DRS Fault-Tolerance

1. Fault-Tolerance
2. Fault-Tolerant Communication
3. Validation
4. Error Handling

Fault-Tolerance

1. Failures, Errors, and Faults
2. Achieving Fault-Tolerance
3. Fault-Tolerance Mechanisms
4. Dependability Requirements

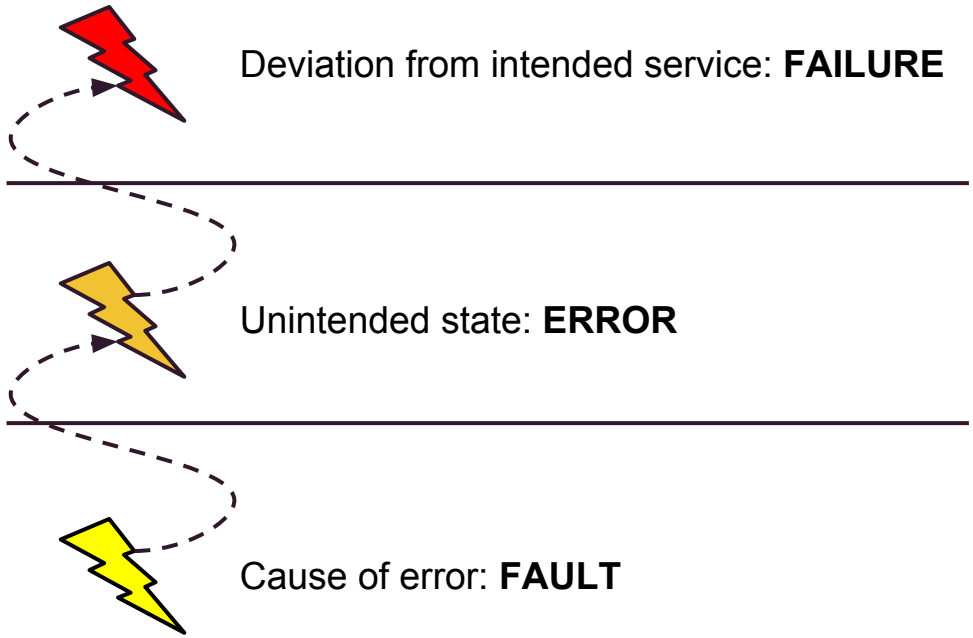
Things Break

Fault-tolerance is required in safety-relevant real-time systems because otherwise a **single component failure** may lead to a catastrophic system failure. Fault-tolerance may be required in non-safety-relevant real-time systems to prevent damage of property and financial loss.

Computer systems are installed to provide dependable service to the system users. Whenever the service of a system deviates from the agreed specification of the system, the system is said to have **failed**.

Failures, Errors and Faults

The relation between causes and effects is very important to understand how systems fail. The following model provides a standard way to classify failure related phenomena in systems. It can be used at all levels of a system. The failure of a subsystem may be seen as the fault of the superior system.



Failures

A failure is an event that denotes a deviation between the actual service and the specified or intended service, occurring at a particular point in real time.

According to the nature of the failure, we distinguish between **value failures** and **timing failures**.

In a **consistent failure** scenario, all users see the (possibly wrong) result, otherwise it is an **inconsistent failure** scenario. If a subsystem either produces correct results or no result at all, we call this type of consistent failure a **fail-silent failure**.

If a system stops operating after the first fail-silent failure, the failure is called a **crash failure**.

Failures

Depending on the effect a failure has on its environment, we distinguish between **benign** and **malign** failures. Applications, where malign failures can occur, are safety-critical applications.

If a failure occurs only once, it is a **single** failure. If the system ceases to provide service until an explicit repair, we call the (single) failure **permanent**. If the system continues to operate after the failure, it is called **transient** failure. If the failure occurs frequently, we call it **intermittent** failure. Transient failures are more frequent than permanent failures.

Errors

An error is a **deviation** between the intended correct state of a system and the current state. Error detection requires knowledge about the intended correct state.

If the error exists only for a short interval of time, and disappears without an explicit repair action, it is a **transient error**. If the error persists permanently until a repair action occurs, we call it a **permanent error**.

It is the goal of fault-tolerant computing to detect and mask or repair errors, before they show up as failures at the system service interface.

Errors

The knowledge about the intended correct state can arise from two different sources:

1. Either from a **priori knowledge** about the intended properties of the states and behavior of the computation, or
2. from the **comparison** of the results of redundant channels

Either from a priori knowledge about the intended properties, or from the comparison of two redundant channels, error detection is based on redundancy.

It is the goal of fault-tolerant computing to detect and mask or repair errors, before they show up as failures at the system-user service interface.

Faults

A fault is the cause of an error and subsequently the possible cause of a failure.

If a fault has its origin in a chance event, e.g., the random break of a wire, is called a **chance fault**. If the fault can be traced to an intentional action by someone, e.g., the introduction of a Trojan horse by a programmer in order to break the security of a system, then, the fault is called an **intentional fault**.

A fault can be caused either by some **physical** phenomenon or by an error in the **design**. In the field of fault-tolerant computing, a number of techniques have been developed that are effective in handling random physical faults. No comparable progress has been achieved to handle design faults. Design faults are difficult to avoid. Rigorous requirements engineering helps to avoid an important class of design faults due to missing requirements.

Faults

Fault boundaries are used to distinguish whether a fault is caused by a phenomenon within the system (**internal fault**) or by some external effects (**external fault;** e.g., a lightning stroke causing spikes in the power supply line). Care must be taken to avoid that a single external fault causes correlated errors in multiple error-containment regions.

Faults that have their origin in the **incorrect development** of the system must be distinguished from faults that are related to **system operation**, e.g., a wrong input by the operator.

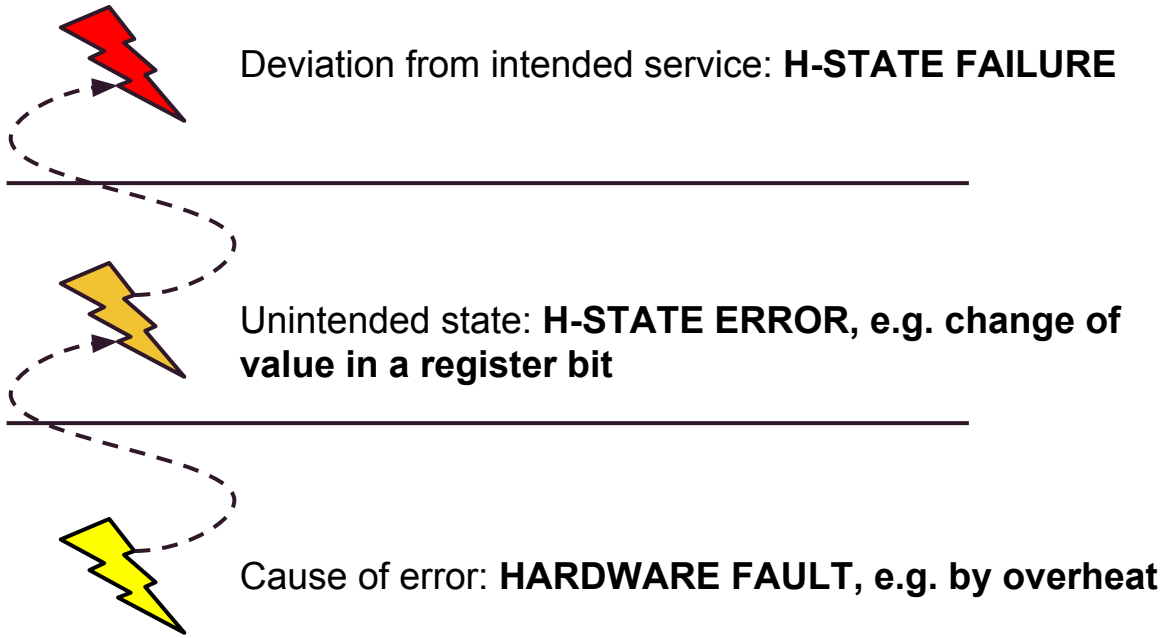
Faults

It is important to distinguish between faults that occur only once and disappear by themselves (**transient fault**; e.g., the mentioned lightning stroke), and faults that remain in the system until they are removed by an explicit repair action (**permanent fault**).

In a system with h-state, a transient fault can cause a permanent error.

Transient Hardware Faults

One of the most common faults are transient hardware faults (caused by e.g. overheat, radiation, vibration, EMI, static discharge, …) .



Dependability Requirements

How can requirements related fault-tolerance be expressed? In industry the following terms are well defined and broadly used:

1. Reliability R
2. Availability A
3. Maintainability M
4. Safety S
5. Security S

Reliability R(t)

R(t) of a system is the probability that a system will provide the specified service **until time t**, given that the system was operational at $t=t_0$.

$$R(t) = e^{-\lambda (t-t0)} \quad 1/\lambda = \text{MTTF (Mean-Time-to-Failure)}$$

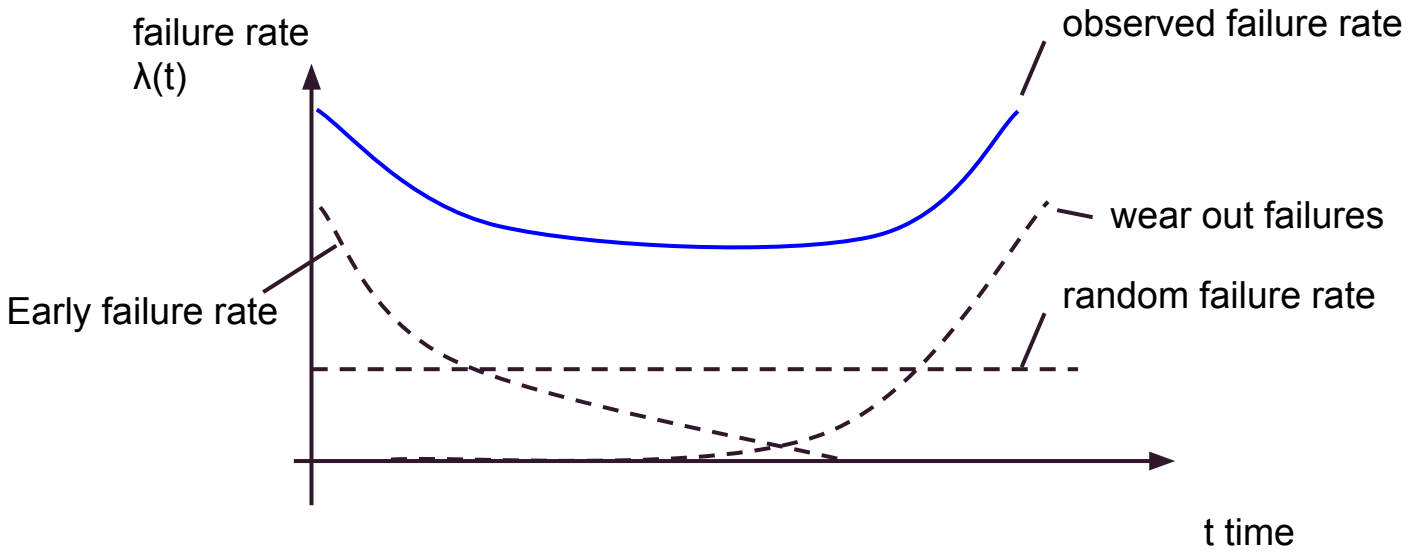
λ .. failure rate [**failures / h**]

e.g. $\lambda \leq 10^{-9}$ failures / h ... ultra-high reliability requirement

e.g. FIT (failure in time) number of failures that can be expected in one billion (10^9) device-hours of operation

Failure Rate

It cannot be assumed that the failure rate of a subsystem is constant. It changes over time. Hardware centered subsystems tend to show a bathtub curve of failure rates over time.



Maintainability

Maintainability is a measure of the time required to repair a system after the occurrence of a benign failure.

M(d) is the probability that the system is restored within a time interval d after the failure.

Mean-Time-to-Repair (MTTR) defines a quantitative maintainability measure. The smaller the MTTR, the better the maintainability of a system.

Availability

Availability A is a measure of the delivery of a correct service with respect to the alternation of correct and incorrect service, and is measured by the fraction of time that the system is ready to provide the service.

$$A = MTTF / (MTTF + MTTR)$$

$$MTBF = MTTF + MTTR$$

MTBF ... Mean-Time-Between-Failures

Availability is high when MTTF is high and MTTR is low.

Safety

Safety is reliability regarding critical failure modes. For a critical failure mode, the cost of a failure can be orders of magnitude higher than the utility of the system during normal operation.

In many cases the design of a safety-critical real-time system must be approved by an independent agency.

Security

It is the ability of the system to **prevent unauthorized access** to information or services.

During last years, security issues have become (very) important in real-time systems.

e.g. starting the engine of a car by inserting an USB stick with specially prepared MP3 music files.

Strategies to Handle Failures

There are different strategies to handle failures depending on the operational requirements of a system:

1. **Fail-stop systems**: after detection of any failure, the system is stopped immediately.
2. **Fail-safe systems**: after detection of any failure, the system actively establishes a safe state of the controlled object.
3. **Fail-operational systems**: failures need to be tolerated within a given fault hypothesis and the operation must be continued.

Fail-operational systems require **fault-tolerance**.

Types of Fault-Tolerance

Systematic fault tolerance provides the fault-tolerance mechanism at the level of the architecture by using replicated hardware.

→ clean architecture

Application-specific fault tolerance mixes fault-tolerance functions and application functions, thus increasing the software complexity but possibly reduces the level of required hardware redundancy.

→ more economic

Achieving Fault-Tolerance

Fault-Tolerance is achieved by masking failures of components. Therefore redundancy is needed.

1. **Information redundancy** provides fault tolerance through replicating or coding of data. Examples of information redundancy are parity memory, ECC (Error Correcting Codes) memory, and ECC codes on data blocks.
2. **Time redundancy** achieves fault tolerance by performing an operation several times. Timeouts and retransmissions in reliable point-to-point and group communication are examples of time redundancy. This form of redundancy is useful in the presence of transient or intermittent faults.
3. **Physical redundancy** deals with replication of devices. It adds extra equipment to enable the system to tolerate the loss of some failed components. RAID disks and backup name servers are examples of physical redundancy.

A Node as a Unit of Failure

For systematic fault-tolerance in a distributed real-time system, a **node** is considered to be an appropriate **unit of failure**. A node is a self-contained unit that provides a function across a small well-defined external interface as well as sufficient measures to prevent the propagation of errors outside the node (**error containment region**). A failure of a node thus corresponds to the failure of the function of the node.

In distributed systems it is common that nodes are replicated to achieve fault-tolerance.

Tolerating Failures

To tolerate k failures of a certain type, we need:

1. k+1 components if the nodes are **fail-silent**. Failures of such nodes are consistent failures of omission (no result in case of failure)
2. 2*k+1 components if the nodes can fail **fail-inconsistent**. Failures of fail-inconsistent nodes may be perceived inconsistent by different observing nodes.
3. 3*k+1 components if the failures are **fail-inconsistent** and agreement between nodes has to be reached [1].

Redundant nodes are grouped together to form fault-tolerant units.

Fault-Tolerant Unit

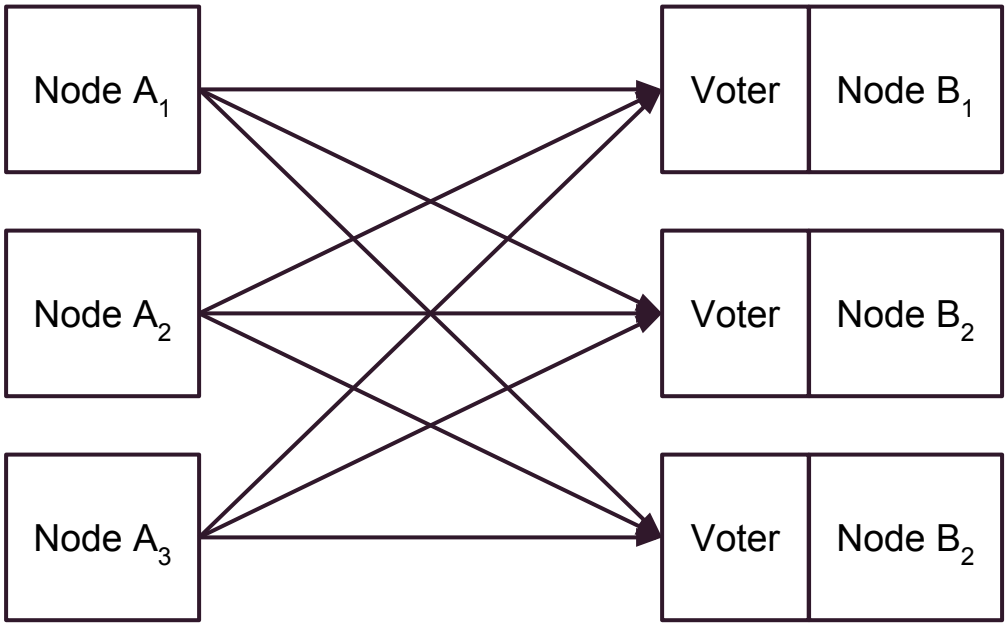
The purpose of a Fault-Tolerant Unit (FTU) is to mask the failures of nodes within a defined fault-hypothesis.

Depending on the required node failure properties, FTUs have to be constructed differently.

The nodes of an FTU perform the same redundant function.

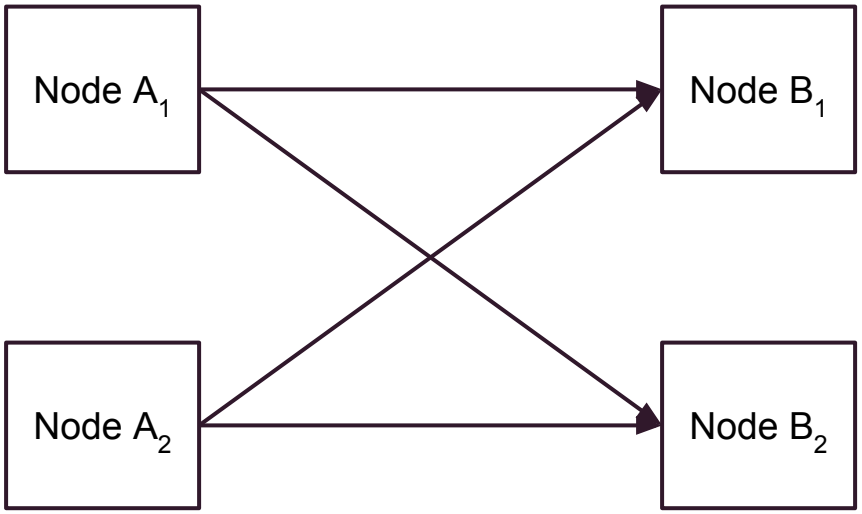
Fail-Inconsistent Nodes

Nodes A_1, A_2 and A_3 are **fail-inconsistent**. Each node B has to select a majority of the values. This system is fault tolerant for k=1 (TMR).



Fail-Silent Nodes

Node A_1 and A_2 are **fail-silent** and deliver a correct value or no value to nodes B. Each node B has to select one of one OR one of two values. This system is fault tolerant for k=1.



Node Properties

The following table gives an incomplete guide how to establish the different node properties. It is clear that fail-silence is the most expensive property at the node level.

Fail-inconsistent node	Model for COTS* components. Voter and voting protocol is complex. Operations need to be synchronous.
Fail-silent node	Self-checking of node in value and time domain to achieve a high error detection coverage - diverse software - multiple execution of software - control flow monitoring - memory protection - time monitoring - error detection and correction codes - avoidance of babbling idiot errors Independent mechanism to switch off the node in case of errors. Replica determinism is required.

* COTS: commercial-off-the-self

Voters

A Triple-Modular Redundancy (TMR) fault-tolerant unit consists of three nodes and voters.

With **exact voting**, a bit-by-bit comparison of the data fields in the redundant result messages is performed. Exact voting requires replica determinate computational channels.

With **inexact voting** two messages are assumed to contain the same result if the results are within some application-specific interval (must be used if no replicated nodes available)

Fault-Tolerance Mechanisms

Fault-tolerance mechanisms at the architecture level must perform two major tasks:

1. **State management**: observe and react on state changes of the nodes in the system. It triggers mode changes and handles state degradation. One important service element is a **membership service**.
2. **Redundancy management**: mask node failures by voting and trigger restart and reintegration of nodes.

Reintegration

The first step after a node failure is a self-test of the node hardware. If this self-test is successful, then, it can be assumed that a transient fault led to the failure of the node. The reintegration can be started immediately.

If the self-test assumes a permanent hardware fault, the node must be replaced. In a fault-tolerant system, is must be possible to replace a node while the system is under power.

The key issue during the reintegration of a node in a real-time system is to find a future point in time when the **h-state** of a node is in synchrony with the node environment.

A point in time, when a component is in **ground state**, is an ideal reintegration point, because the size of the h-state is minimal.

Fault-Tolerant Communication

In the presence of failures of the communication system special protocols are required to perform coordinated and agreed actions among nodes in a distributed real-time system.

The following types of coordination and agreed actions are common:

1. Membership service
2. Message ordering
3. Distributed agreement
4. Atomic multicast

Membership Service

A membership service delivers information about the activity status of nodes in a distributed system.

The membership information can have the following properties:

1. agreed or not agreed between all participating nodes
2. age of membership information
3. fault-tolerance

E.g., Membership information may be used to determine if the system has sufficient nodes available to perform a certain activity, or to enter a degraded mode.

Membership Service

A fault-tolerant membership protocol contains at least three mechanisms:

1. Collection of node statuses
2. Agreement over node status vector
3. Distribution of agreed node status vector

The most difficult problem to solve is the detection of **cliques** of nodes. Such cliques may each contain a subset of nodes which regard the other clique as offline.

Nodes which observe a deviation between membership observation and the agreed membership vector have to assume a failure inside themselves. They have to leave the membership group.

Message Ordering

A message ordering service ensures that messages transmitted between sender and receiver nodes are received in the same order as they were transmitted.

A message ordering service may have several levels of quality [2]:

1. **FIFO ordering** ensures the transmission sequence of one sender is the same as the reception sequence from this sender by all receivers
2. **Causal ordering** ensures that a message transmitted as reaction to the reception of a previous message is received by all receivers after the previous message
3. **Total ordering** ensures that all group members see the same order of received messages

Distributed Agreement

In distributed systems with faulty channels, special protocols are required to reach agreement among nodes about certain values (e.g. mode of operation).

The complexity of agreement protocols depends mainly on the fault-hypothesis for the participating nodes (see fault-tolerance chapter).

The most complex version is the Byzantine Agreement Protocol which can tolerate up to k fail-inconsistent nodes with 3*k+1 nodes and k+1 rounds of message exchange (very expensive and time consuming in hard real-time systems).

Atomic Multicast

This service ensures that in a group all members receive a sent message, or no one receives the message.

This protocol can be used to ensure a consistent input stream of messages for replicated nodes.

Atomic multicast protocols differ in

1. number of messages required to reach agreement
2. number of failures and faulty nodes tolerated
3. time needed to reach agreement

Validation

1) Safety Case
2) Safety Standards
3) Testing
4) Fault Injection
5) Formal Methods

Motivation

Validation deals with the question: “Is this system fit for its purpose?”

Before a safety critical system can be put into operation, convincing evidence must be gathered from independent sources to ensure that the system is safe to operate.

For different areas of application specific standards exist that define what it means “is safe to operate”. This judgement is based on expert knowledge, experience and a contract between society and operator about how much risk can be taken.

e.g. for automobiles the risk is tolerated that a specific number of persons die in car accidents per million cars in operation.

Safety Case

A **safety case** is a combination of a set of arguments supported by an analytical and experimental evidence concerning the safety of a given design. The safety case must argue why it is extremely unlikely that a single fault will cause a catastrophic failure. The arguments included in the safety case will have a major influence on the design decisions. Hence, the outline of the safety case should be planned early in a project.

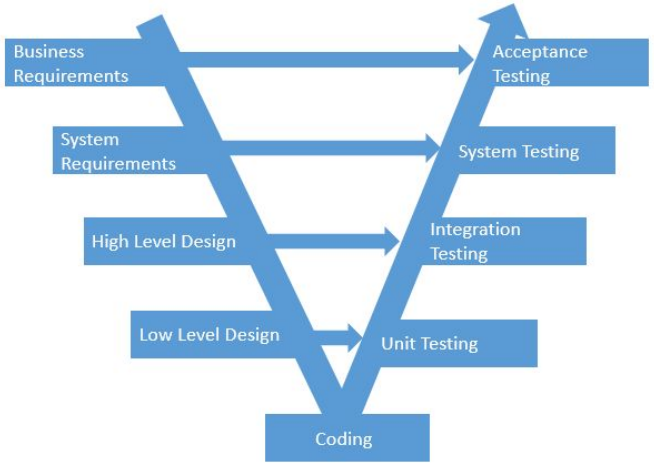
A safety case combines the evidence from independent sources to convince the certification authority that the system is safe to deploy. Examples of sources of evidence are

- disciplined development process
- results from inspection and testing
- formal verification of critical properties
- experience with similar system
- diverse designs, ...

Development Process

The domain specific safety standards usually define the requirements for the development process models.

Typically such models follow the V-model pattern.



Safety Standards

A number of domain specific safety standards exist.

Usually it is the responsibility of the system operator to choose and follow the required safety standard of the given domain.

E.g. There exists a generic safety standard IEC 61508 with domain specific derived standards::

- IEC 61511: Functional safety for process industries
- IEC 61513: Requirements and recommendations for the instrumentation and control for systems important to safety of nuclear power plants.
- EN 50128: Safety relevant electronic systems for railway systems
- IEC 62061: Machinery specific electrical controls systems
- ISO 26262: Functional safety of road vehicles

Safety in Real-Time Systems

The execution of multiple software partitions with different safety requirements leads to a new requirement for “**freedom from interference**” between the software partitions.

For an embedded real-time system software partitioning usually means:

1. hardware supported memory protection
2. time budget monitoring and enforcement
3. control flow monitoring
4. watchdogs
5. multiple execution
6. structured exception handling

...

Testing

In real-time systems, the functional as well as the temporal behavior of the system must be tested under the specified **load-hypothesis** and **fault-hypothesis**.

Testing of a distributed real-time system needs special consideration since it is usually complex to control the load and fault situation over several distributed nodes within the required synchronization.

E.g. execute a distributed state-machine in lock-step mode requires an augmentation of the operating system in all nodes.

The Probe Effect

Observability of the outputs of a subsystem under test and controllability of the test inputs are the core of any testing activity in real-time systems.

The modification of the behavior of the object under test by introduction of a test probe is called the **probe effect**. The challenge in testing distributed real-time systems lies in designing a test environment that is free of the probe effect.

Cluster Simulation

An appropriate solution to establish controllability and observability in a distributed system is to use a **cluster simulation** approach.

In a cluster simulation, one device under test (DUT) is connected to a system which simulates all other nodes in the system. A cluster simulator

- generates the required test input values for the DUT
- checks the outputs of the DUT for correctness
- simulates all sequences and protocols to bring the DUT into the desired mode of operation

There are different approaches for cluster simulation, like software-in-the-loop (software only), hardware-in-the-loop (includes hardware and IO of DUT).

Test Data Selection

During the test phase, only a tiny fraction of the potential input space of a software system can be tested. The challenge for the tester is to find an effective and representative set of test data that will uncover a high percentage of the unknown faults.

Beside the valid value domains, also invalid input values shall be used for testing.

The peak-load scenarios as well as above-peak situations should be tested extensively. To test the correctness of the fault-tolerant mechanism, fault injection can be used.

Fault Injection

Fault injection is the intentional activation of faults by hardware or software means to be able to observe the system operation under fault conditions. During a fault-injection experiment, the target system is exposed to two types of inputs: the injected faults and the input data.

Fault injection serves two purposes during the evaluation of a dependable system.

1. **Test and debugging**: Faults are rare events that occur only infrequently. Fault-tolerance mechanisms require faults for activation
2. **Robustness evaluation**: In many applications it is important to know how robust the system is when the fault-hypothesis does not hold or the load hypothesis does not hold, or both.

Fault Injection

It is possible to inject faults at the physical level of the hardware (physical fault-injection) or into the state of the computation (software implemented fault-injection).

During physical fault-injection the target hardware is subjected to adverse physical phenomena that interfere with the correct operation of the computer hardware (e.g. EMI, ESD, radiation, temperature,..).

For distributed real-time systems, fault injection tests in the communication system are important. At least the resilience against faults from the fault-hypothesis has to be tested. Usually there are tools available for bus fault injection.

There are several tools available for software implemented fault-injection.

Formal Methods

Formal methods support the correct design and validation of safety critical systems.

Fault-tree analysis is a methodology to identify hazards and to increase the safety of complex systems. The fault-tree analysis always begins at the system level with the identification of the undesirable failure event (the top event in the tree).

Failure Mode and Effect Analysis (FMEA) is a technique for systematically analyzing the effects of possible failure modes of components within a system to detect weak spots of the design, and to prevent system failures from occurring.

The FMEA is complementary to the Fault-Tree Analysis. While the Fault-Tree Analysis starts from the top event, FMEA starts with the components and investigates the failure effects on the system.

Sources

[1] The Byzantines General Problem, L. Lamport, R. Shostak, M. Pease, SRI International, 1982, ACM Transactions on Programming Languages and Systems

[2] Distributed Network Systems. From Concepts To Implementations. Weijia Jia,Wanlei Zho, Springer Verlag