



TECHNISCHE
UNIVERSITÄT
WIEN

TTTech

Dependable Systems

191.109

(formerly 182.712)

Überblick

191.109 Dependable Systems

→ Bitte ueber TISS zur LVA registrieren (Ankuendigen werden ueber TISS verschickt).

Vortragende

- Poledna, Stefan
- Puschner, Peter
- Steiner, Wilfried

Einführung in die Laborübung: wird ebenfalls per TISS/TUWEL bereitgestellt

Naechster Pruefungstermin: voraussichtlich April

Inhalt der LVA

- Dependable systems and incidents
- Basic concepts and terminology
- Fault-tolerance and Modeling
- Processes and Certification Standards
- Failure modes and models
- System aspects of dependable computers

Dependable Systems

Part 1: Dependable systems and incidents

Contents

- Dependability Problem Statement
- Examples of dependable systems and incidents
- The Therac-25 accidents
- Unintended Acceleration Incidents
- Reasons for low dependability
- Concept of coupling and interactive complexity

Dependability Problem Statement

Our society depends on a broad variety of computer controlled systems where failures are critical and may have severe consequences on property, environment, or even human life.

Aims of this lectures

- to understand the attributes and concepts of dependability,
- to understand reasons for low dependability and
- gain knowledge on how to build dependable computer systems

NASA Orion



Boeing 787



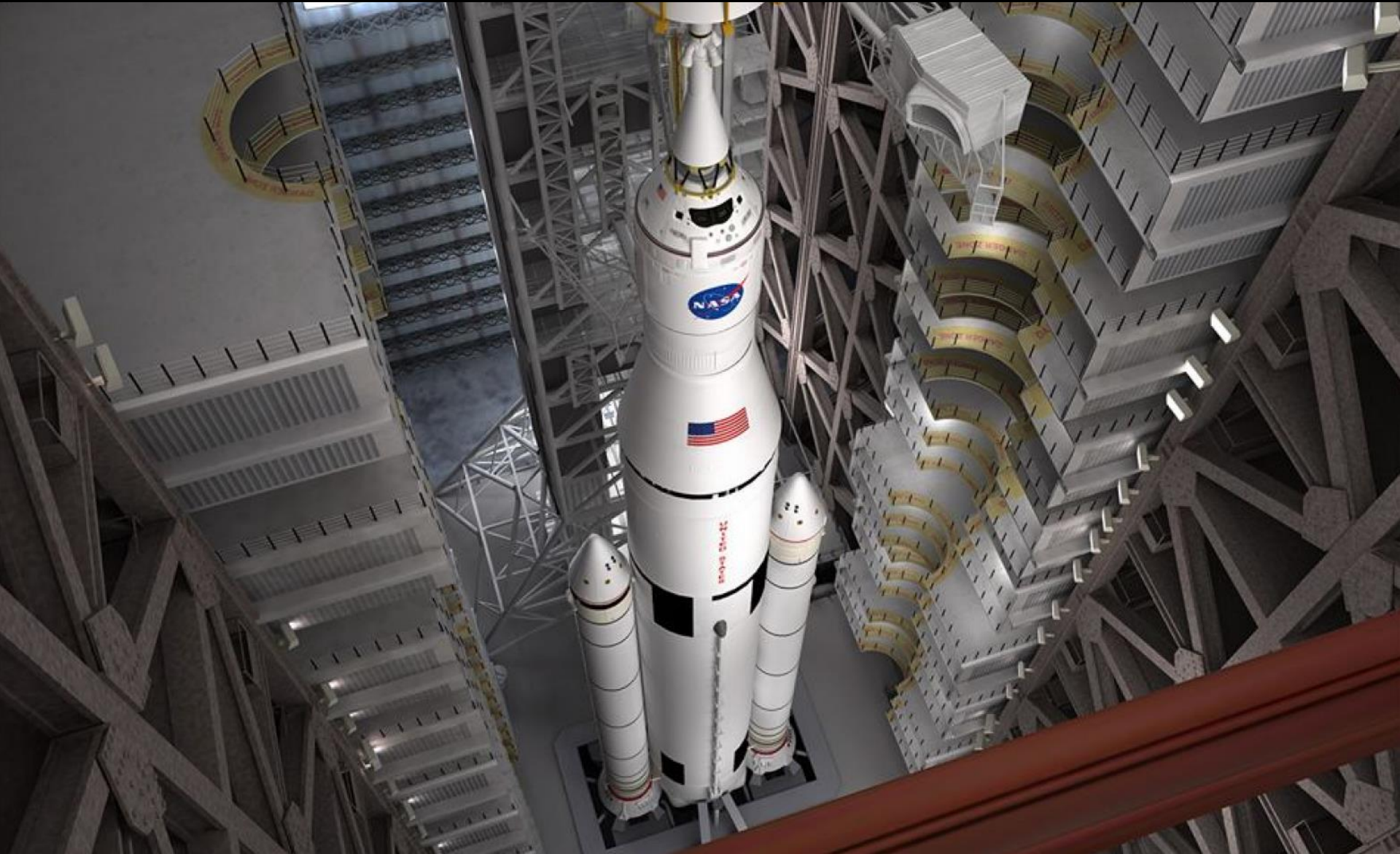
Airbus A380



Audi A8



America's New Rocket: Space Launch System



The Future of Human Space Exploration

NASA's Building Blocks to Mars

U.S. companies provide affordable access to low Earth orbit

Mastering the fundamentals aboard the International Space Station

Pushing the boundaries in cis-lunar space

Developing planetary independence by exploring Mars, its moons, and other deep space destinations

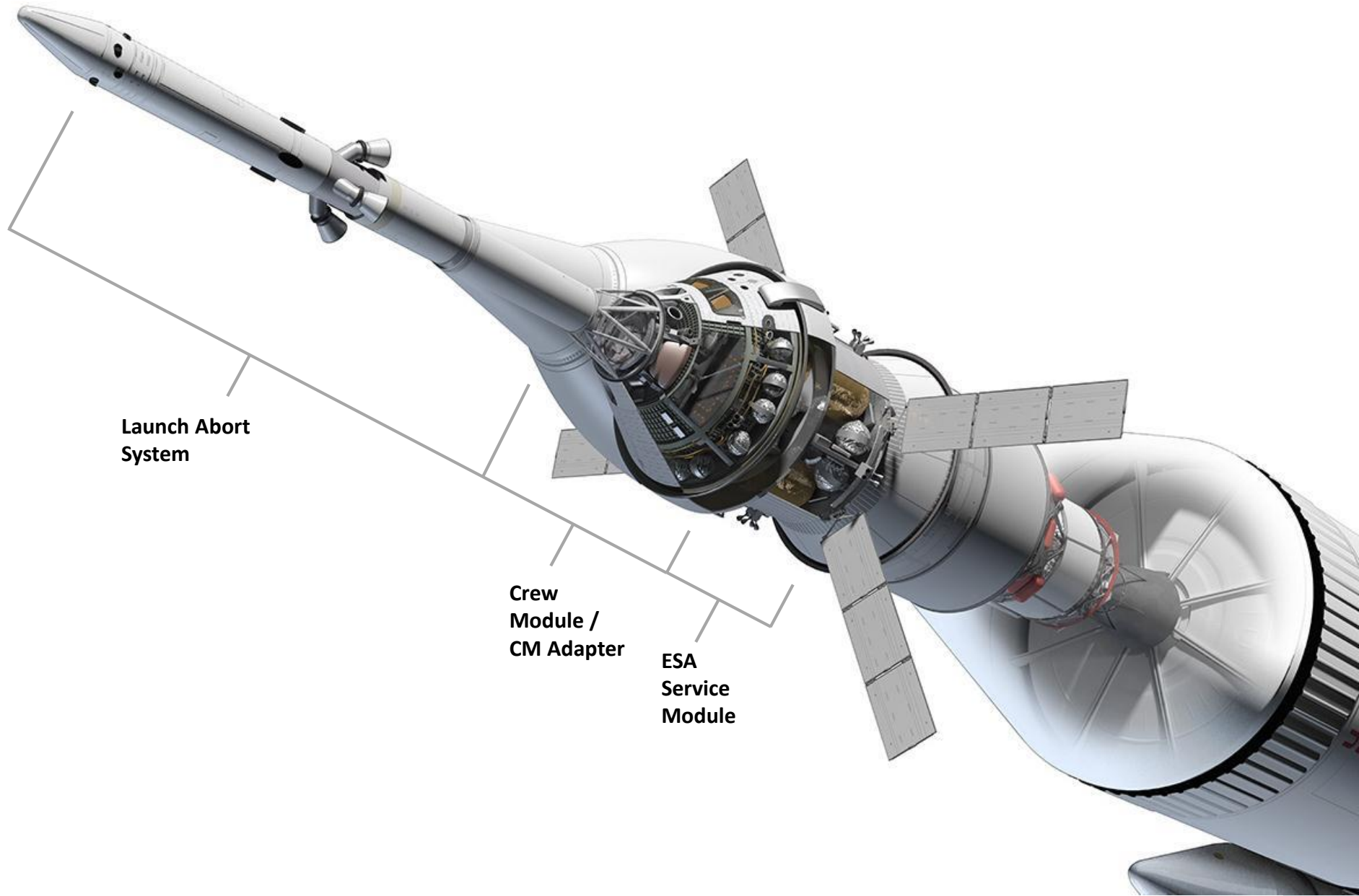
The next step: traveling beyond low-Earth orbit with the Space Launch System rocket and Orion crew capsule

*Missions: 6 to 12 months
Return: hours*

*Missions: 1 month up to 12 months
Return: days*

*Missions: 2 to 3 years
Return: months*

The Orion Spacecraft



Launch Abort System

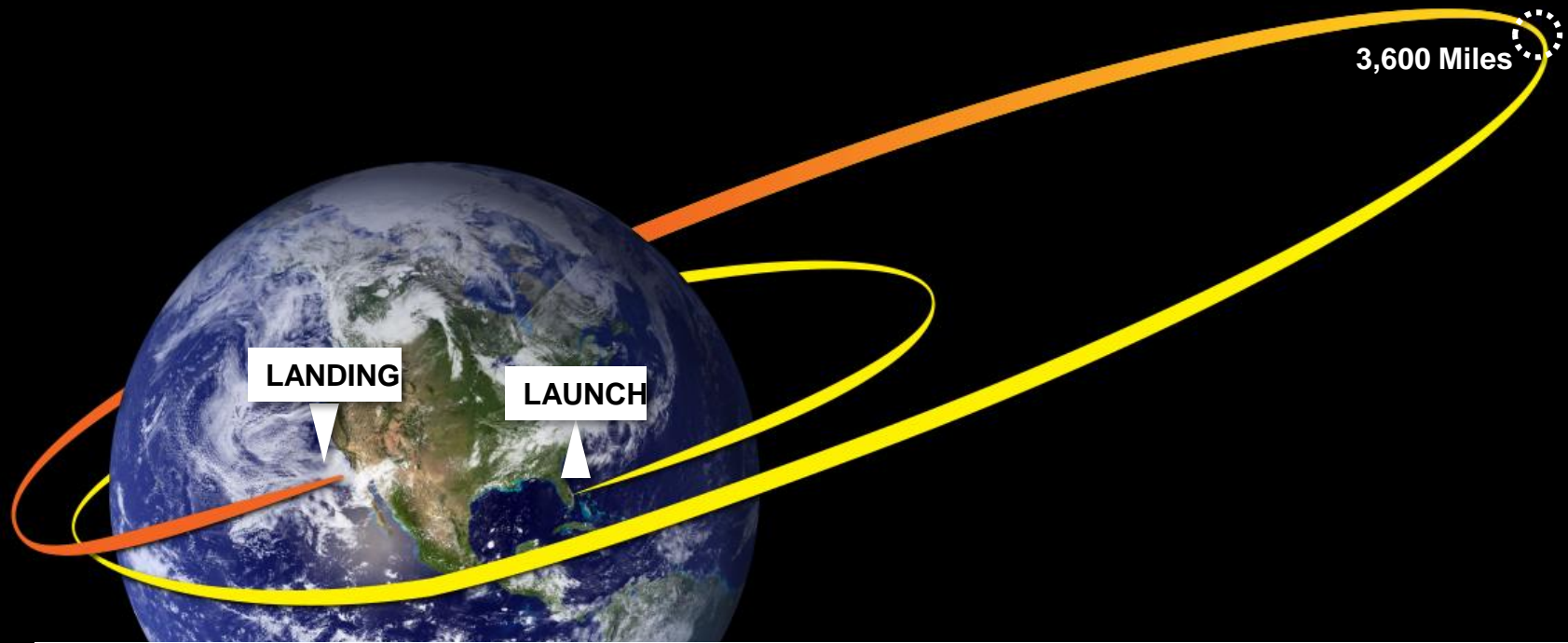
Crew Module /
CM Adapter

ESA Service Module



This year NASA will fly a spacecraft built for humans farther than any has traveled in over 40 years.

2 Orbits | 20,000 MPH entry | 3,600 Mile Apogee | 28.6 Deg Inclination



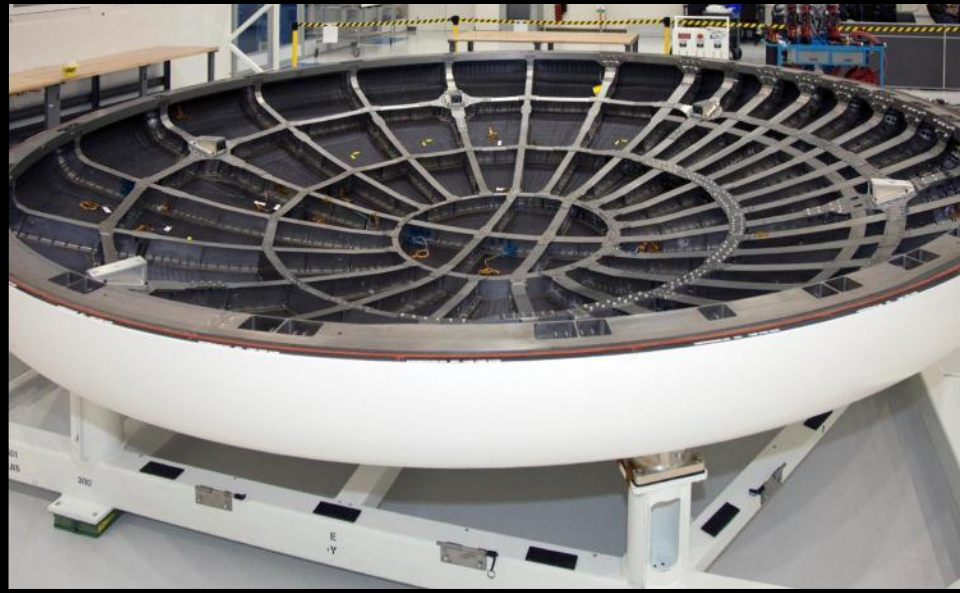
Launched Dec/05, 2014

<https://www.nasa.gov/specials/orionfirstflight/>

EFT-1 WILL EXERCISE 10 TOP LOSS OF CREW RISKS

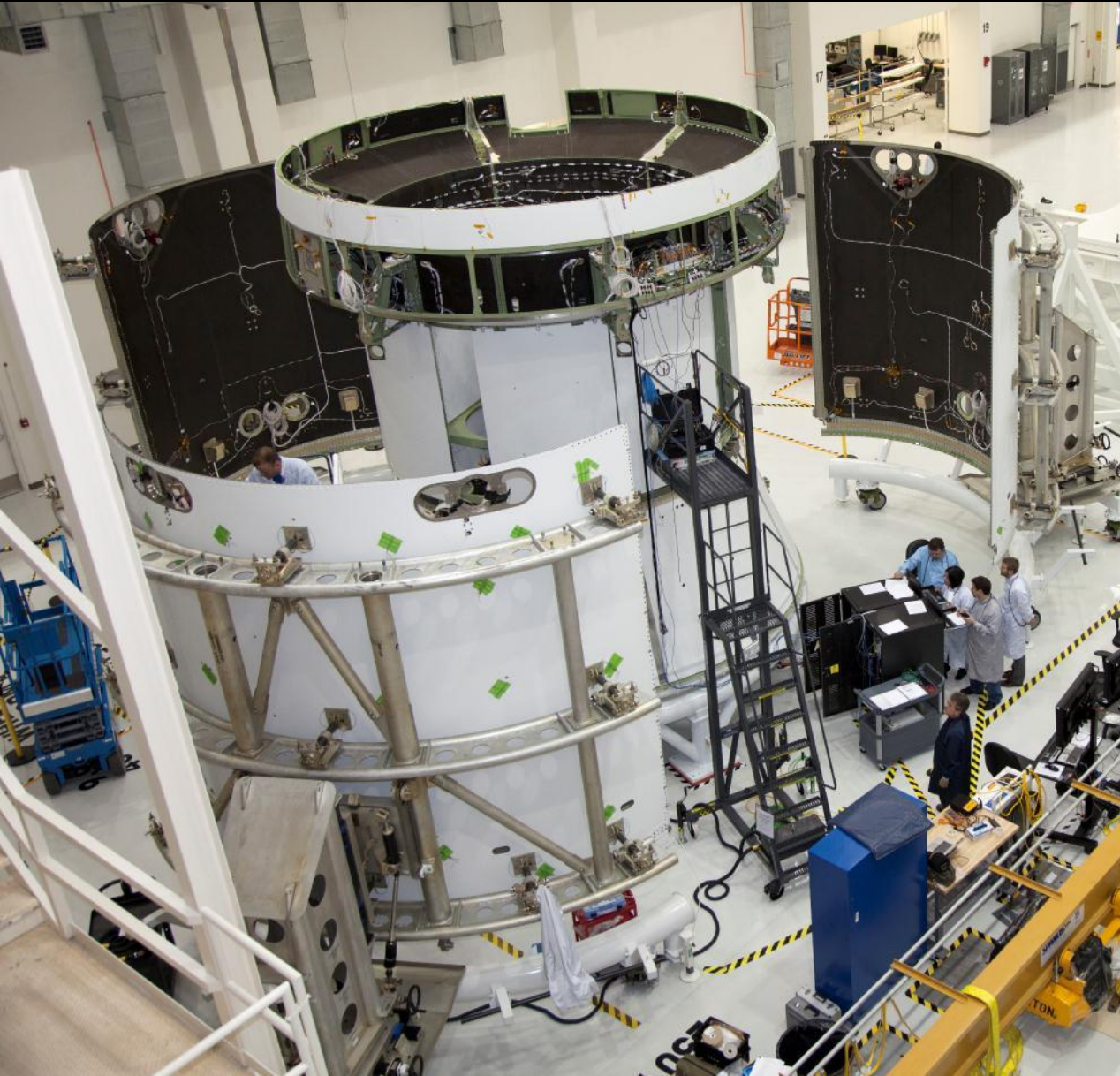
Crew Module

Functional Testing Underway; On Track for May Delivery



Service Module

Assembly Complete – Ready for Integration



Launch Abort System

Assembly Complete – Ready for Integration

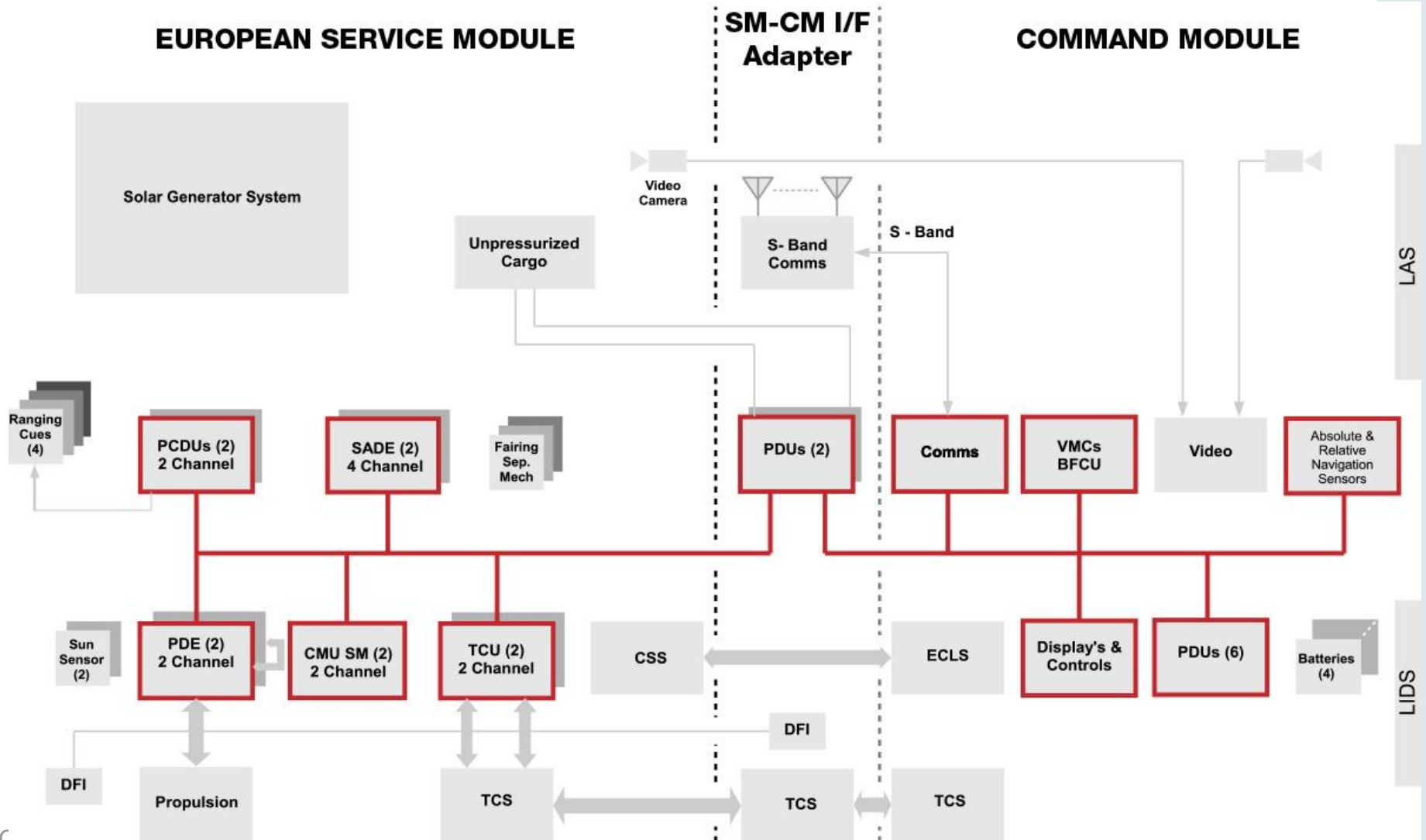


Time Triggered Gigabit Ethernet



The Backbone of Orion's State of the Art, High Reliability Avionics System

48 Network end points | 3 planes of connectivity for every device



Development Agreement Between TTTech and Airbus Safran Launchers

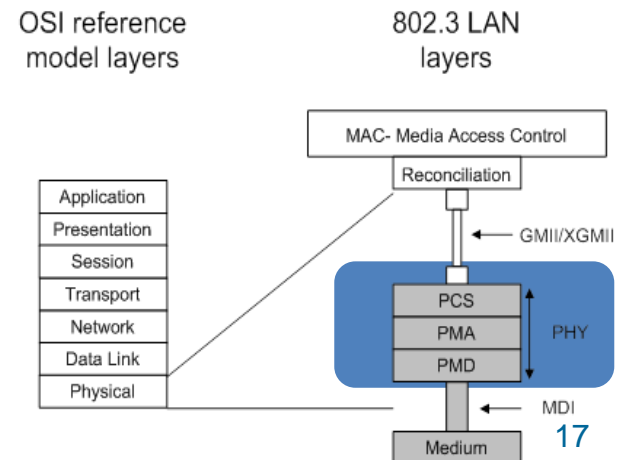
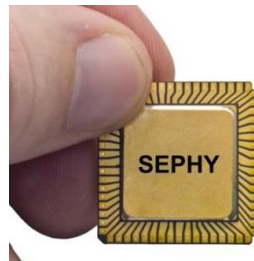
July 08, 2016

TTTech recently signed a long-term development agreement with European space market leader Airbus Safran Launchers to utilize [TTEthernet](#) as avionics backbone in the Ariane 6 family of launchers.

Main elements of the contract between Airbus Safran Launchers and TTTech include product lifetime technical support, software development (firmware, tooling) and co-funding for the production and qualification of 100/1000 Mbps TTEthernet semiconductors intended to be used in all major subsystems of the launcher. The declared goal of Airbus Safran Launchers' parent company and TTTech is a wider usage of Deterministic Ethernet technology in various different spacecraft (including satellites) in order to simplify development, maximize reuse, and thus reduce system lifecycle costs across all space programs. The radiation-tolerant components support both strictly deterministic (Time-Triggered Ethernet and ARINC A664 compliant) as well as standard,



- Increase the European competitiveness by delivering the solution that could be used worldwide
- Develop an ITAR-free and radiation hardened 10/100-Base-T Ethernet transceiver (PHY) for the space market
- ASIC will be used in a harsh environment which can produce undesired effects on electronic devices
- Enable Ethernet based technologies to become an international space standard in future applications
- TRL 7 – a system prototype demonstration in a space environment (the test campaign presented in the project covers all the elements required to guarantee a proper performance of the device under space environment)



Examples of dependable systems and incidents

“Fly-by-wire”

- pilot commands are transmitted as electrical commands
- a flight control system (FCS computer) is used
- the pilot flies the FCS and the FCS flies the plane
- military planes require FCS to get artificial stability
- for civilian use the advantages are:
 - weight savings
 - enhanced control qualities
 - enhanced safety

Fly-by-Wire Incidents

The SAAB JAS Gripen:

- 1989: Crash after sixth test flight due to exceeded stability margins at critical frequency, software was updated
- 1993: Crash on a display flight over the Water Festival in Stockholm, again due to pilot commands the plane became instable
- the cycle time of the Gripen FCS is 200 *ms*
- the probability of instability was estimated by the engineers as “sufficiently low”

The Airbus A320:

- 4 hull losses (plane crashes)
- all crashes are attributed to a mixture of pilot and computer or interface failures

A332, en-route, Atlantic Ocean, 2009

- Jun/1, 2009
- Airbus A330-200 being operated by Air France on a scheduled passenger flight from Rio de Janeiro to Paris CDG as AF447
- exited controlled flight and crashed into the sea with the loss of the aircraft and all 228 occupants
- loss of control followed an inappropriate response by the flight crew to a transient loss of airspeed indications in the cruise which resulted from the vulnerability of the pitot heads to ice crystal icing.

http://www.skybrary.aero/index.php/A332,_en-route,_Atlantic_Ocean,_2009

Patriot vs. Scud

During gulf war a Scud missile broke through the Patriot anti-missile defense barrier and hit American forces killing 28 people and injuring 98.

A software problem

- time is represented as an 32 *bit* integer and converted to 24 *bit* real number
- with the advent of time this conversion loses accuracy
- tracking of enemy missiles becomes therefore faulty
- the software problem was already known, and the update was delivered the next day

Critical Infrastructure Incidents

Bank of America financial system:

- development during 4 years costs \$20 millions
- \$60 millions in overtime expenses
- \$1.5 billion in lost business
- system was abandoned after nearly one year in service

Airport of Denver, Colorado

- one of the largest airports worldwide
- intelligent luggage transportation system with 4000 “Telecars”, 35 *km* rails, controlled by a network of 100 computers with 5000 sensors, 400 radio antennas, and 56 barcode readers
- due to software problems about one year delay which costs 1.1 million \$ per day

More Examples

Harsh environment:

- The “bug”: On a Mark II in 1945 a moth came between relay contacts
- train cars were changed from external to disc brakes, trains vanished from display
- near a broadcast transmission tower it was possible to "hear rock and roll on the toaster"
- an overripe tomato hung over an answering machine, dripping tomato juice into the machine which caused repeated call to the emergency line
- pigeons may deposit a "white dielectric substance" in an antenna horn

Examples may seem funny but:

- systems are designed to endure within a given operational conditions
- it is **very hard** to anticipate the operational conditions correctly
- illustrates difficulties of *good* system design

Which other (recent) incidents are you aware of?

The Therac-25 accidents

The Therac-25 accidents

Therac-25 is a machine for radiation therapy (to treat cancer)

Between June 1985 and January 1987 (at least) six patients received severe overdoses:

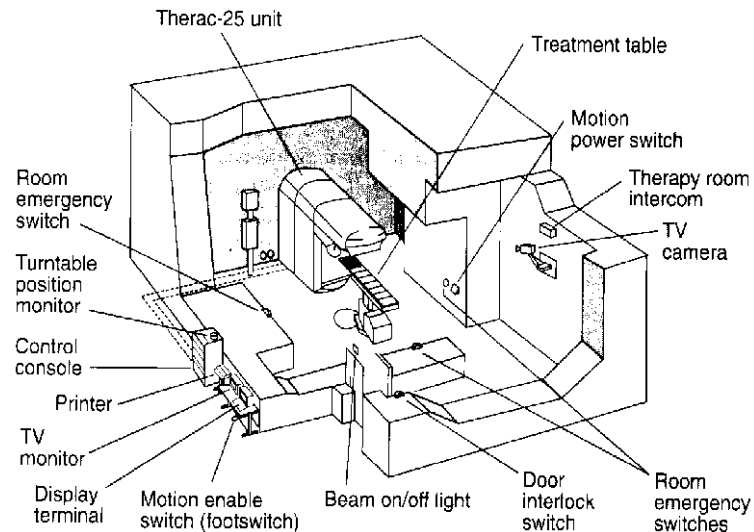
- two died shortly afterwards
- two might have died but died because of cancer
- the remaining two suffered of permanent disabilities

Functional principle

- Therac is a “dual-mode” machine
- electron beams are used for surface tumors
- X-ray for deep tumors
- “scanning magnets” are used to spread the beam and vary the beam energy

X-ray and Electron Mode

- a tungsten target and a “beam flattener” is moved in the path to the rotating turntable
- the target generates the X-rays but absorbs most of the beam energy
- the required energy has to be increased by a factor of 100, compared to electron mode



Typical Therac-25 facility

Major Event Time Line

	1985
Jun	3rd: Marietta, Georgia, overdose. Later in the month, Tim Still calls AECL and asks if overdose by Therac-25 is possible. 26th: Hamilton, Ontario, Canada, overdose; AECL notified and determines microswitch failure was the cause.
Jul	AECL makes changes to microswitch and notifies users of increased safety.
Sep	Independent consultant (for Hamilton Clinic) recommends potentiometer on turntable. Georgia patient files suit against AECL and hospital.
Oct	8th: Letter from Canadian Radiation Protection Bureau to AECL asking for additional hardware interlocks and software changes.
Nov	Yakima, Washington, clinic overdose.
Dec	1986
Jan	Attorney for Hamilton clinic requests that potentiometer be installed on turntable. 31st: Letter to AECL from Yakima reporting overdose possibility.
Feb	24th: Letter from AECL to Yakima saying overdose was impossible and no other incidents had occurred.

Major Event Time Line (cont. 1986)

Mar	21st: Tyler, Texas, overdose. AECL notified; claims overdose impossible and no other accidents had occurred previously. AECL suggests hospital might have an electrical problem.
Apr	7th: Tyler machine put back in service after no electrical problem could be found. 11th: Second Tyler overdose. AECL again notified. Software problem found. 15th: AECL files accident report with FDA.
May	2nd: FDA declares Therac-25 defective. Asks for CAP and proper renotification of Therac-25 users.
Jun	13th: First version of CAP sent to FDA.
Jul	23rd: FDA responds and asks for more information. First user group meeting. 26th: AECL sends FDA additional information.
Aug	30th: FDA requests more information.
Sep	12th: AECL submits revision of CAP.
Nov	Therac-20 users notified of a software bug.
Dec	11th: FDA requests further changes to CAP. 22nd: AECL submits second revision of CAP.

FDA =	US Food and Drug Administration
CAP =	Corrective Action Plan

Major Event Time Line (cont. 1987)

Jan	17th: Second overdose at Yakima. 26th: AECL sends FDA its revised test plan.
Feb	Hamilton clinic investigates first accident and concludes there was an overdose. 3rd: AECL announces changes to Therac-25. 10th: FDA sends notice of adverse findings to AECL declaring Therac-25 defective under US law and asking AECL to notify customers that it should not be used for routine therapy. Health Protection Branch of Canada does the same thing. This lasts until August 1987.
Mar	Second user group meeting. 5th: AECL sends third revision of CAP to FDA.
Apr	9th: FDA responds to CAP and asks for additional information.
May	1st: AECL sends fourth revision of CAP to FDA. 26th: FDA approves CAP subject to final testing and safety analysis.
Jun	5th: AECL sends final test plan and draft safety analysis to FDA.
Jul	Third user group meeting. 21st: Fifth (and final) revision of CAP sent to FDA.
	1988
Jan	29th: Interim safety analysis report issued.
Nov	3rd: Final safety analysis report issued.

Lessons learned from Therac-25 accident:

- Accidents are seldom simple
- Accidents are often blamed to single source
- Management inadequacies, lack of following incident reports
- Overconfidence in software
- Involvement of management, technicians, users, and government
- Unrealistic risk assessment
- Less-than-acceptable software-engineering practices

Who would ride on an autonomous car?

Unintended Acceleration Incidents

Unintended Acceleration Examples

<https://www.youtube.com/watch?v=cOWdWHSgl-4>

I will show a video with accidents for the next 5 minutes.
In case anyone prefers to leave the room, that is of course possible!

Toyota Unintended Acceleration Incident

2007/Sep: Toyota recall to fasten floor mats

2009/Aug: Toyota Lexus ES 350 sedan crash

- unintended acceleration reached 100 mph
- four passengers died, 911 emergency phone call during event
- crash was blamed on wrong floor mats causing pedal entrapment

2009/Oct: Extended floor mat recalls

2010/Jan: Sticky gas pedal recall

2010/Feb: US congressional investigation

2010/May: CBS News “Toyota Unintended Acceleration has killed 89”

2010-2011: NASA investigation of unintended acceleration

- conclusion: no electronic-based cause for unintended high-speed acceleration
- tight timeline and limited information

2012/Dec: Toyota settlement for \$1.6 Billion USD

http://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf

Toyota Unintended Acceleration Incident (cont.)

2013/Oct: Bookout/Schwarz Trial

- 2007 crash of a 2005 Toyota Camry
- Dr. Koopman & Mr. Barr testified as software experts
- Testified about defective safety architecture and software defects

Jury awarded \$3 million compensation

Key technical element of criticism is the Electronic Throttle Control System (ECTS)

http://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf

Electronic Throttle Control System (ETCS)

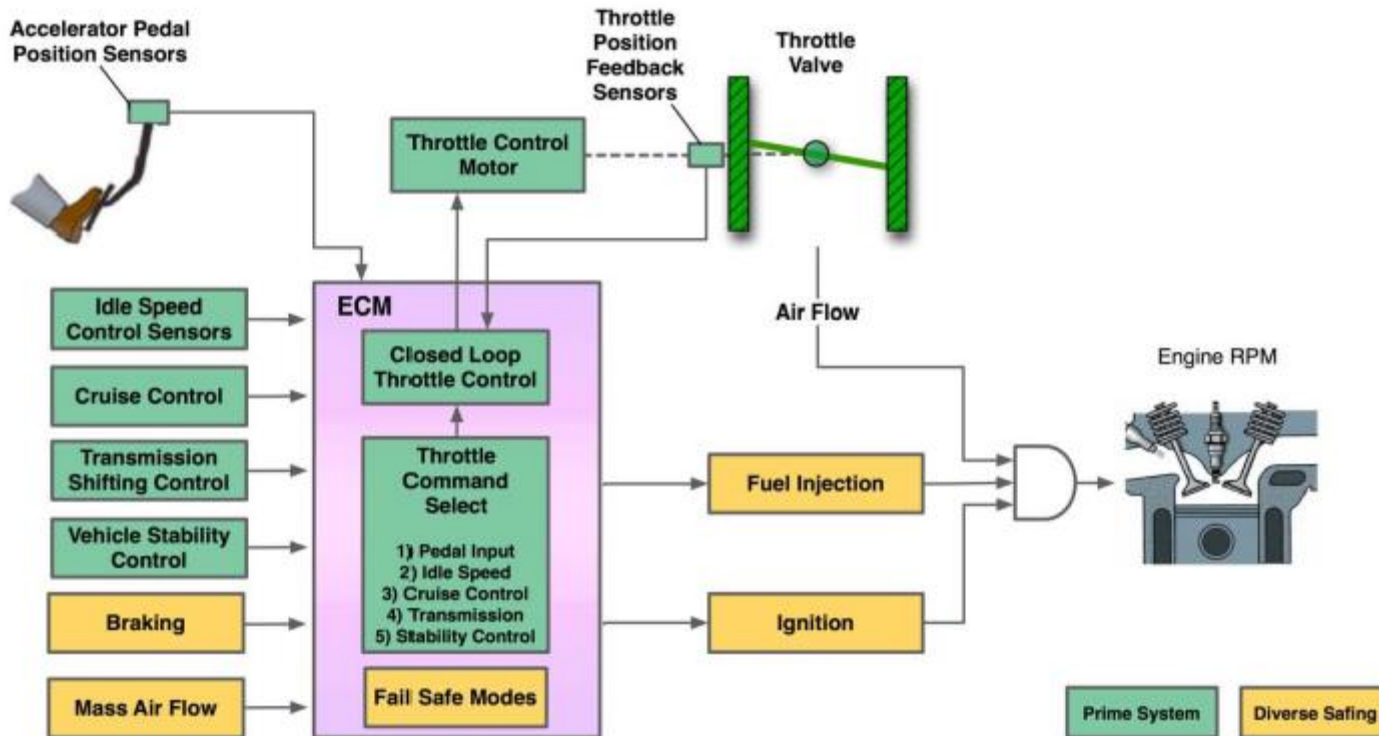


Figure 6.4-1. ETCS-i Major Functions

http://www.nhtsa.gov/staticfiles/nvs/pdf/NASA-UA_report.pdf

ETCS Criticism

Safety architecture

- Shortcomings in failsafes
- Shortcomings in the watchdog design
- Non-independent Fault-Containment Regions

Software Quality

- 256,600 Non-Commented Lines of C source
- 9,273 – 11,528 global variables (ideally 0 writable globals)
- Spagetti code, untestable functions according to McCabe cyclomatic complexity metric
- Use of recursion, no mitigation for stack overflow
- Concurrency issues

ETCS Criticism (cont)

Certification

- Critical SW is typically developed by following standardized processes, e.g., MISRA SW Guidelines
- Toyota does not claim to have followed MISRA
- Mike Barr's team found 80,000 violations of MISRA C

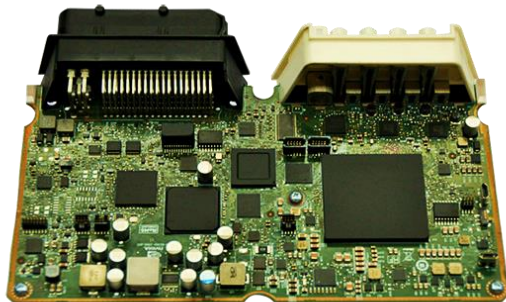
Who would ride on an autonomous car?

System Classification by VDA

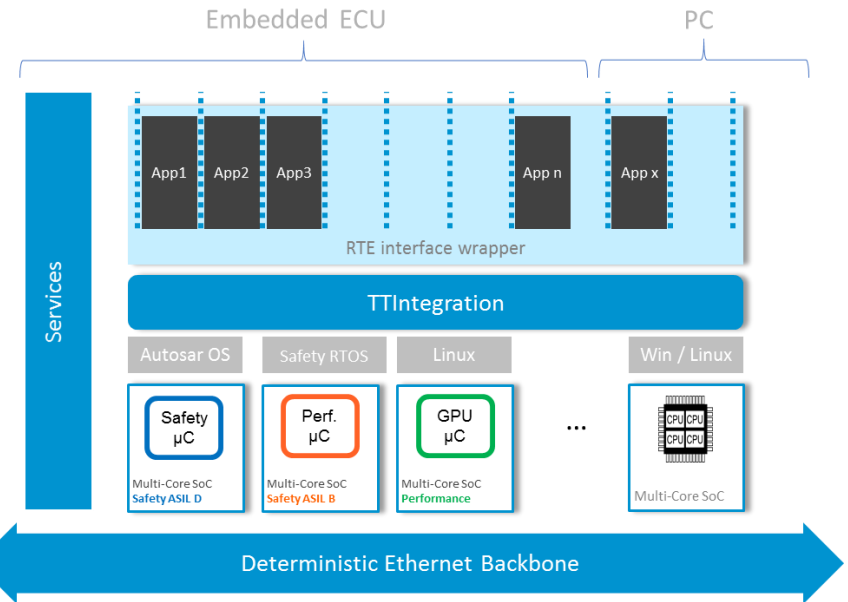


Classification according to VDA

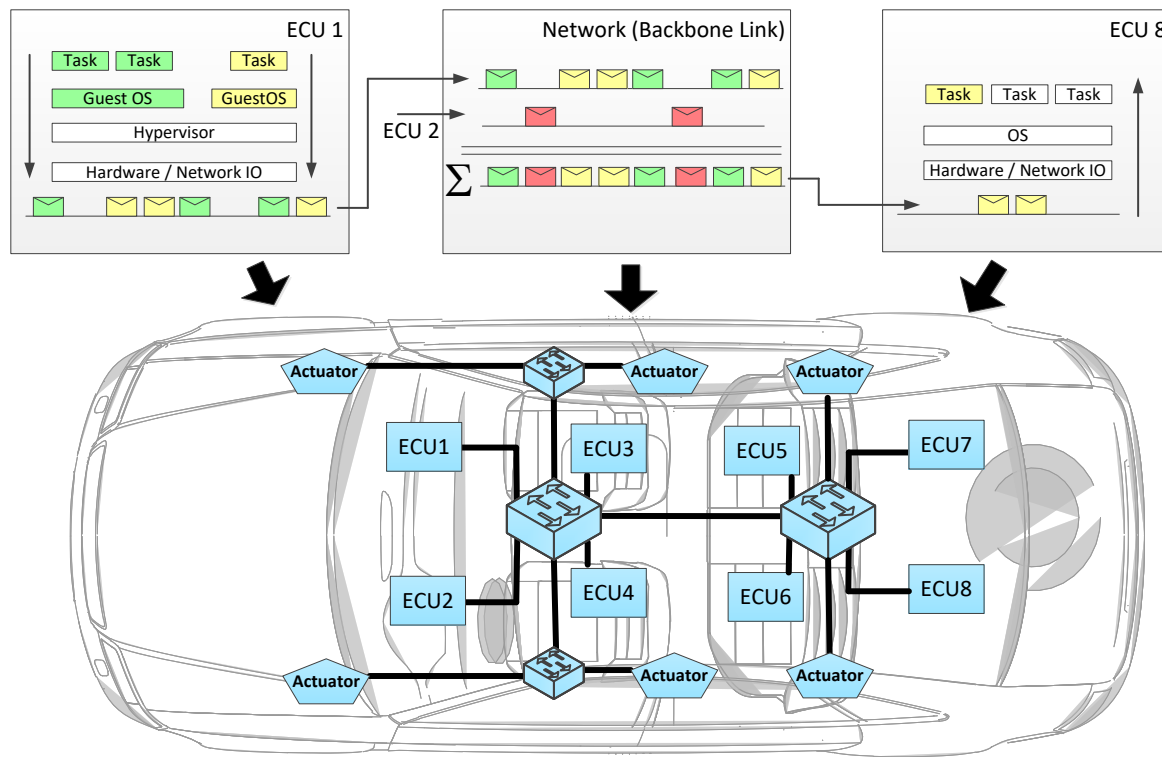
Electronic Control Units



MotionWise



Vehicle-Wide Virtualization



Reasons for low dependability

What would you think are reasons for low dependability?

Reasons for low dependability

- **Chips with everything:**
Computers are increasingly used for all types of devices and services.
- **Interface design:**
Complex systems must have a “friendly” interface that is easy to understand and must not confuse or mislead the user.
- **The “system” includes the operator:**
The total system requires some functions to be carried out by the operator.
- **The “system” includes the documentation:**
Operator failures may occur due to hard to understand or misleading documentation.
- **The “system” includes its operating procedures:**
Just as the operator and the documentation are regarded as part of the system, so must the procedures for using it.

Reasons for low dependability (cont)

- **“System” failures are human failure:**
Not only the operator, but other humans and ultimately the designer are causing system failures.
- **Complexity:**
Problem inherent complexity—not solution induced complexity—is hard to handle.
- **System Structure:**
Unsuitable system structures can lead to low dependability
- **Wrong assessment of peak load scenario:**
Systems can only be designed to handle a priori known peak load scenarios.
- **Wrong assessment of fault hypothesis:**
Systems can only be designed to handle a priori known fault hypothesis.

Reasons for low dependability (cont.)

- **Low dependability of components:**
“A system is as strong as its weakest link”
- **Misunderstanding of application:**
Customer and system manufacturer have different understandings of the services
- **Incomplete problem description:**
Unintended system function due to incomplete problem description
- **Coupling and interactive complexity:**
cf. next slide
- **Discontinuous behavior of computers:**
cf. foil after slide
- **No system is fool-proof**

Concept of coupling and interactive complexity

The concept of coupling and interactive complexity is a model to explain what type of systems are potentially hazardous [Perrow 1984].

- **Tightly coupled systems:**

In a tightly coupled system components affect one another automatically with great rapidity, so that errors propagate too quickly for a human operator to detect, contain and correct them.

- **Interactive complex systems:**

In an interactive complex system components interact in many ways simultaneously, so that the behavior of the system (as a whole) is inherently difficult to understand.

Problem of discontinuous behavior or the Problem of Software

- discrete computers are symbol manipulating machines
- symbols are represented in binary form of 0's and 1's
- computers are finite state machines
- large state space (combinatorial explosion)
- mapping of actual state and input to new state
- in contrast to analogue systems there is no continuous trajectory
- discontinuous trajectories are intractable by simple mathematics
- is worse than chaotic behavior (of analog systems)
- continuous or analog systems have an infinite number of stable states while discrete systems have only a small (finite) number of stable states

Dependable Systems

Part 2: Basic Concepts and Taxonomy

Contents

- The Basic Concepts
- The Threats to Dependability and Security (Fault – Error – Failure)
- The Means to Attain Dependability
- Error Recovery and Redundancy
- On the Importance of the Specification

The Basic Concepts

Avizienis, Algirdas, J-C. Laprie, Brian Randell, and Carl Landwehr. "Basic concepts and taxonomy of dependable and secure computing." *IEEE transactions on dependable and secure computing* 1, no. 1 (2004): 11-33.

Dependability Definitions

How would you define a dependable system?

Dependability Definitions

Original: Dependability is the ability to deliver service that can justifiably be trusted.

Alternate: Dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable.

System Function, Behavior, Structure, and Service

A **system** is an entity that interacts with other entities, i.e., other systems.

For a particular system A, the sum of all the other systems system A is interacting with is referred to as the **environment** of system A.

The **system boundary** is the common frontier between a system and its environment.

System Function, Behavior, Structure, and Service (cont.)

The **function** of a system is what the system is intended to do.

The function is described in the **functional specification**.

The **behavior** of a system is what the system does to implement its function and is described by a sequence of states.

System Function, Behavior, Structure, and Service (cont.)

The **structure** of a system is what enables it to generate the behavior.

In terms of a structure, a system is composed of **components** bound together to interact.

Components are systems which can be *composed of other components*.

Alternatively, a component is said to be **atomic**, in case the inner structure of the component is of no interest.

System Function, Behavior, Structure, and Service (cont.)

A system is the **provider** of a **service** to one or many **users**.
Users are, again, systems.

The **service interface** between the provider and the one or many users is the respective part of the provider's system boundary.

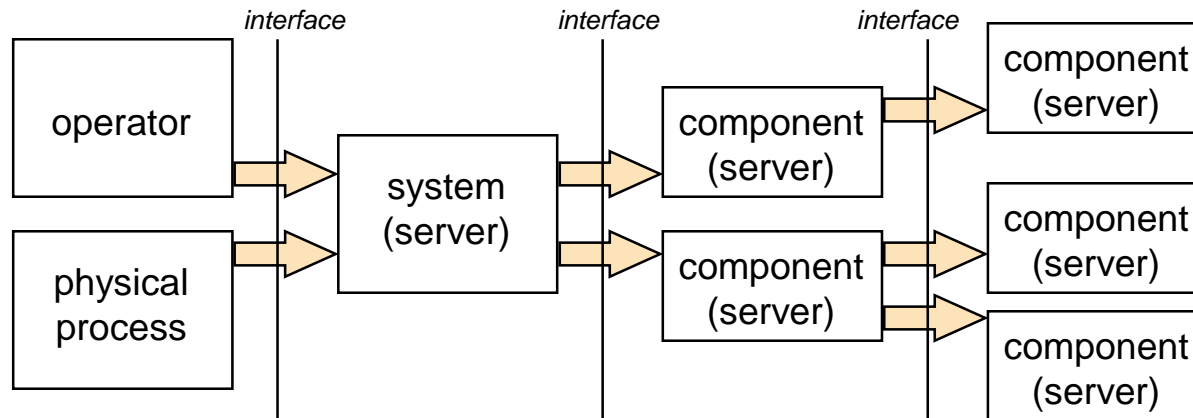
The part of the provider's total state that is perceivable at the service interface is its **external state**. The remaining part is its **internal state**.

The interface of the user at which the user receives the service is the **use interface**.

System Function, Behavior, Structure, and Service (cont.)

Recursive nature of the *depends* (\Rightarrow) relation

- service users depend on the services provided by the system (server)



Definitions of Failure – Error – Fault

Correct service is delivered when the service implements the system function.

A **(service) failure** is an event that occurs when the delivered service deviates from correct service.

- Thus, a failure is a transition from correct service to incorrect service.

The different kinds of incorrect service delivery are referred to as the **failure mode** and these modes are ranked according to **failure severity**.

Definitions of Failure – Error – Fault (cont.)

A service failure means that at least some external state of the provider service deviates from the correct state.

This deviation is called the **error** (i.e., a deviation from the current state from the correct state).

The adjudged or hypothesized cause of an error is called a **fault**.

Attributes of Dependability

Reliability: continuity of correct service.

Availability: readiness for correct service.

Maintainability: ability to undergo modifications and repairs.

Safety: absence of catastrophic consequences on the user(s) and the environment.

Integrity: absence of improper system alterations.

Reliability vs. Availability

Reliability is the probability that the system will conform to its **functional specification** throughout a period of duration t .

Availability is the percentage of time for which the system will conform to its specification (also considering repair actions).

→ Availability is a function of reliability and maintainability.

Reliability vs. Availability (cont.)

Can you think of an example system that needs to be highly-available but reliability is less of an issue?

Reliability vs. Availability (cont.)

Factory automatization:

- the computer has to assure proficient manufacturing
- availability is most important parameter
- reliability is not that important

Satellite:

- once put into operation there is no possibility for maintenance
- mission reliability is most important parameter

Reliability vs. Safety

Reliability is the probability that the system will conform to its **functional specification** throughout a period of duration t .

Safety is the probability that the system will not exhibit **specific undesired behaviors** throughout a period of duration t .

→ In general, not all deviations from the functional specification imply specific undesired behaviors in the sense of the safety definition.

.

Reliability vs. Safety (cont.)

What would be an example of a loss of reliability does/does not lead to a safety incident?

Reliability vs. Safety (cont.)

- **Railway signalling:**
 - red signal is a safe system state
 - safe system state is unreliable
 - safety \neq reliability
- **Fly-by-wire airplane control:**
 - after take off there is no safe (non-functional) system state
 - safety \approx reliability
(degraded modes of operation are possible)
- often there is a conflict between safety and reliability

.

Reliability vs. Safety (cont.)

- often there is a conflict between safety and reliability

→ Why?

.

Attributes of Security

Confidentiality: the absence of unauthorized disclosure of information.

+ **Integrity** (as before)

+ **Availability** (as before)

The Threats to Dependability and Security

Details on: Fault, Error, Failure

Life Cycle of a System

- Development Phase, including
 - initial system conception
 - system design, development, verification, and validation
- Use Phase, including
 - service delivery
 - service outage (service not available)
 - service shutdown (service not needed)
 - maintenance

Faults

Recap:

The adjudged or hypothesized cause of an error is called a **fault**.

Faults – eight elementary fault classes

Classification into **eight elementary fault classes**:

- Phase of creation or occurrence (development vs. use phase)
- System boundaries (internal vs. external)
- Phenomenological cause (natural vs. human-made)
- Dimension (hardware vs. software)
- Objective (malicious vs. non-malicious)
- Intent (deliberate vs. non-deliberate)
- Capability (accident vs. incompetence)
- Persistence (permanent vs. transient)

What would a Software Flaw (i.e., a “bug”) be classified as?

Classification into **eight elementary fault classes**:

- Phase of creation or occurrence (development vs. use phase)
- System boundaries (internal vs. external)
- Phenomenological cause (natural vs. human-made)
- Dimension (hardware vs. software)
- Objective (malicious vs. non-malicious)
- Intent (deliberate vs. non-deliberate)
- Capability (accident vs. incompetence)
- Persistence (permanent vs. transient)

Example Faults: Software Flaws

Software flaws (may) have the following aspects (in red):

- Phase of creation or occurrence (**development** vs. use phase)
- System boundaries (**internal** vs. external)
- Phenomenological cause (natural vs. **human-made**)
- Dimension (hardware vs. **software**)
- Objective (malicious vs. **non-malicious**)
- Intent (**deliberate** vs. **non-deliberate**)
- Capability (**accident** vs. **incompetence**)
- Persistence (**permanent** vs. transient)

Faults – combined fault classes

- A particular fault will typically fall into multiple of the eight elementary fault classes.
- Since three of the elementary fault classes are of particular importance, we use them to derive combined fault classes:
 - Phase of creation or occurrence (**development** vs. use phase) → **Development Faults**
 - System boundaries (internal vs. **external**) → **Interaction faults**
 - Dimension (**hardware** vs. software) → **Physical faults**

Failures

Recap:

A **(service) failure** is an event that occurs when the delivered service deviates from correct service.

- Thus, a failure is a transition from correct service to incorrect service.

Failure Mode Classification – Overview

- Domain:
 - content, early timing failure, late timing failure, halt failure, erratic failure
- Detectability:
 - signaled failures, unsignaled failures
- Consistency:
 - consistent failure, inconsistent failure
- Consequences:
 - minor failure, ..., catastrophic failure

Failure Mode Classification – Domain

- Content
- Early timing failure
- Late timing failure
- Halt failure
 - the external state becomes constant, i.e., system activity is no longer perceptible to the users
 - silent failure mode is a special kind of halt failure in that no service at all is delivered
- Erratic failure
 - not a halt failure, e.g., a babbling idiot failure

Failure Mode Classification – Consistency

When there are more than one users of a service.

- Consistent failure:
 - All users experience the same incorrect service.
- Inconsistent failure
 - Different users experience different incorrect services.

Failure Mode Classification – Consequences, e.g., Aircraft

Minor: **10E-5 per flight hour or greater**

no significant reduction of aeroplane safety, a slight reduction in the safety margin

Major: **between 10E-5 and 10E-7**

significant reduction in safety margins or functional capabilities, significant increase in crew workload or discomfort for occupants

Hazardous: **between 10E-7 and 10E-9**

large reduction in safety margins or functional capabilities, causes serious or fatal injury to a relatively small number of occupants

Catastrophic: **less than 10E-9**

these failure conditions would prevent the continued safe flight and landing of the aircraft

Error

Recap:

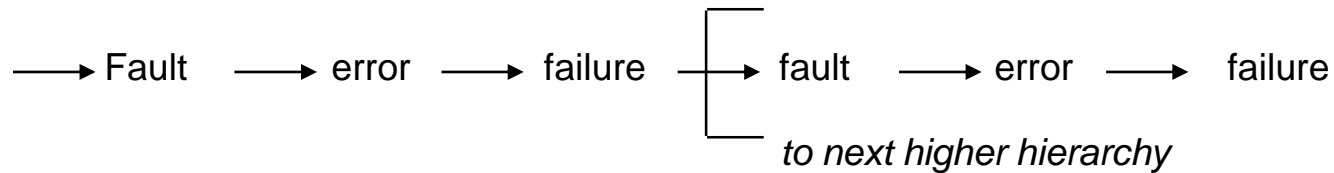
A service failure means that at least some external state of the provider service deviates from the correct state.

This deviation is called the **error** (i.e., a deviation from the current state from the correct state).

Error

- An error is detected if its presence is indicated by an error message or error signal.
- Errors that are present but not detected are *latent* errors.
- Whether or not an error actually leads to a failure depends on the following facts:
 - the system composition and the existence of redundancy (intentional or unintentional redundancy)
 - the system activity after the introduction of an error (the error may get overwritten)
 - the definition of a failure by the user's viewpoint

Fault – Error – Failure Chain



fault → error

- a fault which has not been activated by the computation process is *dormant*
- a fault is *active* when it produces an error

error → failure

- an error is *latent* when it has not been recognized
- an error is *detected* by a detection algorithm/mechanism

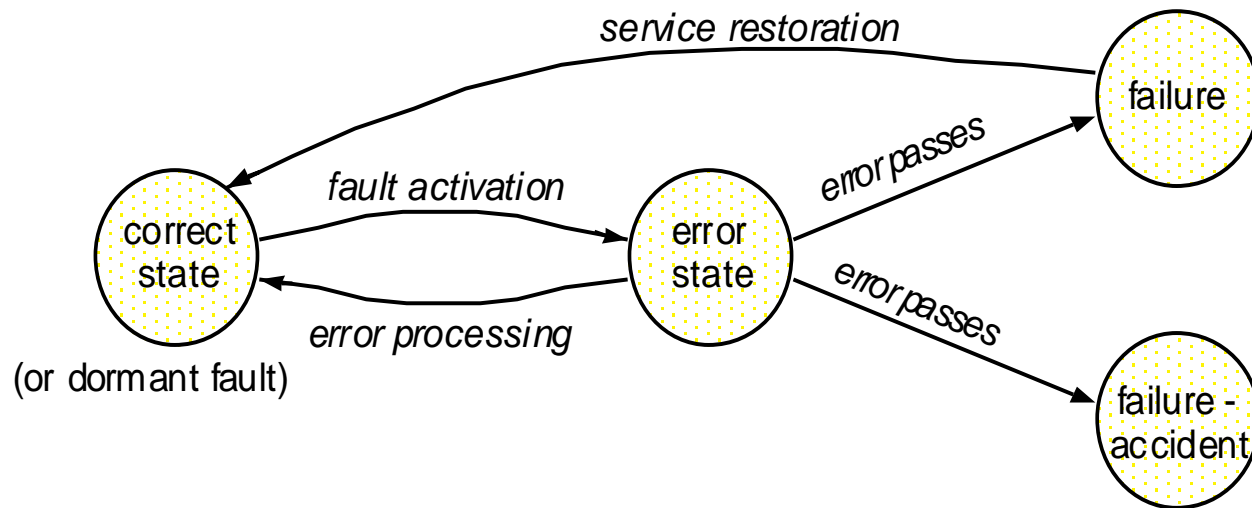
failure → fault

- a failure occurs when an error “passes through” and affects the service delivered
- a failure results in a fault for the system which contains or interacts with the component

Examples for fault/failure chain

- **Program error (software):**
 - a dormant *fault* in the written software (instruction or data)
 - upon activation the fault becomes active and produces an *error* (system state)
 - if the erroneous data affects the delivered service, a *failure* occurs
- **Electromagnetic interference (hardware):**
 - leads to *faulty* input value (either digital or analog)
 - by reading the input the fault becomes active and produces an *error*
 - if the erroneous input value is processed and becomes visible at the interface a *failure* occurs

Fault/failure state transition chart



The Means to Attain Dependability

Means to Attain Dependability and Security

Fault prevention: means to prevent the occurrence or introduction of faults.

Fault tolerance: means to avoid service failures in the presence of faults.

Fault removal: means to reduce the number and severity of faults.

Fault forecasting: means to estimate the present number, the future incidence, and the likely consequences of faults.

Fault prevention

- **hardware components:**
 - environment modifications (temperature)
 - quality changes, use “better” components
 - component integration level, higher integration
 - derating, reduction of electrical, thermal, mechanical, and other environmental stresses
- **software components:**
 - software engineering methodologies
 - OOD and OO languages
 - design rules
 - CASE tools
 - formal methods

Fault removal

- **verification:**
to check, whether the system adheres to the specification.
 - Static analysis: inspections, walk-throughs, data flow analysis, complexity analysis, compiler checks, correctness proofs, petri net models, finite state automata.
 - Dynamic Analysis: testing, black-box, white-box, conformance, fault-finding, functional, timeliness, structural, deterministic, random or statistical
- **diagnosis:**
diagnosing the fault which prevented the verification from succeeding
- **correction:**
perform corrective actions to remove the fault \Rightarrow regression verification

Fault forecasting

- performing an evaluation of the system with respect to faults
- evaluation of aspects such as:
 - reliability
 - availability
 - maintainability
 - safety
- see chapter “Fault-tolerance and modelling”

Fault tolerance

There are four phases, which, taken together, provide the general means by which faults are prevented from leading to system failures.

- **error detection:**
errors are the manifestations of faults, which need to be detected to act upon
- **damage confinement and assessment:**
before any attempt is made to deal with the detected error, it is necessary to assess and confine the extent of system state damage

Fault tolerance (cont.)

- **error recovery:**
error recovery is used to transform the currently erroneous system state into a well defined error-free system state
- **fault treatment and continued service:**
even if the error-free system state has been recovered it is often necessary to perform further actions to prevent the fault from being activated again

Error Recovery and Redundancy

Error recovery

There are two possibilities to transform the currently erroneous system state into an error-free system state:

- **Backward recovery:**

- system state is reset to a previously store error-free system state
- re-execution of failed processing sequence
- typical for data base systems
(it is not possible to predict valid system states)

- **Forward recovery:**

- system state is set to a new error-free system state
- typical for real-time systems with period processing patterns
(it *is* possible to predict valid system states)

Redundancy

A system requires some kind of redundancy to tolerate faults. This redundancy can be implemented in three different domains:

- **Domain of information:**
redundant information e.g. error correcting codes, robust data structures
- **Domain of space:**
replication of components, e.g. 2 CPU's, UPS (uninterruptable power supply)
- **Domain of time:**
replication of computations, e.g. calculate results by same (or different) algorithm a second time, sending messages more than once

Fault-tolerance in the domain of information

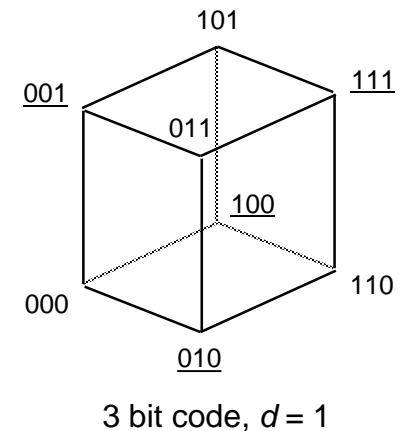
- **error correcting codes:**
 - for all error correcting codes (ECC)

$$(2t + p + 1) \leq d$$

d .. Hamming distance of code

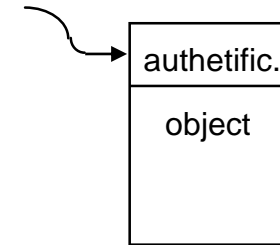
t ... number of single bit errors to be tolerated

p .. number of additional errors that can be detected



Fault-tolerance in the domain of information (cont.)

- **robust data structures:**
 - store the number of elements
 - redundant pointers
(e.g. double linked chains with status)
 - status or type information
(e.g. authenticated objects)
 - checksum or CRC
- **application specific knowledge**

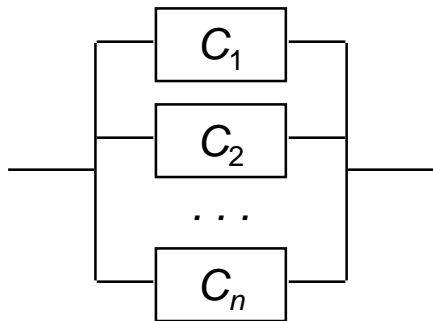


pointer to authenticated object

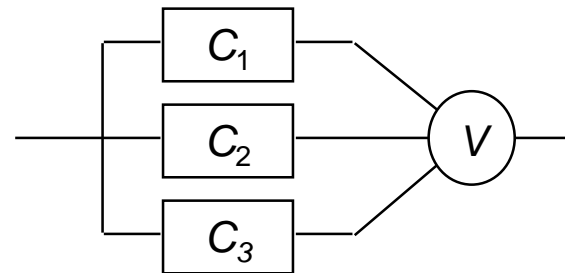
Fault-tolerance in the domain of space

- **active redundancy**

- parallel fail-silent components



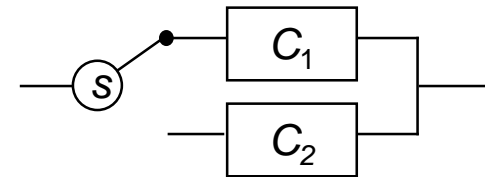
- voting, triple modular redundancy (TMR)



Fault-tolerance in the domain of space (cont.)

passive or standby redundancy

- hot standby:
standby component is operating
- cold standby:
standby components starts only
in case of a failure



Fault-tolerance in the domain of time

Allows tolerance of temporary faults

- **multiple calculation:**
 - a function is calculated n times with the same inputs
 - the result is checked by an acceptance test
 - or the multiple results are voted
- **sending messages multiple times:**
 - message transmission is repeated n times
 - retransmission only in case of failures (positive acknowledge retransmit PAR)
 - retransmission always n times (reduces temporal uncertainty for real-time systems)

On the Importance of the Specification

Specification

The definition of all dependability attributes is based on specifications. A *good* specification must be:

- exact
- consistent
- complete
- authoritative

Importance of specification

Together with the analysis of possible behavior and its consequences, system specification is the most difficult part of building a dependable system.

Specification (cont.)

Multiple levels of specifications

To consider the different aspects and attributes of dependable systems, usually different levels of specifications exists.

An example

<i>level</i>	<i>specification</i>
functional	“all commands have to be carried out correctly”
reliability	“either correct commands or warning indicator”
safety	“recorded info may not be corrupt”

The underground train

The underground train

- an electronically controlled underground train had the following buttons:
 - to open and close doors
 - to start the train
- it was specified that “the train only may start if and only if the start button is pressed and all doors are closed”
- a driver blocked the start train button by means of a tooth pick to start the train immediately if the doors were closed

The underground train (cont.)

What happened?

- one day a door was blocked and the driver went back to close the door, and of course, the train left the station without the driver

What went wrong?

- it was the drivers fault to block the start button with a tooth pick
- but it was also a specification fault since the correct specification should have read: “the train only may start if and only if the start button changes it state to start and all doors are closed”
- in that example it made a big difference whether *state* or *event*-semantics are implemented

Dependable Systems

Part 3: Fault-Tolerance and Modelling

Contents

- Reliability: Basic Mathematical Model
- Example Failure Rate Functions
- Probabilistic Structural-Based Modeling: Part 1
- Maintenance and Repair: Basic Mathematical Model
- Probabilistic Structural-Based Modeling: Part 2
- Open issues of probabilistic structural based models
- Reliability growth models
- Comparison of probabilistic modeling techniques
- Limits of validation for ultra-high dependability
- Example: Hardware Design Analysis at TTTech

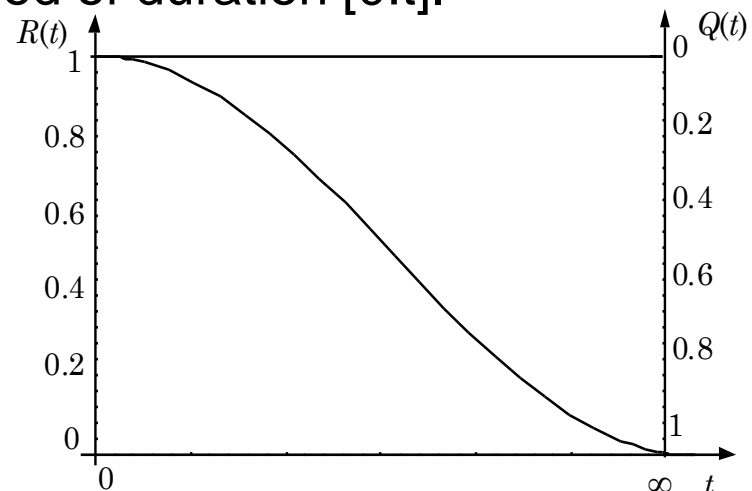
Reliability: Basic Mathematical Model

Failure Probability $Q(t)$

Reliability $R(t)$

- **Failure Probability $Q(t)$** , probability that the system will not conform to its specification throughout a period of duration $[0:t]$.
- **Reliability $R(t)$** , probability that the system will conform to its specification throughout a period of duration $[0:t]$.

- $R(0) = 1$ $R(\infty) = 0$
- $R(t) = 1 - Q(t)$



Failure Probability Density Function

- Def.: The failure density $f(t)$ at time t is defined by the number of failures during Δt .

$$f(t) = \frac{dQ(t)}{dt} = -\frac{dR(t)}{dt}$$

Failure Rate

- Def.: The failure rate $\lambda(t)$ at time t is defined by the number of failures during Δt in relation to the number of correct components at time t .

$$\begin{aligned}\lambda(t) &= \frac{f(t)}{R(t)} \\ &= -\frac{dR(t)}{dt} \frac{1}{R(t)}\end{aligned}$$

- The dimension of failure rate is FIT (failures in time)
 - x FIT = x failures per 10^9 hours

Example Failure Rates in FIT

(according to IEC TR 62380)

- Resistor 0.1 FIT
- Capacitor (ceramic) 2 FIT
- Capacitor (electrolytic) 7 FIT
- Diode 9 FIT
- Inductor 6 FIT
- Transistor (low power) 8 FIT
- Transistor (high power) 46 FIT
- Varistor 1 FIT
- Switching regulator 22 FIT
- Comparator IC 5 FIT
- Flash (46 MBit) 105 FIT
- EEPROM (512 kBit) 33 FIT
- CPU (180 MHz, Dualcore)
300 FIT (Hard Errors) /
2700 FIT (Soft Errors)
- High-side powerswitch 25 FIT
- Shift Register IC (8 Bit) 8 FIT
- 8 to 1 analog multiplexer IC
8 FIT
- CAN transceiver 7 FIT
- RS232 transceiver 9 FIT
- LIN transceiver 7 FIT
- Ethernet PHY 41 FIT
- Signal transformer 34 FIT

Example Failure Rate Functions

Constant Failure Rate

Used to model the normal-life period of the bathtub curve

- **failure rate**

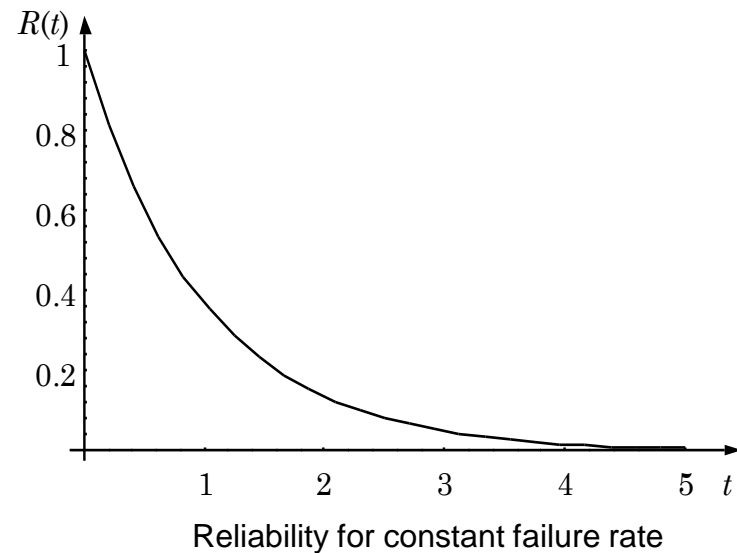
$$\lambda(t) = \lambda$$

- **probability density function**

$$f(t) = \lambda e^{-\lambda t}$$

- **reliability**

$$R(t) = e^{-\lambda t}$$



Weibull distributed failure rate

Used to model infant mortality and wear out period of components.

$\alpha < 1$: failure rate is decreasing with time

$\alpha = 1$: constant failure rate

$\alpha > 1$: failure rate is increasing with time

- **failure rate**

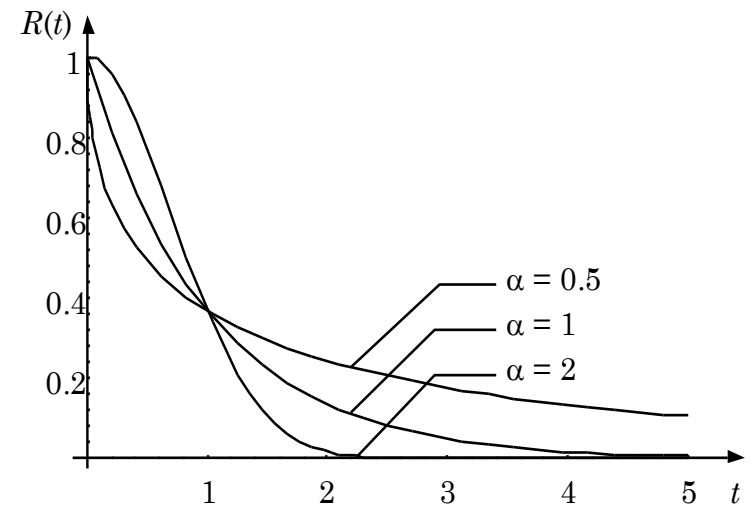
$$\lambda(t) = \alpha\lambda(\lambda t)^{\alpha-1}$$

- **probability density function**

$$f(t) = \alpha\lambda(\lambda t)^{\alpha-1}e^{-(\lambda t)^\alpha}$$

- **reliability**

$$R(t) = e^{-(\lambda t)^\alpha}$$



Reliability for weibull distributed failure rate

Lognormal distributed failure rate

For semiconductors the lognormal distribution fits more data collections than any other and is assumed to be the proper distribution for semiconductor life.

- **failure rate**

$$\lambda(t) = \frac{f(t)}{R(t)}$$

- **probability density function**

$$f(t) = \frac{1}{\sigma t \sqrt{2\pi}} e^{-\frac{1}{2} \left(\frac{\ln t - \mu}{\sigma} \right)^2}$$

- **reliability**

$$R(t) = 1 - \frac{1}{\sigma \sqrt{2\pi}} \int_0^t \frac{1}{x} e^{-\frac{1}{2} \left(\frac{\ln t - \mu}{\sigma} \right)^2} dx$$

Probabilistic Structural-Based Modeling: Part 1

Assumptions

- Identifiable (independent) components,
- Each component is associated with a given failure rate,
- Model construction is based on the structure of the interconnections between components.

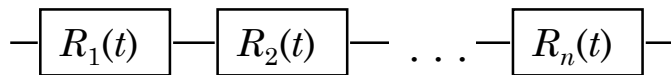
Example Modelling Paradigms

- Simple block diagrams
- Arbitrary block diagrams
- Markov models
- Generalized Stochastic Petri Nets (GSPN)

Simple block diagrams

- assumption of independent components
- combination of series or parallel connected components

Series Connection

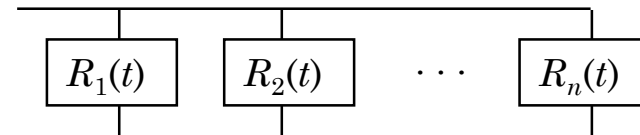


$$R_{series}(t) = \prod_{i=1}^n R_i(t)$$

$$Q_{series}(t) = 1 - R_{series}(t) = 1 - \prod_{i=1}^n R_i(t)$$

$$= 1 - \prod_{i=1}^n (1 - Q_i(t))$$

Parallel Connection



$$Q_{parallel}(t) = \prod_{i=1}^n Q_i(t)$$

$$R_{parallel}(t) = 1 - Q_{parallel}(t) = 1 - \prod_{i=1}^n Q_i(t)$$

$$= 1 - \prod_{i=1}^n (1 - R_i(t))$$

Simple block diagrams (cont.)

Constant failure rate

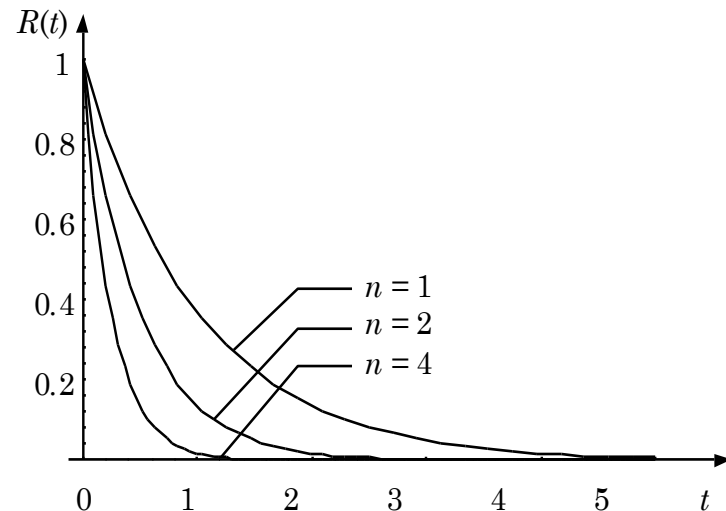
$$\lambda(t) = \lambda$$

$$R(t) = e^{-\lambda t}$$

Series connection

$$R_{series}(t) = \prod_{i=1}^n R_i(t) = \prod_{i=1}^n e^{-\lambda_i t}$$
$$= e^{-t \sum_{i=1}^n \lambda_i}$$

- the resulting failure rate for the system is still constant



Reliability of 1,2 and 4 series connected components with constant failure rate ($\lambda_1 = \lambda_2 = \lambda_3 = \lambda_4$)

Simple block diagrams (cont.)

Parallel connection

$$R_{parallel}(t) = 1 - \prod_i^n (1 - R_i(t))$$

$$= 1 - \prod_i^n (1 - e^{-\lambda_i t})$$

for 3 parallel components this gives:

$$R_{parallel}(t) = 1 - \left((1 - e^{-\lambda_1 t}) (1 - e^{-\lambda_2 t}) (1 - e^{-\lambda_3 t}) \right)$$

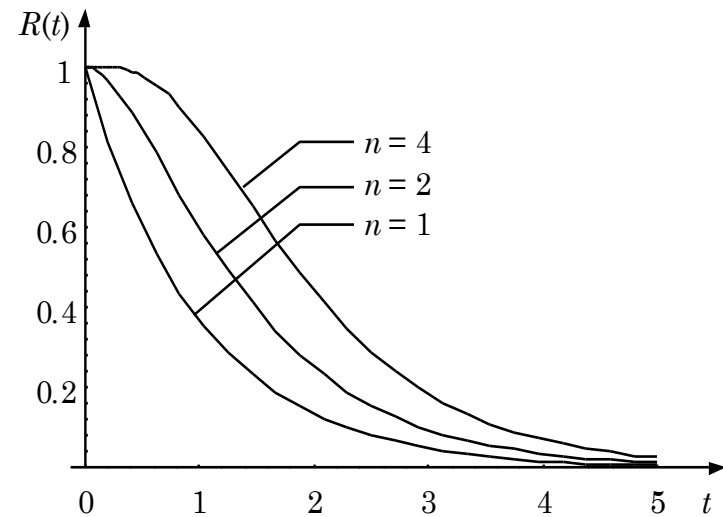
$$= e^{-\lambda_1 t} + e^{-\lambda_2 t} + e^{-\lambda_3 t} + e^{-(\lambda_1 + \lambda_2 + \lambda_3)t} -$$

$$e^{-(\lambda_1 + \lambda_2)t} - e^{-(\lambda_1 + \lambda_3)t} - e^{-(\lambda_2 + \lambda_3)t}$$

under the assumption $\lambda_1 = \lambda_2 = \lambda_3$ it follows

$$R_{parallel}(t) = 3(e^{-\lambda t} - e^{-2\lambda t}) + e^{-3\lambda t}$$

the resulting failure rate is no longer constant

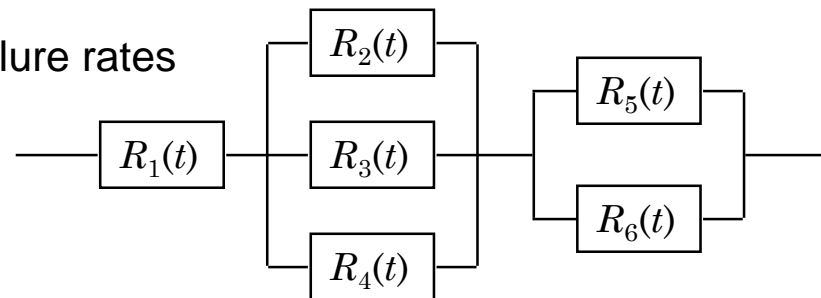


Reliability of 1,2 and 4 parallel connected components with constant failure rate ($\lambda_1 = \lambda_2 = \lambda_3 = \lambda_4$)

Simple block diagrams (cont.)

Pros:

- can be used to model arbitrary combinations of series and parallel connected components
- easy mathematics for constant failure rates

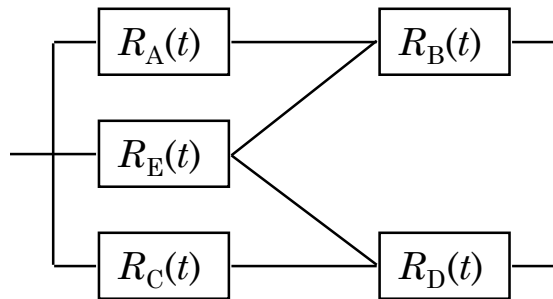


Cons:

- assumption of independent failures
- maintenance cannot be modeled
- restricted to series/parallel connection
- only for active redundancy and fail-silence

Arbitrary block diagrams

no restriction to series/parallel connections



$$R_{block}(t) = R_{AB} + R_{BE} + R_{DE} + R_{CD} - R_{ABE} - R_{ABCD} - R_{BDE} - R_{CDE} + R_{ABCDE}$$

$$R_{ABC} = R_{series}(A, B, C)$$

Inclusion/exclusion principle

1:	A	B			+
2:		B		E	+
3:			D	E	+
4:			C	D	+
12:	A	B		E	-
13:	A	B		D	-
14:	A	B	C	D	-
23:		B		D	-
24:		B	C	D	-
34:			C	D	-
123:	A	B		D	+
124:	A	B	C	D	+
134:	A	B	C	D	+
234:		B	C	D	+
1234:	A	B	C	D	-

Arbitrary block diagrams (cont.)

Active redundancy and voting

- for TMR 2 out of 3 components have to function correctly

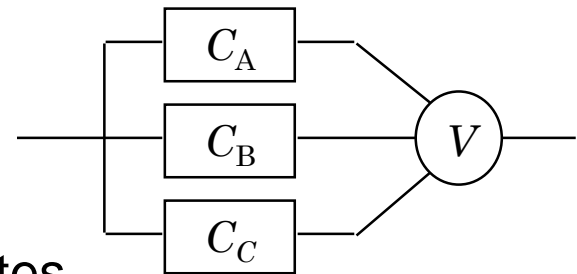
$$R_{TMR}(t) = R(C_A, C_B, C_C, t) + R(C_A, C_B | \bar{C}_C, t) + \\ R(C_A, C_C | \bar{C}_B, t) + R(C_B, C_C | \bar{C}_A, t)$$

- under the assumption of identical failure rates

$$R_{TMR}(t) = R(t)^3 + 3R(t)^2 Q(t)$$

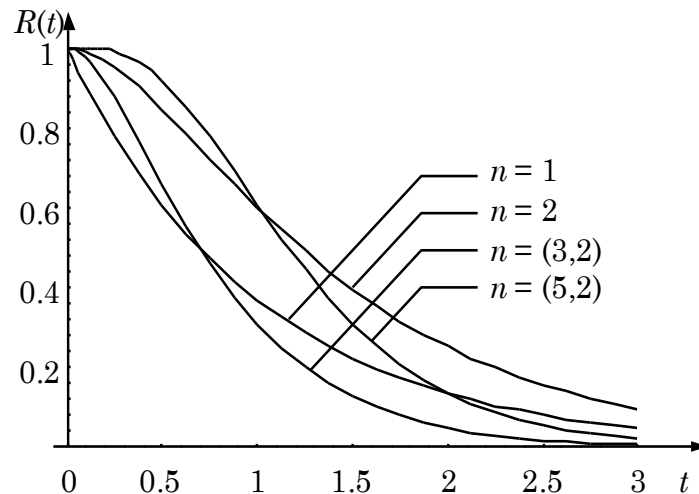
- for general voting systems where c out of n components have to function correctly

$$R_{NMR}(t) = \sum_{k=c}^n \binom{n}{k} (e^{-\lambda t})^k (1 - e^{-\lambda t})^{n-k}$$



Arbitrary block diagrams (cont.)

Parallel fail silent components vs. majority voting



$n = 1$	single component
$n = 2$	two parallel components
$n = (3,2)$	voting, 2 out of 3
$n = (5,2)$	voting, 2 out of 5

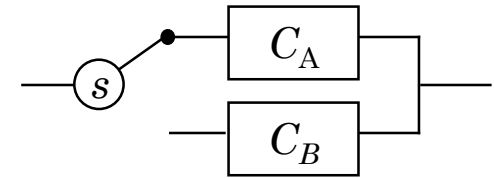
Neglected issues:

- coverage of fail silence assumption
- reliability of voter

Arbitrary block diagrams (cont.)

Passive redundancy

- probability that A is performing correctly plus conditional probability that B is performing correctly and A has failed



$$R(t) = R(C_A) + R(C_B|\bar{C}_A)$$

- under the assumption of constant failure rates $\lambda_A = \lambda_B$

$$R(t) = e^{-\lambda t} + \sum_{x=0}^t R_B(t-x+\Delta x) \frac{[R_A(x) - R_A(x+\Delta x)]\Delta x}{\Delta x}$$

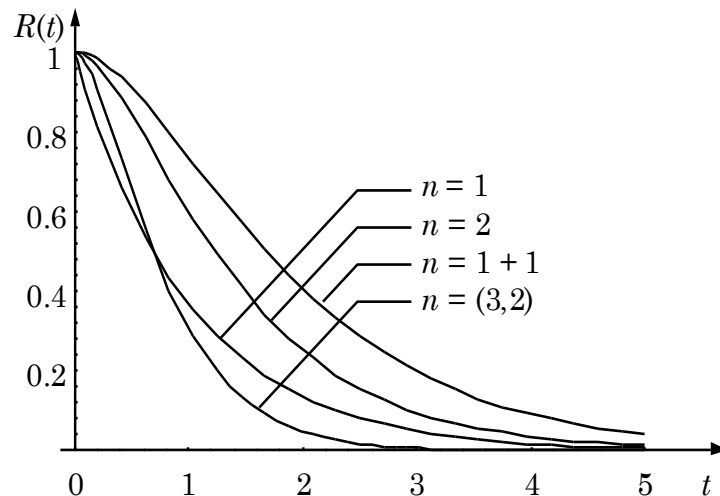
$$\Delta x \rightarrow 0: e^{-\lambda t} + \int_{x=0}^t R_B(t-x)f(x)dx$$

$$= e^{-\lambda t} + \int_{x=0}^t e^{-\lambda(t-x)}\lambda e^{-\lambda x}dx$$

$$= e^{-\lambda t}(1 + \lambda t)$$

Arbitrary block diagrams (cont.)

Passive vs. active redundancy



$n = 1$	single component
$n = 2$	two parallel components
$n = (3,2)$	voting, 2 out of 3
$n = 1 + 1$	one passive backup

Neglected issues:

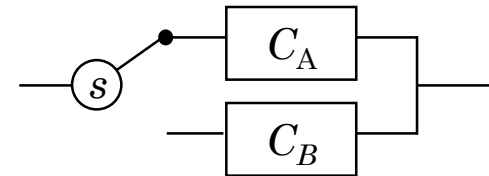
- coverage of fail silence assumption
- reliability of switch

Arbitrary block diagrams (cont.)

Passive redundancy with an unreliable switch

- assumption that the switch functions correctly with probability $R_s(t)$
- the system reliability is the probability that A is performing correctly plus the conditional probability that B is performing correctly and A has failed **and** the switch still functions correctly

$$\begin{aligned} R(t) &= e^{-\lambda t} + \sum_{x=0}^t R_B(t-x+\Delta x) R_s(t) [R_A(x) - R_A(x+\Delta x)] \\ &= e^{-\lambda t} + \int_{x=0}^t e^{-\lambda(t-x)} e^{-\lambda_s t} \lambda e^{-\lambda x} dx \end{aligned}$$

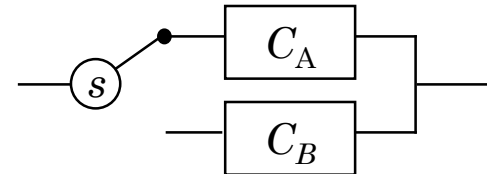


Arbitrary block diagrams (cont.)

Passive red. with limited error detection coverage

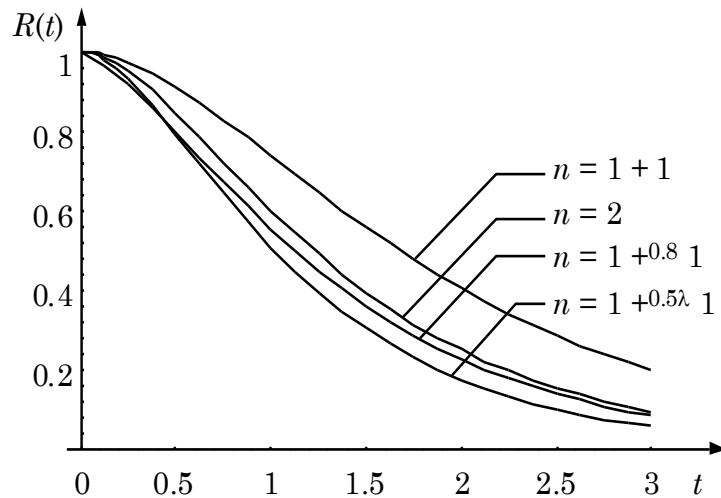
- assumption that errors of component A are not always detected, the error detection coverage is given by c
- the system reliability is the probability that A is performing correctly plus the conditional probability that B is performing correctly and A has failed **and** A's error has been detected

$$\begin{aligned} R(t) &= e^{-\lambda t} + \sum_{x=0}^t c R_B(t-x+\Delta x) [R_A(x) - R_A(x+\Delta x)] \\ &= e^{-\lambda t} + \int_{x=0}^t c e^{-\lambda(t-x)} \lambda e^{-\lambda x} dx \end{aligned}$$



Arbitrary block diagrams (cont.)

Perfect vs. imperfect passive redundancy



$n = 1 + 1$	one passive backup
$n = 2$	two parallel components
$n = 1 + 0.8 \cdot 1$	error detection coverage 80%
$n = 1 + 0.5\lambda \cdot 1$	reliability of switch is 0.5λ

- under practical conditions it is impossible to build an *ideal* passive replicated system
- an unreliable switch with $\lambda_s = 0.5\lambda$ or a switch with error detection coverage of 80% reduces the system reliability below that of active redundant components

Maintenance and Repair

Single parametric measures

- Mean time to failure:

$$MTTF = \int_0^{\infty} t f(t) dt$$

- Mean time to repair:

$$MTTR = \int_0^{\infty} t f_r(t) dt$$

- Mission reliability:

$$R_m = R(t_m) \quad t_m \dots \text{mission duration}$$

- (Steady state) availability:

$$A = \frac{MTTF}{MTTF + MTTR}$$

Mean time to failure

- Constant failure rate:

$$MTTF = \int_0^{\infty} t f(t) dt = \int_0^{\infty} t \lambda e^{-\lambda t} dt = \frac{1}{\lambda}$$

- Serial Connected Components

$$MTTF_{series} = \frac{1}{\lambda_1 + \lambda_2 + \dots + \lambda_n}$$

- Parallel connected components:

$$MTTF_{parallel} = \frac{1}{\lambda} \left(1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \right)$$

- Weibull distributed failure rate:

$$MTTF = \int_0^{\infty} t \alpha \lambda (\lambda t)^{\alpha-1} e^{-(\lambda t)^\alpha} dt = \frac{\Gamma(1 + \alpha^{-1})}{\lambda}$$

- Passive redundancy:

$$MTTF_{passive} = \frac{1}{\lambda_1} + \frac{1}{\lambda_2} + \dots + \frac{1}{\lambda_n}$$

Repair

■ Repair rate

- repair rate $\mu(t)$ analogous to failure rate
- most commonly constant repair rates $\mu(t) = \mu$

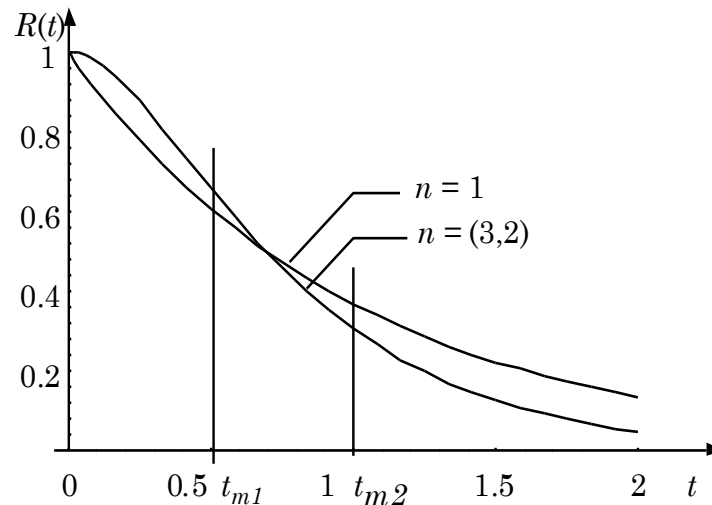
■ Mean time to repair

- analogous to mean time to failure

$$MTTR = \frac{1}{\mu}$$

Mission reliability

- assumption of a mission time t_m
- during mission there is no possibility of maintenance or repair
- typical examples are space flights or air planes
- suitability of architectures depends on mission time



Availability

- the percentage of time for which the system will conform to its specification
- also called steady state or instantaneous availability

$t \rightarrow \infty$:

$$A = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF + MTTR} \quad \text{mean time between failures (MTBF)}$$

- without maintenance and repair

$$MTTR = \infty: \quad A = 0$$

- Mission availability $t \rightarrow t_m$:

$$A_m = \frac{1}{t_m} \int_{t=0}^{t_m} R(t) dt$$

Probabilistic Structural-Based Modeling: Part 2

Markov models

- Suitable for modeling of:
 - arbitrary structures
(active, passive and voting redundancy)
 - systems with complex dependencies
(assumption of independent failures is no longer necessary)
 - coverage effects
- Markov property:
 - The system behavior at any time instant t is independent of history (except for the last state).
- Restriction to constant failure rates

Markov models

The two phases for Markov modeling

- **Model design:**

- identification of relevant system states
- identification of transitions between states
- construction of Markov graph with transition rates

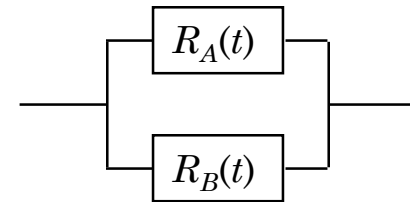
- **Model evaluation:**

- Differential equation
- Solution of equation gives $R(t)$
 - explicit (by hand)
 - Laplace transformation
 - numeric solution (tool based)
- Integration of differential equation gives $MTTF$
 - system of linear equations

Markov models (cont.)

Example model for active redundant system

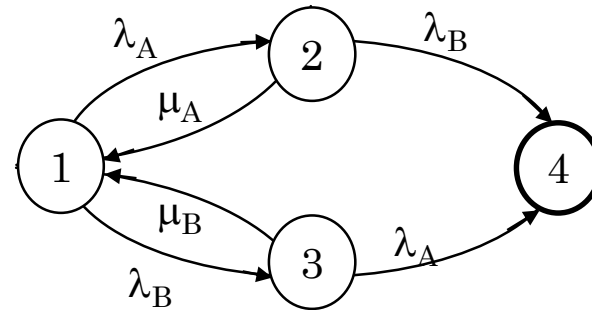
Two parallel connected components A and B with maintenance. Failure rates are λ_A and λ_B , repair rates are μ_A and μ_B .



Identification of system states:

1: A correct	B correct	$P_1(t)$
2: A failed	B correct	$P_2(t)$
3: A correct	B failed	$P_3(t)$
4: A failed	B failed	$P_4(t)$

Construction of Markov Graph



Markov models (cont.)

Active redundancy with identical components

- failure rates: $\lambda_A = \lambda_B = \lambda$ repair rates: $\mu_A = \mu_B = \mu$

- Identification of system states:

1: A correct B correct $P_1(t)$

2: one failed one correct $P_2(t)$

3: A failed B failed $P_3(t)$

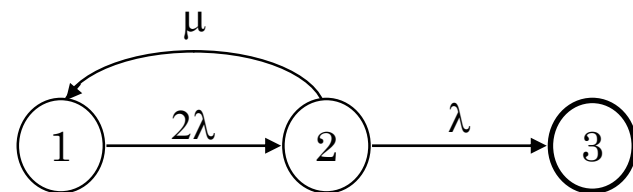
- Differential equations:

$$\frac{dP_1(t)}{dt} = -2\lambda P_1(t) + \mu P_2(t)$$

$$\frac{dP_2(t)}{dt} = 2\lambda P_1(t) - (\mu + \lambda)P_2(t)$$

$$\frac{dP_3(t)}{dt} = \lambda P_2(t)$$

- Construction of Markov Graph



Markov models (cont.)

MTTF evaluation from Markov model

- In a Markov model the MTTF is given by the period during which the system exhibits states that correspond to correct behavior.
- for the active redundant example system:

$$MTTF = \int_{t=0}^{\infty} (P_1(t) + P_2(t)) dt = T_1 + T_2$$

$$T_1 = \int_{t=0}^{\infty} P_1(t) dt \quad T_2 = \int_{t=0}^{\infty} P_2(t) dt$$

- state probabilities for $t = 0$ and $t = \infty$

$$P_1(0) = 1 \quad P_1(\infty) = 0$$

$$P_2(0) = 0 \quad P_2(\infty) = 0$$

$$P_3(0) = 0 \quad P_3(\infty) = 1$$

Markov models (cont.)

MTTF evaluation from Markov model (cont.)

- integration of differential equation

$$\frac{d P_1(t)}{dt} = -2\lambda P_1(t) + \mu P_2(t)$$

$$\frac{d P_2(t)}{dt} = 2\lambda P_1(t) - (\mu + \lambda) P_2(t) \Rightarrow$$

$$\frac{d P_3(t)}{dt} = \lambda P_2(t)$$

$P_1(0) = 1$	$P_1(\infty) = 0$
$P_2(0) = 0$	$P_2(\infty) = 0$
$P_3(0) = 0$	$P_3(\infty) = 1$

$$0 - 1 = -2\lambda T_1 + \mu T_2$$

$$0 - 0 = 2\lambda T_1 - (\mu + \lambda) T_2$$

$$1 - 0 = \lambda T_2$$

- solution of linear equation system

$$T_2 = \frac{1}{\lambda}$$

$$T_1 = \frac{\mu + \lambda}{2\lambda} T_2 = \frac{\mu + \lambda}{2\lambda^2} = \frac{1}{2\lambda} + \frac{\mu}{2\lambda^2}$$

$$MTTF = T_1 + T_2 = \frac{3}{2\lambda} + \frac{\mu}{2\lambda^2}$$

Markov models (cont.)

Effect of maintenance

- repair and failure rate: $\lambda = \frac{1}{1000}$ [h] $\mu = \frac{1}{10}$ [h]

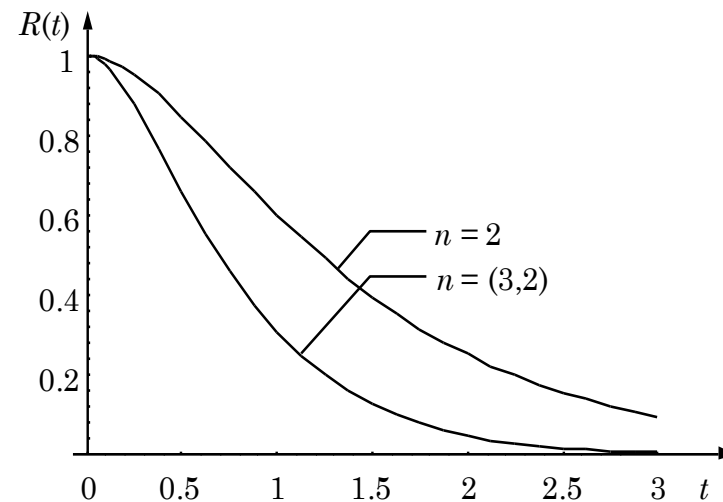
	without maintenance			with maintenance		
	$R(t)$	$MTTF$	h	$R(t)$	$MTTF$	h
2 components in series	$e^{-2\lambda t}$	$\frac{1}{2\lambda}$	500	$e^{-2\lambda t}$	$\frac{1}{2\lambda}$	500
single component	$e^{-\lambda t}$	$\frac{1}{\lambda}$	1000	$e^{-\lambda t}$	$\frac{1}{\lambda}$	1000
2 components in parallel	$2e^{-\lambda t} - e^{-2\lambda t}$	$\frac{3}{2\lambda}$	1500	—	$\frac{3}{2\lambda} + \frac{\mu}{2\lambda^2}$	51500
one passive backup	$e^{-\lambda t}(1 + \lambda t)$	$\frac{2}{\lambda}$	2000	—	$\frac{2}{\lambda} + \frac{\mu}{\lambda^2}$	102000

- for 2 active redundant components the MTTF is improved by a factor 34
- for 2 passive redundant components the MTTF is improved by a factor 51

Markov models (cont.)

Effect of failure semantics and assumption coverage

- comparing a system with two active replicated components to a TMR systems shows that under *ideal* conditions active replication has a higher reliability
- **But:** active replication is based on the assumption that components are fail silent
 - assumption coverage ???
- TMR voting is based on the assumption of fail consistent components
 - assumption coverage ≈ 1 (if properly constructed)



Markov models (cont.)

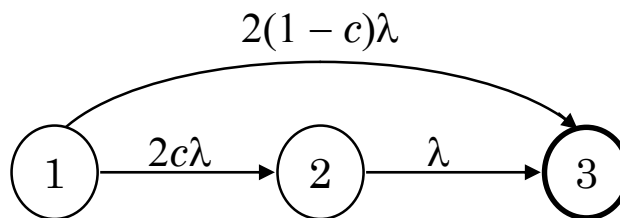
Effect of failure semantics and assumption coverage

- modeling of the TMR was reasonable since assumption coverage of fail consistent behavior ≈ 1
- modeling of the active redundant system was idealistic since assumption coverage of fail silent behavior < 1

- **Markov model:**

λ .. failure rate for active redundant parallel connected components

c .. assumption coverage for fail silent behavior

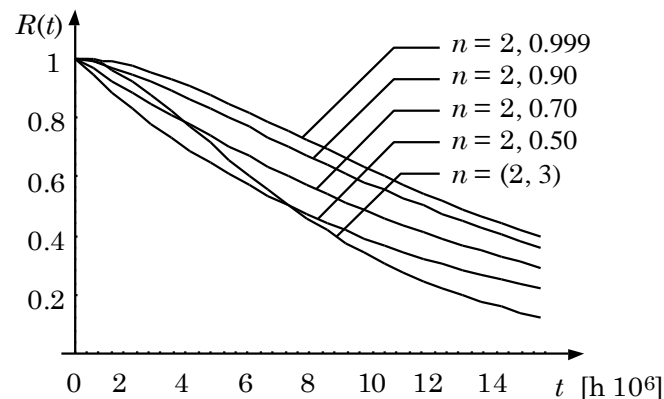


Markov models (cont.)

Effect of failure semantics and assumption coverage

- failure rate of a single component: $\lambda = 100$ FIT

<i>System</i>	<i>Description</i>	<i>MTTF</i>
$n = 2, 0.999$	two parallel components, coverage of fail silent assumption 99.9%	$14.99 \cdot 10^6$
$n = 2, 0.90$	two parallel components, coverage of fail silent assumption 90%	$14.00 \cdot 10^6$
$n = 2, 0.70$	two parallel components, coverage of fail silent assumption 70%	$12.00 \cdot 10^6$
$n = 2, 0.50$	two parallel components, coverage of fail silent assumption 50%	$10.00 \cdot 10^6$
$n = (2, 3)$	TMR system, coverage of fail consistent assumption 100%	$8.33 \cdot 10^6$



Markov models (cont.)

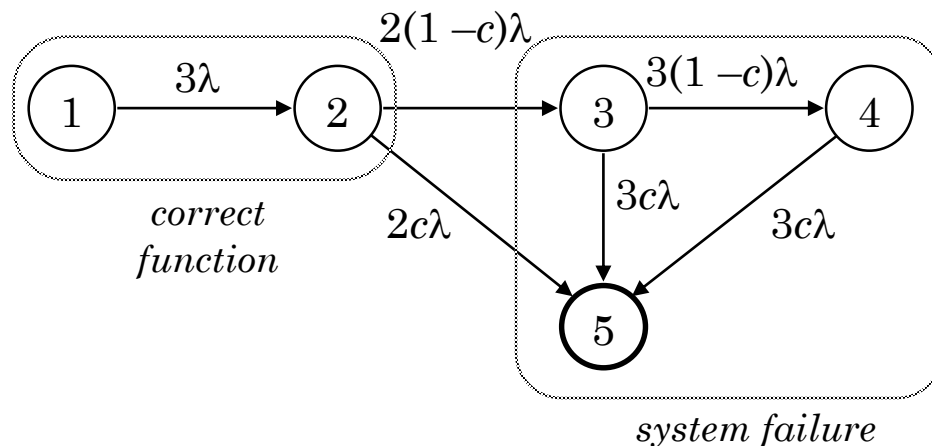
Effect of failure semantics and assumption coverage

- comparing parallel components to a TMR systems shows that the reliability of the parallel system is superior for reasonable assumption coverages
- **Safety:**
from the viewpoint of safety both systems needs to be reevaluated
- $R(t) = S(t)$
In an example, a system consists of two parallel components. The system reliability is equal to the system safety when the system may potentially cause a hazard if it does not function correctly.
- $R(t) < S(t)$
In an example, a system consists of a TMR systems. The reliability is not equal to the safety when the system can enter a safe state although it is not functioning correctly, e.g. all three components disagree.

Markov models (cont.)

Safety of a TMR system

- to model the safety of a TMR system it needs to be differentiated between incorrect function and the unsafe system state
- Markov model:**
 - λ .. failure rate for single component
 - c .. probability of coincident failures of two components



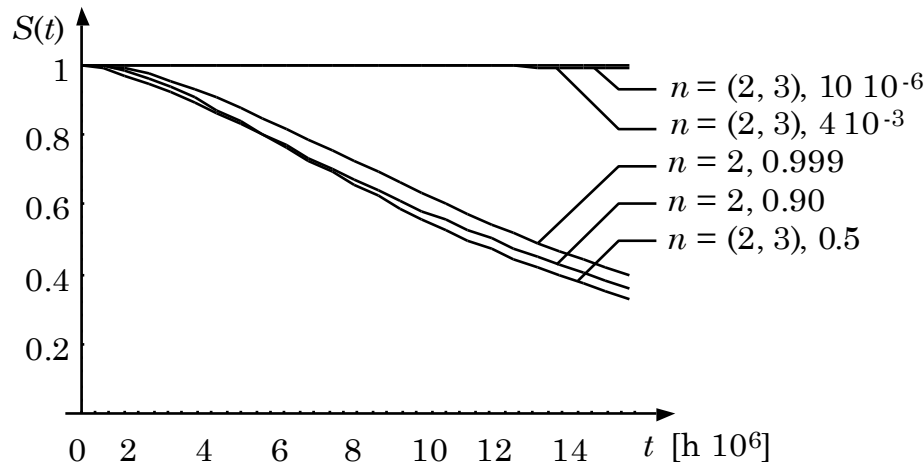
- 1 .. 3 correct components
- 2 .. 2 correct, 1 failed comp.
- 3 .. 1 correct, 2 failed comp.
- 4 .. 3 failed components
- 5 .. unsafe state, ≥ 2 coincident component failures

Markov models (cont.)

Effect of assumption coverage on safety

- failure rate of a single component: $\lambda = 100 \text{ FIT}$

<i>System</i>	<i>Description</i>	<i>MTTF_S</i>
$n = (2, 3), 10 \cdot 10^{-6}$	TMR system, probability of two coincident failures $10 \cdot 10^{-6}$	$333.34 \cdot 10^9$
$n = (2, 3), 4 \cdot 10^{-3}$	TMR system, probability of two coincident failures $4 \cdot 10^{-3}$	$861.71 \cdot 10^6$
$n = (2, 3), 0.5$	TMR system, probability of two coincident failures 0.5	$13.33 \cdot 10^6$
$n = 2, 0.999$	two parallel comp., coverage of fail silent assumption 99.9%	$14.99 \cdot 10^6$
$n = 2, 0.90$	two parallel comp., coverage of fail silent assumption 90%	$14.00 \cdot 10^6$



coincidence probability of two even distributed numbers

16 bit	$10 \cdot 10^{-6}$
8 bit	$4 \cdot 10^{-3}$
1 bit	0.5

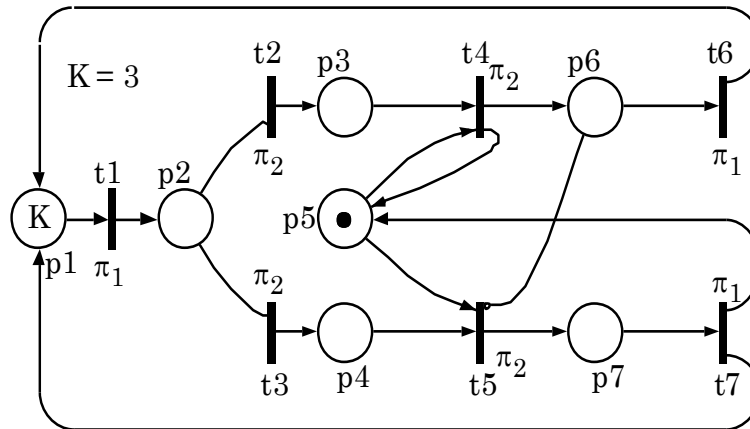
Generalized Stochastic Petri Nets (GSPN)

- because of the very limited mechanisms available, Markov models become easily very complex
- Petri Nets provide much richer mechanisms, they can be used to model and analyze arbitrary systems, algorithms and processes
- basic Petri Nets — which were restricted to discrete behavior only — can be extended to “Generalized Stochastic Petri Nets” by allowing transition delays to be either deterministically equal to zero or exponentially distributed random variables, or to be random variables with different distributions
- it was shown that stochastic Petri Nets are *isomorphic* to continuous Markov chains, i.e. for each stochastic Petri Net there exists a functional equivalent Markov chain (and vice versa)

Generalized Stochastic Petri Nets

Example

Single-writer/multiple-reader access to a shared resource with single access.



p_i ... places

t_i ... transitions

π_i ... transition priorities

- the 3 tokens in place p_1 represents customers that may request the resource
- firing t_1 starts the protocol
- t_2 indicates “read” and t_3 “write” access, respectively
- the single token in p_5 represents the resource

Generalized Stochastic Petri Nets Modeling

To model and analyze a system by means of GSPN the following steps have to be carried out:

- **model construction:** usually by means of structured techniques, bottom-up or top-down
- **model validation:** structural analysis, possibly formal proves of some behavioral properties
- **definition of performance indices:** definition of markings and transition firings (deterministically or stochastic)
- **conversion to Markov chain:** generation of reachability set and reachability graph to obtain the Markov chain
- **solution of the Markov chain**

Tool support for all steps exists. Conversion to a Markov chain and solution can be automated completely.

Generalized Stochastic Petri Nets

Model simulation vs. analytical solutions

- generalized stochastic petri nets are well suited for simulation
- transition rates are not restricted to be deterministic or exponentially distributed
- complex models are computationally expensive (simulation step width and simulation duration)
- too large simulation step width can result in meaningless results (variance of result is too big)

Open issues of probabilistic structural based models

Open issues of probabilistic structural based models

- large gap between system and model
- model construction is time consuming, error prone and unintuitive
- small changes in the architecture result in considerable changes in the model
- model validation for ultra-high dependability

Probabilistic structural modeling and software

Probabilistic structural based models are not well suited for software. They are rather well suited to analyze hardware architectures and design alternatives.

- for software there are no well defined individual components
- complexity of software structures is very high
- for software the assumption of independent failures is too strong
 - one CPU for many processes
 - one address range for many functions
- real-time aspects are not captured
- parallelism and synchronization is not considered (except for GSPN's)

Reliability growth models

Reliability growth models

- no assumption on identifiable components and system structure
- based on the idea of an iterative improvement process:
 - testing → correction → re-testing
- major goals of reliability growth models:
 - disciplined and managed process for reliability improvement
 - extrapolating the current reliability status to future results
 - assessing the magnitude of the test, correction and re-test effort
- allows modeling of wearout *and* design faults
- can be used for hardware and software as well

Reliability growth models (cont.)

Software

- typically continuous time reliability growth
 - the software is tested
 - the times between successive failures are recorded
 - failures are fixed
- observed execution time data $t_1, t_2, t_3, \dots, t_{i-1}$ are realizations of the random variables $T_1, T_2, T_3, \dots, T_{i-1}$
- based on these data the unobserved T_i, T_{i+1}, \dots should be predicted (e.g. $T_i = MTTF$)

But:

- accuracy of models is very variable
- no single model can be trusted to behave well in all contexts

Reliability growth models (cont.)

The prediction system

Software reliability growth models are prediction systems which are comprised of:

- The **probabilistic model**
which specifies the distribution of any subset T_j 's conditional on a unknown parameter α .
- A **statistical inference procedure**
for α involving use of available data (realizations of T_j 's)
- A **prediction procedure**
combining the above two points to allow to make probability statements about future T_j 's

Comparison of probabilistic modeling techniques

Comparison of probabilistic modeling techniques

Method	Advantages	Restrictions and deficiencies
simple block diagrams	simple and easy to understand model, easy to calculate for constant failure rates	restricted to series and parallel connection, assumption of independent failures, maintenance can-not be modelled, only for active redundant systems, not well suited for software
arbitrary block diagrams	can be used to model arbitrary structures	same restrictions as with simple block diagrams, except series and parallel connection, not well suited for software
markov chains	can model arbitrary structures, no restriction to independent failures, complex dependencies can be expressed, modeling of coverage and maintenance, good tool support	compared to GSPN higher model complexity, restriction to constant failure rates, not well suited for software

Comparison of probabilistic modeling techniques (cont.)

Method	Advantages	Restrictions and deficiencies
generalized stochastic petri nets	much richer mechanisms for modeling, allows combination of discrete and stochastic behavior, good tool support, can be used to model algorithmic issues of software	it is difficult to verify that the model agrees with reality (as for any complex model)
reliability growth models	suited for prediction of software reliability, does not make assumptions on the system structure, based on relatively easy obtainable experimental data	accuracy of models is very variable, no general applicable model, user must analyze different models to select suitable one
error seeding	very easy procedure, takes few assumptions on the system	computational complexity (seeded errors by number of test cases), error size needs to be controlled

Limits of validation for ultra-high dependability

Limits of validation for ultra-high dependability

- 10^{-9} catastrophic failure conditions per hour for civil transport airplanes
- experimental system evaluation is impossible for critical applications
- modeling is therefore the only possibility to validate ultra-high dependability

Limits of validation for ultra-high dependability (cont.)

- **Limits for reliability growth models:**

- If we want to have an assurance of high dependability, using information obtained from the failure process, then we need to observe the system for a very long time.

- **Limits of testing:**

- If we see a period of 10^9 hours failure free operation a MTTF of 10^9 hours can be expected without bringing any apriori believe to the problem.

- If a MTTF of 10^6 is required and only 10^3 hours of test are carried out, Bayesian analysis shows that essentially we need to *start* with a 50:50 believe that the system will attain a MTTF of 10^6 .

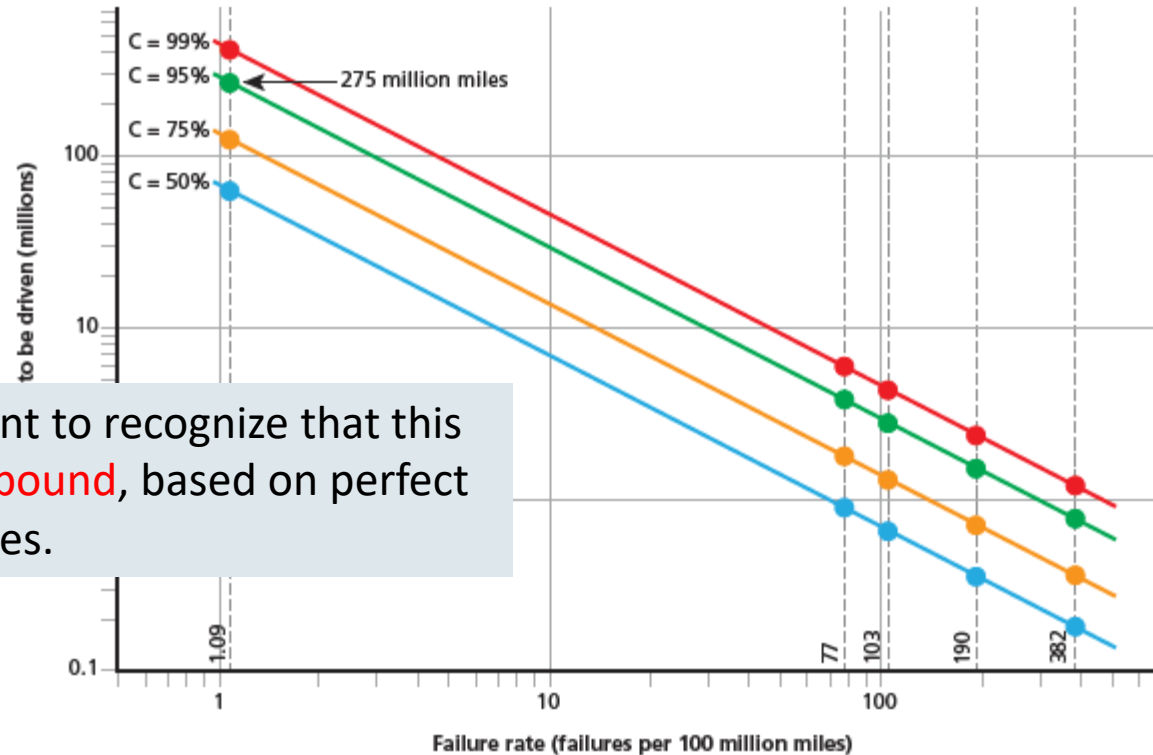


Driving to

How Many Miles Autonomous V

Nidhi Kalra, Susan M.

Figure 1. Failure-Free Miles Needed to Demonstrate Maximum Failure Rates



However, it is important to recognize that this is a **theoretical lower bound**, based on perfect performance of vehicles.

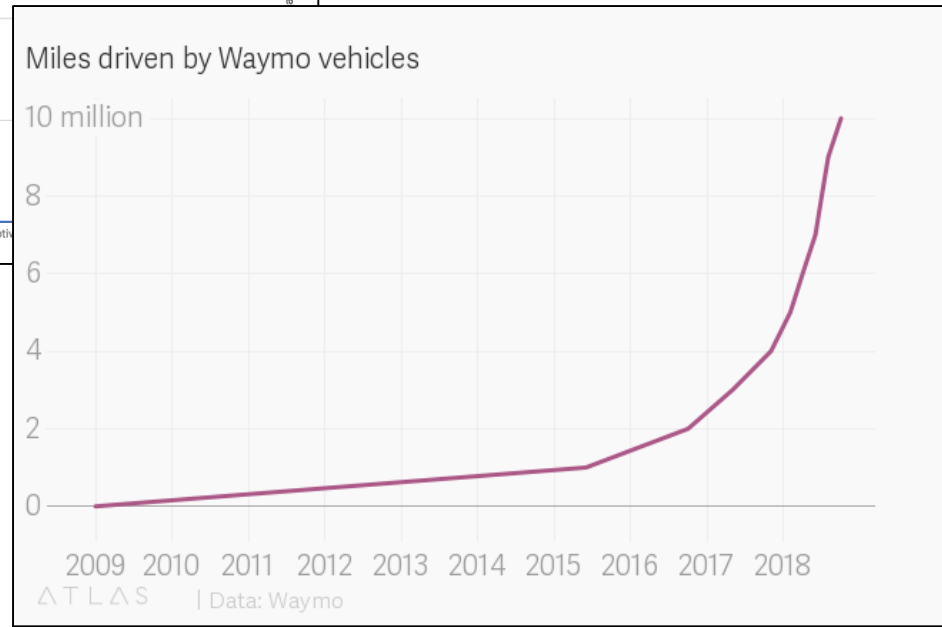
SOURCE: Authors' analysis.

NOTE: The four colored lines show results for different levels of confidence. The five dashed vertical reference lines indicate the failure rates of human drivers in terms of fatalities (1.09), reported injuries (77), estimated total injuries (103), reported crashes (190), and estimated total crashes (382).

RAND RR478-1



<https://arstechnica.com/cars/2018/02/waymo-now-has-a-serious-driverless-car-rival-gms-cruise/>



<https://qz.com/1419747/waymos-self-driving-cars-have-logged-10-million-miles/>

Limits of validation for ultra-high dependability (cont.)

- **Limits of other sources of evidence:**

- Step-wise evolution, simple design, over-engineering can be used only to a limited extent to obtain confidence because there is no continuous system model and there are no identifiable stress factors.

- **Limits of past experience:**

- For software there is no clear understanding of how perceived differences in the design or design methodology affect dependability.

- **Limits of structural modelling:**

- There are obvious limitations with respect to design faults, and software in particular since the assumption of failure independence does not hold.

Limits of validation for ultra-high dependability (cont.)

- **Limits of formal methods and proofs:**

“We believe that proofs may eventually give ‘practically complete’ assurance about software developed for small but well-understood application problems, but the set of these problems is now empty and there is no way of foreseeing whether it will grow to be of some significance.”

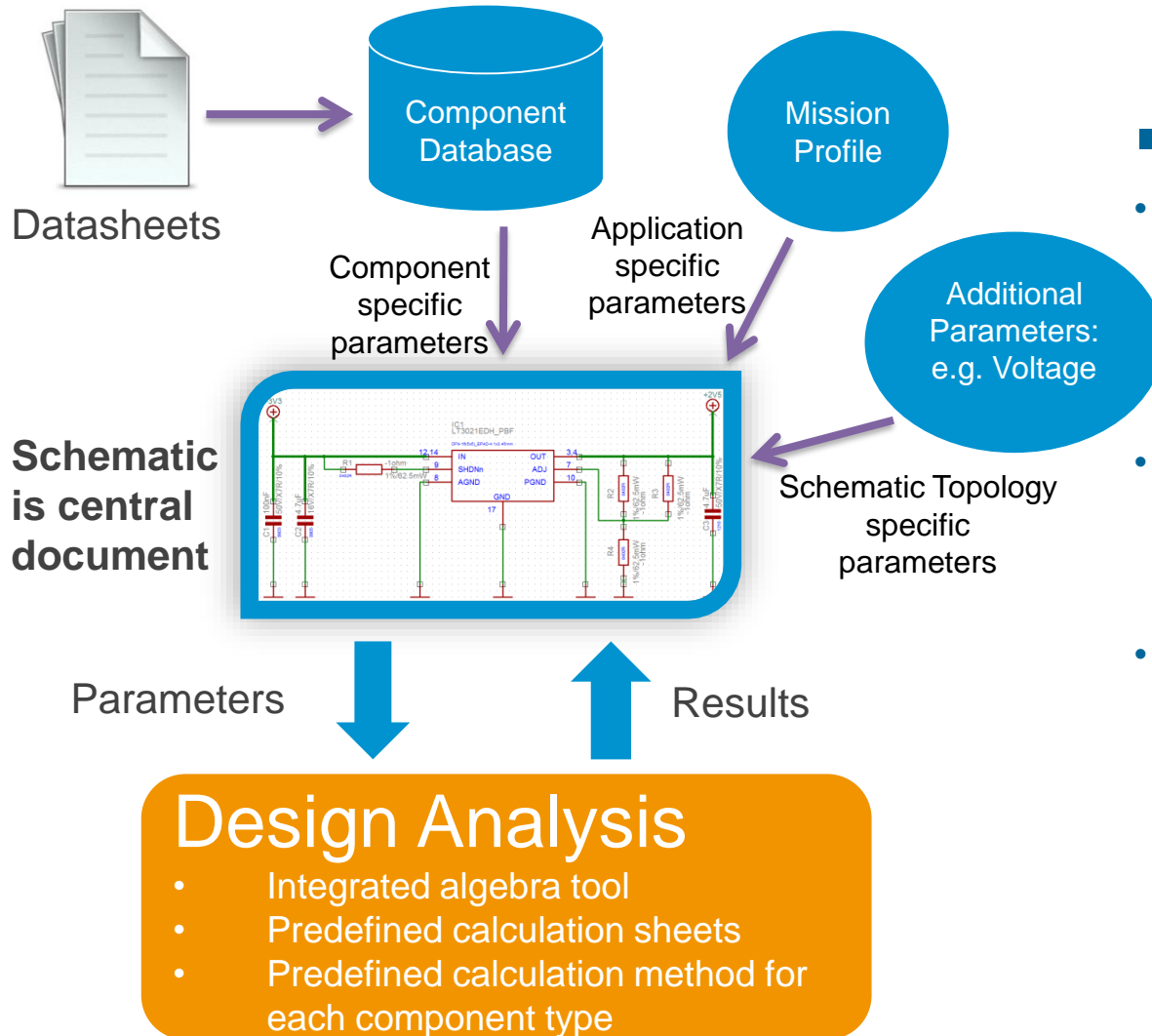
(Littlewood and Strigini, 1993)

Example: Hardware Design Analysis at TTTech

Design Analysis Goals

- Failure Rate Prediction
 - Calculation of component FIT and MTBF values

- IEC TR 62380 Reliability Data Handbook
 - provides elements to calculate failure rate of mounted electronic components
 - Reliability data is taken from field data
 - Failures rates include the influence of component mounting processes



Advantages:

- Per component predefined analysis method
- Analysis within the schematic
- Component parameter changes are automatically adopted in the design analysis
- Analysis's can be sequenced and use results from preceding calculations

Dependable Systems

Part 4: Certification – Processes and Standards

Contents

- Generic Characteristics
- Example: TTTech's Software Development
- Example: Traceability in the Development of an Ethernet Switch
- Certificates
- Standards
 - Safety Integrity Levels (SIL)
 - Automotive SIL (ASIL)
 - Design Assurance Levels (DAL)
- The Safety Case

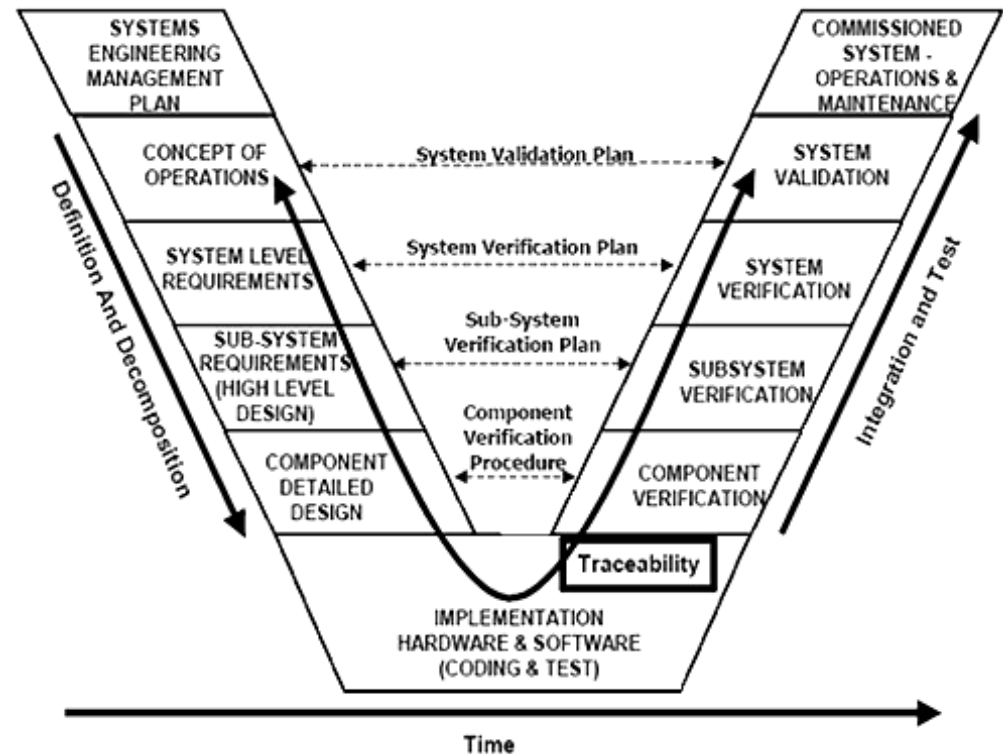
Generic Characteristics of Development Processes for Dependable Systems

Objectives of Development Processes

- The aim of development processes is to minimize the likelihood of **development faults**, i.e., faults that occur during the creation of the system (HW, SW, etc.)
- For example: since the introduction of the DO-178B standard “Software Considerations in Airborne Systems and Equipment Certification” in the 1990s, not a single lethal incident has occurred that would trace back to a software development fault.

Typical activities in such development processes

- Requirements Capturing
- High-Level Requirements Document
- Low-Level Requirements Document
- Conceptual Design Document
- Detailed Design (i.e., implementation)
- Verification and Validation
- **Peer review and auditing**
- Key property of the documents: **traceability**

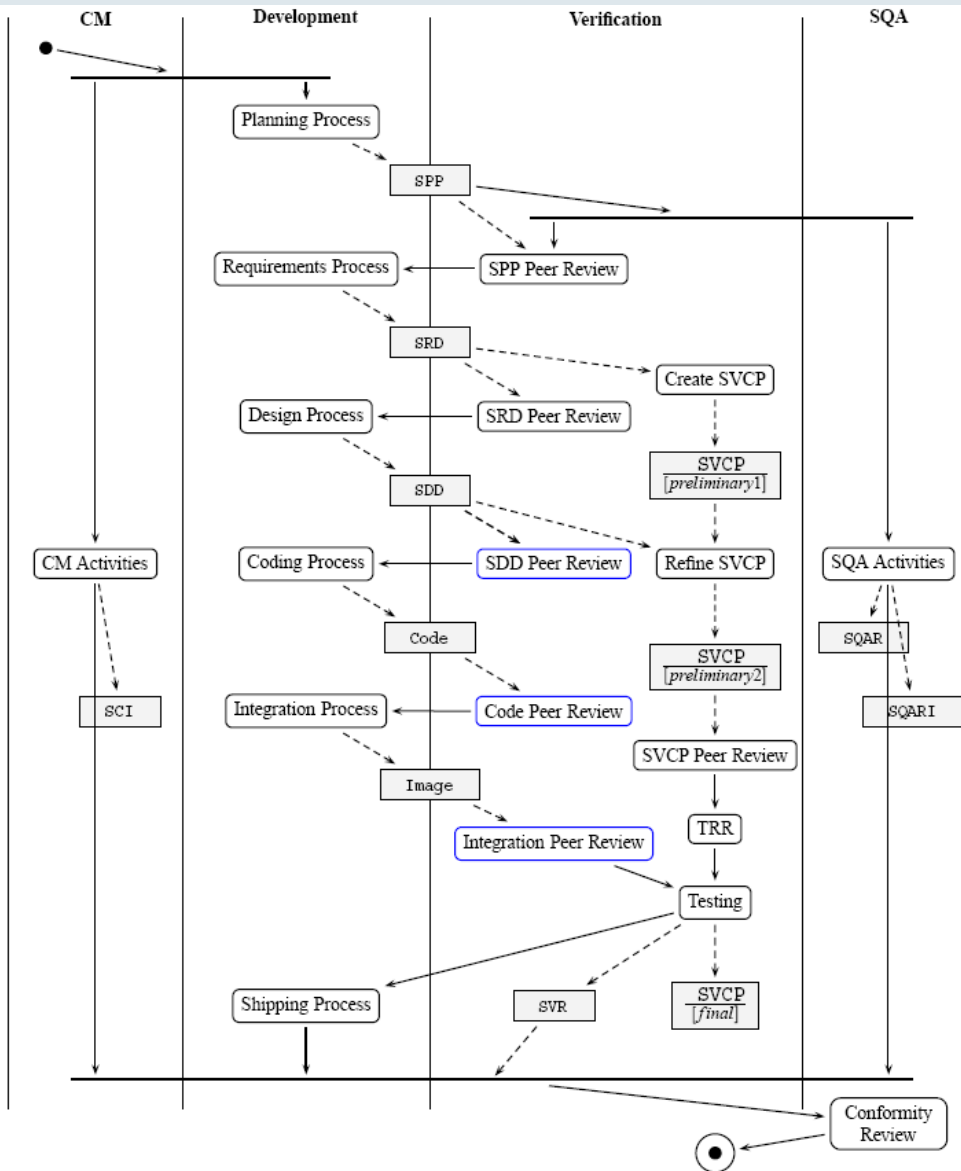


<http://www.mitre.org/sites/default/files/images/selc-te-vv-fig2.gif>

Verification and Validation

- Verification is the process to check whether a product satisfies its requirements.
- Validation is the process to check if the product satisfies its purpose.
- Why is verification different from validation?
 - Sometimes, a product's purpose is not fully described by its requirements.

Example: TTTech's Software Development



- The flowchart to the left shows how software processes are implemented at TTTech.
- Each development process creates an artifact as output (documents or code).
- *Software Verification Cases and Procedures (SVCP)* are developed in parallel to the refinement steps of the development process.
- All development, planning and verification artifacts are *peer reviewed* prior to release.
- The Testing Process creates the *Software Verification Results (SVR)* as objective evidence for the correct implementation of all high- and low-level requirements.
- SQAR, SW Quality Assurance Record
- SQARI, SW Quality Assurance Record Index

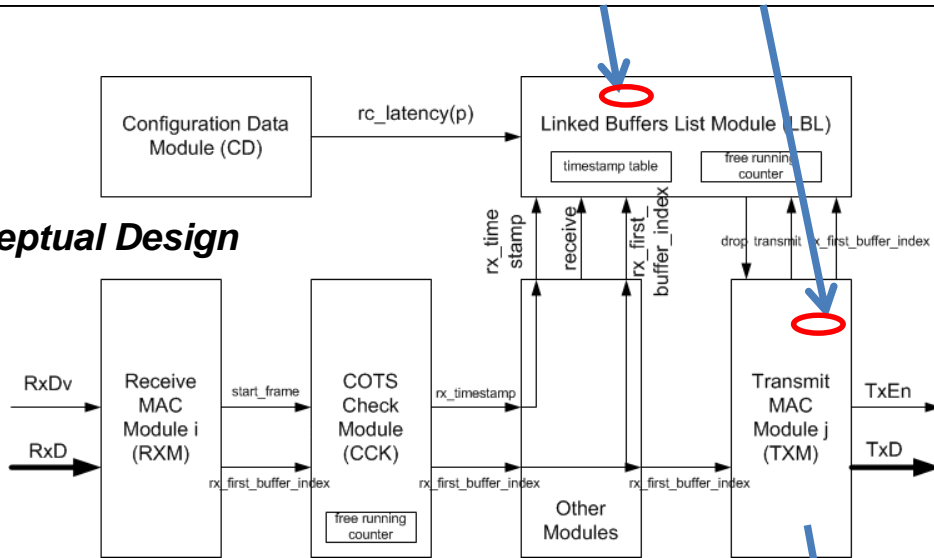
Example: Traceability in the Development of an Ethernet Switch

Requirement

1737 **VLU-4650** The switch shall drop the frames policed at switch ingress, as critical traffic in accordance with
 1738 DCI-2052 and dispatched as rate constrained traffic that reside in the switch more than a statically config-
 1739 urable amount of time (*rc_latency* parameter defined on a per output port basis described by MNI-3724).
 1740 *Guidance: When a CT frame was dropped because of age violation, the diagnosis counter rcframe_age_err*
 1741 *is incremented by 1, according to DCI-2745*

Test

Conceptual Design



Implementation

```

if R.tdma_frame >= SERDES_PORTS_NO then
    V.age_time := ONE_SECOND;
else
    V.age_time := cdi.age_time(stdvec_to_int(R.tdma_frame));
end if;

elsif R.start_tdma_buffer_d = '1' then
    V.tdma_frame := ismi.tdma_frame;

if ctci.valid = '1' and R.one_time = '1' then
    V.ramo_lbl_to_link_ram_addr := ctci.tail_index;
    
```

```

[PURPOSE]
This test case checks if the switch ip drops incoming RC
frames that aged out inside the IP.\

[AUTHOR]
AST

[RESULT]
PASS

[REQUIREMENTS]
\ReqRef{MNI-3700}
\ReqRe{VLU-4650}
\ReqRef{DCI-2745}

[PRECONDITIONS]

\footnotesize
\begin{verbatim}

The test assumes the following configuration is loaded:
General Parameters Table:
- static cots routing := 1
:= 0
:= 0xDEADBEEF
:= 0xFFFFFFFF
:= 500 (4us) - has the resolut
s
    
```

Certificates

What is being certified?

■ Product

- a regulatory body approves that a product has certain characteristics.
- e.g., type certificate of an airplane

■ Company

- a regulatory body approves that a given company follows given standards.
- e.g., ISO 9001

Certificates Examples

- Type Certificate (Aerospace):
 - is issued to signify the airworthiness of an aircraft manufacturing design,
 - is issued by a regulating body (e.g., FAA, EASA).

Standards

Main Aspects of Development Processes

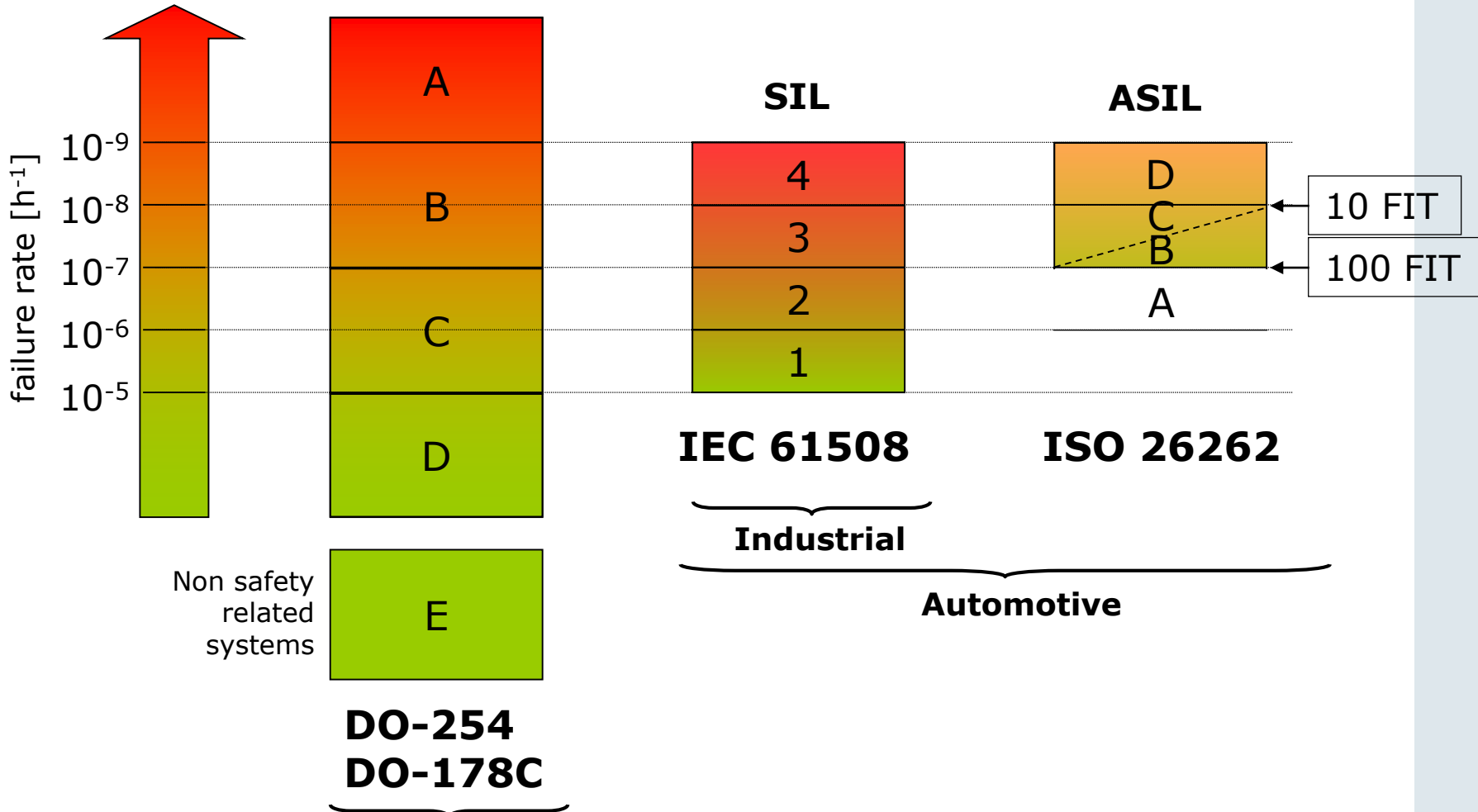
- Requirements on the development process in particular:
 - specification
 - design
 - verification
- Requirements on the safety management.

Example Standards

- IEC 61508 – “Functional Safety”
- ISO 26262 – “Road vehicles – Functional safety”
- ARP 4754 – “Certification Considerations for Highly-Integrated or Complex Aircraft Systems”
- DO 178B/C – “Software Considerations in Airborne Systems and Equipment Certification”

Possible relation between safety standards

Multi-dimensional aspects needs to be considered here

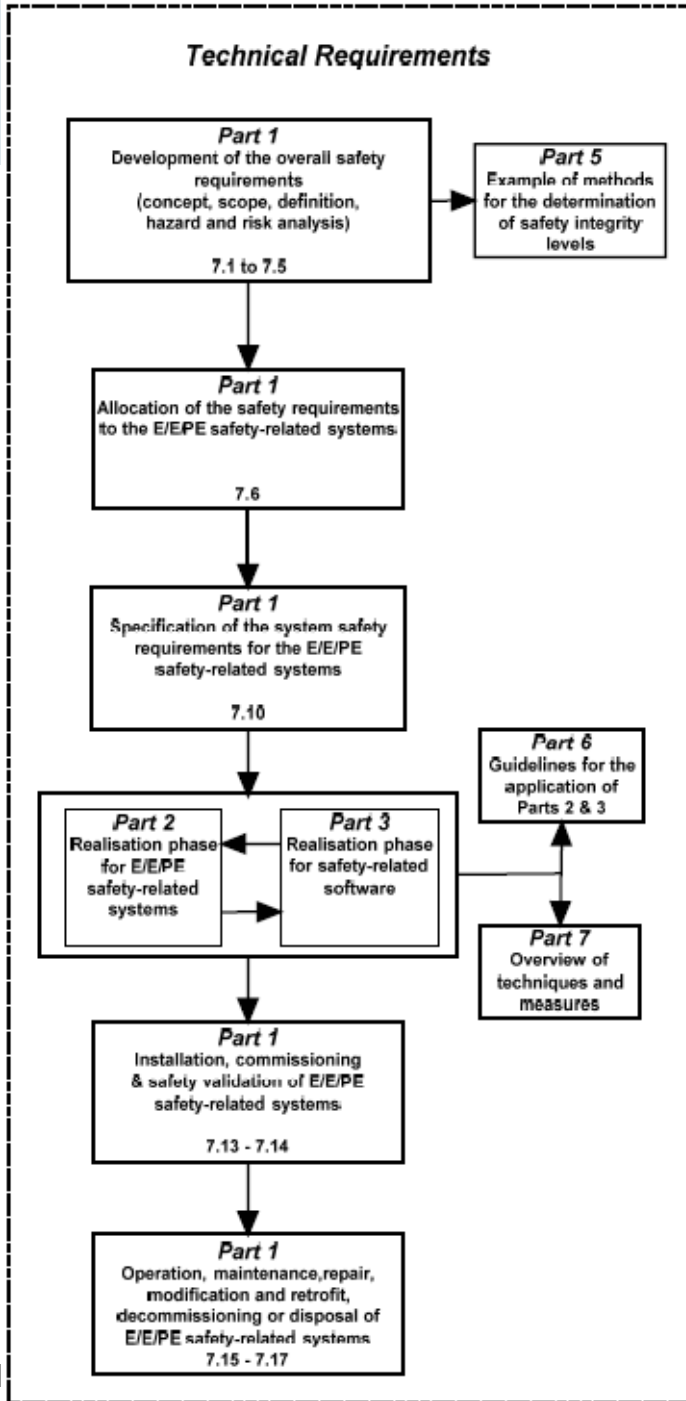


Terminology

- Certification-related standards have been developed in parallel to the academic work. Thus, the terminology as introduced by Avizienis et al. and used in this course, does not always apply.
- Certification-related standards introduce their individual terms and definitions.

Safety Life Cycle Considerations

- A complete framework for the safety life cycle consists of:
 - definition of different life cycle phases
 - specification of which activities to perform in each phase
 - specification of which inputs to provide to each of the activities
 - requirement on which results to achieve.
- Standards vary with respect to their framework completeness.
 - e.g., IEC 61508 defines a complete framew. (see next slide)
 - e.g., DO 178 defines only the results to be achieved



Other Requirements

Part 4
Definitions & abbreviations

Part 1
Documentation
Clause 5 & Annex A

Part 1
Management of functional safety
Clause 6

Part 1
Functional safety assessment
Clause 8

We will discuss later how some of these parts tie into each other.

SW/HW Development Life Cycle

- Standards also vary in imposing requirements on the SW/HW development life cycle.
 - e.g., IEC 61508 does not require any particular SW development process
 - e.g., ISO 26262 defines a V-Model as a reference software development process (see next slide).

1. Vocabulary

2. Management of functional safety

2-5 Overall safety management

2-6 Safety management during the concept phase and the product development

2-7 Safety management after the item's release for production

3. Concept phase

3-5 Item definition

3-6 Initiation of the safety lifecycle

3-7 Hazard analysis and risk assessment

3-8 Functional safety concept

4. Product development at the system level

4-5 Initiation of product development at the system level

4-6 Specification of the technical safety requirements

4-7 System design

4-11 Release for production

4-10 Functional safety assessment

4-9 Safety validation

4-8 Item integration and testing

7. Production and operation

7-5 Production

7-6 Operation, service (maintenance and repair), and decommissioning

5. Product development at the hardware level

5-5 Initiation of product development at the hardware level

5-6 Specification of hardware safety requirements

5-7 Hardware design

5-8 Evaluation of the hardware architectural metrics

5-9 Evaluation of the safety goal violations due to random hardware failures

5-10 Hardware integration and testing

6. Product development at the software level

6-5 Initiation of product development at the software level

6-7 Software architectural design

6-8 Software unit design and implementation

6-9 Software unit testing

6-10 Software integration and testing

6-11 Verification of software safety requirements

8. Supporting processes

8-5 Interfaces within distributed developments

8-6 Specification and management of safety requirements

8-7 Configuration management

8-8 Change management

8-9 Verification

8-10 Documentation

8-11 Confidence in the use of software tools

8-12 Qualification of software components

8-13 Qualification of hardware components

8-14 Proven in use argument

9. ASIL-oriented and safety-oriented analyses

9-5 Requirements decomposition with respect to ASIL tailoring

9-6 Criteria for coexistence of elements

9-7 Analysis of dependent failures

9-8 Safety analyses

10. Guideline on ISO 26262

Safety Integrity Levels

Safety Integrity Levels (IEC 61508)

- **3.5.1 safety function:**
 - function to be implemented by an E/E/PE safety-related system or other risk reduction measures, that is intended to achieve or maintain a safe state for the EUC [Equipment Under Control], in respect of a specific hazardous event (see 3.4.1 and 3.4.2)
- **3.5.4 safety integrity:**
 - probability of an E/E/PE safety-related system satisfactorily performing the specified safety functions under all the stated conditions within a stated period of time
- **3.5.8 safety integrity level SIL:**
 - discrete level (one out of a possible four), corresponding to a range of safety integrity values, where safety integrity level 4 has the highest level of safety integrity and safety integrity level 1 has the lowest

Safety Integrity Levels (IEC 61508) cont.

Is the result of a risk assessment (IEC 61508 – part 5).

Table 2 – Safety integrity levels **target failure measures** for a safety function operating in low demand mode of operation

Safety integrity level (SIL)	Average probability of a dangerous failure on demand of the safety function (PFD _{avg})
4	$\geq 10^{-5}$ to $< 10^{-4}$
3	$\geq 10^{-4}$ to $< 10^{-3}$
2	$\geq 10^{-3}$ to $< 10^{-2}$
1	$\geq 10^{-2}$ to $< 10^{-1}$

Table 3 – Safety integrity levels **target failure measures** for a safety function operating in high demand mode of operation or continuous mode of operation

Safety integrity level (SIL)	Average frequency of a dangerous failure of the safety function [h ⁻¹] (PFH)
4	$\geq 10^{-9}$ to $< 10^{-8}$
3	$\geq 10^{-8}$ to $< 10^{-7}$
2	$\geq 10^{-7}$ to $< 10^{-6}$
1	$\geq 10^{-6}$ to $< 10^{-5}$

Safety Integrity Levels (IEC 61508) cont.

mode of operation: way in which a safety function operates, which may be either

- low demand mode: where the safety function is only performed on demand, in order to transfer the EUC into a specified safe state, and where the frequency of demands is no greater than one per year; or
- high demand mode: where the safety function is only performed on demand, in order to transfer the EUC into a specified safe state, and where the frequency of demands is greater than one per year; or
- continuous mode: where the safety function retains the EUC in a safe state as part of normal operation

Safety Integrity Levels (IEC 61508) cont.

average probability of dangerous failure on demand (PFD_{avg}):

- mean unavailability (see IEC 60050-191) of an E/E/PE safety-related system to perform the specified safety function when a demand occurs from the EUC or EUC control system

average frequency of a dangerous failure per hour (PFH)

- average frequency of a dangerous failure of an E/E/PE safety related system to perform the specified safety function over a given period of time

Safety Integrity Levels (IEC 61508) cont.

- *NOTE 3: Tables 2 and 3 relate the target failure measures, as allocated to a safety function carried out by an E/E/PE safety-related system, to the safety integrity level. It is accepted that it will not be possible to predict quantitatively the safety integrity of all aspects of E/E/PE safety-related systems. Qualitative techniques, measures and judgements will have to be made with respect to the precautions considered necessary to ensure that the target failure measures are achieved...*
- *NOTE 4 For hardware safety integrity it is necessary to apply quantified reliability estimation techniques in order to assess whether the target safety integrity, as determined by the risk assessment, has been achieved, taking into account random hardware failures (see IEC 61508-2, 7.4.5).*

Safety Integrity Levels (IEC 61508) cont.

- Determination of the safety integrity of a safety function is non-trivial as it highly depends on expert knowledge in the application area.
- Various methods are informally presented in IEC to determine the safety integrity (and consequently also the SIL).
- Examples are: ALARP (as low as reasonable possible), and the quantitative method (IEC 61508 – part 5).

Safety Integrity Levels (IEC 61508) cont.

Is the result of a risk assessment (IEC 61508 – part 5).

Table 2 – Safety integrity levels **target failure measures** for a safety function operating in low demand mode of operation

Safety integrity level (SIL)	Average probability of a dangerous failure on demand of the safety function (PFD _{avg})
4	$\geq 10^{-5}$ to $< 10^{-4}$
3	$\geq 10^{-4}$ to $< 10^{-3}$
2	$\geq 10^{-3}$ to $< 10^{-2}$
1	$\geq 10^{-2}$ to $< 10^{-1}$

Table 3 – Safety integrity levels **target failure measures** for a safety function operating in high demand mode of operation or continuous mode of operation

Safety integrity level (SIL)	Average frequency of a dangerous failure of the safety function [h ⁻¹] (PFH)
4	$\geq 10^{-9}$ to $< 10^{-8}$
3	$\geq 10^{-8}$ to $< 10^{-7}$
2	$\geq 10^{-7}$ to $< 10^{-6}$
1	$\geq 10^{-6}$ to $< 10^{-5}$

Safety Integrity Levels (IEC 61508) cont.

- Once the SIL of a given safety function is determined, IEC 61508 (part 2, 3) defines particular requirements. IEC 61508 is product prescriptive, i.e., it requires that the end product implements specific features:

Technique/Measure *		Ref.	SIL 1	SIL 2	SIL 3	SIL 4
Architecture and design feature						
1	Fault detection	C.3.1	---	R	HR	HR
2	Error detecting codes	C.3.2	R	R	R	HR
3a	Failure assertion programming	C.3.3	R	R	R	HR
3b	Diverse monitor techniques (with independence between the monitor and the monitored function in the same computer)	C.3.4	---	R	R	---
3c	Diverse monitor techniques (with separation between the monitor computer and the monitored computer)	C.3.4 ↑	---	R	R	HR

IEC 61508 – part 7

Safety Integrity Levels (IEC 61508) cont.

- IEC 61508 part 7 gives an overview of techniques and measures, e.g. C.3.3 Failure assertion programming

C.3.3 Failure assertion programming

NOTE This technique/measure is referenced in Table A.17 of IEC 61508-2, and Tables A.2 and C.2 of IEC 61508-3.

Aim: To detect residual software design faults during execution of a program, in order to prevent safety critical failures of the system and to continue operation for high reliability.

Description: The assertion programming method follows the idea of checking a pre-condition (before a sequence of statements is executed, the initial conditions are checked for validity) and a post-condition (results are checked after the execution of a sequence of statements). If either the pre-condition or the post-condition is not fulfilled, the processing reports the error.

For example,

```
assert < pre-condition>;
  action 1;
  :
  :
  action x;
assert < post-condition>;
```

References:

Exploiting Traces in Program Analysis. A. Groce, R. Joshi. Lecture Notes in Computer Science vol 3920, Springer Berlin / Heidelberg, 2006, ISBN 978-3-540-33056-1

Software Development – A Rigorous Approach. C. B. Jones, Prentice-Hall, 1980

Automotive SIL (ASIL)

Automotive Safety Integrity Levels (ISO 26262)

- IEC 61508 determines the SIL levels by consideration of the consequence of the hazardous event and by the probability of occurrence of this event.
- The equivalent parameters in ISO 26262 are:
 - Severity:
 - estimate of the extent of harm to one or more individuals that can occur in a potentially hazardous situation
 - Exposure (actually – the probability of exposure)
 - state of being in an operational situation that can be hazardous if coincident with the failure mode under analysis

Automotive Safety Integrity Levels (ISO 26262) cont.

- ISO 26262 defines in addition also a third parameter: the controllability.
- **Controllability:**
 - ability to avoid a specified harm or damage through the timely reactions of the persons involved, possibly with support from external measures
- E.g., in current series implementations of driver assistance systems, the driver is requested to be alert such that he/she can take over in case of an emergency. Typically the driver needs to get in contact with the steering wheel every few seconds. This increases and enforces the controllability.

Automotive Safety Integrity Levels (ISO 26262) cont.

- Classes of Severity:

Class	S0	S1	S2	S3
Description	No injuries	Light and moderate injuries	Severe and life-threatening injuries (survival probable)	Life-threatening injuries (survival uncertain), fatal injuries

- Classes of Probability of Exposure:

	Class				
	E0	E1	E2	E3	E4
Description	Incredible	Very low probability	Low probability	Medium probability	High probability

- Classes of Controllability:

	Class			
	C0	C1	C2	C3
Description	Controllable in general	Simply controllable	Normally controllable	Difficult to control or uncontrollable

Automotive Safety Integrity Levels (ISO 26262) cont.

- QM: Quality Management – there are no hazards associated with the given application
- ASIL A: lowest automotive safety integrity level, moderate additional requirements towards the development process (on top of QM), example sub-system: retractable hardtop for convertibles
- ASIL B: example sub-system: head & tail lights
- ASIL C: example sub-system: electric drivetrain
- ASIL D: highest automotive safety integrity level, rigorous development process requirements, example sub-system: EPS (electro-mechanical power steering)

Automotive Safety Integrity Levels (ISO 26262) cont.

Table 4 — ASIL determination

Severity class	Probability class	Controllability class		
		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

Determination of ASIL

		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

Table 6.4: ASIL assignment

			C1	C2	C3
S1	E1	QM	QM	QM	
	E2	QM	QM	QM	
	E3	QM	QM	A	
	E4	QM	A	B	
S2	E1	QM	QM	QM	
	E2	QM	QM	A	
	E3	QM	A	B	
	E4	A	B	C	
S3	E1	QM	QM	A	
	E2	QM	A	B	
	E3	A	B	C	
	E4	B	C	D	

Table 6.4: ASIL assignment

Light/moderate injury

Severe / lifethreatening injury (survival possible)

Lifethreatening / fatal injury (survival uncertain)

Simply controllable (≥ 99% of drivers are able to control)

Normally controllable (≥ 90% of drivers are able to control)

Difficult to control or uncontrollable

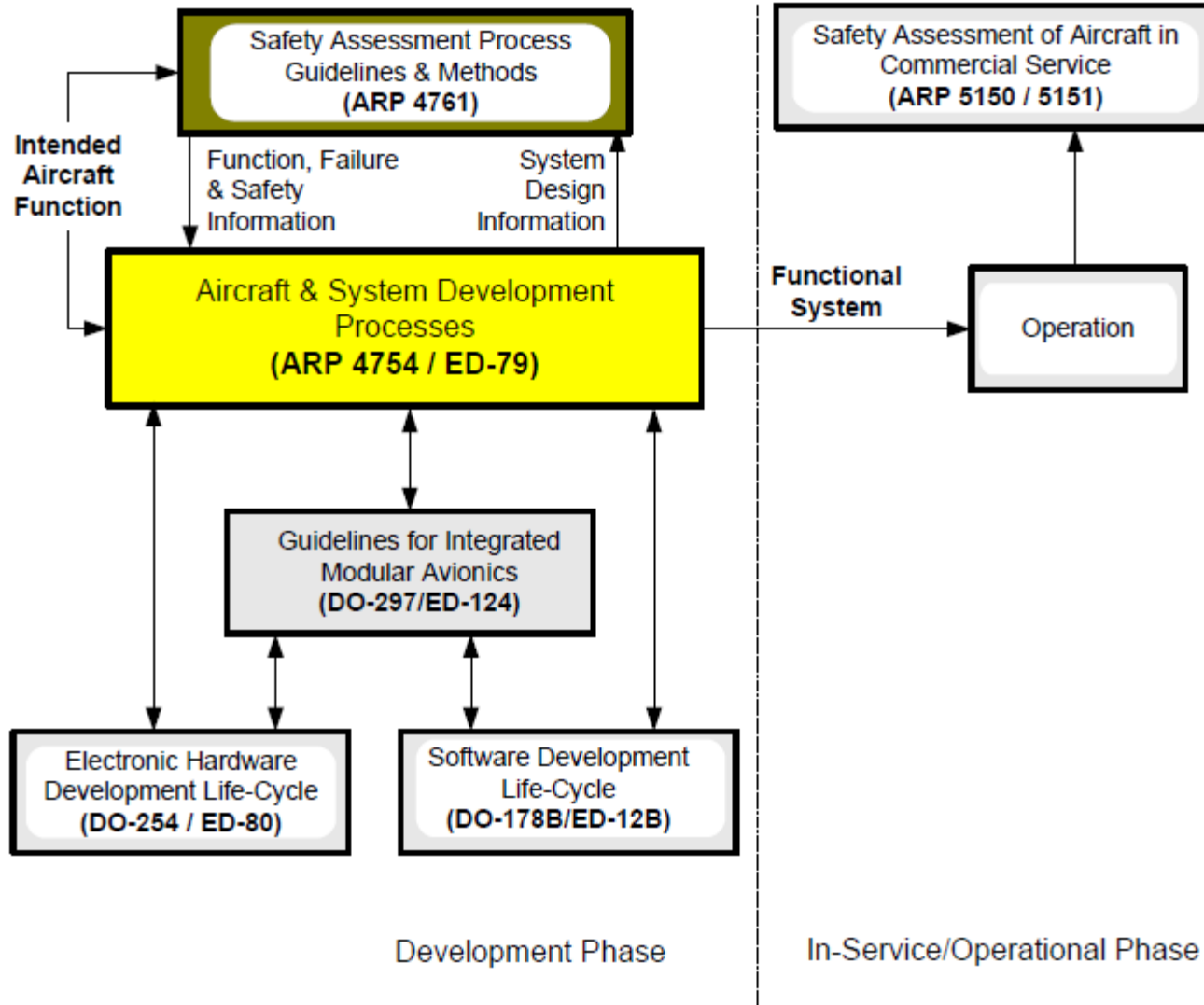
Very low probability

Low (<1% of operating time)

Medium (1-10% of operating time)

High (>10% of operating time)

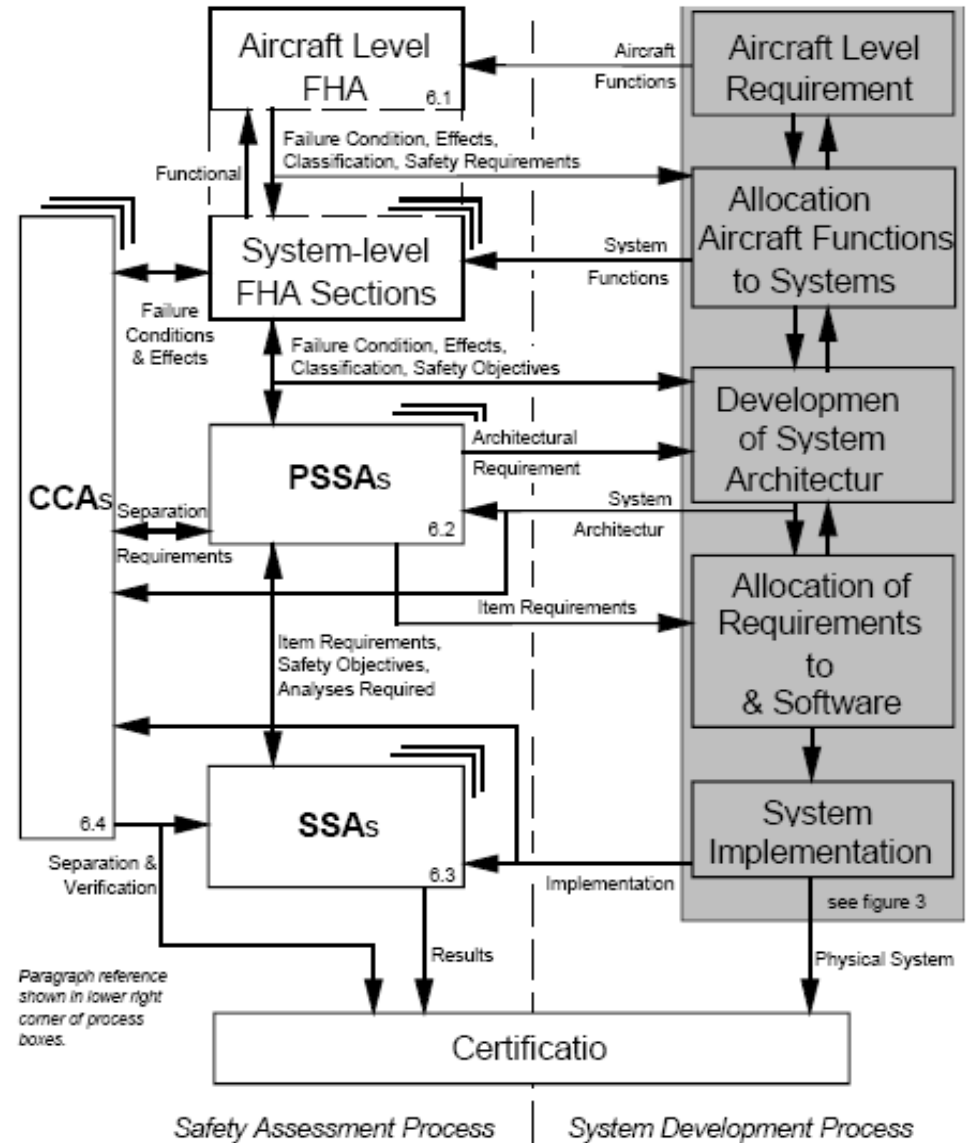
ARP 4754, 4761, DO 178,
DO 254



SAE ARP 4754A

FIGURE 1 - GUIDELINE DOCUMENTS COVERING DEVELOPMENT AND IN-SERVICE/OPERATIONAL PHASES

- Aircraft level functional requirements are allocated to aircraft systems
- Iterative analysis with Functional Hazard Assessment (FHA)
 - Determines severity of failures
- Development of System Architecture
 - Allocation, Redundancy, Partitioning, etc.
- Preliminary System Safety Assessment (PSSA) of design, iteratively (top-down)
 - Determines Safety Requirements and
 - Development Assurance Levels
- Allocation of requirements to hardware and software items
- HW/SW item development according to DO-254 and DO-178B, respectively
- *System Safety Assessments (SSAs)* analyze implementation (bottom-up)



SAE ARP 4754

Common Cause Analysis

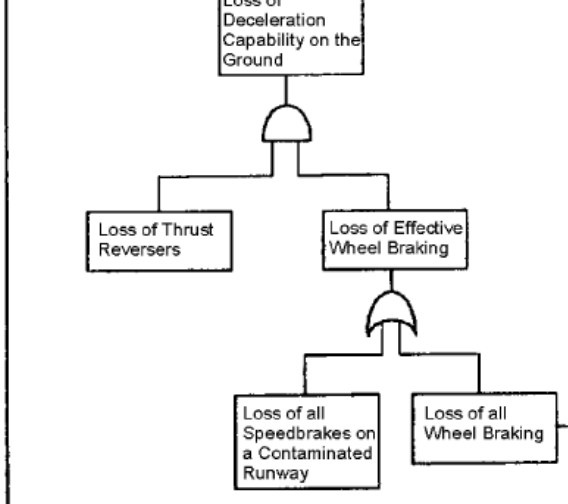
- Common Cause Analysis (CCA) targets design errors that may invalidate subsystem *failure independence assumptions* required by the (P)SSA.
 - **Zonal Safety Analysis:**
should examine each physical zone of the aircraft to ensure that equipment installation and potential physical interference with adjacent systems do not violate the independence requirements of the systems.
 - **Particular Risk Assessment:**
should examine those common events or influences that are outside the system(s) concerned but which may violate independence requirements. These particular risks may also influence several zones at the same time, whereas zonal safety analysis is restricted to each specific zone.
 - **Common Mode Analysis:**
provides evidence that the failures assumed to be independent are truly independent. The analysis also covers the effects of design, manufacturing, and maintenance errors and the effects of common component failures.

Concept & Architecture Development

Aircraft FHA

Funct. Failure Ref.	Function	Phase	Failure Condition	Failure Effect	Classification
1.1.1	Decelerate Aircraft on Ground	Landing RTO	Loss of Deceleration Capability on the Ground	Crew is unable to stop aircraft on runway	Catastrophic
1.1.2	Decelerate Aircraft on Ground	Landing	Unannounced Loss of All Automatic Stopping Functions	Crew must use manual procedures to stop aircraft	Major

Aircraft FTAs



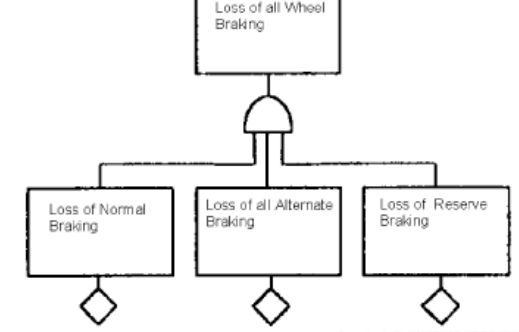
Quantitative

Preliminary Design

System FHAs

Electrical System					
Hydraulic System					
Speedbrake System					
Thrust Reverser System					
Brake System					
Funct. Failure Ref.	Function	Phase	Failure Condition	Failure Effect	Classification
30-40 1.1	Wheel Braking	Landing RTO	Loss of all Wheel Braking	Crew's ability to stop the aircraft on runway to significantly reduced	Hazardous
36-40 1.2	Auto Braking	RTO Landing	Unannounced Loss of Autobraking	Crew must use manual procedures to stop aircraft	Major

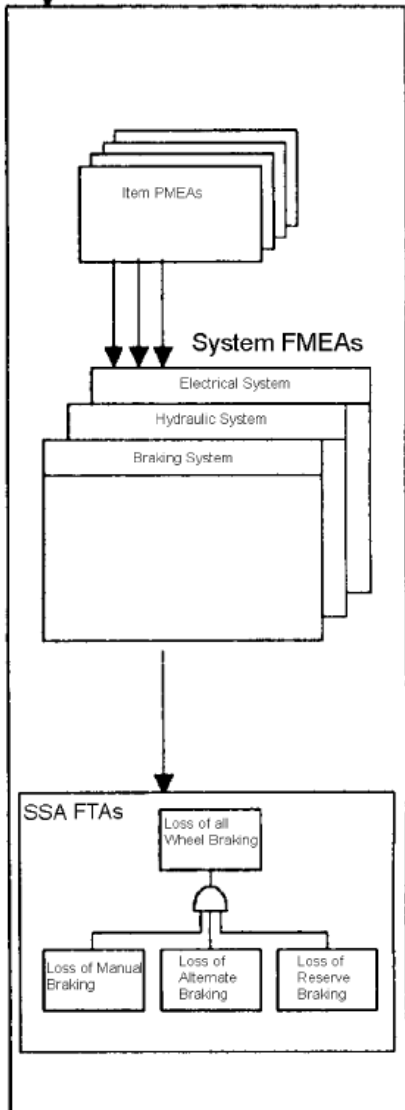
PSSA FTAs



Quantitative

Quantitative

Detailed Design



Quantitative

Failure Mode Classification – Consequences, e.g., Aircraft

Minor: **10E-5 per flight hour or greater**

no significant reduction of aeroplane safety, a slight reduction in the safety margin

Major: **between 10E-5 and 10E-7**

significant reduction in safety margins or functional capabilities, significant increase in crew workload or discomfort for occupants

Hazardous: **between 10E-7 and 10E-9**

large reduction in safety margins or functional capabilities, causes serious or fatal injury to a relatively small number of occupants

Catastrophic: **less than 10E-9**

these failure conditions would prevent the continued safe flight and landing of the aircraft

Design Assurance Levels (ARP, DO 178, DO 254)

- Design Assurance Levels are determined only by the effects on the aircraft:
 - DAL A: Catastrophic
 - DAL B: Hazardous failure condition
 - DAL C: Major
 - DAL D: Minor
 - DAL E: No Effect
- DO 178 and DO 254 are process prescriptive,
 - i.e., the DAL defines which processes need to be executed and how.
- DO 178 and DO 254 are not product prescriptive,
 - i.e., the DAL does not require specific functions in an end product

Assurance Cases / Safety Cases

Definitions

- *“An assurance case provides arguments to justify certain claims about a system, based on evidence concerning both the system and the environment in which it operates.”*
[Rushby]
- A safety case is a special kind of assurance case in which the claims being argued concern safety properties.

Prescriptive Method vs. Performance-Oriented Method

- Prescriptive methods can be product prescriptive and/or process prescriptive.
 - We have discussed IEC 61508 and ISO 26262 as product prescriptive methods.
 - We have discussed DO 178b/c and DO 254 as project prescriptive methods.
- In performance-oriented methods, *“the certification authority specifies a threshold of acceptable performance and a means for assuring that the threshold has been met. [...] it is up to the assurer to decide how to accomplish that goal.”* [Leveson].



TECHNISCHE
UNIVERSITÄT
WIEN

TTTech

Dependable Systems

Part 5: Failure Modes and Models

Contents

- Recap from Part 2: Canonical Failure Classification
- Failure Mode Hierarchy
- Fault-Hypothesis, Failure Semantics, and Assumption Coverage
- Failure Hypothesis Estimation
- Overview of Safety Analysis Methods
- Comparison of Safety Analysis Methods

Recap from Part 2: Canonical Failure Classification

Failures

Recap:

A **(service) failure** is an event that occurs when the delivered service deviates from correct service.

- Thus, a failure is a transition from correct service to incorrect service.

Failure Mode Classification – Overview

- Domain:
 - content, early timing failure, late timing failure, halt failure, erratic failure
- Detectability:
 - signaled failures, unsignaled failures
- Consistency:
 - consistent failure, inconsistent failure
- Consequences:
 - minor failure, ..., catastrophic failure

Failure Mode Classification – Domain

- Content
- Early timing failure
- Late timing failure
- Halt failure
 - the external state becomes constant, i.e., system activity is no longer perceptible to the users
 - silent failure mode is a special kind of halt failure in that no service at all is delivered
- Erratic failure
 - not a halt failure, e.g., a babbling idiot failure

Failure Mode Classification – Consistency

When there are more than one users of a service.

- Consistent failure:
 - All users experience the same incorrect service.
- Inconsistent failure
 - Different users experience different incorrect services.

Failure Mode Classification – Consequences, e.g., Aircraft

Minor: **10E-5 per flight hour or greater**

no significant reduction of aeroplane safety, a slight reduction in the safety margin

Major: **between 10E-5 and 10E-7**

significant reduction in safety margins or functional capabilities, significant increase in crew workload or discomfort for occupants

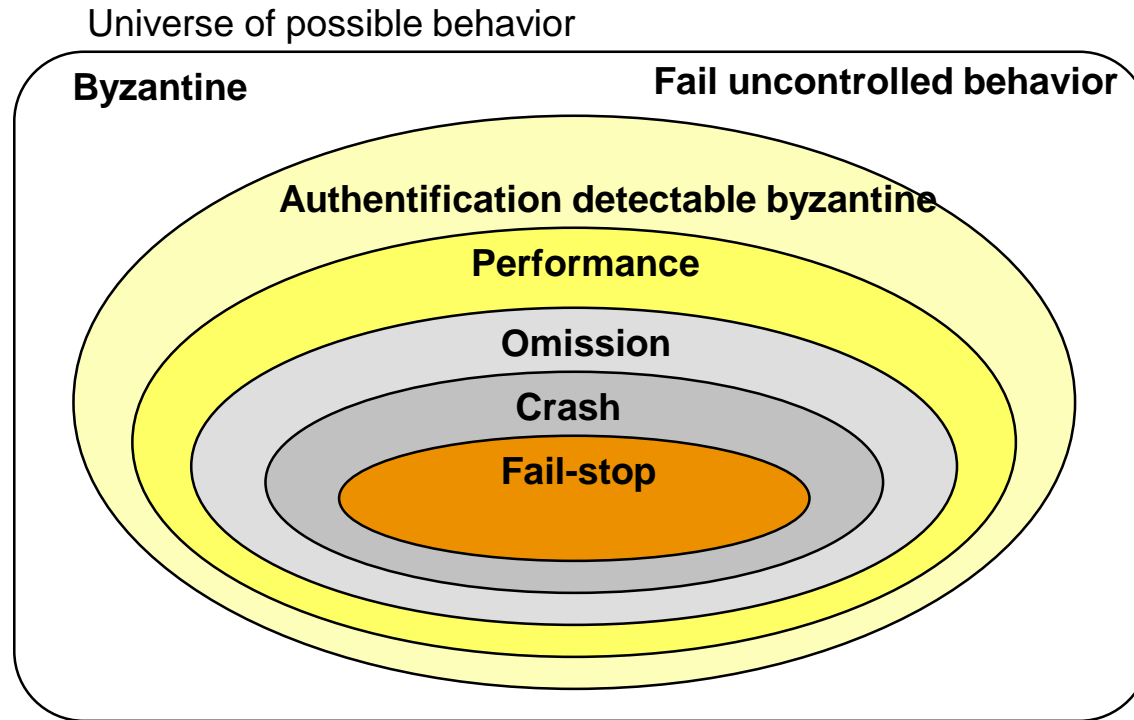
Hazardous: **between 10E-7 and 10E-9**

large reduction in safety margins or functional capabilities, causes serious or fatal injury to a relatively small number of occupants

Catastrophic: **less than 10E-9**

these failure conditions would prevent the continued safe flight and landing of the aircraft

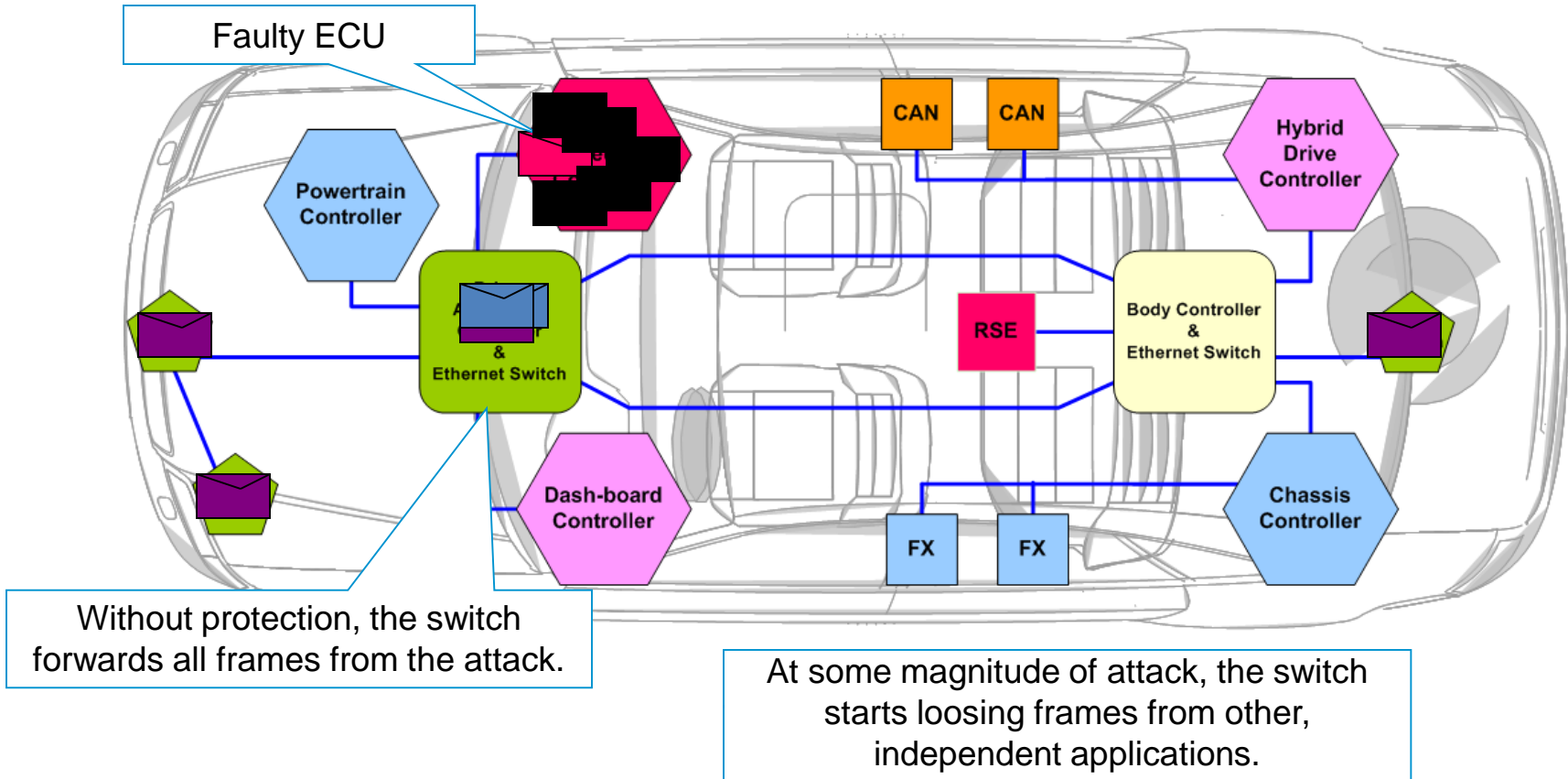
Failure Mode Hierarchy

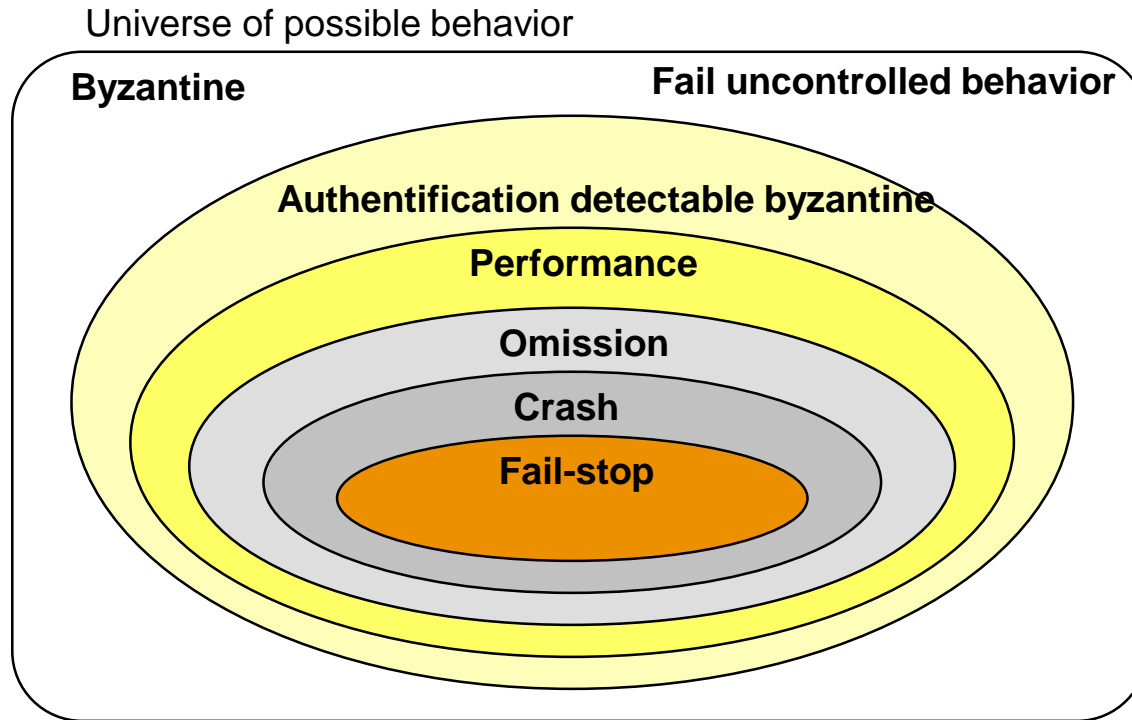


Based on the *strength* of the assumptions the failure modes form a hierarchy.

- byzantine failures are based on the weakest assumption (a non-assumption)
- fail-stop failures are based on the strongest assumptions (only correct results, information about the last correct state in case of a failure)

Failure Example





Based on the *strength* of the assumptions the failure modes form a hierarchy.

- byzantine failures are based on the weakest assumption (a non-assumption)
- fail-stop failures are based on the strongest assumptions (only correct results, information about the last correct state in case of a failure)

Failure Mode Hierarchy (cont.)

- **Byzantine or arbitrary failures:**
there is no restriction on the behavior at the system interface, this mode is often called fail-uncontrolled
(“two-faced” behavior, forging of messages)
- **Authentication detectable byzantine failures:**
the only restriction on the behavior at the system interface is that messages of other systems cannot be forged
(this failure mode applies only to distributed systems)

Failure Mode Hierarchy (cont.)

- **Performance failures:**
under this failure mode systems deliver correct results in the value domain, in the time domain results may be early or late (early or late failures)
- **Omission failures:**
a special class of performance failures where results are either correct or infinitely late (for distributed systems subdivision in send and receive omission failures)

Failure Mode Hierarchy (cont.)

- **Crash failures:**
a special class of omission failures where a system does not deliver any subsequent results if it has exhibited an omission failure once
(the system is said to have crashed)
- **Fail-Stop failures:**
besides the restriction to crash failures it is required that other (correct) systems can detect whether the system has failed or not and can read the last correct state from a stable storage

Fault-Hypothesis, Failure Semantics, and Assumption Coverage

Fault-Hypothesis, etc.

Concepts

- **Fault hypothesis:**
The fault hypothesis specifies anticipated faults which a server must be able to handle (also fault assumption).
- **Failure semantics:**
A server exhibits a given failure semantics if the probability of failure modes which are not covered by the failure semantics is sufficiently low.
- **Assumption coverage:**
Assumption coverage is defined as the probability that the possible failure modes defined by the failure semantics of a server proves to be true in practice conditions on the fact that the server has failed.

Fault-Hypothesis, etc. (cont.)

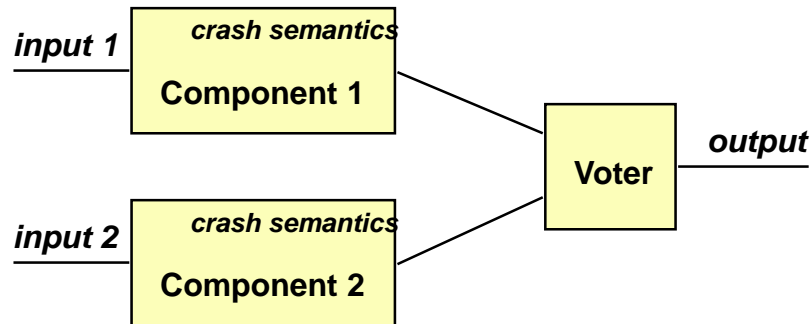
Importance of assumption coverage

- The definition of a proper fault hypothesis, failure semantics and achievement of *sufficient* coverage is one of the most important factors.
- If the fault hypothesis (or failure semantics) is violated a system may fail as a whole.

Fault-Hypothesis, etc. (cont.)

Assumption Coverage Example

If component 1 or 2 violates its failure semantics the system fails, although it was designed to tolerate 1 component failure.



Why?

Fault-Hypothesis, etc. (cont.)

The Titanic or: violated assumption coverage

- **The fault hypothesis:**

The Titanic was built to stay afloat if less or equal to 4 of the underwater departments were flooded.

- **Rationale of fault hypothesis:**

This assumption was reasonable since previously there had never been an incident in which more than four compartments of a ship were damaged.

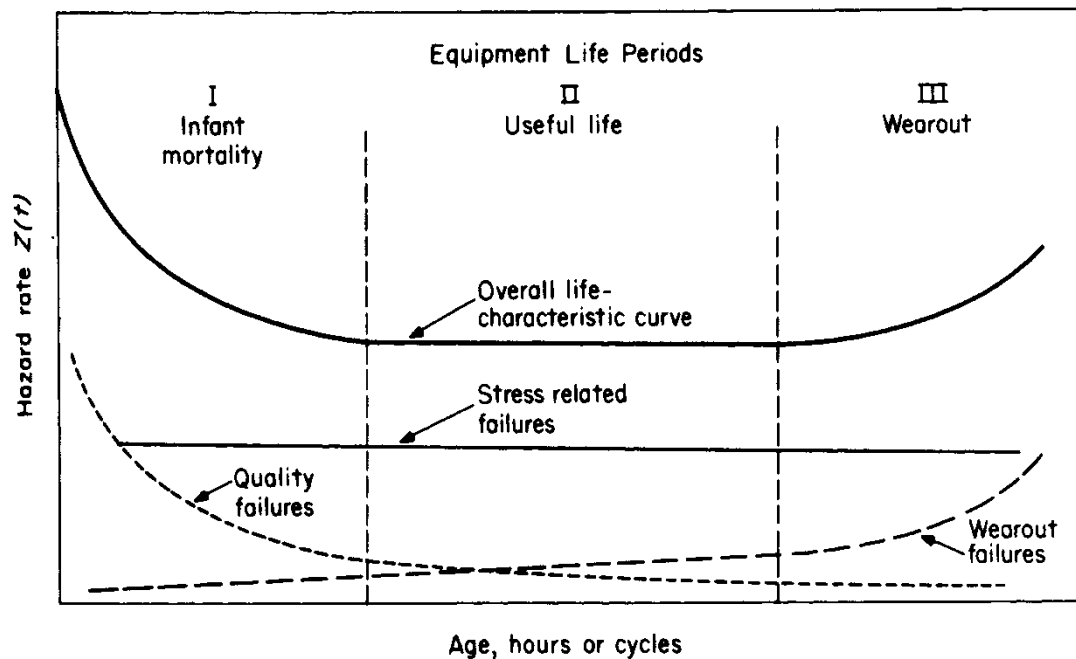
- **But:**

Unfortunately, the iceberg ruptured five spaces, and the following events went down to history.

Failure Hypothesis Estimation

Life-characteristics curve (Bathtub curve)

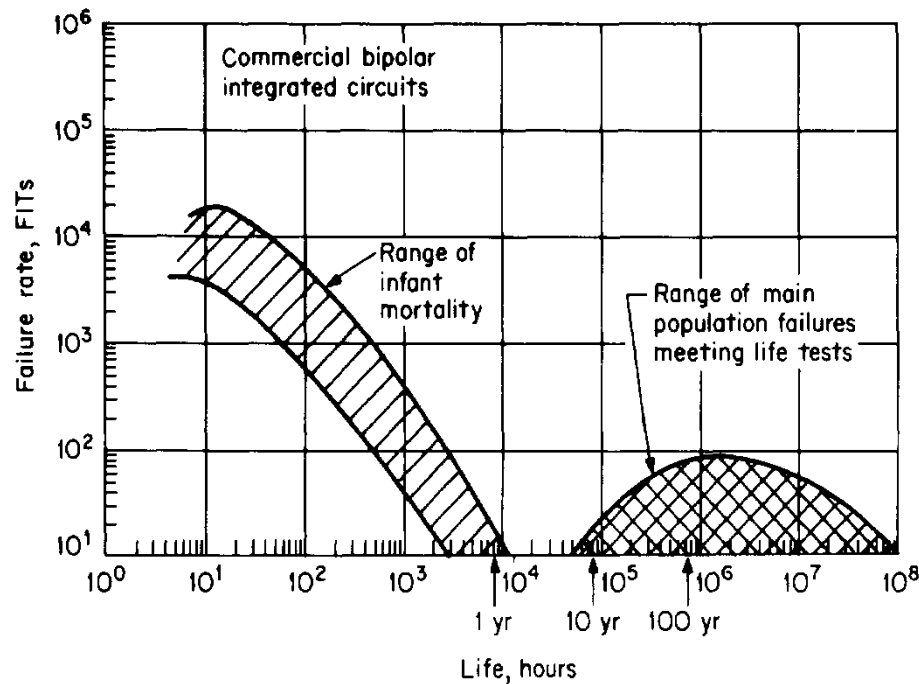
- For semiconductors, out of three terms describing the life characteristics only *infant mortality* and the *constant-failure-rate* region are of concern



Life-characteristics curve, showing the three components of failure

Semiconductor failure rate

- a typical failure rate distribution for semiconductors shows that wear out is of no concern



Semiconductor failure rate

Stress Tests

- semiconductor failures are stress dependent
- the most influential stress factor is temperature

Stress Tests (cont.)

Arrhenius equation

- the basic relationship between the activation rate of failures and temperature is described by the Arrhenius equation

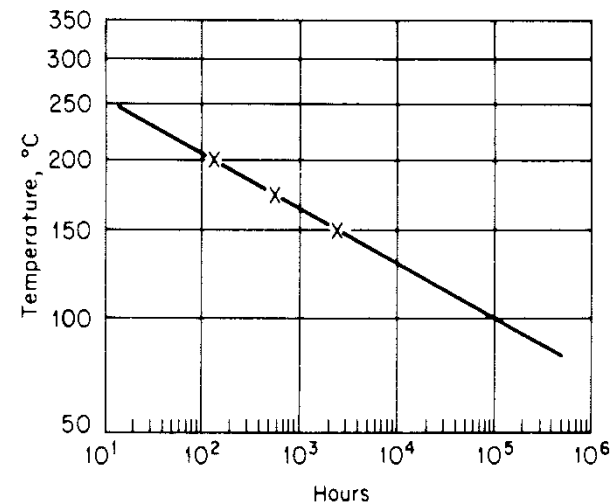
$$R = R_0 e^{-\frac{E_A}{kT}}$$

R_0 .. constant

T .. absolute temperature (K)

E_A .. activation energy (eV)

k .. Boltzmann's constant $8.6 \cdot 10^{-5}$ eV/K



Arrhenius plot ($E_A = 1$ eV)

Stress Tests (cont.)

Accelerated stress testing of semiconductors

- to remove freaks and infant-mortality failures (screening)
- to determine the expected failure rate

Accelerated conditions:

accelerated temperature
cycling of temperature
temperature and voltage stress
temperature, voltage and
humidity stress

lowering of temperature
high temperature and current
 α particles
high voltage gradients

Stress Tests (cont.)

Software stress

- For software there is no sound empirical and mathematical basis to use stress as a method to characterize the behavior of components.
 - it is currently unknown how to characterize stress for software
 - it is impossible to carry out accelerated stress tests to examine failure rates for software
 - for software there is no such relation as the Arrhenius equation which describes the activation rate of failures
 - there is no general possibility to “over-engineer” a system to handle conditions which are more stressful

Hardware/Software Interdependence

- software depends on hardware:
 - software requires hardware to execute (e.g. Intel's Pentium bug)
- hardware depends on software:
 - VLSI design uses software tools
 - PCB layout and routing by software tools
 - EMC analysis by software tools
 - hardware testers are software driven

Overview of Safety Analysis Methods

Safety Analysis (cont.)

Concepts

System Safety: is a subdiscipline of system engineering that applies scientific, management, and engineering principles to ensure adequate safety, throughout the operational life cycle, within the constraints of operational effectiveness, time and cost.

Safety: has been defined as “freedom from those conditions that can cause death, injury, occupational illness, or damage to or loss of equipment or property”. safety has to be regarded as a relative term.

Software Safety: to ensure that the software will execute within a system context without resulting in unacceptable risk

Safety analysis

Overview

- includes complete life cycle of project/product (specification, design, maintenance, modification, ...)
- definition of responsibilities
- communication with other groups
- complete documentation
- analysis of complex processes
- management procedures (specialists, meetings, action reviews, time schedule, ...)

Safety Analysis (cont.)

Major topics of Safety analysis

- which (hazard analysis)
- how (accident sequencing)
- how likely (quantitative analysis)

Safety Analysis (cont.)

Safety analysis methodologies

- Preliminary Hazards Analysis (PHA)
- Hazards and Operability Study (HAZOP)
- Action Error Analysis (AEA)
- Fault Tree Analysis (FTA)
- Event Tree Analysis (ETA)
- Failure Modes and Effect Analysis (FMEA)
 Failure Modes, Effect and Criticality Analysis (FMECA)
- Cause-consequence analysis

Safety Analysis (cont.)

Preliminary hazard analysis (PHA)

- The first step in any safety program is to identify hazards and to categorize them with respect to criticality and probability
 - define system hazards
 - define critical states and failure modes
 - identify critical elements
 - determine consequences of hazardous events
 - estimate likelihood of hazardous events
 - issues to be analyzed in more detail

Safety Analysis (cont.)

Hazards and Operability Study (HAZOP)

Based on a systematic search to identify deviations that may cause hazards during system operation

Intention: for each part of the system a specification of the “intention” is made

Deviation: a search for deviations from intended behavior which may lead to hazards

Guide Words: Guide words on a check list are employed to uncover different types of deviations
(NO, NOT, MORE, LESS, AS WELL AS, PART OF, REVERSE, OTHER THAN)

Team: the analysis is conducted by a team, comprising different specialists

Safety Analysis (cont.)

Example for HAZOP

- **Intention:** pump a specified amount of A to reaction tank B. Pumping of A is complete before B is pumped over.

NO or NOT

- the tank containing A is empty
- one of the pipe's two valves V1 or V2 is closed
- the pump is blocked, e.g. with frozen liquid
- the pump does not work (switched off, no power, ...)
- the pipe is broken

CONSEQUENCE is serious, a possible explosion

MORE

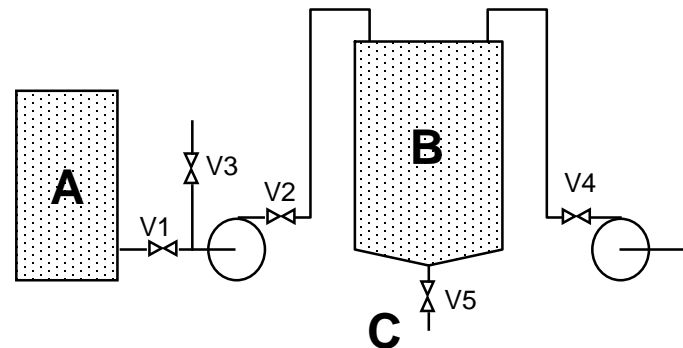
- the pump has a too high capacity
 - the opening of the control valve is too large
- CONSEQUENCE not serious, tank gets overfilled

AS WELL AS

- valve V3 is open, another liquid or gas gets pumped
- contaminants in the tank
- A is pumped to another place (leak in the connecting pipe)

CONSEQUENCE is serious, a possible explosion

...



Safety Analysis (cont.)

Action Error Analysis (AEA)

Considers the operational, maintenance, control and supervision actions performed by human beings. The potential mistakes in individual actions are studied.

- list steps in operational procedures (e.g. “press button A”)
- identification of possible errors for each step, using a check-list of errors
- assessment of the consequences of the errors
- investigations of causes of important errors
(action not taken, actions taken in wrong order, erroneous actions, actions applied to wrong object, late or early actions, ...)
- analysis of possible actions designed to gain control over these process
- relevant for software in the area of user interface design

Taiwan

TransAsia crash pilot pulled wrong throttle, shut down sole working engine

Report into plane crash that killed 43 people in Taiwan says captain had previously failed training on dealing with engine

<https://www.theguardian.com/world/2015/jul/02/transasia-crash-pilot-pulled-wrong-throttle-shut-down-sole-engine>

https://en.wikipedia.org/wiki/Kegworth_air_disaster

Kegworth air disaster

From Wikipedia, the free encyclopedia

Coordinates: 52°49′55″N 1°17′57.5″W﻿ / ﻿



This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed. *(October 2010)* [\(Learn how and when to remove this template message\)](#)

The **Kegworth air disaster** occurred when a [Boeing 737-400](#) crashed on to the embankment of the [M1 motorway](#) near [Kegworth](#), Leicestershire, England, while attempting to make an emergency landing at [East Midlands Airport](#) on 8 January 1989.

[British Midland Flight 92](#) was on a scheduled flight from [London Heathrow Airport](#) to [Belfast Airport](#), when a fan-blade broke in the left engine, disrupting the air conditioning and filling the flight deck with smoke. The pilots believed that this indicated a fault in the right engine, since earlier models of the 737 ventilated the flight-deck from the right, and they were unaware that the 400 used a different system. The crew mistakenly shut down the good engine, and pumped more fuel into the malfunctioning one, which burst into flames. Of the 126 people aboard, 47 died and 74 sustained serious injuries.

The inquiry attributed the blade fracture to metal fatigue, caused by heavy

British Midland Flight 92



Safety Analysis (cont.)








Fault Tree Analysis (FTA)

A graphical representation of logical combinations of causes that may lead to a hazard (top-event). Can be used as a quantitative method.

- identification of hazards (top-events)
- analysis to find credible combinations which can lead to the top-event
- graphical tree model of parallel and sequential faults
- uses a standardized set of symbols for Boolean logic
- expresses top-event as a consequence of AND/OR combination of basic events
- minimal cut set is used for quantitative analysis

Safety Analysis (cont.)

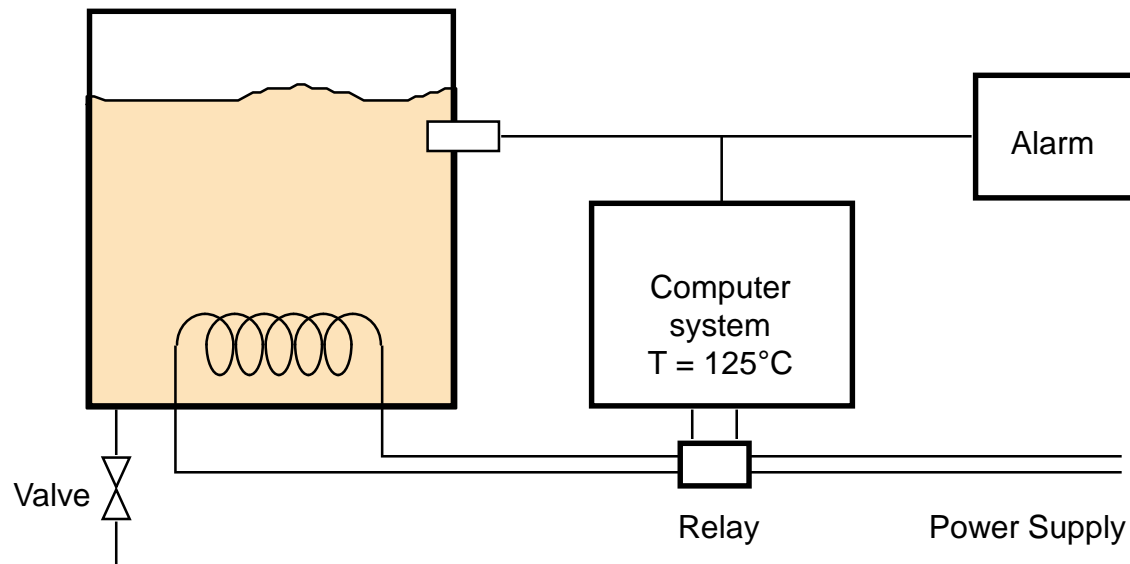
Symbols used in fault tree analysis

Symbol	Designation	Function
	BASIC EVENT	Basic event or failure
	UNDEVELOPED EVENT	Causes are not developed
	EVENT	Event resulting from more basic events
	CONDITIONAL EVENT	Event that can occur normally
	AND gate	Output event occurs only if all input events occur simultaneously
	OR gate	Output event occurs if any one of the input events occurs
	TRANSFER SYMBOL	Represents an event which comes from another lower-order fault tree or which is to be transferred to a higher-order tree

Safety Analysis (cont.)

An Example for fault tree analysis

In a container two chemicals react with each other over a period of 10 hours at a temperature of 125 °C. If the temperature exceeds 175 °C toxic gas is emitted. The temperature is controlled by a computer system.

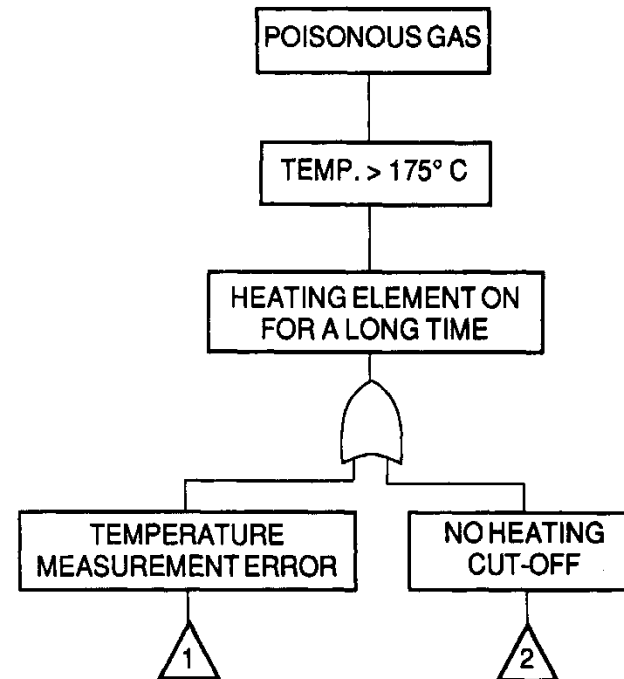


Safety Analysis (cont.)

An Example for fault tree analysis (cont.)

Identification of the top-event:

Emission of poisonous gas
is the top event

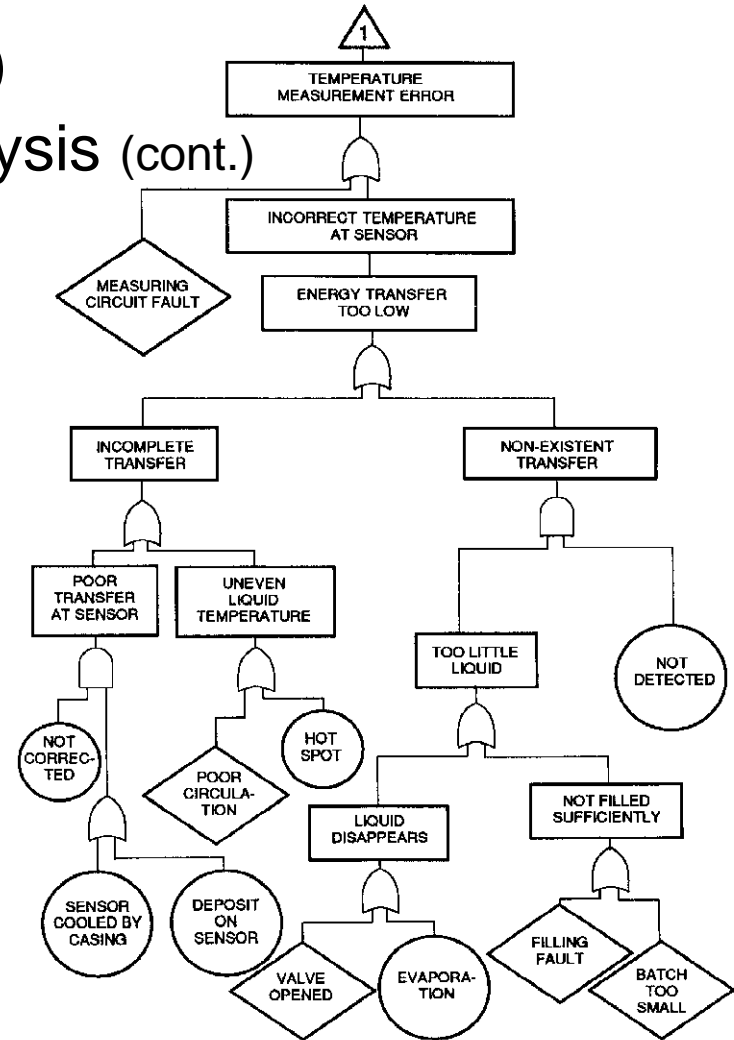


The upper part of the fault tree

Safety Analysis (cont.)

An Example for fault tree analysis (cont.)

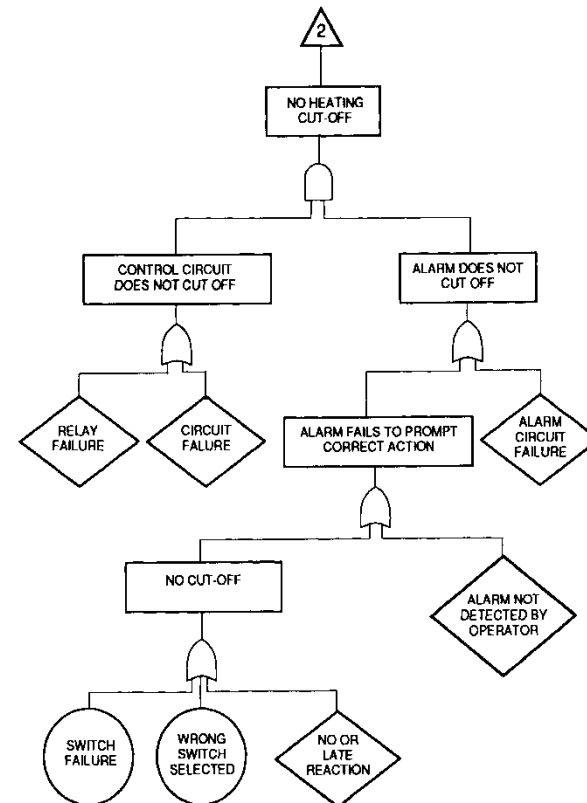
Subtree for temperature measurement failure



Safety Analysis (cont.)

An Example for fault tree analysis (cont.)

Subtree for heating cut off failure



Autonomous Driving Example

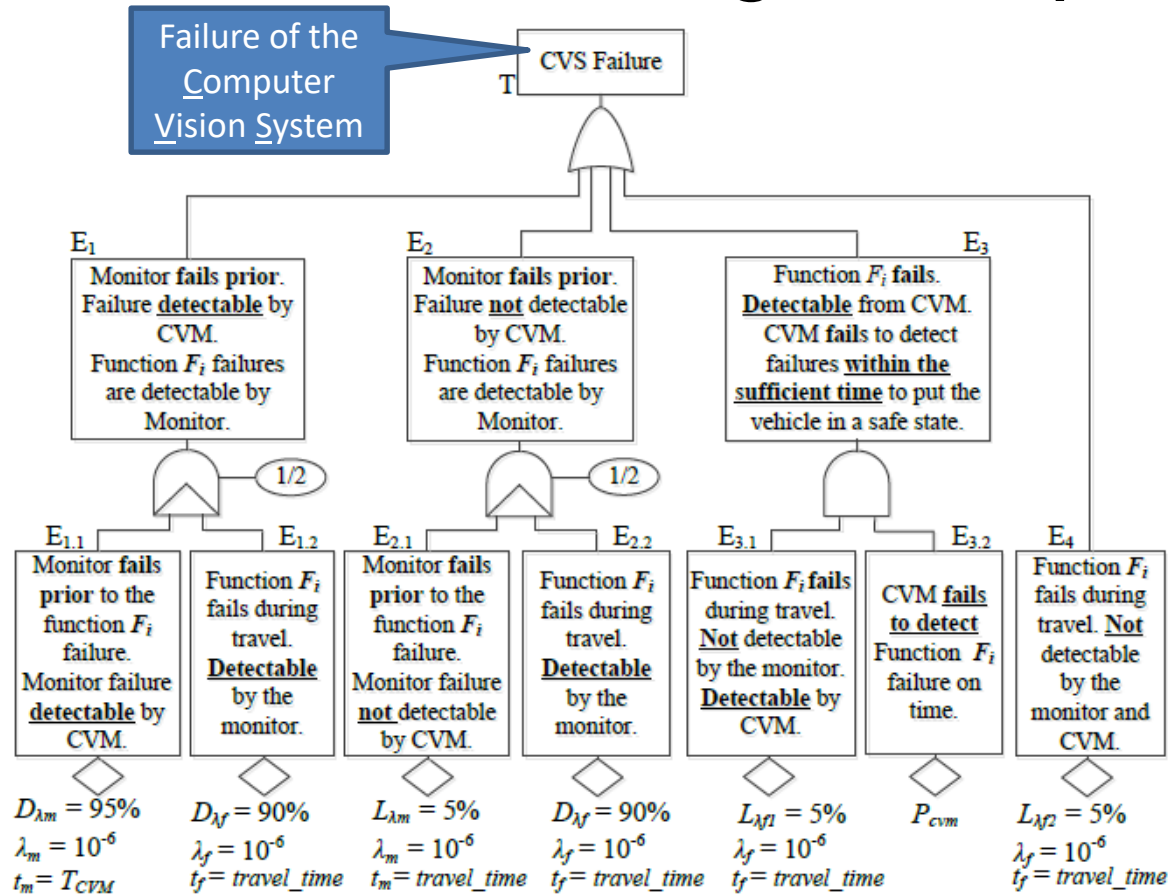


Fig. 4. High-level FTA for vision-based ADAS with CVM.

Safety Analysis (cont.)

Event Tree Analysis (ETA)

Models the potential consequences of faults which are considered as events. Can be used as a quantitative method.

- identification of basic events
- start with basic events and describe possible consequences of this event
- binary decision for consequences of events
- opposite of FTA which starts with top events

Safety Analysis (cont.)

Failure Modes and Effect Analysis (FMEA)

A common method where the designer in a systematical way has to answer the questions “How can the component fail?” and “What happens then?”.

- the system is divided up into different components in the form of a block diagram
- failure modes are identified for all components
- causes, consequences and the significance of failures are assessed for each failure mode

Safety Analysis (cont.)

Failure Modes and Effect Analysis (FMEA) (cont.)

- an investigation is made into how the failure can be detected
- if necessary, recommendations for suitable control measures are made
- analysis is supported by tabular sheets (e.g. IEC standard 1985)
- failure mode, effects and criticality analysis (FMECA) puts special emphasis on the criticality aspect

*Hazard assessment
criterias
according to VDA
(Verein Deutscher
Automobilhersteller)*

Safety Analysis (cont.)

An example FMEA hazard assessment

<i>Severity of consequence</i>		<i>Probability of occurrence</i>		<i>Probability of detection</i>	
10	Very severe System operation has to be abandoned or even a safety critical state may be reached	10	High It is almost certain that the failure will occur with high probability	500	10^{-6}
9	Severe Failure causes disturbance of end user (no safety critical failures or violations of regulations)	9	Moderate The component is similar to component designs which already have caused problems in the past	50	10^{-6}
8	Moderate Failure causes inconvenience of the end user, restricted system operation will be perceived by the customer	8	Small The component is similar to component designs which have caused problems in the past, but the extent of problems was relatively low	5	10^{-6}
7	Minor Failure causes only minor inconvenience of the end user, only minor restrictions of the system operation are perceivable	7	Very small The component is similar to component designs which had very low failure rates in the past	100	10^{-9}
6	Improbable It is very improbable that the failure will be perceived by the end user	6	Improbable It is very improbable that a failure occurs	1	10^{-9}
5				10	Unprobable It is impossible or at very improbable that the failure can be detected
4				9	Very low It is possible to detect the fault before the system fails
3				8	Small
2				7	
1				6	
				5	Moderate
				4	
				3	
				2	High
				1	Very High It is certain that the faults get detected before the system fails

Safety Analysis (cont.)

An example FMEA hazard assessment (cont.)

<i>Function</i>	<i>Failure Mode</i>	<i>Cause</i>	<i>Effect</i>	<i>Controls</i>	<i>Severity</i>			
					<i>Probability</i>	<i>Detection</i>	<i>Product</i>	
speed sensor	open	connector or harness	no operation possible	supplier quality control and end of line testing	9	4	3	108
		computer	no operation possible	computer supplier quality control and end of line testing	9	3	3	81
		sensor	no operation possible	sensor supplier quality control, module and end of line testing	9	4	3	108
	short to supply	connector or harness	no operation possible	supplier quality control and end of line testing	9	2	3	54
		computer	no operation possible	computer supplier quality control and end of line testing	9	2	3	54
		sensor	no operation possible	sensor supplier quality control, module and end of line testing	9	2	3	54
	short to ground	connector or harness	no operation possible	supplier quality control and end of line testing	9	1	3	27
		computer	no operation possible	computer supplier quality control and end of line testing	9	1	3	27
		sensor	no operation possible	sensor supplier quality control, module and end of line testing	9	1	3	27

Safety Analysis (cont.)

Cause-consequence analysis

Combination of fault tree analysis and event tree analysis

- starts at a critical event
- works forward by using event tree analysis (consequences)
- works backward by using fault tree analysis (causes)
- very flexible
- well documented method

Comparison of Safety Analysis Methods

Comparison of Safety Analysis Methods

Method	Advantages	Restrictions and deficiencies
Preliminary hazards analysis	A required first step.	None.
Hazards and operability study	Suitable for large chemical plants. Results in a list of actions, design changes and cases identified for more detailed study. Enhances the information exchange between system designers, process designers and operating personnel.	Technique is not well standardized and described in the literature. Most often applied to continuous processes.
Action error analysis	Gives the computer system designer proposals for proper interface design. Helps the personnel or users to monitor the process during operation and helps to prevent operator mistakes.	AEA is an analysis of the technical system, and does not analyze the behavior of operators. The thoughts and intentions of human beings, i.e. the reasons for mistakes, are not considered.

Comparison of Safety Analysis Methods (cont.)

Method	Advantages	Restrictions and deficiencies
Fault tree analysis	Well accepted technique. Very good for finding failure relationships. A fault oriented technique which looks for the ways a system can fail. Makes it possible to verify requirements, which are expressed as quantitative risk values.	Large fault trees are difficult to understand, bear no resemblance to system flow charts, and are mathematically not unique. It assumes that all failures are of binary nature, i.e. a component completes successfully or fails completely.
Event tree analysis	Can identify effect sequences and alternative consequences of failures. Allows analysis of systems with stable sequences of events and independent events.	Fails in case of parallel sequences. Not suitable for detailed analysis due to combinatorial explosion. Pays no attention to extraneous, incomplete, early or late actions.

Comparison of Safety Analysis Methods (cont.)

Method	Advantages	Restrictions and deficiencies
Failure modes and effects analysis	Easily understood, well accepted, standardized technique. Non-controversial, non-mathematical. Studies potential failures and their effects on the function of the system.	Examines non-dangerous failures and is therefore time consuming. Often combinations of failures and human factors not considered. It is difficult to consider multiple and simultaneous failures.
Cause-consequence analysis	Extremely flexible and all-encompassing methodology. Well documented. Sequential paths for critical events are clearly shown.	Cause-consequence diagrams become too large very quickly (as FTA, ETA). They have many of the disadvantages of fault tree analysis.

Problems with software safety analysis

- relatively new field
- lack of systematic engineering discipline
- no agreed or proven methodologies
- time and cost
- complexity
(understanding of the problem domain, separation of knowledge)
- discrete nature of software
(difficulties with large discrete state spaces)
- real-time aspects
(concurrency and synchronization)
- (partially) invalid assumption of independent failures

Dependable Systems

Part 6: System aspects of dependable computers

Contents

- System design considerations
- Fault-tolerance: systematic vs. application-specific
- The problem of *Replica Determinism*
- Services for replicated fault-tolerant systems
 - Basic Services
 - Clock Synchronization Services
 - Communication Services
 - Replica Control Services

System design considerations

System design considerations

Fault-tolerance is not the only means for dependability.

To achieve given dependability goals the following aspects need consideration (usually in the order given here):

- perfection
 - maintenance
 - fault-tolerance
 - systematic fault-tolerance
 - application-specific fault-tolerance
- low
↓ design complexity
high

Perfection vs. fault-tolerance

Perfection is easier than fault-tolerance:

- if it is possible to attain a given dependability goal by means of perfection then use perfection in favor of fault-tolerance
- perfection leads to conceptual simpler systems
- lower probability of design faults
- does not require error detection, damage confinement and assessment, error recovery and fault treatment to tolerate faults
- steady reliability improvement of hardware components supports perfection

But, perfection is limited:

- perfection is limited by the dependability of individual components
- very high dependability goals can only be reached by maintenance or by fault-tolerant systems

Maintenance vs. fault-tolerance

Maintenance is easier than fault-tolerance:

- if it is possible to attain a given dependability goal (availability) by means of maintenance then use maintenance in favor of fault-tolerance
- maintenance adds to system complexity, but is still considerable simpler than fault-tolerance
- maintenance has lower probability of design faults than FT
- maintenance requires error detection and damage confinement, but no error recovery and fault treatment at system level
- there is also trade off between maintenance and reliability (connector vs. solder joint), i.e., some maintenance measures may reduce reliability

But, maintenance is limited:

- maintenance is limited by the dependability of individual components
- applicability of maintenance is limited (cf. next slide)

Limitations of maintenance

- maintenance (without fault-tolerance) is only applicable if system down times are permitted
 - fail-stop or fail-safe systems allow down times:
(train signaling, anti-lock braking system, ...)
 - fail-operational systems do not allow down times:
(fly-by-wire, reactor safety system, ...)
- only restricted reliability and safety improvements by preventive maintenance
- preventive maintenance is only reasonable if:
 - replacement units have constant or increasing failure rate
 - infant mortality is well controlled and failure rates are sufficiently low

The maintenance procedure

The maintenance procedure consists of the following phases:

- error detection
- call for maintenance
- maintenance personnel arrival
- diagnosis
- supply of spare parts
- replacement of defect components
- system test
- system re-initialization
- resynchronization with environment

Aspects of maintenance

- maintenance costs vs. system costs
- error latency period, error propagation and error diagnosis
- maintenance personnel
(number, education, equipment, location, service hours, etc.)
- spare part supply, stock or shipment
- Maintainability of a system depends on the:
 - quality and availability of documentation
 - including test plans
 - design of the system structure with maintenance in mind
 - implementation of appropriate error messages
 - size and interconnection of replacement units
 - accessibility of replacement units
 - mechanical stability of replacement units

Determining factors for SRU size

The size of the smallest replaceable unit (SRU) is determined by the following factors:

<i>factor (increases)</i>	<i>SRU size</i>
qualification of service personnel	decreases
effort for diagnosis	decreases
cost of SRU	increases
spare part costs ¹	increases
maintainability	increases
maintenance duration	decreases

¹Cost for parts which are used to construct SRU's

Diagnosis support for maintenance

- diagnosis support is very important and therefore needs to be considered during system design
- self diagnosis with meaningful messages
- needs to completely cover the error domain
- maintenance documentation:
 - symptom → cause and affected SRU
 - error symptom/cause matrix indicates for each symptom all possible SRU's that may cause the symptom
 - sparse matrices indicate good diagnosability
- expert system support for diagnosis
- duration of diagnosis is important for MTTR

Fault-tolerance: systematic vs. application-specific

Application-specific fault-tolerance

- the computer system interacts with some physical process, the behavior of the process is constrained by the law of physics
- these laws are implemented by the computer system to check its state for reasonableness
- for example:
 - the acceleration/deceleration rate of an engine is constrained by the mass and the momentum that affects the axle
 - signal range checks for analog input signals
- reasonableness checks are based on application knowledge
- fail-stop behavior can be implemented based on reasonableness checks

Application-specific fault-tolerance

- the laws of physics constraining the process can be used to perform state estimations in case some component has failed
- for example:
 - if the engine temperature sensor fails, a simple state estimation could assume a default value
 - a better state estimation can be based on the ambient temperature of the engine, engine load and thermostatic behavior of the engine
- the speed of a vehicle can be estimated if the engine speed and the transmission ratio is known
- state estimations are based on application knowledge
- fail-operational behavior can be implemented based on reasonableness checks and state estimations

Systematic fault-tolerance

- does not use application knowledge, makes no assumptions on the physical process or controlled object
- uses replicated components instead
- replicas must be designed to deliver corresponding results in the absence of faults
- if among a set of replicated components, some—but not all—fail then there will be divergence among replicas
- information on divergence is used for fault detection

- The problem of replica determinism: due to the limited accuracy of any sensor that maps continuous quantities onto computer representable discrete numbers it is impossible to avoid nondeterministic behavior.

Systematic fault-tolerance (cont.)

- systematic fault-tolerance requires agreement protocols due to replica nondeterminism
- the agreement protocol has to guarantee that correct replicas return corresponding results
- fail-stop behavior can be implemented by using the information of divergent results, i.e., when replicas diverge then the system stops
- fail-operational behavior can be implemented by using redundant components, i.e., NooM: “N-out-of-M” replicas provide corresponding results (e.g., TMR – 2oo3)

Comparison of fault-tolerance techniques

Systematic fault-tolerance	Application-specific fault-tolerance
<ul style="list-style-type: none">• replication of components• divergence among replicas in case of faults• no reasonableness checks necessary• requires replica determinism• no application knowledge necessary• exact distinction between correct and faulty behavior	<ul style="list-style-type: none">• no replication necessary• —• reasonableness checks for fault detection• —• depends on application knowledge• fault detection is limited by a <i>gray zone</i>

Comparison of fault-tolerance techniques (cont.)

Systematic fault-tolerance

- no state estimations necessary
- independence of application areas
- service quality is independent of whether replicated components are faulty or not
- correct system function depends on the number of correct replicas and their failure semantics
- only backward recovery

Application-specific fault-tolerance

- state estimations for continued service
- missing or insufficient reasonableness checks for some application areas
- quality of state estimations is lower than quality delivered during normal operation
- correct system function depends on the severity of faults and on the capability of reasonableness checks and state estimations
- forward and backward recovery

Comparison of fault-tolerance techniques (cont.)

Systematic fault-tolerance

- additional costs for replicated components (if no system inherent replication is available)
- no increase in application complexity
- considerable increase of system level complexity
- separation of fault-tolerance and application functionality
- fault-tolerance can be handled transparently to the application

Application-specific fault-tolerance

- no additional costs for replicated components
- considerable increase in application complexity
- no increase of system level complexity
- application and fault-tolerance are closely intertwined
- — // —

Systematic *and* application-specific fault-tolerance

- under practical conditions there will be a compromise between systematic and application-specific fault-tolerance
- usually cost, safety and reliability are the determining factors to choose a proper compromise
- software complexity plays an important role:
 - for complex systems software is almost unmanageable without adding fault-tolerance (fault containment regions and software robustness)
 - therefore systematic fault-tolerance should be applied in favor of application-specific fault-tolerance to reduce the software complexity
 - systematic fault-tolerance allows to test and to validate the mechanisms independently of the application software (divide and conquer)

The problem of *Replica Determinism*

The problem of *Replica Determinism*

- For systematic fault-tolerance it is necessary that replicated components show consistent or deterministic behavior in the absence of faults.
- If for example two active redundant components are working in parallel, both have to deliver corresponding results at corresponding points in time.
- This requirement is fundamental to differentiate between correct and faulty behavior.
- At a first glance it seems trivial to fulfill replica determinism since computer systems are assumed to be examples of deterministic behavior, but in the following it is shown that computer systems behave only almost deterministically.

Nondeterministic behavior

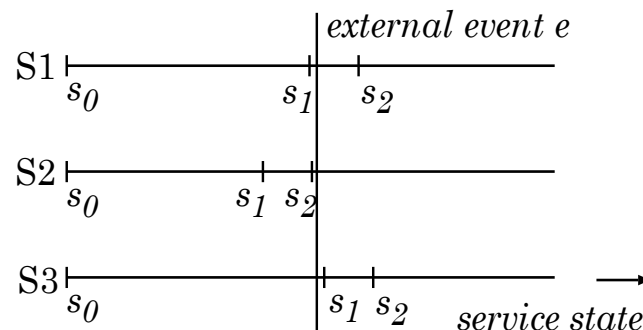
- **Inconsistent inputs:**

If inconsistent input values are presented to the replicas then the results may be inconsistent too.

- a typical example is the reading of replicated analogue sensor
 $\text{read}(S1) = 99.99 \text{ }^\circ\text{C}$, $\text{read}(S2) = 100.00 \text{ }^\circ\text{C}$

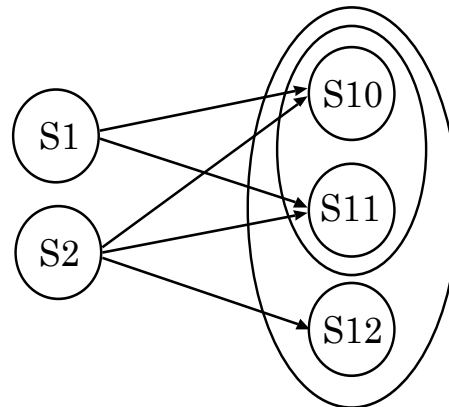
- **Inconsistent order:**

If service requests are presented to replicas in different order then the results will be inconsistent.



Nondeterministic behavior (cont.)

- Inconsistent membership information:
Replicas may fail or leave groups voluntarily or new replicas may join a group.
If replicas have inconsistent views about group membership it may happen that the results of individual replicas will differ.



Nondeterministic behavior (cont.)

- **Nondeterministic program constructs:**

Besides intentional nondeterminism, like random number generators, some programming languages have nondeterministic program constructs for communication and synchronization (Ada, OCCAM, and FTCC).

```
task server is  
  entry service_1();  
  ...  
  entry service_n();  
end server;
```

```
task body server is  
begin  
  select  
    accept service_1() do  
      action_1();  
    end;  
  ...  
  or  
    accept service_n() do  
      action_n();  
    end;  
  end select;  
end server;
```

Nondeterministic behavior (cont.)

- **Local information:**

If decisions with a replica are based on local knowledge (information which is not available to other replicas) then the replicas will return different results.

- system or CPU load
- local time

- **Timeouts:**

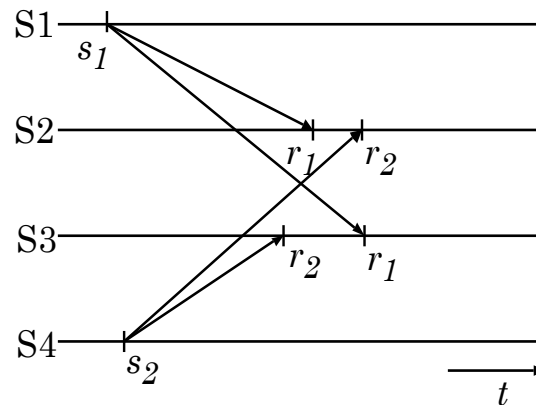
Due to minimal processing speed differences or due to slight clock drifts it may happen that some replicas locally decide to timeout while others do not.

Nondeterministic behavior (cont.)

- **Dynamic scheduling decisions:**
Dynamic scheduling decides in which order a series of service requests are executed on one or more processors. This may cause inconsistent order due to:
 - non-identical sets of service requests
 - minimal processing speed differences

Nondeterministic behavior (cont.)

- **Message transmission delays:**
Variabilities in the message transmission delays can lead to different message arrival orders at different servers (for point-to-point communication topologies or topologies with routing).



The consistent comparison problem:

- computers can only represent finite sets of numbers
- it is therefore impossible to represent the real numbers exactly, they are rather approximated by equivalency classes
- if the results of arithmetic calculations are very close to the border of equivalency classes, different implementations can return diverging results
- different implementations are caused by: N-version programming, different hardware, different floating point libraries, different compilers
- for example the calculation of $(a - b)^2$ with floating point representation with a mantissa of 4 decimal digits and rounding where $a = 100$ and $b = 0.005$ gives different result for mathematical equivalent formulas.

$$(a - b)^2 = 1.000 \cdot 10^4$$

$$(a - b)^2 = a^2 - 2ab + b^2 = 9.999 \cdot 10^3$$

Fundamental limitations to replication

The real world abstraction limitation:

- dependable computer systems usually interface with continuous real-world quantities:

quantity	SI-unit
distance	meter [m]
mass	kilogram [kg]
time	second [s]
electrical current	ampere [A]

- these continuous quantities have to be abstracted (or represented) by finite sets of discrete numbers
- due to the finite accuracy of any interface device, different discrete representations will be selected by different replicas

Fundamental limitations to replication (cont.)

The impossibility of exact agreement:

- due to the real world abstraction limitation it is impossible to avoid the introduction of replica non-determinism at the interface level
- but it is also impossible to avoid the once introduced replica nondeterminism by agreement protocols completely
- exact agreement would require ideal simultaneous actions, but in the best case actions can be only simultaneous within a time interval d

Fundamental limitations to replication (cont.)

Intention and missing coordination:

- replica nondeterminism can be introduced intentionally
- or unintentionally by omitting some necessary coordinating actions

Replica control

- Due to these fundamental limitations to replication it is **necessary** to enforce replica determinism which is called replica control.

Internal vs. external replica control

Internal replica control:

- avoid nondeterministic program constructs, uncoordinated timeouts, dynamic scheduling decisions, diverse program implementations, local information, and uncoordinated time services
- can only be enforced partially due to the fundamental limitations to replication

External replica control:

- control nondeterminism of sensor inputs
- avoid nondeterminism introduced by the communication service
- control nondeterminism introduced by the program execution on the replicated processors by exchanging information

Def.: Replica Determinism

Correct replicas show *correspondence* of service outputs and/or service states under the assumption that all servers within a group start in the same initial state, executing *corresponding* service requests *within a given time interval*.

- this generic definition covers a broad range of systems
- *correspondence* and *within a given time interval* needs to be defined according to the application semantics

Groups, resiliency and replication level

- Replicated entities such as processors are called groups.
- The number of replicas in a group is called replication level.
- A group is said to be n -resilient if up to n processor failures can be tolerated.

Services for replicated fault-tolerant systems

- Basic Services
- Clock Synchronization Services
- Communication Services
- Replica Control Services

Services for replicated fault-tolerant systems

- **Basic Services**
- Clock Synchronization Services
- Communication Services
- Replica Control Services

Basic services for replicated fault-tolerant systems

- **Membership:**

Every non-faulty processor within a group has timely and consistent information on the set of functioning processors which constitute the group.

- **Agreement:**

Every non-faulty processor in a group receives the same service requests within a given time interval.

- **Order:**

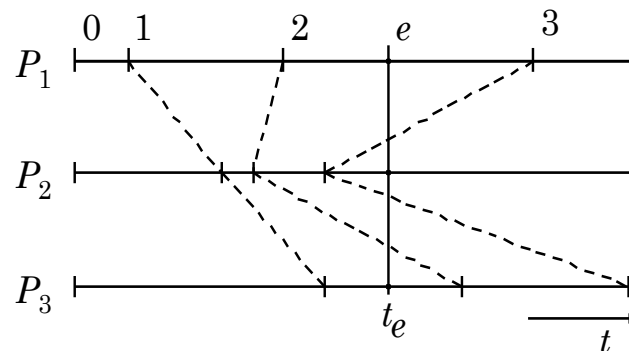
Explicit service requests as well as implicit service requests, which are introduced by the passage of time, are processed by non-faulty processors of a group in the same order.

Services for replicated fault-tolerant systems

- Basic Services
- **Clock Synchronization Services**
- Communication Services
- Replica Control Services

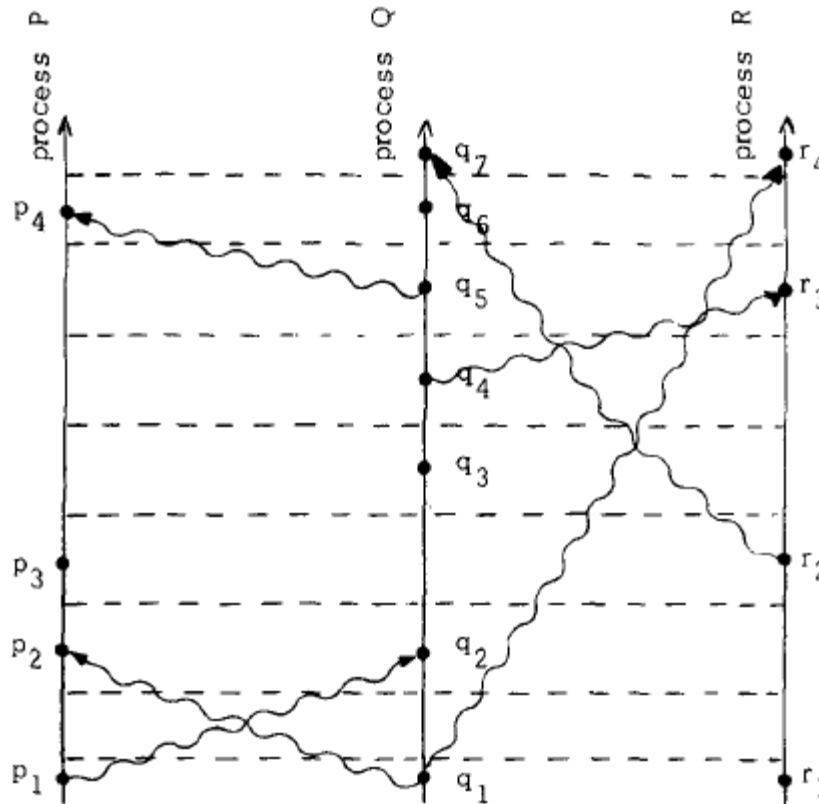
Logical Clocks

- all members in a group observe the same events in the same order
- this applies to process internal events and external events such as service requests and faults
- external events need to be reordered according to the internal precedence relation and individual processing speeds



Logical Clocks

We want to define a “happened before” relation between events in the distributed system (\rightarrow) that defines a partial order of events and captures **potential causality**, but excludes external clandestine channels.



Lamport, Leslie. "Time, clocks, and the ordering of events in a distributed system." *Communications of the ACM* 21, no. 7 (1978): 558-565.

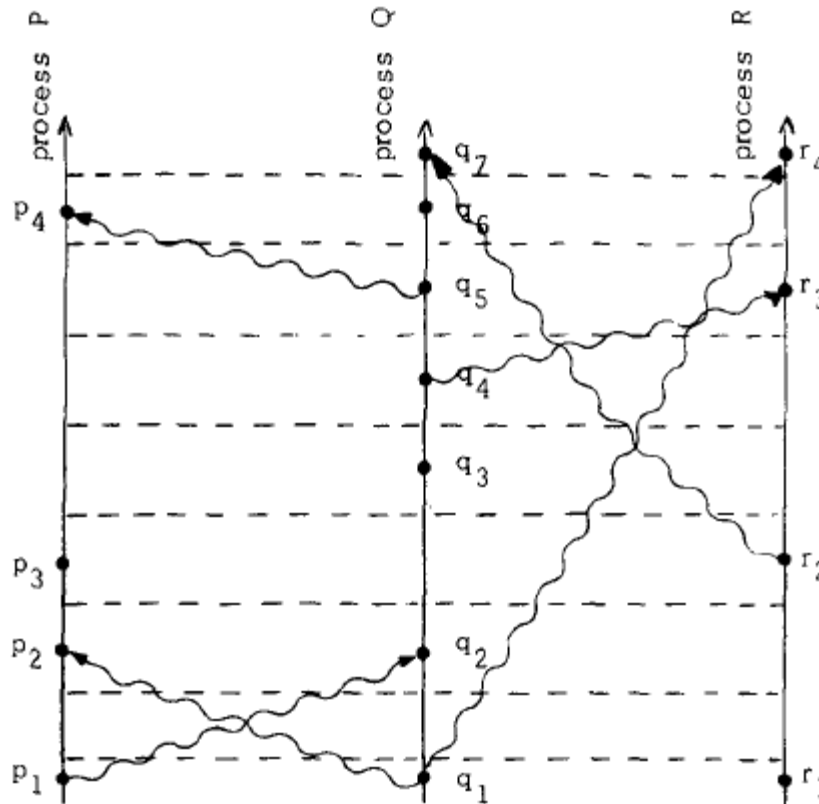
Logical Clocks

Can you find examples $p_x \rightarrow r_y$ in the figure?

Def.: The relation \rightarrow on a set of events in a distributed system is the smallest relation satisfying the following three relations:

1. If a and b are events performed by the same process, and a is performed before b then $a \rightarrow b$.
2. If a is the event of sending of a message by one process and b the receiving of the same message by another process, then $a \rightarrow b$.
3. Transitivity: if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

Two distinctive events are said to be concurrent if neither $a \rightarrow b$ nor $b \rightarrow a$.



Lamport, Leslie. "Time, clocks, and the ordering of events in a distributed system." *Communications of the ACM* 21, no. 7 (1978): 558-565.

Logical Clocks (cont.)

Logical Clocks implement a distributed algorithm over local variables (i.e., the logical clocks LC) that satisfies the following clock condition:

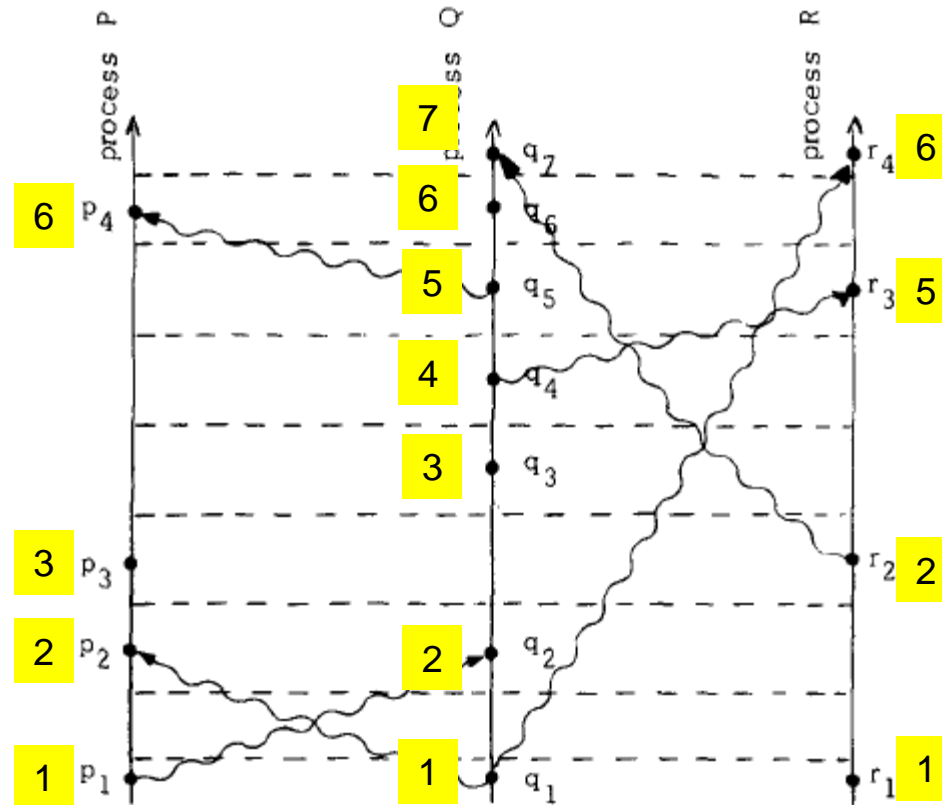
Clock Condition: $a \rightarrow b$ then $LC(a) < LC(b)$
(we cannot expect the converse condition to hold as well)

An algorithm implementing logical clocks (i.e. satisfying the Clock Condition):

1. Each process P_i increments LC_i between two successive events.
2. If event a is the sending of a message m by process P_i , then the message m contains a timestamp $T_m = LC_i(a)$. Upon receiving a message m , process P_j sets LC_j greater than or equal to its present value and greater than T_m .

Example

1. Each process P (and Q,R) increments its LC between two successive events
2. If event a is the sending of a message m by process P, then the message m contains a timestamp $T_m = LC_P(a)$. Upon receiving a message m, process Q sets LC_Q greater than or equal to its present value and greater than T_m .



Lamport, Leslie. "Time, clocks, and the ordering of events in a distributed system." *Communications of the ACM* 21, no. 7 (1978): 558-565.

Logical Clocks (cont.)

- The algorithm defines no total order since independent processes may use the same timestamp for different events.
- A possible solution is to break ties by using a lexicographical process order.
- Logical clocks have no gap-detection property.
- **Gap-detection:**
Given a local process with local clock LC and given two events e and e' with clock values $LC(e) < LC(e')$ (and only this information) and let's further assume that we record all events and their timestamps on LC. Then, when looking at this list determine whether some other event e'' is missing in this list such that $LC(e) < LC(e'') < LC(e')$.

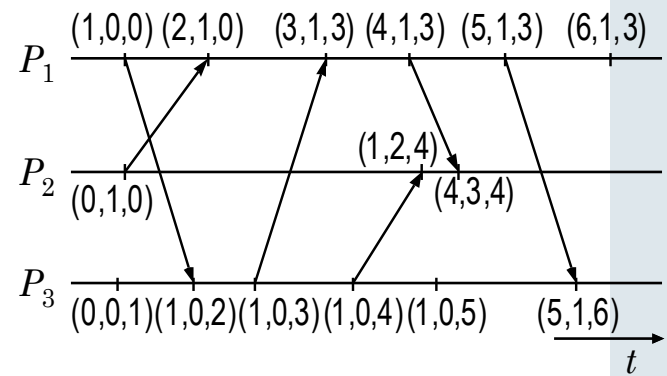
Logical Clocks (cont.)

- The gap-detection property is necessary for stability and a bounded action delay, i.e., before an action is taken it has to be guaranteed that no earlier messages are delivered
- Stability and action delay are based on potential causality, two events e and e' are potential causal related if $e \rightarrow e'$.
- Vector clocks are an extension of logical clocks which have gap-detection property.

Vector Clocks

- vector clocks are an extension of logical clocks which have gap-detection property
- An algorithm implementing vector clocks:

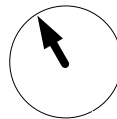
- Each process P_i increments VC_i between two successive events
- Upon receiving a message m , a process P_j sets all vector elements VC_j to the maximum of VC_j and T_m , where T_m is message m 's vector clock timestamp. Afterwards the element $VC_j[j]$ is incremented.



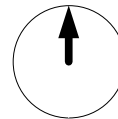
- Potential causality for vector clocks $e \rightarrow e' \equiv VC(e) < VC(e')$
 $-VC < VC' \equiv (VC \circ VC') \wedge (\forall i:1 \checkmark i \checkmark n: VC[i] \checkmark VC'[i])$

Real-Time Clocks

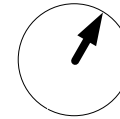
*In an ensemble of clocks, the **precision Π** is defined as the maximum distance between any two synchronized non-faulty clocks at any point in real time.*



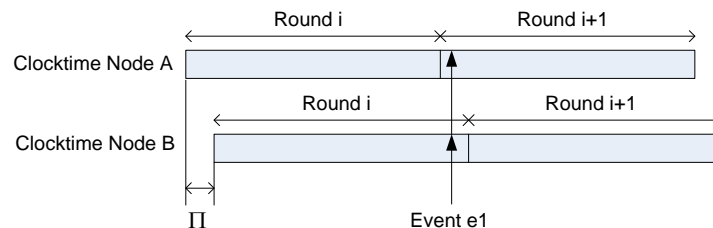
Late Clock



Perfect Clock

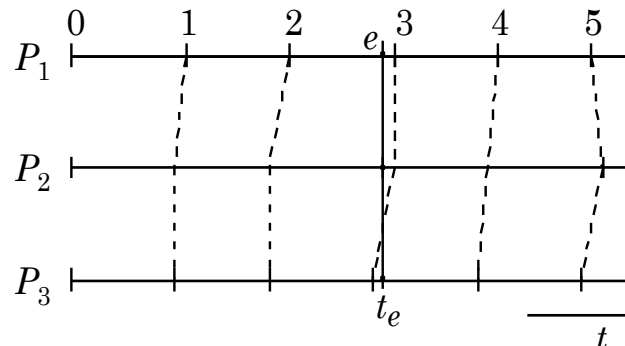


Early Clock



Real-Time Clocks

- all processors have access to a central real-time clock or
- all processors have local real-time clocks which are approximately synchronized
- the synchronized clocks define a global time grid where individual clocks are off at most by one tick at any time instant t
- the maximum deviation among clocks is called precision
- t -precedent events (events that are at least t real-time steps apart) can be causally related regardless of clandestine channels



Comparing Real-Time and Logical Clocks

real-time clocks	logical clocks
synchronous system model	asynchronous system model
higher synchronization overhead	little delays and synchronization overhead if only system internal events are considered
needs to achieve consensus on the systematic clock error of one tick	external events need to be reordered in accordance to logical time
stability within one clock tick	unbounded duration for stability, requires consistent cut or vector clock
potential causality for t -precedent external events	potential causality only for closed systems
bounded action delay (total order)	unbounded action delay (no total order)

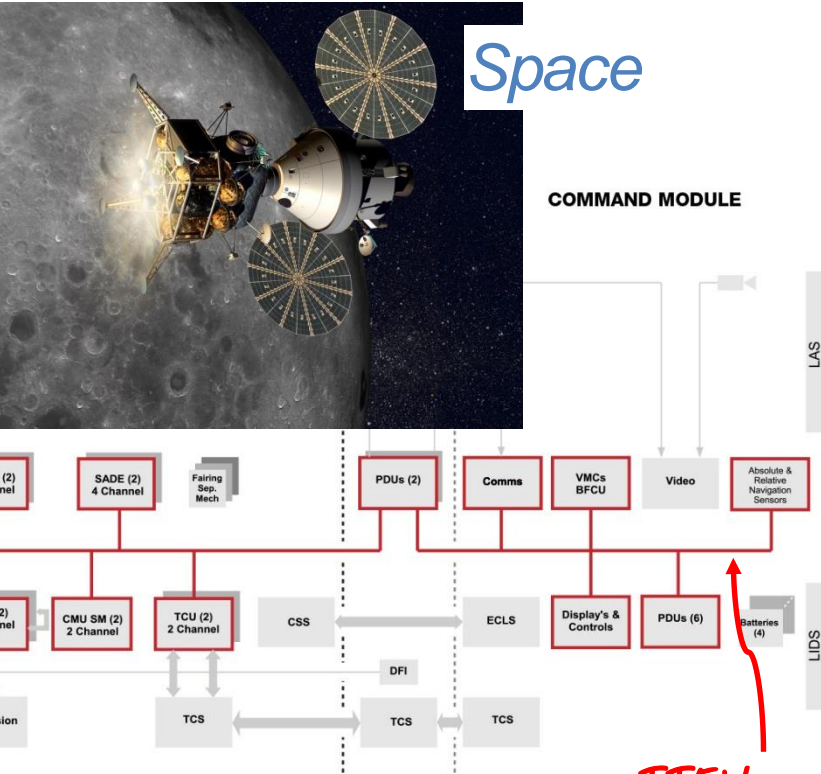
Services for replicated fault-tolerant systems

- Basic Services
- Clock Synchronization Services
- **Communication Services**
- Replica Control Services

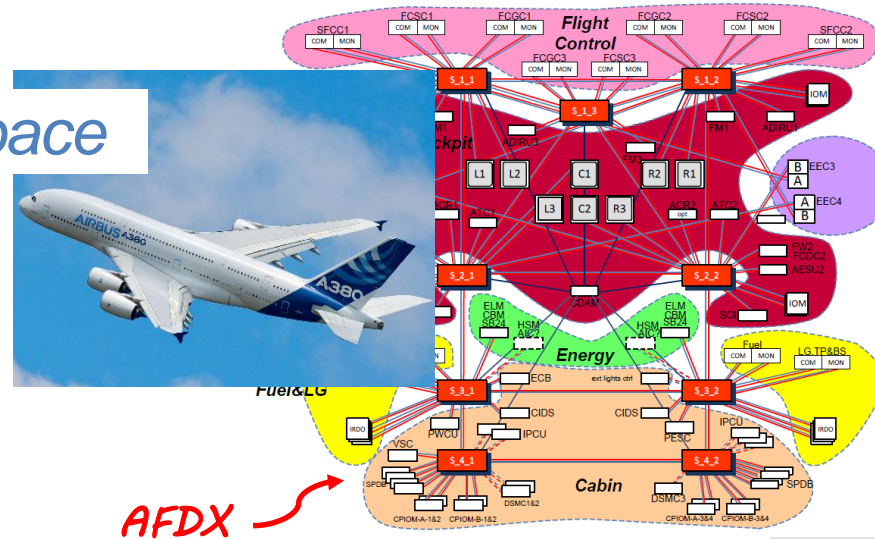
Communication Services

The following arguments motivate the close interdependence of fault-tolerant computer systems, communication and replica control:

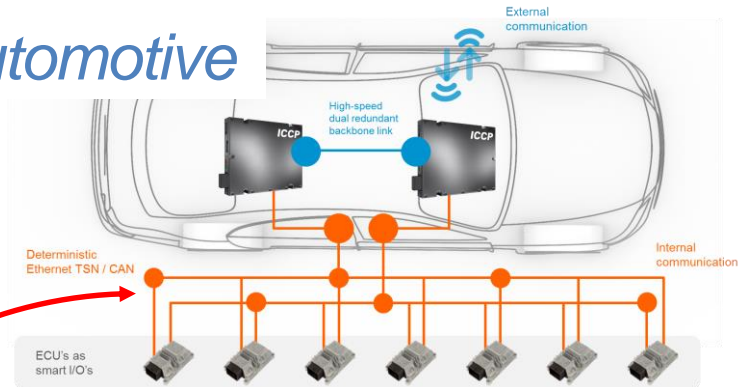
- fault-tolerant systems are built on the assumption that individual components fail independently
- this assumptions requires the physical and electrical isolation of components at the hardware level
- these properties are best fulfilled by a distributed computer system where nodes are communicating through message passing but have no shared resources except for the communication media
- furthermore it has to be guaranteed that faulty nodes are not able to disturb the communication of correct nodes and that faulty nodes are not allowed to contaminate the system



Aerospace



Automotive



... and many more ...

Time-Sensitive Networking

Requirements for networks for dependable cyber-physical systems:

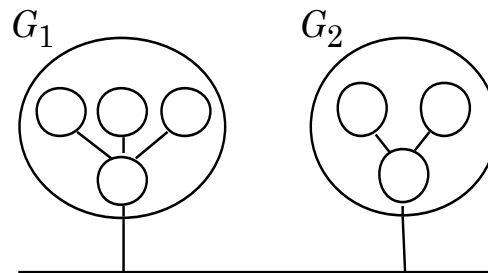
- Real-Time message deliver (often $< 1\text{ms}$)
- High Dependability (Safety, Reliability)
 - Failure probability as low as 10^{-9} (sometimes even less)
 - No single point of failure (sometimes even no dual)
- Mixed time- and safety-criticality
 - Share a single network between different application types.
- High Performance
 - Aerospace is at 100Mbit/sec, will transit to 1Gbit/sec
 - Automotive has a need for $> 1\text{Gbit/sec}$ already (bc. AD)
- Security (rather new requirement)

Services for replicated fault-tolerant systems

- Basic Services
- Clock Synchronization Services
- Communication Services
- **Replica Control Services**

Central Replica Control

- **Strictly central replica control principle:**
 - there is one distinguished processor within a group called leader or central processor
 - the leader takes all nondeterministic decisions
 - the remaining processors in the group, called followers, take over the leaders decisions



Central Replica Control (cont.)

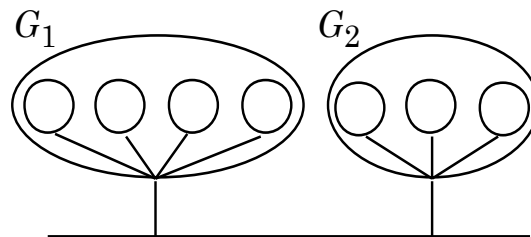
- Strictly central replica control requires a communication service assuring reliable broad- or multicast.
- **Reliable broadcast:** A distinguished processor, called the transmitter, sends its local service request to all other processors in the group, fulfilling the following properties:
 - **Consistency:** All correct processors agree on the same value and all decisions are final.
 - **Non-triviality:** If the transmitter is non faulty, all correct processors agree on the input value sent by the transmitter.
 - **Termination:** Each correct processor decides on a value within a finite time interval .

Central Replica Control (cont.)

- Failures and Replication
 - semi-active and passive replication
 - the leading processor is required to be fail restrained
 - byzantine or performance failures of the leader cannot be detected by other processors in the group (“heartbeat” or “I am alive” messages)
 - to tolerate t failures with crash or omission semantics $t + 1$ processors are necessary

Distributed Replica Control

- **Strictly distributed replica control principle:**
 - there is no leader role, each processor in the group performs exactly the same way
 - to guarantee replica determinism the group members have to carry out a consensus protocol on nondeterministic decisions



Distributed Replica Control (cont.)

- Any (partially) distributed replica control strategy requires a communication service assuring consensus.
- Consensus: Each processor starts a protocol with its local input value, which is sent to all other processors in the group, fulfilling the following properties:
 - Consistency: All correct processors agree on the same value and all decisions are final.
 - Non-triviality: The agreed-upon input value must have been some processors input (or is a function of the individual input values).
 - Termination: Each correct processor decides on a value within a finite time interval.

Distributed Replica Control (cont.)

Failures and Replication

- active replication
- no restricted failure semantics of processors
- to tolerate t crash or omission failures $t + 1$ processors are necessary
- to tolerate t performance failures $2t + 1$ processors are necessary
 - e.g., if a faulty message is too early $2t$ would be insufficient to identify the correct timing
- to tolerate t byzantine failures $3t + 1$ processors are necessary
- for crash or omission failures it is sufficient to take 1 processor result

Replica Control Strategies

Active replication:

- all processors in the group are carrying out the same service requests in parallel
- strictly distributed approach, nondeterministic decisions need to be resolved by means of an agreement protocol
- the communication media is the only shared resource
- **Advantages:**
 - unrestricted failure semantics
 - no single point of failure
- **Disadvantages:**
 - requires the highest degree of replica control
 - high communication effort for consensus protocols
 - problems with dynamic scheduling decisions and timeouts

Replica Control Strategies (cont.)

Semi-active replication:

- intermediate approach between distributed and centralized
- the leader takes all nondeterministic decisions
- the followers are executing in parallel until a potential nondeterministic decision point is reached
- **Advantages:**
 - no need to carry out a consensus protocol
 - lower complexity of the communication protocol (compared to active replication)
- **Disadvantages:**
 - restricted failure semantics, the leader's decisions are single points of failures
 - problems with dynamic scheduling decisions and timeouts

Replica Control Strategies (cont.)

Passive replication:

- only one processor in the group – called primary – is active
- the other processors in the group are in standby
- checkpointing to store last correct service state and pending service requests
- **Advantages:**
 - requires the least processing resources
 - standby processors can perform additional tasks
 - highest reliability of all strategies (if assumption coverage = 1)
- **Disadvantages:**
 - restricted failure semantics (crash or fail-stop)
 - long resynchronization delay

Replica Control Strategies (cont.)

Lock-step execution:

- processors are executing synchronized
- the outputs of processors are compared after each single operation
- typically implemented at the hardware level with identical processors
- **Advantages:**
 - arbitrary software can be used without modifications for fault-tolerance (important for commercial systems)
- **Disadvantages:**
 - common clock is single point of failure
 - transient faults can affect all processors at the same point in the computation
 - high clock speed limits number and distance of processors
 - restricted failure semantics

Dependable Systems

Part 7: System Aspects of dependable computers
(cont.)

Contents

- Consensus
- Interactive Consistency Algorithms
- Broadcast Properties and Algorithms
- Checkpointing
- Stable Storage
- Diagnosis
- Fault-Tolerant Software

Consensus

Consensus

- Each processor starts a protocol with its local input value, which is sent to all other processors in the group, fulfilling the following properties:
 - Consistency: All correct processors agree on the same value and all decisions are final.
 - Non-triviality: The agreed-upon input value must have been some processors input (or is a function of the individual input values).
 - Termination: Each correct processor decides on a value within a finite time interval.

Consensus (cont.)

- The consensus problem under the assumption of byzantine failures was first defined in 1980 in the context of the SIFT project which was aimed at building a computer system with ultra-high dependability. Other names are
 - byzantine agreement or byzantine general problem
 - interactive consistency

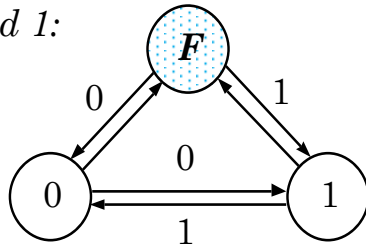
Impossibility of deterministic consensus in asynch. systems

- asynchronous systems cannot achieve consensus by a deterministic algorithm in the presence of even one crash failure of a processor
- it is impossible to differentiate between a late response and a processor crash
- by using coin flips, probabilistic consensus protocols can achieve consensus in a constant expected number of rounds
- failure detectors which suspect late processors to be crashed can also be used to achieve consensus in asynchronous systems

Byzantine Failure Behaviour

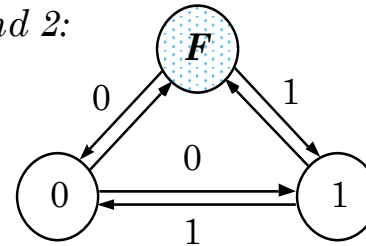
$n \geq 3t + 1$ processors are necessary to tolerate t failures

Round 1:



$Maj(0, 0, 1) = 0$ $Maj(0, 1, 1) = 1$

Round 2:



$Maj(0, 0, 1) = 0$ $Maj(0, 1, 1) = 1$

■ Situation:

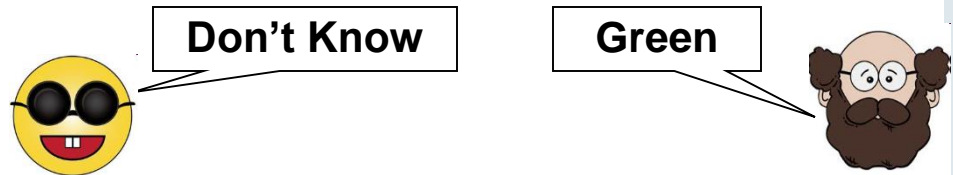
What is the color of the house?



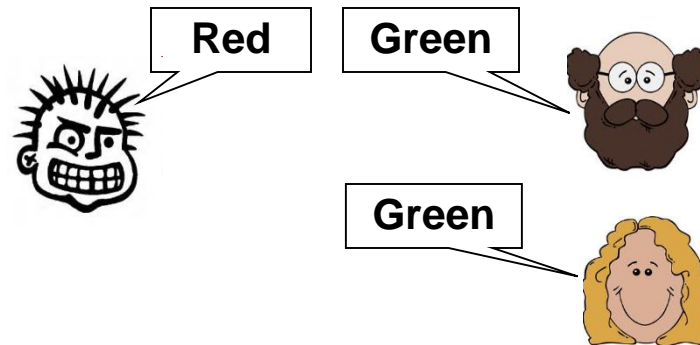
No Failure



Fail-Silence Failure



Fail-Consistent Failure



Situation:

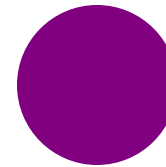
What is the color of the house?



Static Situation – one Truth

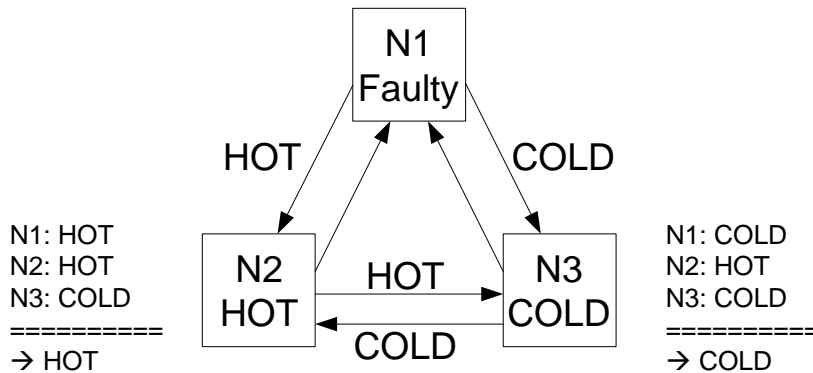
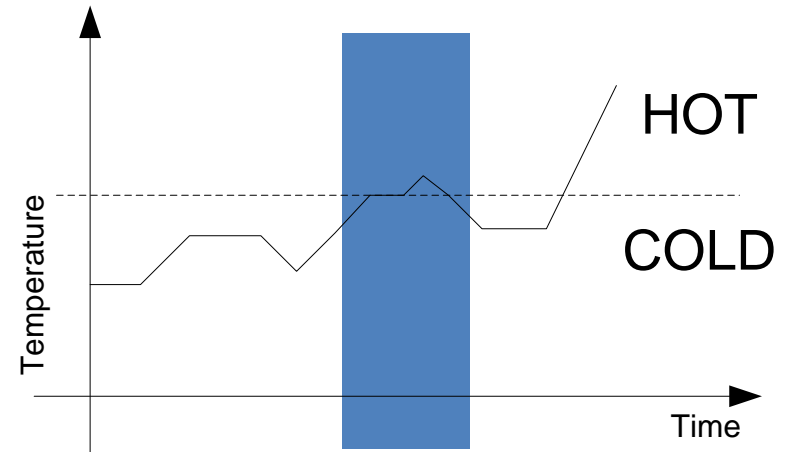
Situation:

What is the color of the ball ?



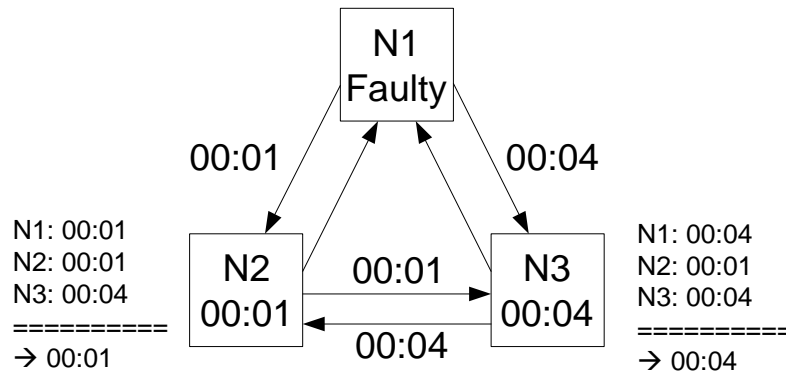
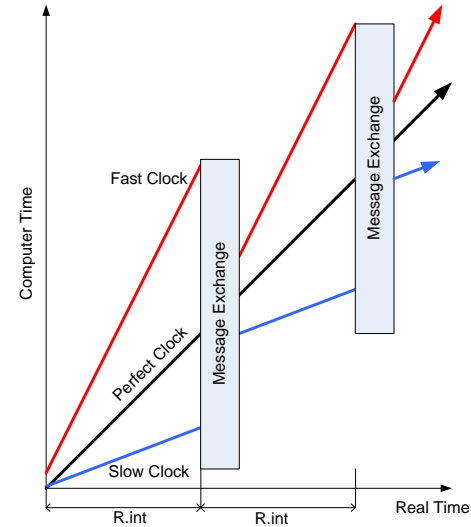
Dynamic Situation – >one Truth

A distributed system that measures the temperature of a vessel shall raise an alarm when the temperature exceeds a certain threshold.
The system shall tolerate the arbitrary failure of one node.
How many nodes are required?
How many messages are required?



In general, three nodes are insufficient to tolerate the arbitrary failure of a single node.
The two correct nodes are not always able to agree on a value.
A decent body of scientific literature exists that address this problem of dependable systems, in particular dependable communication.

A distributed system in which all nodes are equipped with local clocks, all clocks shall become and remain synchronized.
The system shall tolerate the arbitrary failure of one node.
How many nodes are required?
How many messages are required?



In general, three nodes are insufficient to tolerate the arbitrary failure of a single node.
The two correct nodes are not always able to bring their clocks into close agreement.
A decent body of scientific literature exists that address this problem of fault-tolerant clock synchronization.

Interactive Consistency Algorithms

Assumptions about the message passing system

- A1: Every message that is sent by a processor is delivered correctly by the message passing system to the receiver.
- A2: The receiver of a message knows which node has sent a message.
- A3: The absence of messages can be detected.

Recursive Algorithm for $n \geq 3t + 1$

ICA(t):

1. The transmitter sends its value to all the other $n - 1$ processors.
2. Let v_i be the value that processor i receives from the transmitter, or else be the default value if it receives no value. Node i acts as the transmitter in algorithm *ICA*($t - 1$) to send the value to each other of the other $n - 2$ receivers.
3. For each processor i , let v_j be the value received from processor j ($j \neq i$) in step 2. Processor i uses the value *Majority*(v_1, \dots, v_{n-1}).

ICA(0):

1. The transmitter sends its value to all the other $n - 1$ processors.
2. Each processor uses the value it receives from the transmitter, or uses the default value, if it receives no value.

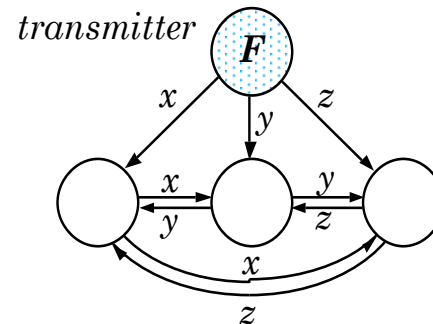
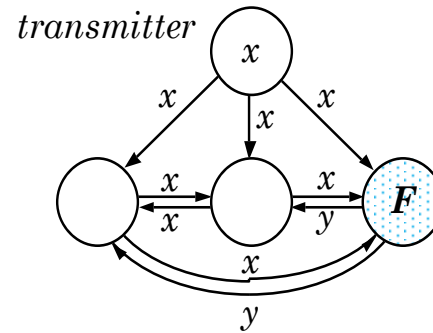
Example (n=4, t=1)

Case 1, one of the receivers is faulty:

- all correct processors decide x

Case 2, the transmitter is faulty:

- depending on the majority function all processors decide either x, y or z



Interactive consistency with signed messages

- if a processor sends x to some processor it appends its signature, denoted $x : i$
- when some processor receives this message and passes it further then $x : i : j$
- the algorithm for $n \geq t + 1$
- V_i is the set of all received messages which is initially $V_i = 0$
- The function $Choice(V_i)$ selects a default value if $V_i = 0$, it selects v if $V_i = \{v\}$ in other cases it could select a median or some other value.

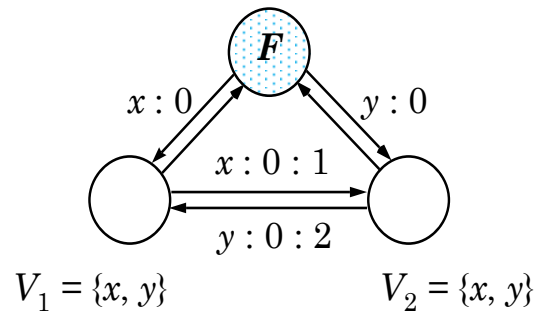
Interactive consistency with signed messages (cont.)

SM(t):

1. The transmitter signs its value and sends it to all other nodes
2. $\forall i$:
 - (A) If processor i receives a message of the form $v : 0$ from the transmitter then (i) it sets $V_i = \{v\}$, and (ii) it sends the message $v : 0 : i$ to every other processor.
 - (B) If processor i receives a message of the form $v : 0 : j_1 : j_2 : \dots : j_k$ and v is not in V_i , then (i) it adds v to V_i , and (ii) if $k < t$ it sends the message $v : 0 : j_1 : j_2 : \dots : j_k : i$ to every other node processor than j_1, j_2, \dots, j_k .
3. $\forall i$: when processor i receives no more messages, it considers the final value as $Choice(V_i)$.
 - The function $Choice(V_i)$ selects a default value if $V_i = 0$, it selects v if $V_i = \{v\}$ in other cases it could select a median or some other value.

Example ($n=3, t=2$)

- we again consider the case of the faulty transmitter:
- because of the signed messages it becomes clear that the transmitter is faulty

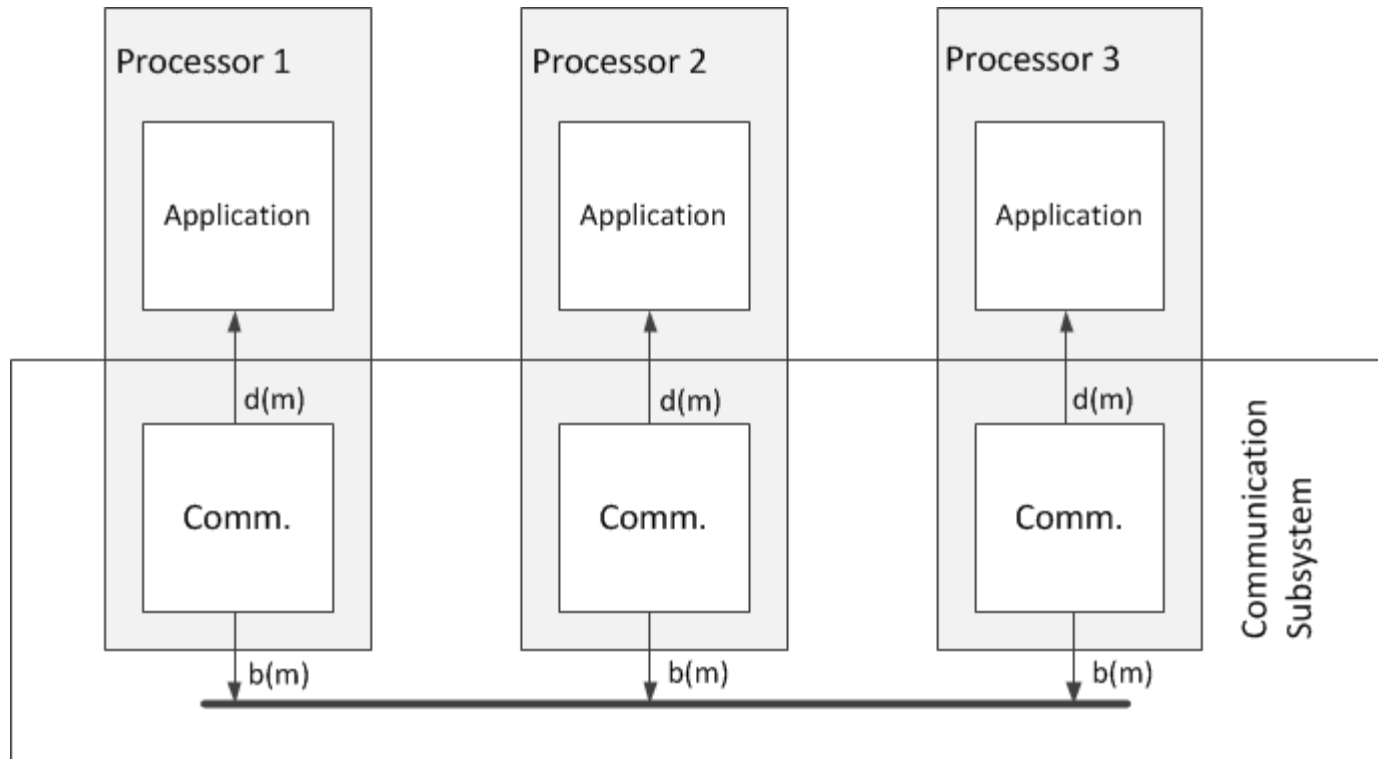


Complexity of consensus

- $ICA(t)$ and $SM(t)$ require $t + 1$ rounds of message exchange
- $t + 1$ rounds are optimal in the worst case, the lower bound for early stopping algorithms is $\min(f + 2, t + 1)$
- for $ICA(t)$ the number of messages is exponential in t , since $(n - 1)(n - 2) \dots (n - t - 1)$ are required $O(n^t)$, similarly the message complexity for $SM(t)$ is exponential
- the lower bound is $O(nt)$, for authentication detectable byzantine failures, performance or omission failures the lower bound is $O(n + t^2)$
- practical experience has shown that the complexity and resource requirements of consensus under a byzantine failure assumption are often prohibitive (up to 80% overhead for SIFT project)

Broadcast Algorithms

Terminology and Concepts



$d(m)$... deliver message m
 $b(m)$... broadcast message m

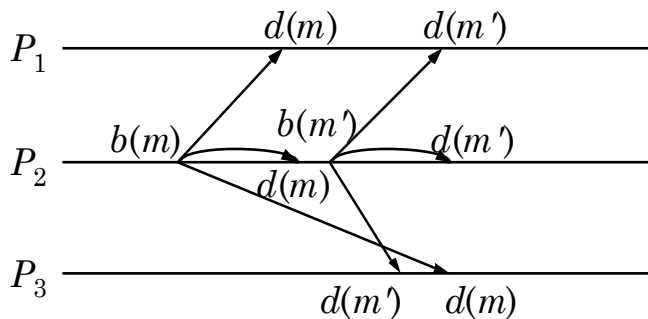
Reliable broadcast

- A distinguished processor, called the transmitter, sends its local service request to all other processors in the group, fulfilling the following properties:
 - **Consistency:** All correct processors agree on the same value and all decisions are final.
 - **Non-triviality:** If the transmitter is non faulty, all correct processors agree on the input value sent by the transmitter.
 - **Termination:** Each correct processor decides on a value within a finite time interval.
- Reliable broadcast is a building block for the solution of a broad class of problems in fault-tolerant computer systems
- Often there are additional requirements to reliable broadcast protocols (cf. next slides)

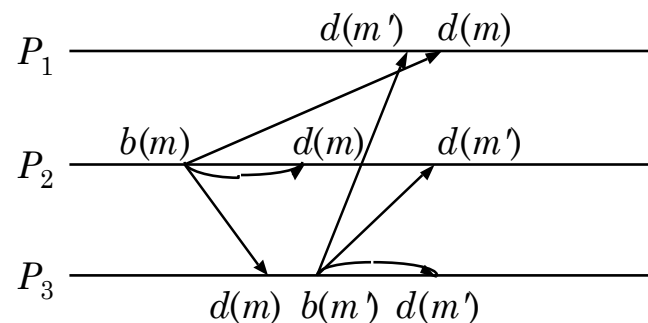
FIFO Broadcast

- FIFO Broadcast = Reliable Broadcast + FIFO order
- **FIFO Order:** If a process broadcasts m before **the same process** broadcasts m' , then no correct process delivers m' unless it has previously delivered m .

non-FIFO Broadcast



Problem with FIFO Broadcast

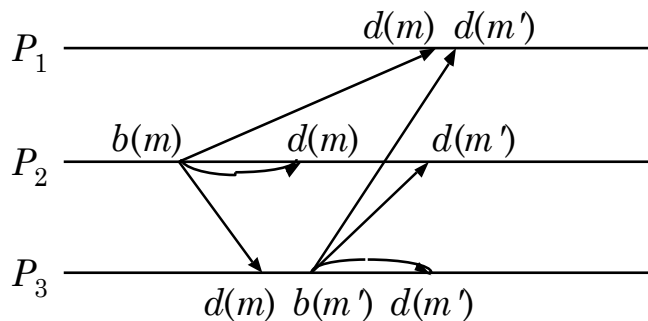


$b(m)$... broadcast message m $d(m)$... deliver message m

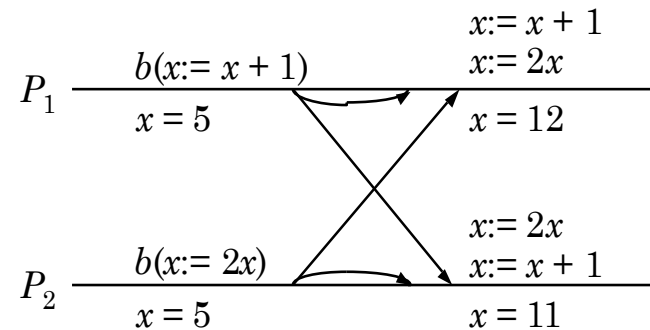
Causal Broadcast

- Causal Broadcast = Reliable Broadcast + Causal order
- (Potential) Causal Order: If the broadcast of m causally (\rightarrow) precedes the broadcast m' , then no correct process delivers m' unless it has previously delivered m .

Causal Broadcast



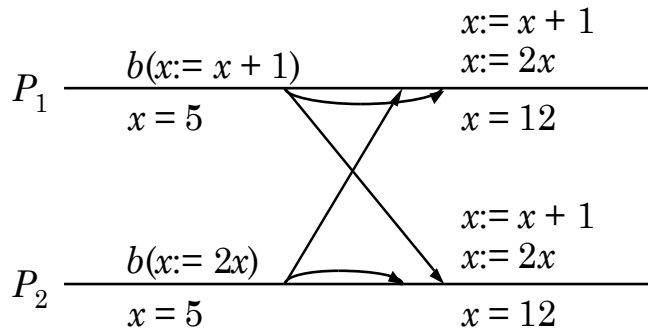
Problem with Causal Broadcast



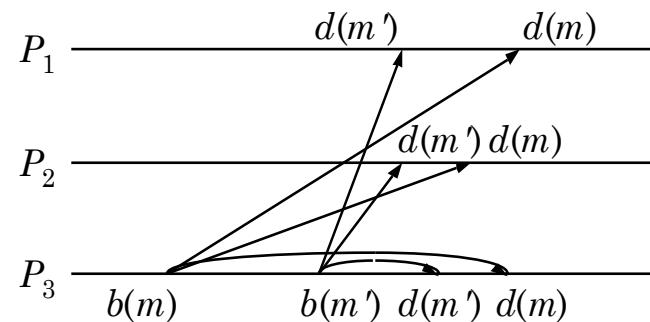
Atomic Broadcast

- Atomic Broadcast = Reliable Broadcast + Total order
- **Total Order:** If correct processes P_1 and P_2 deliver m and m' , then P_1 delivers m before m' if and only if P_2 delivers m before m' .

Atomic Broadcast



Atomic Broadcast is not FIFO

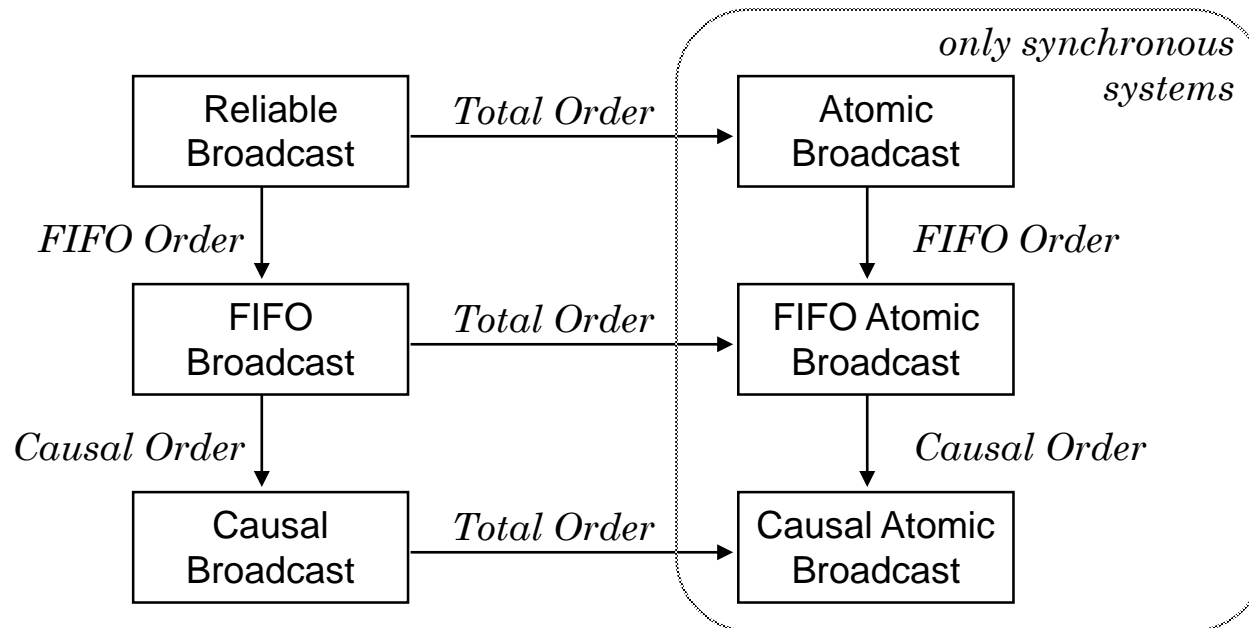


Extensions of Atomic Broadcast

FIFO Atomic Broadcast = Reliable Broadcast + FIFO Order + Total Order

Causal Atomic Broadcast = Reliable Broadcast + Causal Order + Total Order

Relationships among broadcast protocols:



Reliable broadcast protocol

- **Diffusion algorithm:** To R-broadcast m , a process p sends m to itself. When a process receives m for the first time it relays m to all its neighbors, and then R-delivers it.

broadcast(R, m):
send(m) to p

deliver(R, m):
upon receive(m) do
 if p has not previously executed deliver(R, m)
 then
 send(m) to all neighbors
 deliver(R, m)

- in synchronous systems the diffusion algorithm may be used as well, but it additionally guarantees real-time timeliness

Atomic Broadcast Protocols

- **Transformation:** any {Reliable, FIFO, Causal} Broadcast algorithm that satisfies real-time timeliness can be transformed to {Atomic, FIFO Atomic, Causal Atomic} Broadcast.

- **broadcast**(A^* , m):
broadcast(R^* , m)

- **deliver**(A^* , m):
upon deliver(R^* , m) do
schedule deliver(A^* , m) at time $TS(m) + \Delta$

- $TS(m)$ is the timestamp of message m
- the maximum delay for message transmission is Δ
- if two messages have the same timestamp then ties can be broken arbitrarily, e.g. by increasing sender id's

FIFO and Causal Broadcast

- **FIFO Transformation:** Reliable broadcast can be transformed to FIFO broadcast by using sequence numbers.
- **Causal Transformation:** All messages that are delivered between the last broadcast and this send operation are “piggy-packed” when sending a message.

–**broadcast**(C, m):

 broadcast(F, $\langle rcntDlvr \parallel m \rangle$)

$rcntDlvr := \perp$

–**deliver**(C, –):

 upon deliver(F, $\langle m_1, m_2, \dots, m_l \rangle$) do

 for $i := 1 \dots l$ do

 if p has not previously executed deliver(C, m_i)

 then

 deliver(C, m_i)

$rcntDlvr := rcntDlvr \parallel m_i$

- $rcntDlvr$ is the sequence of messages that p C-delivered since its previous C-broadcast

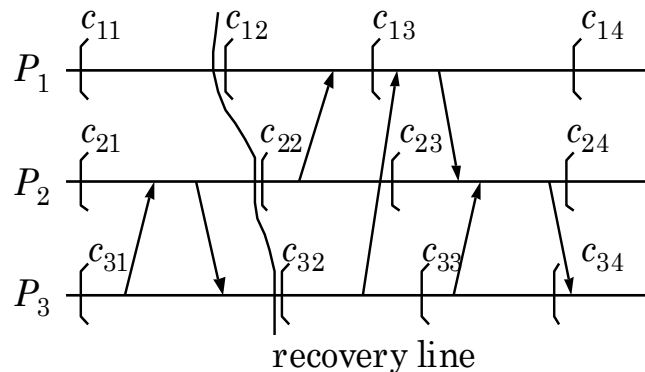
Checkpointing

Backward or rollback recovery

- systematic fault-tolerance is often based on backward recovery to recover a consistent state
- in distributed systems a state is said to be *consistent* if it *could* exist in an execution of the system
- **Recovery line:** A set of recovery points form a consistent state—called recovery line—if they satisfies the following conditions:
 - (1) the set contains exactly one recovery point for each process
 - (2) No **orphan** messages: There is no receive event for a message m before process P_i 's recovery point which has not been sent before process P_j 's recovery point.
 - (3) No **lost** messages: There is no sending event for a message m before process P_i 's recovery point which has not been received before process P_j 's recovery point.

The domino effect

- the consistency requirement for recovery lines can cause a flurry of rollbacks to recovery points in the past



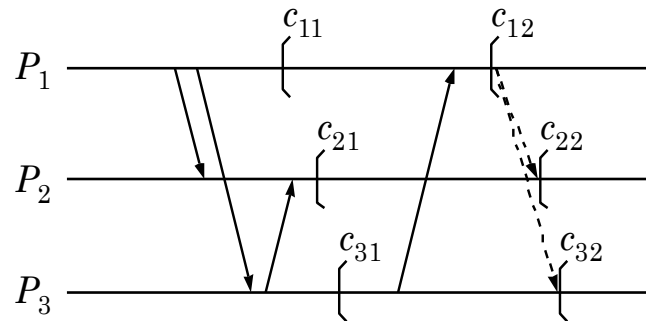
- to avoid the domino effect:
 - coordination among individual processors for checkpoint establishment
 - restricted communication between processors

A distributed checkpointing and rollback algorithm

- this protocol allows lost messages
- there are two kinds of checkpoints:
 - **permanent**: they cannot be undone
 - **tentative**: they can be undone or changed to permanent
- the checkpointing algorithm works in two phases:
 - (1) An **initiator** process P_i takes a tentative checkpoint and requests all processes to take tentative checkpoints. Receiving processes can decide whether to take a tentative checkpoint or not and send their decision to the initiator. There is no other communication until phase 2 is over.
 - (2) If the initiator process P_i learns that all tentative checkpoints have been taken then it reverts its checkpoint to permanent and requests others do the same.
- this protocol ensures that no orphan messages are in the recorded state (processes are not allowed to send messages between phase 1 and 2)

A distributed checkpointing and rollback algorithm (cont.)

- it is not always necessary to record the state of a processor during checkpointing:



- the set $\{c_{12}, c_{21}, c_{32}\}$ is also a consistent set, hence it is not necessary for P_2 to take checkpoint c_{22} , but the set $\{c_{12}, c_{21}, c_{31}\}$ would be inconsistent
- each process assigns monotonically increasing numbers to the messages it sends:

$last_recd_i(j)$ last message number that i received from j after i took a checkpoint

$first_sent_i(j)$ first message number that i sent to j after i took a checkpoint

- if P_i requests P_j to take a tentative checkpoint it adds $last_recd_i(j)$ to the message
- P_j takes a checkpoint only if $last_recd_i(j) \geq first_sent_j(i)$

A distributed checkpointing and rollback algorithm (cont.)

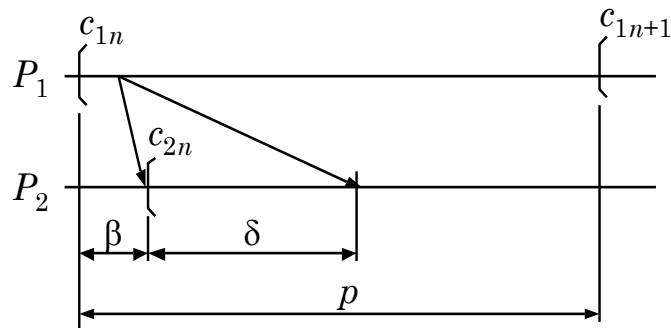
- furthermore if P_i has not received a message from P_j since its last checkpoint then there is no need for P_j to establish a new checkpoint if P_i establishes one
- to make use of this P_i maintains a set $ckpt_cohort_i$ which contains the processes from which received messages since its last checkpoint

```

upon receipt from  $i$  “take tentative checkpoint” ||  $last\_recd(j)$  do
  if  $willing\_to\_ckpt_j$  and  $(last\_recd(j) \geq first\_send_j(i))$  then
    take tentative checkpoint
    for all  $r$  in  $ckpt\_cohort_i$  do
      send to  $r$  “take tentative checkpoint” ||  $last\_recd(r)$ 
    for all  $r$  in  $ckpt\_cohort_i$  await( $willing\_to\_ckpt_r$ )
    if any  $r$  in  $ckpt\_cohort_i$  and  $(willing\_to\_ckpt_r = \text{“no”})$  then
       $willing\_to\_ckpt_i := \text{“no”}$ 
    send to  $r$   $willing\_to\_ckpt_i$ 
  upon receipt from  $i$   $m := \text{“make checkpoint permanent”}$  or
     $m := \text{“undo tentative checkpoint”}$ 
    execute command in  $m$ 
    for all  $r$  in  $ckpt\_cohort_i$  send to  $r$   $m$ 
  
```

Synchronous checkpointing

- based on synchronized clocks check points are established with a fixed period p by all processes, where β is the clock synchronization precision and δ temporal uncertainty of message transmission



- if a message is sent during $[T - \beta - \delta, T]$ it will be received before $T + \beta + \delta$
- to achieve a consistent state two possibilities exists:
 - prohibit message sending during interval β after checkpoint establishment
 - establish checkpoint earlier, at $kp - \beta - \delta$ and log messages during the critical instant

Stable Storage

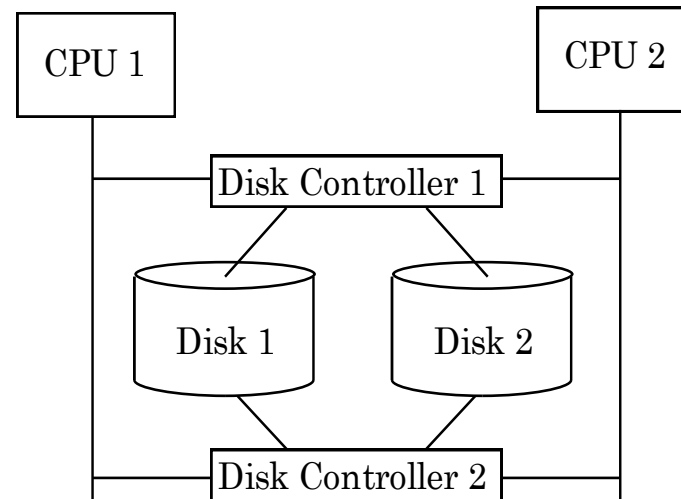
Stable Storage

- stable storage is an important building block for many operations in fault-tolerant systems (fail-stop systems, dependable transaction processing, ...)
- there are two operations which should work correctly despite of faults (as covered by the fault hypothesis):
 - procedure** `writeStableStorage(address, data)`
 - procedure** `readStableStorage(address)` **returns** `(status, data)`
- many failures can be handled by coding (CRC's) but other types cannot be handled by this technique:
 - Transient failures:** The disk behaves unpredictably for a short period of time.
 - Bad sector:** A page becomes corrupted, and the data stored cannot be read.
 - Controller failure:** The disk controller fails.
 - Disk failure:** The entire disk becomes unreadable.

Disk shadowing

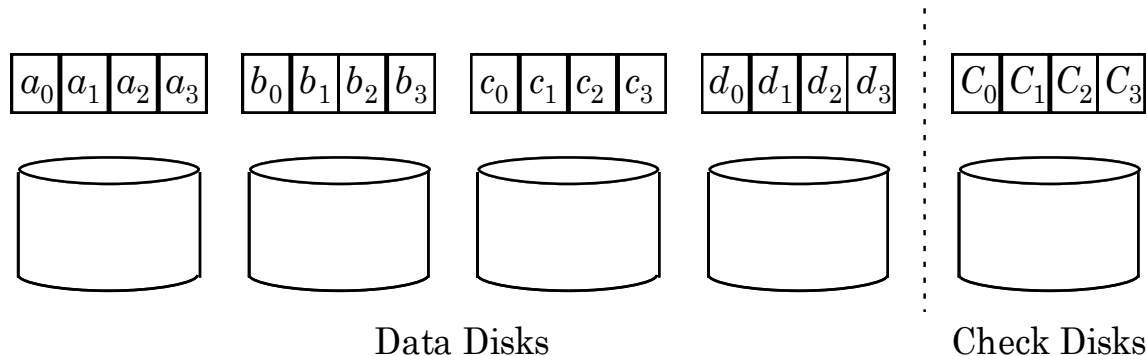
- a set of identical disk images is maintained on separate disks
- in case of two disks this technique is called *disk mirroring*
- for performance and availability reasons the disks should be “dual-ported” (e.g. Tandem system)

$$MTTF_{mirror} = \frac{MTTF}{2} \frac{MTTF}{MTTR}$$



Redundant Array of Inexpensive Disks (RAID)

- data is spread over multiple disks by “bit-interleave” (individual bits of a data word are stored on different disks)
- in the following example single bit failures can be tolerated since a parity bit is stored on a check disk and disks are assumed to detect single bit failures



- RAID's provide high reliability and I/O throughput (parallel read/write)

$$MTTF_{RAID} = \frac{MTTF}{G + C} \frac{MTTF / (G + C - 1)}{MTTR}$$

G .. data disks C .. check disks

Diagnosis

Fault diagnosis in distributed systems

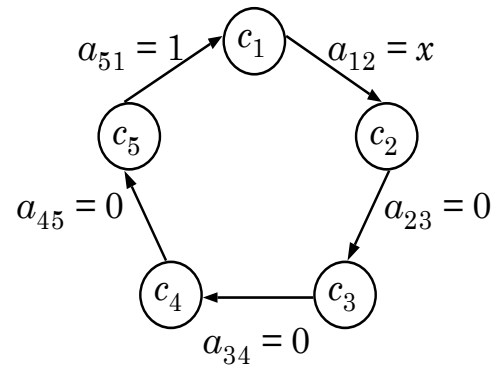
- **Problem:** Each non-faulty component has to detect the failure of other components in a finite time.
- while it is the goal to identify all failed components there are theoretical upper bounds on the number of failed components that can be identified
- **PMC model:**
 - a system S is composed out of n components $C = \{c_1, c_2, \dots, c_n\}$
 - components are either correct or faulty as a whole, they are the lowest level of abstraction that is considered
 - each component is powerful enough to test other components
 - *tests* involve application of stimuli and the observation of responses, tests are assumed to be *complete* and *correct*
 - correct components always report the status of the tested components correctly
 - faulty components can return incorrect results of the tests conducted by them (byzantine failure assumption)

Syndromes

- each component belonging to C is assigned a particular subset of C to test (no component tests itself)
- the complete set of test assignments is called **connection assignments**, it is represented by a graph $G = (C, E)$
 - each node in C represents a component
 - each edge represents a test such that (c_i, c_j) iff c_i tests c_j .
 - each edge is assigned an outcome a_{ij} ,
 - $a_{ij} = 0$ if c_i is correct and c_j is correct
 - $a_{ij} = 1$ if c_i is correct and c_j is faulty
 - $a_{ij} = x$ if c_i is faulty (x is in $\{0|1\}$)
- the set of all test outcomes is call the **syndrome** of S

An example system

- the system consists of five components, it is assumed that c_1 is faulty



- the syndrome for this system is a 5 bit vector $(a_{12}, a_{23}, a_{34}, a_{45}, a_{51}) = (x, 0, 0, 0, 1)$ (x is in $\{0|1\}$)
- t -diagnosable:** A system is t fault diagnoseable if, given a syndrome, all faulty units in S can be identified, provided that the number of faulty units does not exceed t .
- a system S with n components is t -diagnoseable if $n \geq 2t + 1$ and each component tests at least t others, no two units test each other

Central diagnosis algorithms

- A simple algorithm is to take an arbitrary component and suspect it to be either correct or faulty. Based on this guess, and the test results of other components are labeled, if a contradiction occurs, the algorithm backtracks. Complexity $O(n^3)$
- the best known algorithm has a complexity of $O(n^{2.5})$

The adaptive DSD algorithm

- an adaptive distributed system-level diagnose algorithm that is round based
- it stabilizes within n rounds and has no bound on t , provided the communication is reliable
- each component i holds an array $TESTED_UP_i$
- $TESTED_UP_i[k] = j$: component i has received information from a correct component saying that k has tested j to be fault free

The adaptive DSD algorithm (cont.)

- each component executes the following algorithm periodically

```
t := i
repeat
  t := (t + 1) mod n
  request t to forward  $TESTED\_UP_t$  to i
until (i test t as "fault free")
 $TESTED\_UP_i[i] := t$ 
for j := 1 to n - 1
  if  $i \leq t$ 
     $TESTED\_UP_i[j] := TESTED\_UP_t[j]$ 
```

- the algorithm stops if the first fault free component is found
- this component is marked as fault free in $TESTED_UP_i[i]$
- the information of $TESTED_UP_t$ is copied to $TESTED_UP_i$, which forwards the diagnostic information in reverse order through the system

The adaptive DSD algorithm (cont.)

- if a component wants to diagnose the system state it executes the following algorithm:

```
for j:= 1 to n STATEj := "faulty"
t:= i
repeat
  STATEt := "fault-free"
  t:= TESTED_UPt
until t = i
```
- the diagnosis algorithm constructs a cycle that contains all correct components
- if the length of the cycle is l then after l rounds all vectors $TESTED_UP$ will be updated
- since the cycle is constructed by ascending component indices, the repeat loop in the algorithm collects all correct components and updates $STATE$ accordingly

Fault-Tolerant Software

Fault tolerant software

- to tolerate software faults the system must be capable to tolerate design faults
- in contrast, for hardware it is typically assumed that the design is correct and that components fail
- software requires **design diversity**
- **But:** especially for software, perfection is much easier and better understood than fault-tolerance

Exception handling

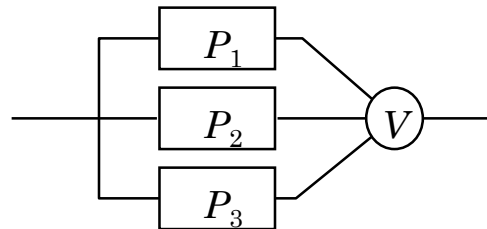
- to detect erroneous states of software modules the exception mechanism can be used (software and hardware mechanisms for detection of exceptional states)
- a procedure (method) has to satisfy a *pre condition* before delivering its intended service which has to satisfy *post conditions* afterwards
- the state domain for a procedure can be subdivided:

anticipated exceptional domain	standard domain
unanticipated exceptional domain	

- an *exception mechanism* is a set of language constructs which allows to express how the standard continuation of module is replaced when an exception is raised
- exception handlers allow the designer to specify recovery actions (forward or backward recovery)

N-Version Programming

- n non-identical replicated software modules are applied
- instead of an acceptance test a voter takes a m out of n or majority decision



- majority voting can tolerate $(n - 1)/2$ failures of modules
- modeling of n -version programming is equivalent to active redundant systems with voting
- *driver* program to invoke different modules (different processes for module execution), wait for results and voting
- require more resources than recovery blocks but less temporal uncertainty (response time of slowest module)

N -Version Programming (cont.)

- n -version programming is approach to systematic fault-tolerance:
 - there is no application specific acceptance test necessary
 - exact voting on every bit is systematic
- **But:** problem of replica nondeterminism:
 - the real-world abstraction limitation is no problem
(all modules get exactly the same inputs from driver program)
 - consistent comparison problem: diverse implementations, different compilers, differences in floating point arithmetic, multiple correct solutions (n roots of n th order equation), ...
- **Problems:**
 - there is **no** systematic solution for the consistent comparison problem
 - either very detailed specification with many agreement points (limits diversity)
 - or approximate voting to consider nondeterminism (application-specific)

N self-checking programming

- n versions are executed in parallel (similar to N -version programming)
- each module is self-checking, an acceptance test is used (similar to recovery blocks)
- mixture of application specific and systematic fault-tolerance
- requires no backward recovery and no voting

