safety does not imply that system failure occurs very infrequently. These aspects of system failure are further discussed in the sections below.

---

### 3.2.2 Failure Hazards

---

**Hazards** are actual or potential, unplanned (but not necessarily unexpected) undesired conditions or events that *could* result from system failure [Roland90, Leveson95]. Examples of such events are:

- death, injury, occupational illness or discomfort
- damage to, or loss of equipment or property
- damage to the natural environment
- adverse operational and environmental conditions such that the operator cannot cope without applying exceptional skills or strength
- loss of revenue due to service interruption or loss of sales
- loss of goodwill and customer satisfaction
- loss of opportunity
- extension or termination of a product's development schedule
- loss of organizational or corporate morale, confidence, prestige, or public image
- legal, moral, or political liability.

**Safety** is the relative freedom from being the cause or subject of uncontrolled hazards. Hence, it is a state in which the real or perceived risk is lower than the upper limit of the acceptable risk. Safety look at consequences of failures, independent of whether the system is reliable, i.e., delivers its required or expected service [Mulazzani85]. For instance, a bomb that cannot explode is safe, but not reliable.

**Hazard severity** is the worst *credible* known or potential consequence that could ultimately result from system malfunction. This severity is ranked with classes or ranges. Example of a 4-level ranking is Catastrophic, Major, Minor, and Negligible Effect [ACAMJ25]. A 2-level ranking could be Benign versus Malign, based on whether the consequences are of the same order of magnitude as the benefits derived from the service delivered by the non-failed system [Laprie85]. For instance, death in a car crash versus the benefit of personal transportation. The number of severity levels is academic, but the boundaries must be clearly defined and the ranges must be consistent without overlap.

Severity classification is form of Hazard Assessment. It is preceded by identification of possible failure modes at various levels of the system, and the associated effects. See Chapter X. This assessment is performed for several reasons:

- to meet contractual or system certification requirements.
- as part of an assessment of the risks versus the benefits derived from the potentially risky function or activity, or similarly, to evaluate the incremental cost of hazard abatement efforts against the resulting reduction in mishap costs. I.e., used as a management tool [Icarus94].
- to provide inputs to trade studies for system architecture and design.

**Hazard probability** is the expected likelihood that a system failure with hazardous consequences will occur. As with hazard severity, the probability is ranked with qualitative or quantitative classes or ranges. For example, a qualitative scale can range from “frequent” to “extremely improbable” [ACAMJ25]. A rating of “absolutely impossible” is typically not an option. The classes are scaled for the mission length or life span of an individual system, or aggregated for the entire installed equipment base or “fleet”. The probability or frequency data is derived from historical data for similar systems or components, or predicted based on analysis. A powerful tool for obtaining a low hazard probability is a system architecture in which it takes *multiple* independent faults before a hazardous failure occurs. I.e., the elimination of single-point failures.

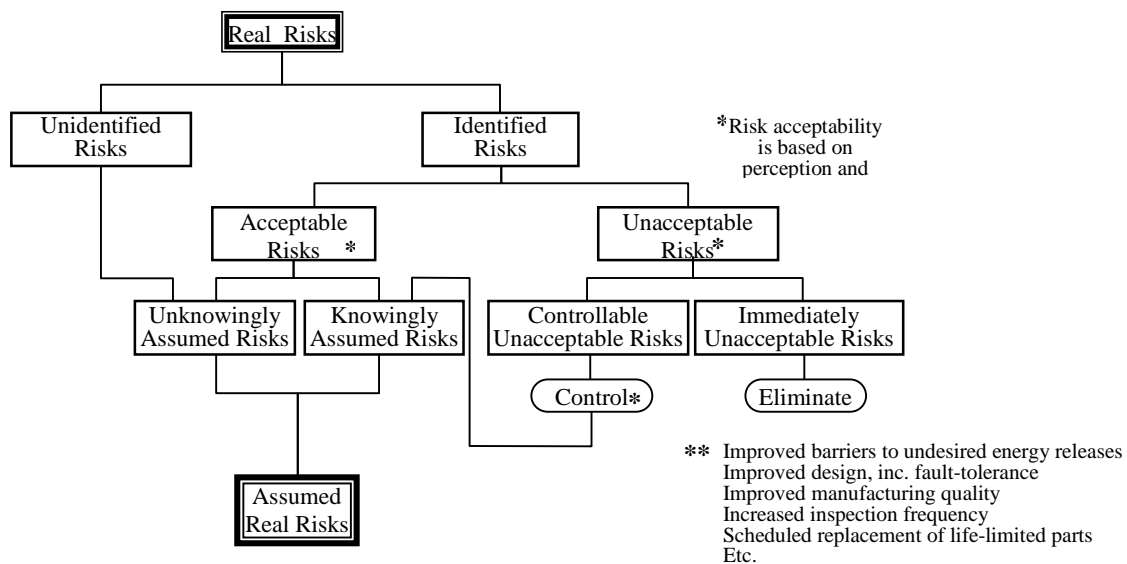
This results in a series reliability model in which the event probabilities are *multiplied* to arrive at the probability of the unsafe event. See Chapter #.

### 3.2.3 Failure Risks

The **risk** attribute combines the hazard *severity* with the hazard *probability*, resulting in a hazard-risk index or matrix:

$$\begin{aligned}\text{Risk} &= (\text{Severity of mishap consequences}) \times (\text{Expected number of mishaps}) \\ &= (\text{Severity}) \times (\text{Expected mishap frequency per unit of exposure}) \times (\text{Amount of exposure})\end{aligned}$$

Such indexing allows prioritization of hazards based on the *relative* level of risk. For instance, severity-probability combinations can be marked as unacceptable (i.e., requiring redesign), undesirable, conditionally acceptable, or acceptable. Common sense dictates that the allowed probability of a particular hazard be inversely proportional to the associated severity. The probability classes are usually not linear, e.g., adjacent classes  $[10^{-5}, 10^{-3})$  and  $[10^{-3}, 10^{-1})$  both span 2 orders of magnitude but one has a size of 0.00099 and the other of 0.999. The same applies to severity classes [Krantz71]. Hence, the entries in the risk index do not represent constant risk amounts. Figure 3-3 illustrates the components of real risks, and how they are reduced and combined into to residual risks.



**Figure 3-3** *Components of Risk*

### 3.2.4 Failure Effect Controls

**Controls** are the means and mechanisms that are in place (or are required) to reduce the risks of system failure to an acceptable level. Some control mechanisms act on the hazard itself, whereas others control the losses or damage:

- System architecture and design that contains a backup or other form of redundancy, or includes a failure path that directs failures in a way that limits the safety impact. E.g., rip-stop textiles and similar techniques applied in mechanical and civil engineering objects.
- Warning devices. They annunciate impending or actual hazards to user-systems or the operator.
- Safety devices. These contrivances are intended to prevent injury, damage, or hazardous operation. They contain or absorb the harmful energy, prevent build up of dangerous energy levels, form a barrier for that energy, or divert it away from vulnerable targets. E.g., pressure relief valves, protective shrouds in jet engines, radiation containment buildings, fuses, protective clothing, interlocking switches on heavy machinery, and brakes. These devices must be checked regularly for latent faults.
- Standard and emergency operating procedures, and training. These are typically not acceptable as the sole means of control for catastrophic hazards.

---

### 3.2.1      Failure Accountability

---

Failure **accountability** indicates who has to fix the failed system, or has to “pay” for or accepting or correcting the damage ensuing from system failure, whether this damage is physical, mental, environmental, or financial. This may involve contractual obligations for the assumption of warranty and repair costs by the system manufacturers who guarantees a certain minimum or average fault-free operating period. The accountability often depends on actual or perceived negligence and, in certain societies, on which associated person or business entity has a large financial basis or “deep pockets”.

### 3.4 REFERENCES AND BIBLIOGRAPHY

- [ACAMJ25] FAA/JAA Advisory Circular/Advisory Material Joint AC/AMJ 25.1309: “*System Design and Analysis*”, Draft Diamond Revised, April 1998, 39 pp. [25.1309-1b to be published soon]
- [Anderson81] Anderson, T., Lee, P.A., pp. 52-58 of “*Fault Tolerance, Principles and Practice*”, Prentice-Hall, 1981, 369 pp., ISBN 0-13-308254-7
- [ARP4761] Section 2.2 “*Definitions*” of SAE Aerospace Recommended Practice ARP4761 “*Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*”, Society of Automotive Engineers (SAE), December 1996, 331 pp.
- [Avizienis85] Avizienis, A.: “*The N-version approach to fault-tolerant software*”, IEEE Trans. on Software Engineering, Vol. SE-11, No. 12, December 1985, pp. 1491-1501
- [Avizienis86] Avizienis, A., Laprie, J.-C.: “*Dependable Computing: From Concepts to Design Diversity*”, Proc. of the IEEE, Vol. 74, No. 5, May 1986, pp. 629-638
- [Avizienis87] Avizienis, A.: “*A Design Paradigm for Fault-Tolerant Systems*”, Proc. 7<sup>th</sup> AIAA/IEEE Digital Avionics Systems Conf. (DASC), Washington/DC, November 1987, pp. X-Y
- [Avizienis99] Avizienis, A., He, Y.: “*Microprocessor Entomology: A Taxonomy of Design Faults in COTS Microprocessors*”, pp. 3-23 of “*Dependable Computing for Critical Applications – Vol. 7*” (Proc. of the 7<sup>th</sup> Int’l IFIP Conf. On Dependable Computing for Critical Applications, San Jose/CA, Jan. 1999), October 1999, 424 pages, ISBN 0-7695-0284-9
- [Barborak93] Barborak, M., Malek, M., Dahbura, A.: “*The Consensus Problem in Fault-Tolerant Computing*”, ACM Computing Surveys, Vol. 25, No. 2, June 1993, pp. 171-220
- [Butler93] Butler, R.W., Finelli, G.B.: “*The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software*”, IEEE Trans. on Software Engineering, Vol. SE-19, No. 1, January 1993, pp. 3-12
- [Ciemens81] Ciemens, P.L.: p. 12 of “*A Compendium of Hazard Identification & Evaluation Techniques for System Safety Application*”, Sverdrup Technology, Inc., November 1981
- [Dolev83] Dolev, D., Lynch, N.A., Pinter, S.S., Stark, E.W., Weihl, W.E.: “*Reaching approximate agreement in the presence of faults*”, Proc. 3<sup>rd</sup> IEEE Symp. on Reliability in Distributed Software and Database Systems, Clearwater Beach/FL, Oct. 1983, pp. 145-154; Also: J. of the ACM, Vol. 33, No. 3, July 1986, pp. 499-516
- [DSVH89] Glossary of “*Digital Systems Validation Handbook – Volume II*”, DOT/FAA/CT-88-10, February 1989
- [Dunn86] Dunn, W.R.: “*Software Reliability: measures and effects in Flight Critical Digital Avionics Systems*”, Proc. 7<sup>th</sup> AIAA/IEEE Digital Avionics Systems Conf. (DASC), Fort Worth/TX, October 1986, pp. 664-669
- [EIA632] “*Processes for Engineering a System*”, ANSI/EIA-632-98, Electronic Industry Alliance, January 18, 1999
- [Evans99] Evans, R.A.: “*The Language of Statistics & Engineering*”, Proc. 1999 Annual Reliability and Maintainability Symp. (RAMS), Washington/DC, January 1999, pp. xi-xii
- [Fogel63] Fogel, L.J.: “*Biotechnology: Concepts and Applications*”, Prentice-Hall, 1963, 826 pp., Library of Congress Nr. 63010246/L/r83
- [Fuller95] Fuller, G.L.: “*Understanding HIRF - High Intensity Radiated Fields*”, Avionics Communications, Inc., 1995, 123 pp., ISBN 1-885544-05-7
- [Donchin95] Donchin, Y., Gopher, D., Olin, M., Badihi, Y., Biesky, M., Sprung, C., Pizov, R., Cotev, S.: “*A look into the Nature and Causes of Human Errors in the Intensive Care Unit*”, Critical Care Medicine, Vol. 23, No. 2, pp. 294-300
- [Gordon91] Gordon, A.M.: “*A practical Approach to Achieving Software Reliability*”, Computing & Control Engineering Journal, November 1991, pp. 289-29
- [Hecht86] Hecht, H., Hecht, M.: “*Software Reliability in the System Context*”, IEEE Trans. on Software Engineering, Vol. SE-12, No. 1, January 1986, pp. 51-58
- [Hughes89] Hughes, D., Dornheim, M.A.: “*United DC-10 crashes in Sioux City, Iowa*”, Aviation Week & Space Technology, 24 July 1989, pp. 96, 97
- [Hughes95] Hughes, D., Dornheim, M.A.: “*Accidents direct focus on cockpit automation*”, Aviation Week & Space Technology, January 30, 1995, pp. 52-54
- [Icarus94] Icarus Committee of the Flight Safety Foundation: “*The Dollars and Sense of Risk Management and Airline Safety*”, Flight Safety Digest, Vol. 13, No. 12, December 1994, pp. 1-6
- [IEC61508] “*Functional Safety of Electrical/Electronic Programmable Electronic Safety-Related Systems*”, International Electrotechnical Commission (IEC), Geneva, 1998

- [IFIP10.4]      Working Group 10.4 “*Dependable Computing and Fault Tolerance*” of Technical Committee 10 “Computer Systems Technology” of the International Federation of Information Processing (IFIP); <http://www.dependability.org/wg10.4/>
- [Knight86]      Knight, J.C., Leveson, N.G. “*An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming*”, IEEE Transactions on Software Engineering, Vol. SE-12, No. 1, January 1986, pp. 96-109
- [Knight90]      Knight, J.C., Leveson, N.G., “*A Reply to the Criticisms of the Knight and Leveson Experiment*”, ACM Software Engineering Notes, Vol. 15, No. 1, January 1990, pp. 24-35
- [Krantz71]      Krantz, D.H., Luce, R.D., Suppes, P., Tversky, A.: “*Foundations of Measurement, Volume 1: Additive and Polynomial Representations*”, Academic Press, 1971, ISBN 0124254012
- [Lala94]      Lala, J.H., Harper, R.E.: “*Architectural principles for safety-critical real-time applications*”, Proceedings of the IEEE, Vol. 82, No. 1, January 1994, pp. 25-40
- [Lamport82]      Lamport, L., Shostak, R., Pease, M.: “*The Byzantine Generals Problem*”, ACM Trans. on Programming Languages and Systems, Vol. 4, No. 3, July 1982, pp. 382-401
- [Laprie85]      Laprie, J.-C.: “*Dependable Computing and Fault Tolerance: Concepts and Terminology*”, Proc. 15<sup>th</sup> Fault Tolerant Computing Systems Conf. (FCTS), city, month, 1985, pp. 2-11
- [Laprie91]
- [Lardner34]      Lardner, D.: “*Babbage’s Calculating Engines*”, Edinburgh Review, No. CXX, July 1834; also: pp. 174-185 of Chapter 4 of “*Charles Babbage and his Calculating Engines*”, Morrison, P., Morrison, E. (Eds.), Dover Publications, Inc., Lib. Of Congress Nr. 61-19855
- [Leveson95]      Leveson, N.G.: “*Safeware: System Safety and Computers*”, Addison-Wesley, 1995, 704 pp., ISBN: 0-201-11972-2
- [Leveson97]      Leveson, N.G., Palmer, E.: “*Designing Automation to Reduce Operator Errors*”, Proc. of the 1997 IEEE International Conference on Systems, Man and Cybernetics, Orlando/FL, October 1997, 7 pp.
- [Littlewood92]      Littlewood, B., Strigini, L.: “*The Risks of Software*”, Scientific American, November 1992, pp. 62-75
- [Littlewood93]      Littlewood, B., Strigini, L.: “*Validation of Ultrahigh Dependability for Software-based Systems*”, Communications of the ACM, Vol. 36, No. 11, November 1993, pp. 69-80
- [McElvany91]      McElvany-Hugue, M., “*Fault type enumeration and classification*”, ONR Report ONR-910915-MCM-TR9105, November 11, 1995, 27 pp.
- [McGough83]      McGough, J.: “*Effects of Near-Coincident Faults in Multiprocessor Systems*”, Proc. of the 4<sup>th</sup> IEEE/AIAA Digital Avionics Systems Conf. (DASC), Seattle/WA, Oct.-Nov. 1983, pp. 16.6.1-16.6.7
- [McGough89]      McGough, J.: “*Latent Faults*”, Chapter 10 Digital Systems Validation Handbook - Vol. II, DOT/FAA/CT-88/10, February 1989, pp. 10.1-10.37
- [Meissner89]      Meissner, C.W., Dunham, J.R., Crim, G. (Eds): “*NASA-LaRC Flight-Critical Digital Systems Technology Workshop*”, NASA Conference Publication CP-10028, April 1989, 185 pp.
- [Mulazzani85]      Mulazzani, M.: “*Reliability versus safety*”, Proc. IFAC SAFECOMP '85 Conf., Como/Italy, 1985, pp. 141-146
- [Nanya89]      Nanya, T., Goosen, H., “*The Byzantine Hardware Fault Model*”, IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems, Vol. 8, No. 11, November 1989, pp. 1226-1231
- [Pfleeger92]      Pfleeger, S.L.: “*Measuring software reliability*”, IEEE Spectrum, August 1992, pp. 56-60
- [Prasad96]      Prasad, D., McDermid, J., Wand, I.: “*Dependability terminology: similarities and differences*”, IEEE Aerospace and Electronic Systems Magazine, January 1996, pp. 14-20
- [Pullum99]      Pullum, L.L.: “*Software Fault Tolerance*”, Tutorial Notes of the 1999 Annual Reliability & Maintainability Symp. (ARMS), Washington/DC, January 1999, 22 pp., ISSN 0897-5000
- [Redmill97a]      Redmill, F., Dale, C. (Eds.): “*Life Cycle Management for Dependability*”, Springer-Verlag, 1997, 235 pp., ISBN 3-540-76073-3
- [Redmill97b]      Redmill, F., Rajan, J. (Eds.): “*Human Factors in Safety-Critical Systems*”, Butterworth-Heinemann Publ., 1997, 320 pp., ISBN 0-7506-2715-8
- [Risks]      “*The Risk Digest*”, on-line digest from the Forum On Risks To The Public In Computers And Related Systems, under auspices of the Association for Computing Machinery (ACM) Committee on Computers and Public Policy, Annual volumes since 1985; <http://catless.ncl.ac.uk/Risks/>
- [Roland90]      Roland, H.E., Moriarty, B.: Chapter 1 of “*System Safety Engineering and Management*”, 2<sup>nd</sup> edition, John Wiley & Sons, 1990, 367 pp., ISBN 0-471-61861-0
- [Rushby93]      Rushby, J. “*Formal Methods and the Certification of Critical Systems*”, Computer Science Lab. of SRI Int'l Tech. Report CSL-93-7, Dec. 1993; also published as: “*Formal Methods and Digital Systems Validation for Airborne Systems*”, NASA Contractor Report CR-4551

- [Shin87] Shin, K.G., Ramanathan, P.: “*Diagnosis of Processors with Byzantine Faults in a Distributed Computing System*”, Proc. IEEE 17<sup>th</sup> Int'l Symp. on Fault-Tolerant Computing (FTCS), Pittsburg/PA, July 1987, xx pp.
- [Shooman93] Shooman, M.L.: “*A study of occurrence rates of EMI to aircraft with a focus on HIRF*”, Proc. 12<sup>th</sup> AIAA/IEEE Digital Avionics Systems Conf. (DASC), Seattle/WA, October 1993, pp. 191-194
- [Siewiorek92] Siewiorek, D.P., Swarz, R.S. (Eds.): Chapter 2 in “*Reliable Computer Systems - Design and Evaluation*”, 2<sup>nd</sup> ed., Digital Press, 1992, 908 pp., ISBN 1-55558-075-0
- [Storey96] Storey, N.: p. 123 of “*Safety-critical computer systems*”, 1996, Addison-Wesley, 453 pp., ISBN 0-201-42787-7
- [Sullivan92] Sullivan, M.S., Chillarege, R.: “*A Comparison of Software Defects in Database Management Systems and Operating Systems*”, Proc. 22<sup>nd</sup> Int'l Symp. On Fault Tolerant Computing (FTCS), Boston/MA, July 1992, pp.475-484
- [Voges88] Voges, U. (Ed.): “*Software Diversity in Compurized Control Systems*”, (Dependable Computing and Fault-Tolerant Systems Vol. 2), Springer-Verlag, 1988, 216 pp., ISBN 0-387-82014-0
- [Walter88] Walter, C.J.: “*MAFT: an Architecture for Reliable Fly-By-Wire Flight Control*”, Proc. 8<sup>th</sup> Digital Avionics Systems Conf. (DASC), San Jose/CA, October 1988, 7 pp.
- [Walter95] Walter, C.J., Monaghan, T.P.: “*Dependability Framework for Critical Military Systems Using Commercial Standards*”, presented at the 14<sup>th</sup> AIAA/IEEE Digital Avionics Systems Conf. (DASC), Boston/MA, November 1995, 6 pp.
- [Wichman93] Wichman, B.A.: “*Microprocessor Design Faults*”, Microprocessors and Microsystems, Vol. 17, No. 7, 1993, pp. 399-401
- [Wiener88] Wiener, E.L., Nagel, D.C.: “*Human Factors in Aviation*”, Academic Press, 1988, 684 pp., ISBN 0-12-750031-6
- [Yount85] Yount, L.J.: “*Generic Fault-Tolerance Techniques for Critical Avionics Systems*”, Proc. AIAA Guidance, Navigation and Control Conf., Snowmass/CO, August 1985, 5 pp.
-