Version: 12 September 2003

3 DEPENDABILITY IMPAIRMENTS: FAULTS, ERRORS, AND FAILURES

Please send comments & suggestions to
Frank Dörenberg via email
textbook@nonstopsystems.com
For research and educational purposes, an electronic copy
of most of the referenced articles is available on request.

3 DEPENDABILITY IMPAIRMENTS

The purpose of any system is to provide its intended, expected, and agreed upon-service (i.e., functions, operations, and behavior). If the system does this in an acceptably correct and timely manner, it is called *dependable*. The transition to a state in which the system is unable to deliver its service in such manner, is referred to as a **failure** or **outage**.

Failures are the manifestations of this inability *at the system boundary*, i.e., *at the service interface* with the users. Obviously, these manifestations are caused by effects and events that are present or occur *within the system*. They may also enter into the system via specifications that do not adequately capture that intended, expected, and agreed upon-service. All these effects and events in effect pose prevent the system from being and staying dependable. Hence, they are collectively referred to as **impairments** [Laprie85, IFIP10.4, Prasad96, Avizienis87/01], and form the second branch of the dependability tree, see Figure 3-1.

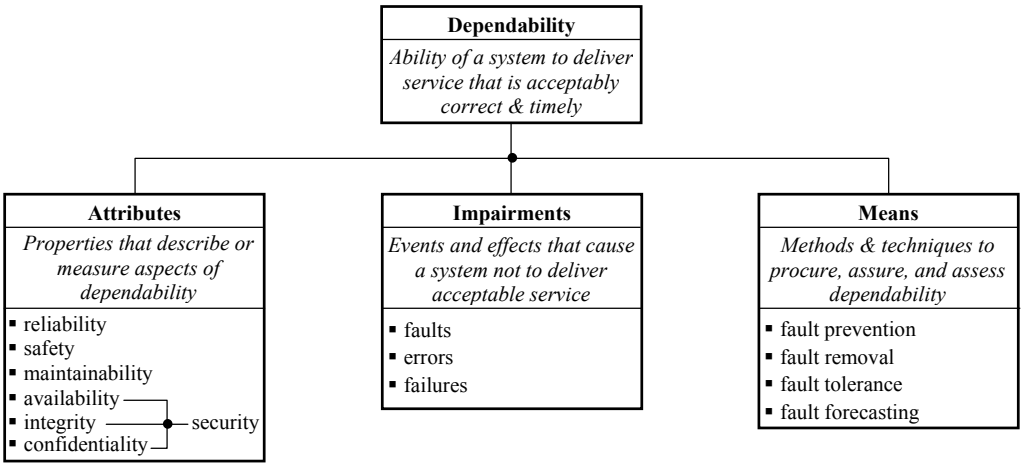


Figure 3-1 ***Impairments: the second branch of the dependability tree***

As the architects, developers, designers, assessors, and users of systems, we need to be able to establish, evaluate, discuss, and communicate the hierarchical, causal relationships between these impairments [Avizienis87]. We may need to know how they enter into the system, how they are activated, and how they propagate to the system's service interface. We also need to be able to specify the system's dependability goals and requirements. This includes the number and type of impairments that the system must provide protection against. Also, without insight into how the impairments occur and develop, it is hard to systematically evaluate means to prevent them, reduce their severity, or model them, both quantitatively and qualitatively.

Obviously, we need a hierarchical framework of terminology and classification of impairments. As known intuitively and from experience, undesirable events are typically the end result of a chain of multiple events. System failure is such an undesirable event. A standard way of decomposing the causal chain that leads up to this top-level failure event is: fault $\Rightarrow$ error $\Rightarrow$ failure [Laprie85/90, Avizienis86, Meissner89, IEC61508]. This relationship is illustrated in Figure 3-2. Many definitions of these terms exist, and they are sometimes used interchangeably.

All devices, components, and other system resources (incl. humans) have an internal state, value, operational mode, or condition. We speak of an **error**, if this state deviates from the one that is correct or desired. The latter is based on specification, computation, observation, theory, or whatever determines "customer satisfaction". A system transitions to its “failed” state, when an error prevents it from delivering the required service performance. I.e., a failure is the system-level manifestation of an error: the error has propagated to the service interface of the system,

and affects the system's users and environment. Conversely, a failure does *not* occur, as long as the error does not propagate to the system boundary. The term “error” is sometimes reserved for software, as well as associated data items and structures.

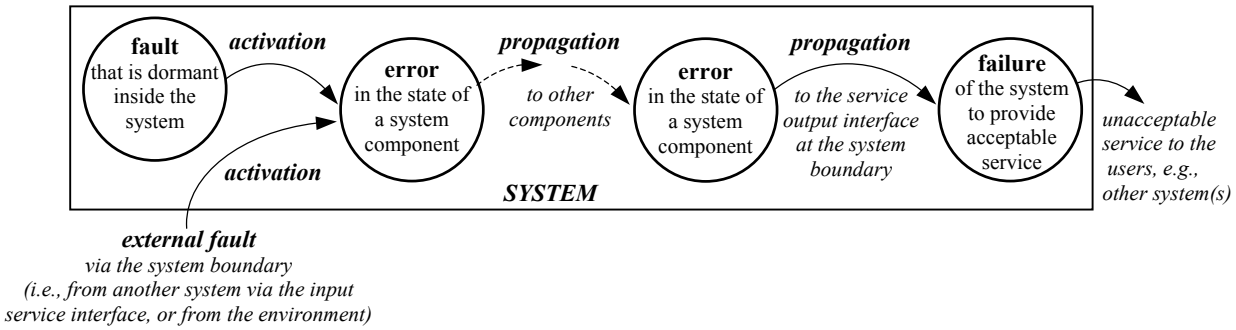


Figure 3-2 ***The causal relationship between faults, errors, and failures***

The flaw, whether hypothesized or actually identified is the cause of an error. This adjudged cause is referred to as a **fault**. It can occur within *any* subsystem or component. This applies equally in the domains of software, electronic and mechanical hardware, optical, chemical, and nuclear systems or system elements. Faults can be seemingly small, but the associated error(s) and failure effects large. E.g., a simple inadvertent bit-flip caused by electromagnetic interference in a gigabyte memory could change a positive bank account balance into a negative balance, or change the direction of movement of control rods in a nuclear power plant. If a single fault causes a failure, we speak of a **single-point-failure** (SPF). This primarily of concern in systems with redundancy.

Example 3-1 (hardware)

Cosmic radiation causes a bit-flip in a cell of an SRAM-based Field Programmable Gate Array (FPGA) that implements a flip-flop function. As a result, the Boolean output of that flip-flop is stuck. This fault is activated as soon as the flip-flop's clock signal attempts to transfer a changed flip-flop input to that output. The fault produces an incorrect flip-flop state, which is an error. In turn, the flip-flop output is interpreted by software as flag that indicates imminent collision with a nearby vehicle, and eventually an alert is asserted to the vehicle's driver. I.e., this inadvertent alert appears at the system's service interface: a failure has occurred. *Note*: per our definition, that same bit-flip represents a failure at the level of the cell, if the flip-flop is considered as a system by itself rather than as a component.

Example 3-2 (software)

Software: p 14 of 21 pp. Fundamental Concepts of Dependability; Therac coding fault, error, failure.

Example 3-3 (interaction)

Interaction: p 14 of 21 pp. Fundamental Concepts of Dependability “inappropriate human-system interaction by an operator during the operation of the system is an external fault (from system point of view); the resulting altered processed data is an error, etc.” “fault in operating manual Some to be prevented/handled, others

Example 3-4 (liveware)

A maintenance person cuts her hand on the sharp corner of a computer box, and punctures a blood vessel. This physical change is a fault and causes the bio-system to lose vital fluids, which is an error. The blood coagulates at the puncture and seals it. This

containment prevents exsanguination. So, no failure occurs, though the bio-system may show a temporary performance reduction or limitation.

Example 3-5 (specification)

A customer forgets to explicitly specify that the units to be used in its thruster control system software must be standard metric kgms (kilogram, meter, second). However, as on some previous programs, the supplier uses units in archaic British fps (feet, pound, second). As a result, the control software expects input data for force control in *feet pounds* instead of *newtons* (a factor of appr. 4.5 difference). Due to schedule and cost pressures, this fault is not caught during design reviews, simulations, or testing. As a result, the Mars Climate Observer interplanetary space probe of the customer (NASA) is off by a catastrophic 100 km after the 500 million km voyage, and disintegrates in the Martian atmosphere [Oberg99].

Example 3-5 (security)

Attack, external fault → vulnerable system, default username/password combination "guest/guest", internal active fault → intrusion = failure, if compromised security / system depends on it

---

Any item or system comprises multiple interacting sub-systems or components. At the same time, a system itself can be considered as a sub-system or component of a larger super-system. This depends on how far we zoom into, or out of, the system under discussion. Hence, failure of a sub-system appears as an external fault to a user system at the next level of the system hierarchy, etc. This chain effect is illustrated in Figure 3-3. How far the iterative decomposition should be carried through, depends on the system context and the reasons for investigating the causal relationships. The tracing must contribute to understanding and …beyond which control over root cause (physical) through design, component selection, process,

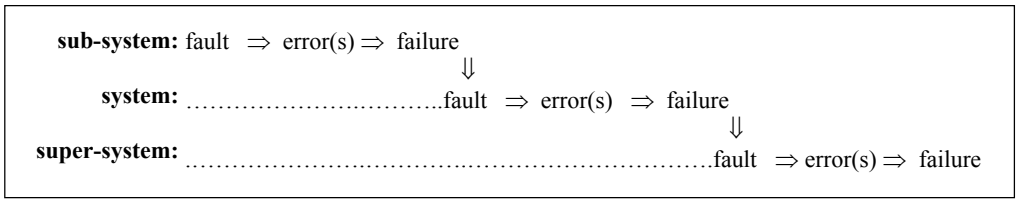


Figure 3-3 ***Causal relationship between faults, errors, and failures***

As outlined above, a failure is the result of an error, which in turn is caused by a fault. The inverse 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. A corrupted memory location or register may be overwritten with a correct value before the corrupted data is used or takes effect. Sometimes the faulty state is (temporarily) the same as the correct state: even a broken clock tells the correct time twice a day.

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. In other words: ***failure-free* does not imply *fault-free***. 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.

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 [James00]. As illustrated in Figure 3-4, the life-cycle covers system development and manufacture, through installation, operation, maintenance, and disposal.

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 for the operating environment. Other forms of environmental interference are caused by contaminants such as corrosive or conductive fluids or vapor, sand or dust, and power or temperature fluctuations. Another source of faults is the interaction with other systems, whether physical or via data. Human induced faults can be either inadvertent, or deliberate (whether with malicious intent or not) [Martin01].

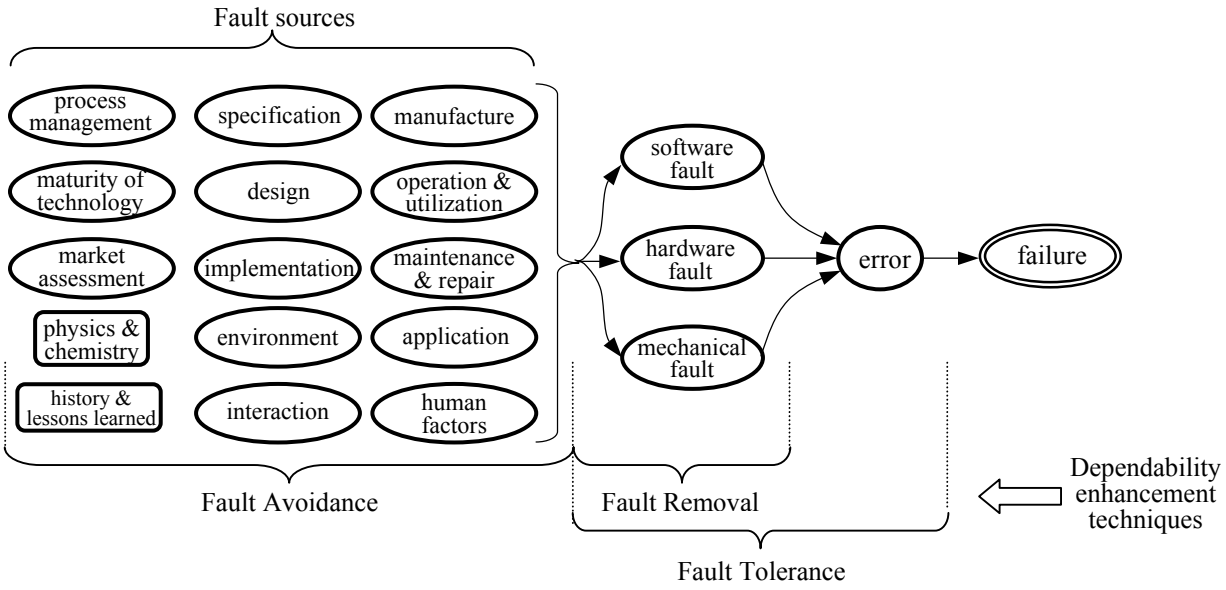


Figure 3-4 ***Dependability means related to impairments ref this fig***

Faults in the software domain, often referred to as "bugs"[1], are related to the generation and execution of software 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 software related fault and failure cases is captured in [Risks]. They are generally dependent on data values, data sequences, timing [Storey96], or hardware resource allocation and management thereof. 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 actually a *physical*

[1] The first real *computer* bug (a moth caught in a relay) was found in 1945 in the Harvard Mark II computer by Grace Hopper, one of the original programmers. She glued the bug into her logbook, which is now in the collection of the Smithsonian National Museum of American History in Washington/DC. The term appears to have been in use to denote a technical problems well before that event, e.g., the Oxford English Dictionary cites the following 1889 newspaper quotation: "*Mr. Edison, I was informed, had been up the two previous nights discovering 'a bug' in his phonograph.*"

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.

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, limited coverage of testing and debugging, 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].

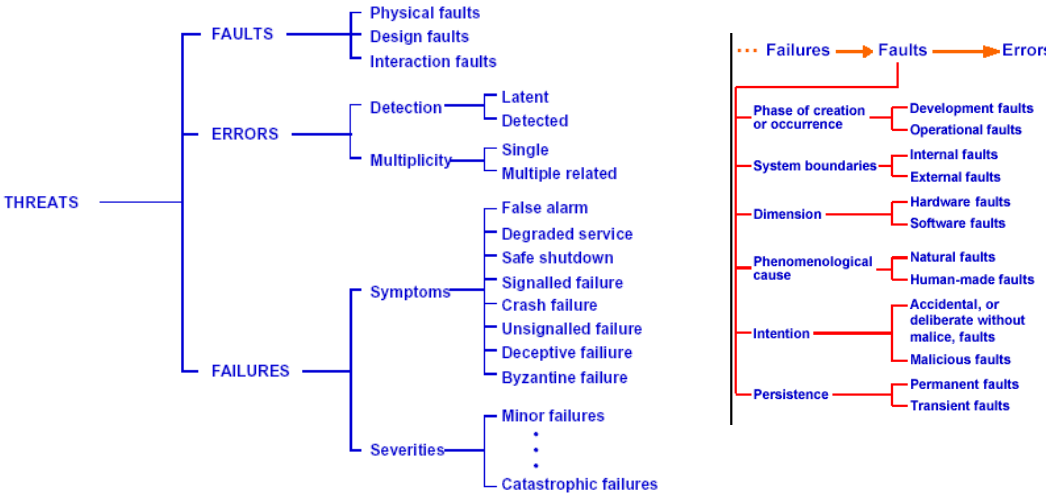
Many types of these impairments can occur during the life of a system. They must be identified and classified, together with:

- how, and under which conditions they enter into the system
- where in the (sub)system in each of them occurs, or can occur
- if, and how they propagate through the system hierarchy
- what effect they have at the system boundary and beyond.

It is important to determine the scope of all the fault types that must be considered during system specification and design or evaluation. There are many methods and techniques for the prevention, containment, detection, removal, and handling of faults, recovery from them or tolerating them. The coverage and efficacy (and cost) of these approaches is highly dependent on the fault type and the number of sequential or simultaneous faults that must be accommodated.

FAULT CLASSIFICATION

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 deal with this, we must aggregate fault types into manageable classes or categories, each representing a major fault **attribute** or **model**. Table 3-1 shows a general fault classification based on commonly used attributes [Avizienis76/86, Laprie90, McElvany91, Powell92]. 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, Transient, Intermittent
Consistency	Symmetrical, Asymmetrical = failure?
Intent	Benign, Malicious, Byzantine, Accidental, Intentional
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
Diagnosability	Diagnosable, Not Diagnosable
Coincidence	Distinct, Coincident, Near-Coincident
Phase of creation	Development, Manufacture, Operation, Support
Source	Hardware, Software, Human Interface, Interaction; Physical, Human-Made (= all)
Origin	Internal, External
Mechanism	Physical processes

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.

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 it has adverse effects, and its manifestations can be detected. This may be through direct observation at the faulty system resource, or be inferred from erroneous sub-system behavior and system performance. Note that the same fault may have very different activation time, depending on the system. E.g., a bad memory location may be accessed several times a second, or once a year.

Detection does not imply that the faulty element has been identified and located, only that its presence has been noted (e.g., as reflected in an error flag, signal, or message). 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 a system [Somani94], 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]. However, most detection and monitoring schemes must apply a certain amount of confirmation delay (persistence time or cycles) so as not to cause nuisance alerts.

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 and is stable. 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 sub-system. 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. Examples of hard faults are broken drive belts, burnt-out light bulbs, and crashed hard-drive heads, and software bugs. 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, Sosnowsky94]. 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 temporary electromagnetic interference, electrical power drops or surges, mechanical shock, or irradiation [Norman96, Ohlsson98, Cataldo01, Baumann01]. 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.

Many applications include fault diagnostic functions that detect faults and identify (i.e., "isolate") the faulty system resource. It is important that such diagnostics be able to discriminate between permanent and transient faults [Pizza98]. Often a different response is specified for these two types. Making no distinction, or incorrectly declaring a transient fault "permanent" (or vice versa), has implications for system operating and maintenance costs: replacing a basically healthy resource that checks out OK when tested, unnecessarily interrupting revenue service, running the risk of exhausting system resources such as spares that are still considered "not permanently failed", inadvertently operating with a resource that is truly faulty (even though healthy spares may be available), etc.

A recurring transient fault is called **intermittent**. Such faults are only present occasionally and inconsistently [Anderson81]. I.e., they occur irregularly and infrequently with respect to the mission duration, and are hard to reproduce. They are often 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 an integrated circuit).
- a 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.
- ever increasing processor clock-speeds, reducing noise margins
- hardware and software states that vary as a function of processing load or activity.

Software itself cannot fail intermittently, but marginal software process timing can cause intermittent faults, as can hardware resource allocation problems. Intermittent faults are often hard to trouble shoot due to their seemingly elusive, inconsistent, and unpredictable occurrence. Intermittent errors can result from intermittent faults, or from hard faults that have activation trigger conditions that occur intermittently. More complex systems often include extensive logging of the fault circumstances (a snapshot of system state, mission phase, internal temperature and voltages, the value of control inputs and outputs, etc.) to help correlate the fault to its cause.

Fault Consistency

Fault-tolerant computer architectures usually comprise three or more computing nodes. They can be configured such that each node monitors the others, and their opinions are consolidated for the purpose of reaching consensus on the identification and exclusion of a faulty node. Such

decisions require that the fault manifests itself in the same way to all[2] observing nodes. This is referred to as **fault consistency**.

A fault is **symmetrical** if its manifestations are seen *identically* by all non-failed observers. Conversely, a fault is called **asymmetrical** if it produces different (i.e., inconsistent) symptoms for different observers or users. This may lead to different conclusions and actions in those users.

A typical example of asymmetry is a data transmitter whose output is fanned out to several receivers, effectively creating a number of transmitter-receiver pairs. Individually, the transmitter and receivers may operate within their respective specifications, but combined (end-to-end) this is not necessarily the case for all pairs. Similarly, asymmetry can result when several independent data acquisition units (sensors) measure the same physical parameter and send their measurements to multiple voters. If a sensor sends different values to different voters, the voter outputs will be inconsistent.

Fault Intent

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 that would be 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].

Depending on the system, it may be 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, or damage to the system during any life-cycle phase. These intrusions may be malicious, for the explicit purpose of causing faults and their consequences. Examples of such faults are benign or malicious computer viruses, jamming of radio transmissions or of network traffic (e.g., denial-of-service attacks), and the destructive acts of vandals, terrorists, and otherwise disgruntled, disturbed, or hostile persons (justifiably or not). Such faults, their sources, and protection against them, is the domain of system security [Martin01]. This is further discussed in Chapter "Security"

[2] depending on the system architecture, it may suffice that a *majority* of healthy nodes agree, rather than *all*. In a redundant system with a sufficient number of nodes, schemes for multiple rounds of data exchange and voting can be implemented such that asymmetrical and even arbitrary ("byzantine") faults can be handled. See section ###.

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 whether they are invoked by the same cause, and whether they occur at the same time. See sections 3.1.7 (Fault Cause) and 3.1.11 (Fault Coincidence). Both of these attributes determine if the multiple faults can be treated as separate, statistically unrelated single faults.

Fault Cause (single fault)

The **fault cause** indicates whether the occurrence of a fault is random or a deterministic event. By definition, **random faults** are non-deterministic. Occurrences of such faults have a statistical distribution. The term "random" is used to label events, such as measurement of time to failure, that can result in different outcomes, even though it repeated in the same manner every time. In the end, every variation, failure, event, and uncertainty has an assignable root cause that can be identified if sufficient effort is expended[3] [Evans99]. 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 actually wear out, though degrading of performance over time may be observed. This is called software aging, see Section 3.2.11 "Fault Source". If software has a fault, then that fault is always present, whether observable or not. The related error is produced deterministically, each time the fault is activated or triggered by a particular data value, data sequence, timing with respect to other processes, or access to hardware resources. However, as trigger conditions occur randomly, software failures may appear to happen randomly as well.

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, Butler93]. What is generally meant by "reliable" software, is that each software 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.

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

[3] "*Nothing in nature is random… A thing appears random only through the incompleteness of our knowledge*", Baruch Spinoza, Dutch philosopher 1632-1677

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 Appendix “”Probability and Statistics”. 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-cause or common-mode faults.

Impact [Tang92]

- Significantly degrade availability, reliability, and transient performance.
- Single failure tolerance is not enough.
- Models assuming failure independence are not appropriate.
- Partial coverage models need to be modified.

[Lala94]
[Buchner94] Buchner, H.: "Occurrence of common mode failure", Reliability Engineering and System Safety, Vol. 45, No. 1_2, 1994, pp. 201-204

Common-cause or **common-mode faults** (CMFs) occur if a *single* condition or event causes a fault in *multiple* system elements. The invoked faults need not affect components that are identical or redundant, nor is it necessary that they happen exactly at the same time, though it is often assumed that their occurrences overlap.

The common fault causes can be very diverse, e.g.:

- Common requirements and design
- Common environment (internal or external): vibration, pressure, cooling, electromagnetic interference (EMI), irradiation, explosion or fire, contamination by fluids, uncontained rupture of rotating parts or of a pressure vessel, etc.
- Common, shared, or interconnected hardware: a processor and memory, time base or clock, synchronization mechanism, data bus, or power supply
- Common operating system on a shared computing platform
- Common, shared, or interconnected application software
- 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
- Common routing of redundant network lines (or control cables[4]) through the same conduit.

This type of fault can have serious consequences, if it causes an error *and* prevents detection of that error. 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 and type of faults. This redundancy may be exhausted at once, if a single cause affects several or all of those resources.

Example 3-6 (hardware)

---

[4] this was the case in the original McDonnel-Douglas DC-10 aircraft design.

The rotor disk of a jet engine disintegrates and the shrapnel severs all supply lines of the aircraft's redundant hydraulic system [Hughes89].

Example 3-7 (cascading)

A320: triple hydraulic, dispatch with one out, pump on second system fails, power transfer motor/pump (no exchange of fluid) overloads

Example 3-7 (environmental)

A320: batch of temperature sensitive components end up installed in duel redundant control computer subject to the same environmental conditions; first one fails, reversion to second box, that fails shortly thereafter.

A special type of common-cause fault is the **generic fault** [Yount85, Poledna96]. 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 concurrently 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 providing protection against generic faults and their consequences, is to ensure that redundant resources fail independently: by having different (diverse) faults, errors, and failures. This is generally achieved by applying **design diversity**, also known as **dissimilarity**. This concept dates back about 175 years (!) to the era of Charles Babbage's calculating machine[5]. In the software domain dissimilarity is usually referred to as **N-version** programming [Avizienis85/95, Voges88, Poledna96, add refs]. 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. Care must be taken not to compromise version-independence when implementing mechanisms to limit divergence between versions, especially as differences between non-failed versions may be legitimate. The level of achievable version-independence is open for debate [Knight86, Knight90].

Caveat

The power of dissimilarity is based on the assumption that that it makes simultaneous common-mode (*generic*) faults extremely (or at least sufficiently) improbable. However, it must be realized that dissimilarity provides little or no advantage in protecting against common-mode environmental faults (interference, temperature, vibration, power, etc.), and flaws in common requirements that drive the design and implementation. Dissimilarity also does not reduce the probability of simultaneous *random* faults, whether version-independent or not. It may allow a shift away from proving absence of generic faults, to demonstrating the ability to handle them. Such an architectural approach may lead to a reduction in the required product certification and design assurance level. Mechanisms put in place to ensure consistency, convergence and tracking between dissimilar versions must be carefully evaluated for their effect on version-independence.

---

5 "*The most certain and effectual check upon erors which arise in the process of computation, is to cause the same computations to be made by separate and independent computers; and this check is rendered still more decisive if they make their computations by different methods*" [Lardner34].

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 impractical, 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, 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
- different Hardware Design Languages (HDLs) for identical or different programmable logic devices (PLDs)
- 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
- generation of aerodynamic control forces via airplane control surface deflection vs. engine thrust vectoring
- data bus with copper wire vs. optic fiber (electromagnetic interference, cross-coupling)
- different algorithms for the same function
- integrated circuits with same form, fit, and function, but implemented with different technologies, cell libraries, or processes.
- different circuits and devices to implement the same function, e.g., discrete components vs. integrated circuits, or analog vs. digital
- different tools for requirements capture, design, development, implementation, test and verification
- AC electrical power from a public utility company vs. from a diesel generator vs. from a battery or solar powered inverter[6].
- Operating environment: environment diversity [refs]

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. [Kanoun01].

3.2.6 Fault Extent

The **fault extent** describes the scope of a fault: how far it propagates through the system, and which area of the system is affected. The boundary of this area is called the Fault Containment Region (FCR). It is typically desirable that this boundary be as close to the fault site as possible. Such containment often limits the (possibly damaging) effects of the fault, and also facilitates fault handling strategies such as redundancy and management thereof.

[6] A single lighning strike could damage the power connection to the public electricity network and to the backup generator. If this happens at an unmanned (telecom) facility. If the power situation is not monitored, or the monitors are not tested, the battery backup will be depleted and cause loss of the facility.

Faults can propagate in many ways. For instance, an electrical fault can propagate electrically, as data, thermally (increased dissipation or even fire), or physically (smoke, molten isolation or dielectric material), etc. Uncontained faults propagate *outside* the system boundary, to connected systems, users, or the system environment in general.

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** or **global**. Hence, the fault extent depends on the system view: a system is always a sub-system in a higher level system.

3.2.7 Fault Value

A faulty system may produce values (internally, or output) that do not fall within the set of specified or expected values. Similarly, it may have an erroneous state or status. The **fault value** indicates what this value is, and how it changes with time.

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

- an extreme value, e.g., stuck at logical level 1 or 0, the maximum position of an actuator, the supply voltage or electrical ground potential. This is also known to as a "hard-over", and it is generally assumed that a hard-over state or value is reached rapidly (with respect to the system's time-constants).
- the component's value at moment of the fault occurrence: the value is "frozen"
- some predetermined or default 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 fixed or arbitrary limits. The rate of change can be fixed or (erratically) varying in sign, magnitude, and rate.

3.2.8 Fault Observability and Diagnosability

A fault is **observable** or **visible** 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 this contributes to undependability 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 systems 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. E.g., A/D, D/A+MUX

Closely related to observability, is fault diagnosability.Does not necessarily mean it cannot be made diagnosable. WEBSEARCH
Def. ; vs observability vs isolation. Somani + ..
(Berlin) a [FT] system of *n* units is a one step *t*-diagnosable system if all faulty units within the [redundant] system can be located without replacement [of what?], provided the number of faulty units does not exceed *t*. [Preparata/Metze/Chien, 1967]: 2*t*+1 <= y, at least *t* units must test each of the *n* units.

SIAM Journal on Computing, Volume 23, Number 5, pp. 895-905
© 1994 Society for Industrial and Applied Mathematics
Diagnosis of **t/(t +1)**-Diagnosable Systems, `A. Das, K. Thulasiraman, V. K. Agarwal:` A classic PMC (Preparata, Metze, and Chien) multiprocessor system [F. P. Preparata, G. Metze, and R. T. Chien, IEEE Trans. Electr. Comput., EC-16 (1967), pp. 848--854] composed of n units is said to be **t/(t+1)**diagnosable [A. D. Friedman, A new measure of digital system diagnosis, in Dig. 1975 Int. Symp. Fault-Tolerant Comput., 1975, pp. 167-170] if, given a syndrome (complete collection of test results), the set of faulty units can be isolated to within a set of at most **t+1** units, assuming that at most **t** units in the system are faulty. This paper presents a methodology for determining when a unit **v** can belong to an allowable fault set of cardinality at most **t**. Based on this methodology, for a given syndrome in a **t/(t+1)**-diagnosable system, the authors establish a necessary and sufficient condition for a vertex v to belong to an allowable fault set of cardinality at most t and certain properties of **t/(t+1)**-diagnosable systems. This condition leads to an $O(n^{3.5}) t/(t+1)$-diagnosis algorithm. This **t/(t+1)**-diagnosis algorithm complements the **t/(t+1)**-diagnosability algorithm of Sullivan [The complexity of system-level fault diagnosis and diagnosability, Ph.D. thesis, Yale University, New Haven, CT, 1986]

3.2.9 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 another member of the same redundant set of monitored parameters or resources. 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.

3.2.10 **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., [EIA63, NASA, SAE, IEC, ISO9004]. They typically distinguish phases and activities corresponding to the five-phase listed 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 dependability and cost.

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.

The system development phase must pay close attention to the Human-Machine Interface (HMI), as the system will be installed, operated, and maintained by humans.3.2.12
[Strigini01] The reliability of diverse systems: a contribution using modeling of the fault creation process

The **manufacturing phase** includes the procurement, fabrication and screening of components, tools, and test equipment. The components 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 and vibration (so-called *shake-and-bake*), to weed out “weak” systems or components. Early in this 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 hardware, software, and mechanical 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., to solve component obsolescence when “drop in” replacement parts can no longer be economically procured) are also subject to development and installation faults [Kuhn97]. Experience with the fielded system can be fed back to any of the preceding phases to improve the system.

Eventually, all systems reach the end of their *useful life* (utility), *technological life* (obsolescence), or *economical life* (operation and support). They are then retired, decommissioned, discarded, or replaced, or otherwise permanently taken out of service. This is referred to as the **disposal** phase [ISO14040]. 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.

3.2.11 Fault Source

The **fault source** indicates whether a fault is a **hardware** fault, a **software** fault, or related to the **human interface** between the system and the operator, user, or maintainer. 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.

Generally, software is deemed reliable if it generates correct internal states and outputs, when it is supplied with valid inputs or sequences thereof, and that the response to invalid inputs is deterministic. Software is “correct” if it operates as specified. This, by the way, does not imply that the specification on which the software design is based, is valid and correct.

Software, as captured in executable code, does not degrade with age. It is time-invariant and does not wear out, *if left untouched*. However, the real-time service performance provided by the combination of the operating system, middleware[7], and application software may worsen over time [Eick01, Yurcik01]. This is called **software aging**. It is caused by progressive depletion and exhaustion of computing resources such as swap space and free memory, due to flawed dynamic allocations and de-allocations of those resources. Examples are:

- **memory fragmentation.** When a typical memory management function such as a processor’s Memory Management Unit (MMU) receives a memory allocation request, it assigns the next block of available contiguous memory that is large enough to fit the request. When that memory is de-allocated, it is released as a contiguous block. There are situations where many allocations and de-allocations of various block sizes are performed that overlap in time. This may result in a memory structure in which there are fewer and fewer large contiguous blocks available, even when the total amount of free memory is large: the memory is fragmented. If this happens, only small(er) allocation requests can be

[7] Middleware is an abstraction layer between the operating system and the application software, to mask heterogeneity of networks and hardware, especially in systems that are distributed (location, concurrency, replication, mobility, faults). This effectivey raises the Application Programming Interface (API) level, and may help avoiding faults in the aplication software. It may be used to mask heterogeneity of operating systems and programming languages as well [Bakken02].

satisfied, and the memory management function takes more and more time to find a sufficiently large block of free memory (e.g., on a hard disk).

- **memory leakage** occurs when an allocated memory block is not freed, even though it should have been released, if it will never be used again by the process to which it was originally allocated. E.g., dynamic memory is allocated to a program such as application code or a library. However for some reason, the associated allocated memory block is not released when the program is terminated by the operating system. Hence, the block cannot be allocated again. An example of this is an unreleased file-lock. If such leakage occurs repeatedly, the amount of unusable memory grows. "Garbage collection" techniques can be used to automatically recycle memory blocks for which it is certain that their continued allocation is no longer needed. This is done in languages such as Java and ADA95. Memory leakage is a special case of untimely release of allocated hardware resources, to the exclusion of access by other users. This so-called "hogging" can also happen with processor time, input/output peripherals or data bus access, etc.
- **false-sharing** can occur when multiple processors access (read or write) different words that are located in the same cache block. If a processor accesses the same data item again, it may no longer be in that cache block, due to an intervening access to that same cache block by another processor.

Memory fragmentation and leakage should not be confused with **bloating**: excessive use of memory and CPU time. This is often caused by "vapor ware": new versions of popular PC application software that tend to have many useless "new and improved" features, and each upgrade tends to further expand, so as to use up whatever computing resources are available, both storage space and processor throughput.

Trivedi.**Software failure due to resource exhaustion**:…The estimation of the rate of resource exhaustion and consequently the expected time of software failure has been the focus of research on "software rejuvenation" [Trivedi00, **refs**] techniques. Periodically restarting a process/rebooting a node, or doing a prediction-based rejuvenation based on the observed rate of resource exhaustion may help prevent the software from crashing (operating system, middleware, application). For more details please see ISSRE '98 paper and the ISSRE '99 paper.Software Aging: deterioration in the availability of OS resources, data corruption and numerical error accumulation. **Software Rejuvenation**: a proactive fault management technique aimed at cleaning up the system internal state to prevent the occurrence of more severe crash failures in the future. Though can become inadequate over time if utilization profile changes or functional demands change. Must be considered in embedded environment [unless "static software"] Cannot be viewed separately from hardware processing, esp. if real-time system. Must consider combined h/w & software [ref] + new faults introduced when "upgrading" or fixing bugs. (not properly done regression testing)

abstraction layer between the O/S and the application software

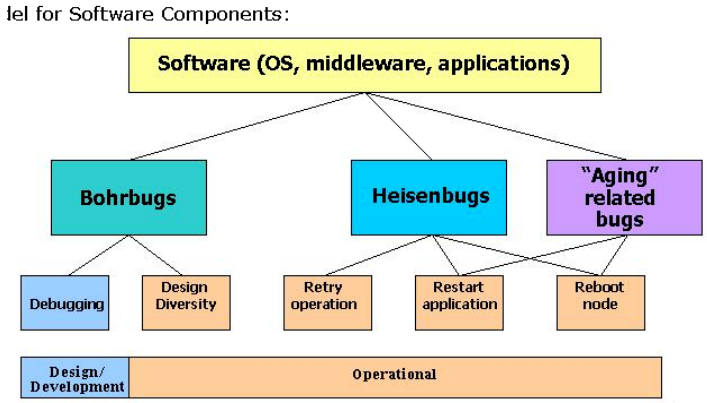
Software also does not fail randomly. However, software errors
This contrasts with user experience and often wellpublished occasions of software errors [ref]. Although software errors are essentially due to design and specification errors they frequently manifest themselves in an apparently random fashion and can be modeled as a stochastic process similar to hardware [Chapter 4 in Lyu]. Encountering those data/sequences.

Defect trigger distribution **Software faults**: a.k.a. "bugs".
Sometimes classified based on the elusiveness (reproducibility) of their activation conditions, . In analogy with phenomena in nuclear physics, a distinction is made between Heisen-bugs [Gray86] and Bohr-bugs.

Trivedi: software-reliability.com
Bohrbugs: Software faults should ideally have been removed during the debugging phase. Even though software may have been thoroughly tested, it still may have some design [+implementation] faults that are yet to be revealed. In fact, such faults may exist even in mature

software such as commercial operating systems. This is particularly true when software such as the operating system is composed of old as well as relatively new parts due to updates, new features, etc. If the failing operation can be safely skipped, the software can recover. Otherwise, if that operation is crucial, the software will fail. The only way out is design diversity wherein applications providing the same functionality but using different design/implementations can take over. For more on design diversity and fault-tolerant software click here.

Heisenbugs: Obvious design [+implementation] faults in software are likely to have been detected and removed during testing and subsequently as a result of feedback during field use. However, even mature software can be expected to have what are known as "Heisenbugs"[GRAY 1986]. These are bugs in the software that are revealed only during specific collusions of events. For instance a sequence of operations may leave the software in a state that results in an error on an operation executed next. Synchronization oversights in multithreaded software are another example, where errors occur during some executions, but do not occur when repeated. Such errors are said to be caused by transient faults. Simply retrying a failed operation, or if the application process has crashed, restarting the process (the restarting could be done by middleware providing Software Implemented Fault Tolerance, SIFT) might resolve the problem.



Errors are caused by the software's data, data-sequence, and (inter-)process timing [+ data/parameter exchange mechanism + access to hardware resources] dependencies. It is fair to assume that software itself is time invariant: it does not degrade. Data and timing conditions that evoke a software error, will evoke that error each time these same conditions occur. Hence, it is the probability of encountering these specific conditions that determines the software "reliability".

Life testing for flight-critical reliability levels of 10^{-9} [10E-9 issue comes from where?] per flight hour is impractical or infeasible [Butler93], large, complex. The test time would be so large that even parallel testing of 10 or 100 systems would not reduce this to acceptable levels [and may never excite the fault]. This type of testing is also cost prohibitive, based on the price of such systems and of the required test environment. Unlike hardware, accelerated testing of software cannot be done for real-time systems like Flight Control computers. They have typical processes iteration rates of 1 – 100 Hz, and their throughput margins do not allow significantly faster processing. Real hardware-in-the-loop: inertias. Limitations are also posed by the generation of input test-vectors, and the evaluation of the responses (the reader may verify how many years it takes[8] to check all possible combinations of two 32-bit words at 1 nanosecond per comparison). This leaves Fault Avoidance [does not protect!] and Fault Tolerance as the way to protect against "incorrect" software. Self-Testing (BIT) can be used to determine the integrity of the program memory devices. This contributes to system reliability by detecting the effects of incorrect installation, spontaneous de-programming, radiation damage, etc.

Quality, metric per SLOC: [Stark94]

8 $2^{32} \times 2^{32}$ comparisons, 3.15×10^7 sec/year, 10^9 comparisons/sec $\rightarrow$ 585 years

Software fault-mechanism: poorly modeled & understood (as compared to hardware). No physical change. Permanent.
Software quality: number of defects shipped.
Software reliability modeling & prediction issues are discussed in Chapter 4.

Sheer unlimited number of forms, e.g., spec faults, coding faults, logical errors, stack overflows, improperly initialized variables, etc. Systematic/systemic?, latent. Data dependent, data sequence dependent, timing dependencies. [Storey96].There are many types of software faults, e.g. [Beizer90, Sullivan91/92 Littlewood92]: Examples shown in Figure 3-X.

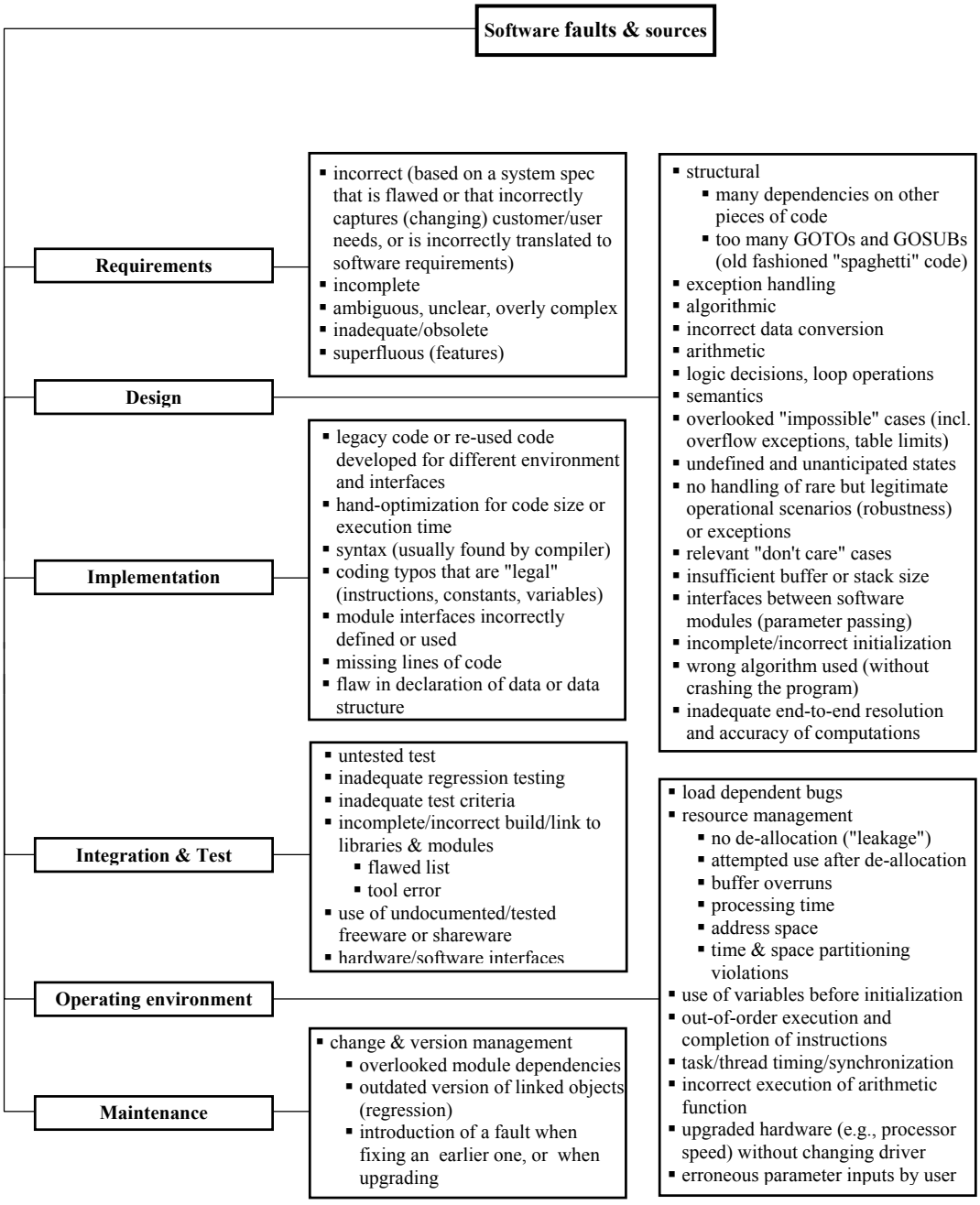
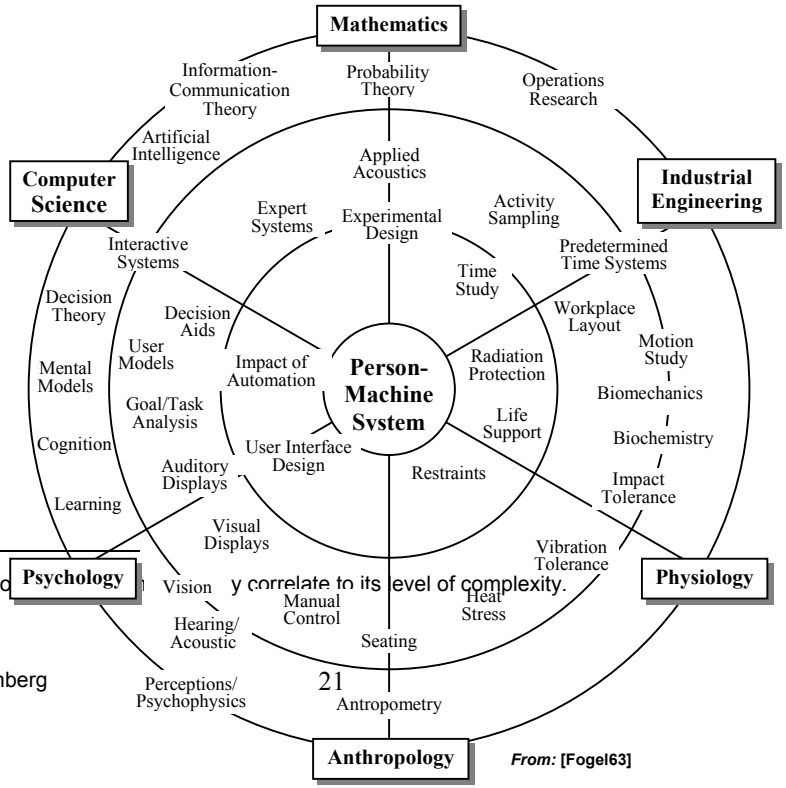


Figure 3-X ***Taxonomy of software faults and their sources***

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[9] 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 [DO254]. As with complex software, it may not be feasible to guarantee correctness of a complex device. Its operation, failure modes, and failure effects are difficult to comprehend without the aid of analytical methods (and sometimes even with such aids). Its development must be subject to the same rigorous assurance measures as would have to be used for an equivalent software implementation.

The system development phase must pay close attention to the Human-Machine Interface (HMI), as 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, aural and visual annunciators such as displays (incl. symbology), operating procedures, etc. The operator or user also has *mental* interaction with the system, e.g., by directing its functionality (task distribution and structure) and interpreting its behavior. Cognitive aspects of automation must be devised carefully, as they are 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) and Human-Computer Interface (HCI) design involves many sciences and engineering disciplines, as illustrated in Figure 3-2 [Fogel63, Donchin95]. [Gertman94] Human Reliability & safety Analysis Data Handbook; [Kuhn97] 50% of telephone US, one of the worlds largest distributed systems, outage time (freq?) caused by humans, procedures etc…



From: **[Fogel63]**

9 A device's gate co… y correlate to its level of complexity.

Figure 3-2 ***Relationship of Human Factors to other areas of science and technology***

The HMI is especially important in control situations for complex, safety-critical systems such as airplanes, air traffic control, and nuclear power plants [Redmill97b, ISO13407]. Systematic consideration of human participation and interaction is generally is lacking in the development of information technology (IT) systems [Hansen01].

Fault Origin

The **fault origin** indicates whether the fault is **internal** or **external**, with respect to the system boundary.

External: Interference, HMI-interaction, inputs, environmental.

Internal: open/short, threshold change, physical phenomena inside the system, heat from dissipation
Intruders: external, even though they may be inside the organization, eg authorized users

Fault Mechanism

A **fault mechanism** is the process by which stresses can damage hardware and mechanical items. When considering a particular item by itself, this is often also referred to as the *failure* mechanism. A fault mechanism is called **overstress** when it is caused by a single stress event that exceeds the item's intrinsic strength. We speak of **wear-out** when damage accumulates incrementally until it exceeds the item's endurance limits. In a well-designed system with appropriately selected components, overstress and wear-out faults should no occur within the intended usage life of that system. Stresses that cause a component to fail need not be permanent or irreversible. E.g., excessive flexing or twisting of interconnection wiring (ribbon cables, flex leads, etc.) may cause excessive crosstalk or short-circuiting that completely disappears without permanent damage when the mechanical load is removed. Connectors are also prone to temporary fault mechanisms [Mroczkowski92].

All physical materials, whether in mechanical or solid-state devices, are subject to mechanical failure mechanisms. Abroad taxonomy of such mechanisms is shown in Figure 4-3 [Dasgupta91/92a/92b/93a/93b, Li93, Engel93, NSWC94, CPE95]. The trend towards shrinking of device features does not only apply to electronic components. More and more, Micro-Mechanical Devices (MMDs) and Micro-Electro-Mechanical Systems (MEMS) are being developed for a growing number of applications such as medical implants (tiny pumps that administer medicine) and micro-sieves for blood screening, solid-state chemical sensors, and small accelerometers in automobile airbag deployment controllers. As the device features are reduced, even minute

defects incurred during manufacture and application stresses may become relatively more important. Fault mechanisms specifically for solid-state devices and materials are shown in Figure 4-4 [Sadlon93, Hu94, Young94, Al-Sheikhly95, Diaz95, Knepley99, Baumann01, Clement01].

SEUs/MBU/ references, why/where need to be considered, spacecraft, satellites solar wind, avionics; latitude/altitude; on ground also [Cataldo01].

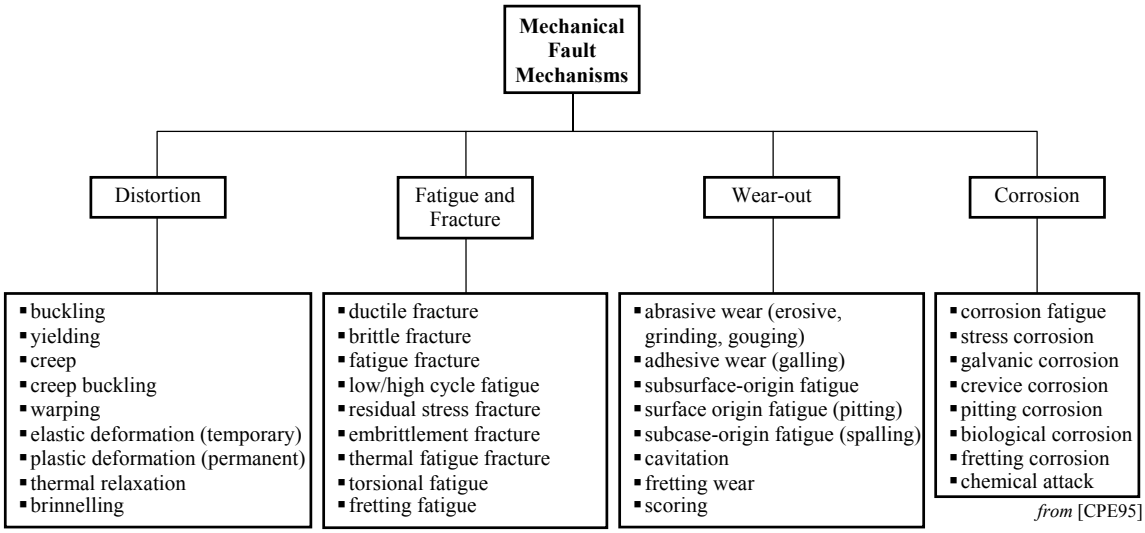


Figure 4-3 ***Taxonomy of mechanical fault mechanisms***

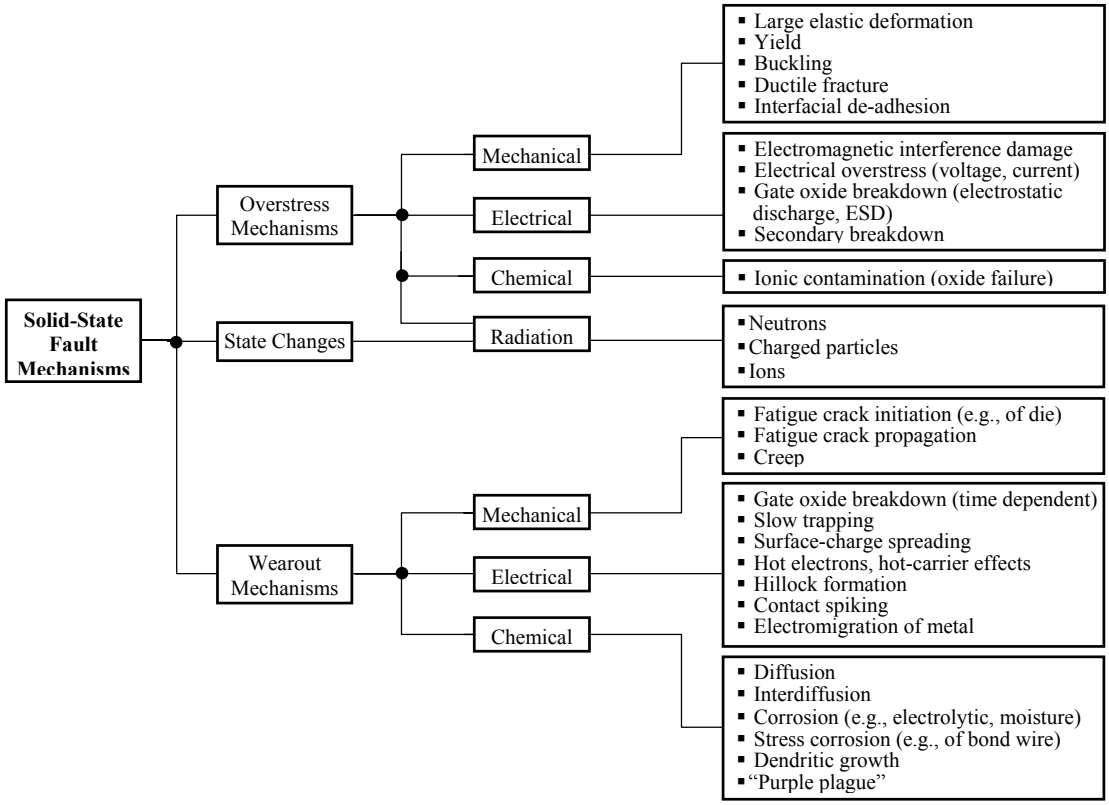


Figure 4-4 ***Taxonomy of fault-mechanisms of solid-state devices***

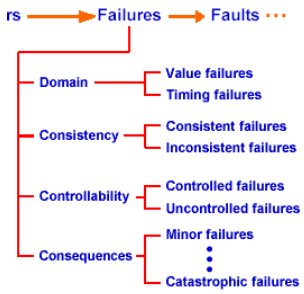
3.2.12 FAILURE CLASSIFICATION

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 intractable failure modes. These may have to be lumped so they can be treated in a practical manner.

The failure modes affect the correctness and timing of service 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:

- Behavior type
- Hazards and Risks
- Failure accountability
- Failure effects and control mechanisms



3.2.13 Failure Behavior Type

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

- Omission
- Commission
- Incorrect results
- Fail-silent
- Fail-stop
- Fail-active
- Fail-passive or fail-neutral
- Graceful degradation or fail-soft
- Arbitrary or Byzantine

Classified based on:

- Correctness of the time at which results are produced or service is delivered

- Correctness of the produced results
- Completeness and level of performance of the functionality provided

[Cristian91/95, Hadzilacos93]

Response; server “liveness”: client *eventually* receives a response or reply to a request.

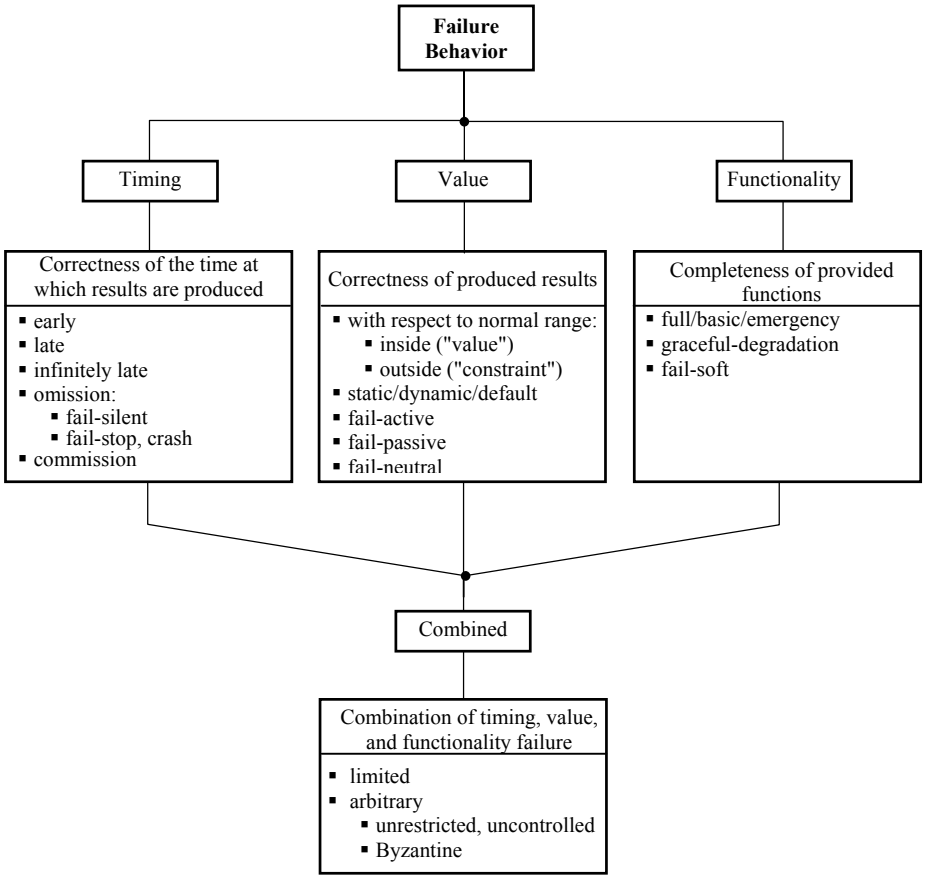


Figure 4-5 ***Taxonomy of failure behaviors***

Timing of service delivery = early/commission/omission/late; omission= silent/stop/controlled?

Real-time systems (RTS) are characterized by data manipulations and data transfers that must meet temporal completion deadline(s). In other words, an RTS is:

any processing system
that must respond to externally generated stimuli
within a finite period that is specified, agreed-upon, or expected

Hence, dependability of such systems is not only determined by correctness of logical and value results, but also by the timeliness of the delivered service. E.g., failure to respond in time may be just as bad as giving the wrong response. Real-time systems are typically embedded systems, as they are a subsystem of a larger engineered system. They can be classified based on the severity or "hardness" of deadlines and the length of response times [Burns01]:

- **soft**: deadlines are important, but it is acceptable to occasionally miss them. E.g., a network or database server that increase access response times to one or more users in order to accommodate more users. I.e., there is a benefit associated with missing the deadline.
- **firm**: a soft RTS that does not benefit from late delivery of service, e.g., displaying text that is entered via an operator console or keyboard.
- **hard**: responses must occur within the imposed deadlines. This is typical for closed-loop control systems. A hard RTS generally has a deadline period or "window" with a lower bound and an upper bound. Early or unexpected responses, even with a correct value, are also considered failures.
- **real**: a hard RTS with very short response times, e.g., missile actuator control system.

A single system may contain subsystems or functions that have different real-time hardness levels. Embedded processors and controllers

Value = constraint/value? See "faults". We identify two classes of object failures: responsive and non-responsive. With responsive failures, a faulty object responds to every operation, but its responses may be incorrect. With non-responsive failures, a faulty object may also "hang" without responding. In each class, we define crash, omission, and arbitrary modes of failure.
Value:
Incorrect results: computations, or other forms of processing in the system, cause value faults that propagate to the system boundary; the faults may occur even if the system is supplied with correct input data. The incorrect results can be static (stuck or frozen), or dynamic. The latter can be arbitrary, oscillatory, "jumpy", etc. In real-time control systems, timing fault (early or late) must also be considered as value faults.

- A constraint of run-time assertion instructions is not satisfied
- attempt to follow a null pointer (software pointer that points to nowhere), when trying to access non-existent array elements
- lack of computational precision (cumulative)
- An exception can be raised either by a language-defined primitive operations, e.g., constraint-error on arithmetic or an index out of bounds

Omission failures: the system does not complete all of its tasks in a timely fashion (i.e., by a specified deadline), performs them only partially, or not at all. The latter is also known as a "system crash". As a result, the system may temporarily or permanently generate no outputs (send-omission). However, if and when the system *does* generate outputs, they are correct and on time.

Commission failures: the system, such as a processing node, produces an event (e.g., an output signal, a message, or a computational result) when none should have been produced [Broster01]. This failure category also covers events that are produced too early, which can cause problems in systems with real-time constraints, including data buses (setup times of clocked

signals). Systems where results (iterations) must be produced at regular intervals (with tolerances), vs. systems in which certain maximum reaction time applies (e.g., transactions), or reaction time window (e.g., data line/signal set-up on data/address bus).

A unit is called **fail-silent** if it can only exhibit omission failures. That is, internal faults never cause an incorrect result to be sent out: the unit either delivers correct performance or ceases all activities at its interfaces, independent of the continuation of internal activities. When the unit resumes generating outputs, they are correct. A variation of this behavior is that no "bad but un-flagged" outputs ever leave the unit: the unit always correctly indicates ("flags") the correctness and timeliness of all outputs to user-systems. In other words, in case of failure, the unit either produces no output, or produces an output to indicate failure. Such behavior is usually easy to detect by user-systems, and a desirable attribute of building blocks for fault-tolerant systems. When used at the level of execution of elementary software instructions, and in communication or transaction tasks, this *all-or-nothing* property of fail-silent failures is referred to as **atomicity** (i.e., indivisibility, as "atomic" literally means un-cuttable).

Fail-stop is the extreme case of fail-silent behavior: all faults cause a permanent omission failure. That is, a faulty process stops before writing (permanent) data, or communicating with other processes [Schneider84, Chandra98]. This is also referred to as **halt-on-failure** behavior. In some literature, the terms fail-silent and fail-stop are used interchangeably. In certain application domains such as that of control systems, a fail-silent or fail-stop system is also referred to as **fail-passive** or **fail-neutral**, if the system automatically disengages, goes off-line, reverts to a bypass mode, or forces its actuators to a bypass mode. Sometimes it is referred to as a **crash failure**. A system that is not fail-stop, by definition continues to generate incorrect results in the presence of a particular fault or any fault. Such a system is therefore called **fail-active**.

Caveat

The descriptors "active", "passive", "silent", and "stop" only apply to the system *behavior* in the presence of one or more faults. They do *not* characterize the *consequences* of that behavior! In no way do any of these failure types imply the presence or absence of hazardous or unsafe conditions as a result of the failure. E.g., a desktop computer that erroneously turns into a random number generator is probably quite safe (though the annoyed and disgruntled user may not be); similar behavior of a control computer in a petrochemical plant most likely is not safe. A system is called **fail-safe** if its failure does not cause (unacceptable) hazards. This typically requires a system design in which only multiple, independent design errors remain as the only reasonably probable causes of a catastrophic failure consequences. By definition, critical systems are never fail-safe, even if they are fail-passive. Section ## discusses the issue of hazards and risks in more detail.

For many systems, the preferred failure behavior is retention of *degraded-but-correct* functionality and performance, rather than incorrect or total loss of functionality. That is, the system continues to provide primary functions despite one or more faults, and loss of the shedded functionality is not hazardous. The resulting situation may be a nuisance, increase operator workload or required vigilance, or reduce productivity. Such behavior is called **graceful degradation** or **fail-soft**, and is especially preferred for (safety) critical applications, where it is part of the specification and design [Herlihy91]. Take for example a monitoring system that sheds certain automatic protection modes upon failure of a particular sensor, and annunciates this to the operators. A process control computer may have a problem with its main application processor, and activate simplistic closed-loop control modes on the I/O processor. A faulty network or web server may continue to operate, but provide limited throughput or increased response time. Another example is a

mechanical actuation system with two independently controlled motors whose outputs are velocity-summed through differential gearing. If one control computer channel fails such that its motor stops, the other channel of the drive system can still provide actuation without loss of torque, but at half speed (provided the motor of the failed channel cannot be back-driven).

When no *a priori* assumptions are made, or can be made, regarding the malicious characteristics of faulty components we speak of **Byzantine** faults and failures. They exhibit **unrestricted**, **uncontrolled**, **arbitrary**, **deceptive** misbehavior (timing, value, state, functionality). This type of pernicious fault is called "Byzantine", in analogy with the so-called Byzantine Generals Problem [Lamport82, Walter88, Barborak93, Vaidya95, Malkhi97, Postma98, Castro99]. Consistency of replicated data in a multi-server system, as seen by the server clients. Distributed replicated data service to improve availability and reliability of important data used over a network; problem: keeping all servers [data copies] in a distributed system up to date and correct. Application in systems requiring consensus, most distributed systems.

Byzantine Generals Problem

This problem refers to a scenario in which a number of Byzantine army divisions are besieging a town. The commanding generals (redundant processing nodes) have to reach agreement (global consensus) on whether to attack or to retreat (agree on mode switching or on computed parameter values). Each general has a messenger (data-link) to send his own, local opinion to each of the other generals, as well as relay the opinions received from those generals. The sending generals and messengers can be loyal or treacherous (originating or conveying wrong, distorted, or inconsistent messages to different generals, impersonate other generals). Receiving generals (monitors) can have interpretation problems.

The provable, classical solution to the Byzantine problem requires a system comprising at least $3m+1$ nodes, where m is the maximum number of 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 that cannot provide Byzantine resilience [ref + illustrate impossibility].

EXAMPLE

Multiple rounds of interactive voting are needed to reach global consensus and to identify a "malicious" node. At least $2m+1$ communication paths are required, and at least $m+1$ rounds of communication.

Exact agreement vs approximate agreement (not suitable for binary parameters or decisions [refs]

Various protocols are used to arrive at consensus between healthy processing nodes:

- *Authenticated Byzantine Agreement Protocol* (ABAP) [Lamport##], using non-forgeable signed messages to achieve agreement on a single value.
- *Degradable Byzantine Agreement Protocol* (DBAP) [Vaidya95] same as ABA, up to a certain number of faults $u < m$. After that, the agreement rules are relaxed and all healthy nodes agree on at most 2 values: one group of healthy nodes agrees on one value, the other healthy nodes agree on a default value. Here, m is largest number of faults for which ABA can be achieved, and u is largest number of faults for which DBA can be achieved. This is called m/u Degradable Agreement. It requires at least $2m+u+1$ nodes.
- *Quorum-based Byzantine Agreement Protocol* (QBAP) [Malkhi97] called *b*-masking quorum

[Rajagopal, J.]

It is clear from the above enumeration that “fail-stop” and “Byzantine” are opposite extremes of the spectrum of failure behavior models. The fail-silent model is a idealistic design goal, but typically too simplistic for use in the design and evaluation of highly complex systems, especially in safety-critical applications that involve complex software [Schneider84, Chandra98]. Byzantine models (and the associated protection mechanisms) sometimes focus on malicious behavior that is much less probable in real systems than the combination of conditions required for such conditions to occur BS! Check Kim00. This is then done the detriment of paying attention to more likely undesirable non-Byzantine events [Kim00].

Caveat

It is important to realize that the characterization of a system with the above categories of failure behavior is meaningless, *unless* this is accompanied by a statement of the type and number of faults for which the particular behavior applies. E.g., a system may be fail-stop to any single hardware fault but not to a software fault. Or, the system is fail-stop to any single fault, but certain subsequent faults may defeat the mechanism that provides the fail-stop behavior.

If stated failure behavior applies with multiple faults in the system and there is a fault handling mechanism, then the minimum time between subsequent faults may have to be stated, as well as their sequence. Obviously, systems that use majority-voting cannot be expected to handle the scenario in which the majority of voted parameters become faulty at (nearly) the same time, but are in agreement with each other.

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 or errors in the system. I.e., there is no failure at system level, and, hence, no failure behavior to be characterized. 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.

3.2.14 Failure Effects

Failure modes are ways in which a system can fail or has failed. The actual or perceived consequences of those failure modes are called the **failure effects** [ARP4761, Walter95. Obviously we are primarily interested in undesired consequences. **Failure effects** = system behavior in presence of failure. Entirely system (and combo with user) specific.

The safety of system components, and to identify design modifications and corrective actions needed to mitigate the effects of a failure on the system. It is used in planning system maintenance activities, subsystem design, and as a framework for system failure detection and isolation. In the military, aerospace and nuclear industries, where safety issues are of prime importance, FMECA has become an essential process in the design of systems from early in the development stage to design and test.

Its [FMECA] primary purpose is to identify all catastrophic, critical and safety related failure modes at the earliest possible time [in the product development process] so they can be eliminated or minimized [controlled] through design changes.

Hazards are actual or potential, unplanned (but not necessarily unexpected) undesired conditions or events that *could* result from system failure, design inadequacies, environmental conditions, or personnel error ("operator failure"). **Format as for Rel def?**

Two major aspects of hazards must be considered:

- **severity**, which is the worst *credible* known or potential consequence that could ultimately result from system malfunction.
- **probability**, which is the expected or actual likelihood that a system failure with hazardous consequences of a particular severity level will occur.

Combined, these two attributes are referred to as **risk**:

Risk = (*Severity of mishap consequences*) **x** (*Expected number of mishaps*)
= (*Severity*) **x** (*Expected mishap frequency per unit of exposure*) **x** (*Amount of exposure*)

Software risks: [Littlewood92, Leveson95]
Computer risks: [Neumann95]

Sometimes *failure conditions* are also called risks. The concept of "risk" is discussed in the next section???

System failures that cause uncontrolled hazards are referred to as **unsafe**.

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 what are deemed acceptable risk levels.

From risk/hazard/safety. The occurrence of a physical **mishap** event generally requires three elements [Ciemens81]:

the accumulation, release, and transfer of unwanted energy
to a vulnerable target
in the absence of adequate barriers

These three elements of a mishap event also point the way to strategies for energy flow control and protection of the target. Energy can be in the form of heat, shock or vibration, electromagnetic fields [Fuller95, Shooman93, EN292], electrical power, nuclear or particle radiation (ionizing, non-ionizing), kinetic energy (linear or rotational), potential energy (pressure, spring loading, mass-at-height), chemical (corrosive, toxic, medication), etc.

This concept of *energy* flows can easily be expanded to include unwanted *data* flows. Data: uncontained, prevent warning/detection, scalpel brain surgeon robot, x-ray imaging controller radiation [thera5], flight control w. control surface deflection control, crossroad with traffic lights; non-phys: bank funds transfer / transaction; privacy, billing off by factor of million, tax assessments, name confusion leading to false arrests, (some correctable) [Neumann92/95]

Controls are the means and mechanisms that are in place (or are required) to reduce the risks of hazardous system failure to an acceptable level. Some control mechanisms act on the hazard itself, whereas others control the losses or damage by protecting the target [IEC61508]:

- 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 around the fan blades 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.
- Modification of the energy release-path

[Ashley01] Concorde's comeback

The release of energy can be modified by providing a safer path, e.g., using a fuse or frangible structure, or testing a radar transmitter with a dummy-load instead of an actual antenna. Barriers are safety devices and features that contain or divert the energy, or protect the target. This reduces the probability of energy release, and reduces the severity of the consequences of a release. Physical barriers can be placed around the energy source (containment structure, shield or shroud), or around the target (hardening, conditioning, absorbing). The energy and target can also be kept separate by a temporal barrier, e.g., a slow release at safe levels, or sufficient warning time for evacuation. Spatial barriers provide a distance between the energy and the target, such that the energy is sufficiently dissipated or diluting by the time it reaches the target. It is usually necessary to periodically check safety devices and features for latent faults. Barriers can also be provided by following procedures for responses to cautions and warnings. Such procedures are based on knowledge, skills, supervision, and training. As single barriers are not always effective, multiple barriers are common. The number of barriers depends on the level and acceptability of the risk that is associated with the hazardous events.

Logical barriers Partitioning (h/w, s/w, containment) + refs (for data "energy"); partitioning violation & protection [MMU normally provides space partitioning protection, but its failure may cause uncontained violation.
[Rushby99]

- Logical Barriers: network firewalls damage confinement and assessment – diagnosis and evaluation of error spreading
  - firewalling – structure of the system preventing error spread
  - modular decomposition – reduces possibility of error spread
  - atomic transactions – moving the system from one consistent state to another

protection mechanisms – access permissions for specified operations

3.2.15 **Failure Accountability**

Failure **accountability** indicates who has to fix the failed system, or has to "pay" for accepting or correcting the damage ensuing from system failure, whether this damage is physical, psychological, 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 aberrant societies, which associated person or business entity has a large financial basis or "deep pockets".

▪ PROBLEMS AND EXERCISES

3.3 REFERENCES

[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]

[Al-Sheikhly94] Al-Sheikhly, M., Christou, A.: "*How radiation affects polymeric materials*", IEEE Trans. On Reliability, Vol. R-43, No. 4, December 1994, pp. 551-556

[Anderson81] Anderson, T., Lee, P.A.: "*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.

[Al-Sheikhly94] Al-Sheikhly, M., Christou, A.: "*Tutorial – how radiation affects polymeric materials*", IEEE Trans. on Reliability, Vol. R-43, No. 4, December 1994, pp. 551-556

[Ashley01] Ashley, S.: "*Concorde's comeback*", American Scientific, August 2001, pp. 12, 13

[Avizienis76] Avizienis, A.: "*Fault tolerant systems*", IEEE Trans. on Computers, Vol. C-25, Vol. 12, pp. 1304-1312

[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. 7th AIAA/IEEE Digital Avionics Systems Conf. (DASC), Washington/DC, USA, November 1987, pp. X-Y

[Avizienis95] Avizienis, A.A.: " *The Methodology of N-Version Programming*", Chapter 2 of "*Software Fault Tolerance*", Lyu, M. (editor), John Wiley & Sons Ltd, 1995, pp. 23-46

[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 7th Int'l IFIP Conf. On Dependable Computing for Critical Applications, San Jose/CA, USA, January 1999), October 1999, 424 pages, ISBN 0-7695-0284-9

[Avizienis00] Avizienis, A., Laprie, J.-C., Randell, B.: "*Fundamental concepts of dependability*", Proc. 3rd IEEE Information Survivability Workshop (ISW2000), Boston/MA, USA, 2000, 6 pp.

[Bakken02] Bakken, D.E.: "Middleware", to appear in Encyclopedia of Distributed Computing, Kluwer Academic Press, 2002, 5 pp.

[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

[Baumann01] Baumann, R.*: "Soft errors in advanced semiconductor devices Part 1: the three sources of radiation*", IEEE Trans. on Device and Materials reliability, Vol. 1, No. 1, 2001, 7 pp.

[Broster01] Broster, I., Burns, A.: "*The babbling idiot in event-triggered real-time systems*", Proc. Of the Work-In-Progress Session, 22nd IEEE Real-Time Systems Symp., 2001, 4 pp.

[Burns01] Burns, A., Wellings, A.: "*Real-Time Systems and Programming Languages*", 3rd Edition, Addison Wesley, 2001, 611 pp., ISBN 0201729881

[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, Jan. 1993, pp. 3-12

[Castro99] Castro, M., Liskov, B.: "*Practical Byzantine Fault Tolerance*", Proc. 3rd ACM Symp. on Operating Systems Design and Implementation, New Orleans/LA, USA, February 1999, pp. 173-186

[Cataldo01] Cataldo, A.: "*SRAM soft errors cause hard network problems*", Electronic Engineering Times, 20 August 2001, pp. 1, 92

[Chandra98] Chandra, S., Chen, P.M.: "*How fail-stop are faulty processes?*", Proc. 28th Annual Int'l Symp. on Fault Tolerant Computing (FTCS-28), Munich, Germany, June 1998, 10 pp.

[Ciemens81] Ciemens, P.L.: p. 12 of "*A Compendium of Hazard Identification & Evaluation Techniques for System Safety Application*", Sverdrup Technology, Inc., November 1981

[Clement01] Clement, J.J.: "*Electromigration modeling for integrated circuit interconnect reliability analysis*", IEEE Trans. on Device and Materials Reliability, Vol. 1, No. 1, 2001, 10 pp.

[Cristian91] Cristian, F.: "*Understanding Fault-Tolerant Distributed Systems*", Communications of the ACM (CACM), Volume 34, No. 3, March 1991, pp. 56-78

[Cristian95] Cristian, F., Aghili, H., Strong, R., Dolev, D.: "*Atomic broadcast: From simple message diffusion to Byzantine agreement*", Information and Control, Vol. 118, No. 1, January 1995; revision of same title in Proc. 15th Int'l Conf. On Fault-tolerant Computing (FTCS-15), Ann Harbor/MI, USA, 1985

[Dasgupta91] Dasgupta, A., Pecht, M.: "*Material Failure Mechanism and Damage Models*", IEEE Trans. on Reliability, Vol. 40, No. 5, December 1991, pp. 531-536

[Dasgupta92a] Dasgupta, A., Hu, J.M..: "*Failure-Mechanism Models for Excessive Elastic Deformation*", IEEE Trans. on Reliability, Vol. 41, No. 1, March 1992, pp. 149-154

[Dasgupta92b] Dasgupta, A., Hu, J.M..: "*Failure-Mechanism Models for Brittle Fraction*", IEEE Trans. on Reliability, Vol. 41, No. 3, September 1992, pp. 328-335

[Dasgupta93a] Dasgupta, A., Haslach, H.W.: "*Mechanical Design Failure Models for Buckling*", IEEE Trans. on Reliability, Vol. 42, No. 1, March 1993, pp. 9-16

[Dasgupta93b] Dasgupta, A.: "*Failure mechanism models for cyclic fatigue* ", IEEE Trans. on Reliability, Vol. R-42, No. 4, December 1993, pp. 548-555

[Diaz95] Diaz, C., Kang, S.M., Duvvury, C.: "*Electrical overstress and electrostatic discharge*", IEEE Trans. on Reliability, Vol. R-44, No. 1., March 1995, pp. 2-5

[Dolev83] Dolev, D., Lynch, N.A., Pinter, S.S., Stark, E.W., Weihl, W.E.: "*Reaching approximate agreement in the presence of faults*", Proc. 3rd IEEE Symp. on Reliability in Distributed Software and Database Systems, Clearwater Beach/FL, USA, Oct. 1983, pp. 145-154; Also: J. of the ACM, Vol. 33, No. 3, July 1986, pp. 499-516

[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

[DO254] RTCA Document DO-254 "*Design Assurance Guidance for Airborne Electronic Hardware*" (Radio Technical Commission for Aeronautics) RTCA, Inc., 19 April 2000, 137 pp.

[Dunn86] Dunn, W.R.: "*Software Reliability: measures and effects in Flight Critical Digital Avionics Systems*", Proc. 7th AIAA/IEEE Digital Avionics Systems Conf. (DASC), Fort Worth/TX, USA, October 1986, pp. 664-669

[EIA632] "*Processes for Engineering a System*", ANSI/EIA-632-98, Electronic Industry Alliance, January 18, 1999

[Eick01] Eick, S., Graves, T., Karr, A., Marron, J., Mockus, A.: "*Does code decay? Assessing the evidence from Change Management data*", IEEE Trans. on Software Engineering, Vol. SE-27, No. 1, January 2001, pp. 1-12

[EN292] European Machinery Directive EN292-1 and -2: "*Safety of Machinery, Basic concepts, general principles of design*", January 1995; also ISO 12100-1 and -2

[Engel93] Engel, P.A.: "*Failure models for mechanical wear modes and mechanisms*", IEEE Trans. on Reliability, Vol. R-42, No. 2, June 1993, pp. 262-267

[Evans99] Evans, R.A.: "*The Language of Statistics & Engineering*", Proc. 1999 Annual Reliability and Maintainability Symp. (RAMS), Washington/DC, USA, 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

[Frank00] Frank, M.V.: "*Theory and Applications of Risk Assessment in Aerospace*", presented at IEEE Reliability Society, San Diego Chapter, Safety Factor Associates, Inc., 19 June 2000, 77 slides

[Fuller95] Fuller, G.L.: "*Understanding HIRF – High Intensity Radiated Fields*", Avionics Communications, Inc., 1995, 123 pp., ISBN 1-885544-05-7

[Gertman94] [Gertman94] Gertman, D.I., Blackman, H.S.: "*Human Reliability & Safety Analysis Data Handbook*", John Wiley & Sons, 3rd ed., 1994, 472 pp., ISBN 0-47159-110-6

[Gordon91] Gordon, A.M.: "*A practical Approach to Achieving Software Reliability",* Computing & Control Engineering Journal, November 1991, pp. 289-29

[Gray86] Gray, J.: "*Why do Computers Stop and What Can be Done About it?", Proc. of 5th Symp. on Reliability in Distributed Software and Database Systems*, January 1986, pp. 3-12,

[Hansen01] Hansen, C.K. (ed.): "*The status of reliability engineering technology in 2001*", IEEE Reliability Society Newsletter, January 2001, pp. 21-29

[Hadzilacos93] Hadzilacos, V., Toueg, S.: "*Fault-tolerant Broadcasts and Related Problems*", Chapter 5 (pp. 97-145) in "*Distributed Systems"*, Mullender, S.J. (editor), Addison-Wesley/ACM Press, 1993, ISBN 0-201-62427-3, 601 pp.

[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

[Herlihy91] Herlihy, M.P.; Wing, J.M., "*Specifying graceful degradation*", IEEE Trans. on Parallel and Distributed Systems, Vol. 2, No. 1, Jan. 1991, pp. 93-04,

[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, Switzerland, 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)

[ISO13407] International Standards Organization (ISO) Standard 13407:1999: "*Human-Centered Design Process for Interactive Systems*", June 1999, 26 pp.

[ISO9004] International Standards Organization (ISO) Standard 9004, Figure 1,: "*Quality Management Systems – Guidelines for Performance Improvement*", December 2000, 66 pp.

[ISO14040] International Standards Organization (ISO) Standard 14040 "*Environmental Management – Life Cycle Assessment – Principles and Framework*", 1st Edition, June 1997, 17 pp.

[James00] James, M.N.: "*Failure as a design criterion*", class notes for first module of DSGN118 "Design as a generic tool" course, http://www.tech.plymouth.ac.uk/sme/FailureCases/Failure.htm, University of Plymouth, 2000, 18 pp.

[Kanoun01] Kanoun, K.: "*Real-world design diversity: a case study on cost*", IEEE Software, July/August 2001, pp. 29-33

[Kim00] Kim, K.H.: "*Issues insufficiently resolved in Century 20 in the fault-tolerant distributed computing field*", Proc. 19th IEEE Symp. on Reliable Distributed Systems (SRDS), Nürnberg, Germany, October 2000, pp. 106-115

[Knepley99] Knepley, J.W.: "*Understanding electronic part failure mechanisms*", Tutorial at 1999 Annual Reliability & Maintainability Symp. (RAMS), Washington/DC, USA, January 1999, 22 pp.

[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

[Kuhn97] Kuhn, D.R.: "*Sources of Failure in the Public Switched Telephone Networks*", IEEE Computer, Vol. 30, No. 4, April 1997, pp. 31-36

[Kuamoto96] Kumamoto, H., Henley, E.J.*: "Probabilistic Risk Assessment and Management for Engineers and Scientists*", 2nd Ed., IEEE Press, 1996, 592 pp., ISBN 0-7803-1004-7

[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. 15th Fault Tolerant Computing Systems Conf. (FTCS-85), Ann Arbor/MI, USA, June 1985, pp. 2-11

[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.

[Li93] Li, J., Dasgupta, A.: "*Failure mechanism models for creep and creep rupture*", IEEE Trans. on Reliability, Vol. R-42, No. 3, September 1993, pp. 339-353

[Lin90] Lin, T.-H., Shin, K.G.: "*A Bayesian approach to fault classification*", Proc. ACM Joint Int'l Conf. on Measurement and Modeling of Computer Systems, Univ. of Colorado, Boulder/CO, USA, 1990, pp. 58-66

[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

[Malkhi97] Malkhi, D., Reiter, M.: "*Byzantine Quorum Systems*", Proc. 29th Annual ACM Symp. on the Theory of Computing (STOC), El Paso/TX, USA, May 1997, pp. 569-578

[Martin01] Martin, R.A. :"*Managing vulnerabilities in networked systems*", IEEE Computer, November 2001, pp. 32-38

[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 4th IEEE/AIAA Digital Avionics Systems Conf. (DASC), Seattle/WA, USA, 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.

[Mroczkowski92] Mroczkowski, R.S., Maynard, J.M.: "*Estimating the reliability of connectors*", IEEE Trans. on Reliability, Vol. R-40, No. 5, December 1991, pp. 507-512

[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

[Neumann92] Neumann, P.G.: "*Aggravation by computer life, death, and taxes*", Communications of the ACM, Vol. 35, No. 7, July 1992, p. 122

[Neumann95] Neumann, P.G.: "*Computer-Related Risks*", ACM Press / Addison Wesley, 1995, 384 pp., ISBN 0-201-55805-X

[Normand96] Normand, E.: "*Single-Event Effects in Avionics*", IEEE Trans. on Nuclear Science, Vol. 43, No. 2, April 1996, pp. 461-474

[Oberg99] Oberg, J.: "*Why the Mars Probe went off course*", IEEE Spectrum, December, 1999, pp. 34-39

[Ohlsson98] Ohlsson, M., Dyreklev, P., Johansson, K., Alfke, P.: "*Neutron single event upsets in SRAM-based FPGAs*", Proc. IEEE Nuclear and Space Radiation Effects Conference, 1998 (NSREC98), Newport Beach/CA, USA, 1998, 4 pp.

[Pfleeger92] Pfleeger, S.L.: "*Measuring software reliability*", IEEE Spectrum, August 1992, pp. 56-60

[Pizza98] Pizza, M., Strigini, L., Bondavalli, A., Di Giandomenico, F.: "*Optimal discrimination between transient and permanent faults"*, Proc. 3rd IEE High-Assurance Systems Engineering Symp. (HASE), 1998, 10 pp.

[Poledna96] Poledna, S.: "*Fault-tolerant real-time systems – the problem of replica determinism*", 1996, Kluwer Academic Publ., 147 pp., ISBN 0-7923-9657-X

[Postma98] Postma, A.: "*Classes of Byzantine fault-tolerant algorithms for dependable distributed systems*", PhD thesis, University of Twenthe, The Netherlands, February 1998, 315 pp., ISBN 90-365-1081-3

[Powell92] Powell, D.: "*Failure mode assumptions and assumption coverage*", Proc. 22nd IEEE Int'l Symp. on Fault Tolerant Computing (FTCS-22), Boston/MA, USA, July 1992, pp. 386-395

[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, USA, 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*", 2nd 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

[Rushby99] Rushby, J.: "*Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance*", NASA Contractor Report NASA/CR-1999-209347, 1999, 75 pp.

[Schneider84] Schneider, F.B.: "*Byzantine Generals in Action: implementing fail-stop processors*", ACM Transactions on Computers, Vol. 2, No. 2, May 1984, pp. 145-154

[Shin87] Shin, K.G., Ramanathan, P.: "*Diagnosis of Processors with Byzantine Faults in a Distributed Computing System*", Proc. IEEE 17th Int'l Symp. on Fault-Tolerant Computing (FTCS), Pittsburg/PA, USA, July 1987, pp. 55-60

[Shooman93] Shooman, M.L.: "*A study of occurrence rates of EMI to aircraft with a focus on HIRF*", Proc. 12th AIAA/IEEE Digital Avionics Systems Conf. (DASC), Seattle/WA, USA, October 1993, pp. 191-194

[Siewiorek92] Siewiorek, D.P., Swarz, R.S. (Eds.): Chapter 2 in "*Reliable Computer Systems – Design and Evaluation*", 2nd ed., Digital Press, 1992, 908 pp., ISBN 1-55558-075-0

[Somani94] Somani, A.K, Sharma, T., Nguyen, P.H.: "*Reliability computation of systems with latent failures and monitoring*", Proc. 1994 Annual Reliability And Maintainability Symp. (RAMS), Annaheim/CA, USA, January 1994, pp. 195-200

[Sosnowsky94] Sosnowsky, J.: "*Transient fault tolerance in digital systems* ", IEEE Micro, Vol. 14, No. 1, February 1994, pp. 24-35

[Stark94] Stark, G.E.: "*Technologies for improving the dependability of software-intensive systems: a review of NASA experience and needs*", Proc. 1994 Annual Reliability and Maintainability Symp. (RAMS), 14 pp.

[Storey96] Storey, N.: p. 123 of "*Safety-critical computer systems*", 1996, Addison-Wesley, 453 pp., ISBN 0-201-42787-7

[Strigini01] Strigini, L., Popov, P.: "*The reliability of diverse systems: a contribution using Modeling of the fault creation process"*, CSR Technical Report, January 2001, 32 pp.

[Sullivan91] Sullivan, M.S., Chillarege, R.: "*Software defects and their impact on System Availability – A study on field failures in Operating Systems*", Proc. 21st Int'l Symp. on Fault Tolerant Computing (FTCS), Montreal, Canada, 1991, pp. 2-9

[Sullivan92] Sullivan, M.S., Chillarege, R.: "*A Comparison of Software Defects in Database Management Systems and Operating Systems*", Proc. 22nd Int'l Symp. on Fault Tolerant Computing (FTCS), Boston/MA, USA, July 1992, pp. 475-484

[Trivedi00] Trivedi, K.S., Vaidyanathan, K., Go¡seva-Popstojanova, K.*: "Modeling and analysis of software aging and rejuvenation*", Proc. 33rd Annual Simulation Symp., Washington/D.C., USA, April 2000, 10 pp.

[Vaidya95] Vaidya, N.H., Pradhan, D.K.: "*Degradable Byzantine Agreement*", IEEE Trans. on Computers, Vol. 44, No. 1, January 1995, p. 146-150

[Voges88] Voges, U. (Ed.): "*Software Diversity in Computerized 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. 8th Digital Avionics Systems Conf. (DASC), San Jose/CA, USA, October 1988, 7 pp.

[Walter95] Walter, C.J., Monaghan, T.P.*: "Dependability Framework for Critical Military Systems Using Commercial Standards*", presented at the 14th AIAA/IEEE Digital Avionics Systems Conf. (DASC), Boston/MA, USA, 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

[Young94] Young, D., Christou, A.: "*Failure mechanism models for electromigration*", IEEE Trans. on Reliability, Vol. R-43, No. 2, June 1994, pp. 186-192

[Yount85] Yount, L.J.: "*Generic Fault-Tolerance Techniques for Critical Avionics Systems*", Proc. AIAA Guidance, Navigation and Control Conf., Snowmass/CO, USA, August 1985, 5 pp.

[Yurcik01] Yurcik, W., Doss, D.: "*Achieving fault-tolerant software with rejuvenation and reconfiguration*", IEEE Software, Vol. 18, No. 4, July/August 2001, pp. 48-52