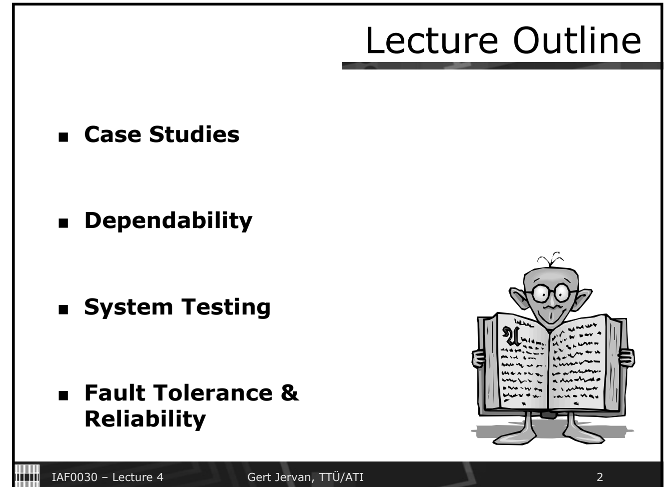


Case Studies – variant 1

- Teosta õnnetuse analüüs.
  - Õnnetuse ülevaade
  - Identifitseeri õnnetuse otsene põhjus ning too välja kaudsed tegurid
  - Kirjelda õnnetuseni viinud sündmuste kulgu ja nende konteksti
  - Millised (ennekõike arvutisüsteemidega seotud) protsessid ja meetodid olid ebapiisavad või vigased ning andsid oma osa õnnetuse tekkimisse? Kus tehti vigu?
  - Mida on sellest õpitud? Kas peale õnnetust on välja antud mingeid juhiseid või soovitusi?
  - Kuidas oleks võimalik tulevikus selliseid õnnetusi vältida (süstemaatilised meetodid)
  - Koosta õnnetuse „fault tree” või „event tree” (lähtudes ühest sündmusest). max. 1 lk.
  - Kas antud süsteemile on võimalik määrata SIL? Põhjenda.
  - Kokkuvõte

Case Studies – variant 2

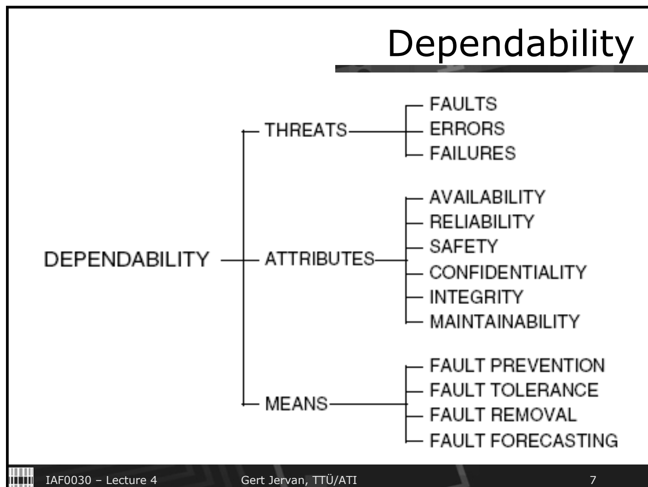
- Teosta süsteemi ohutusanalüüs. Eeldan süsteemi, mis on ohutus-kriitiline (safety-critical)
  - Süsteemi lühikirjeldus
  - Süsteemi kasutusvaldkonnad
  - Süsteemi kasutamisega kaasnevad riskid erinevates kasutusvaldkondades. Mida on tehtud nende riskide vähendamiseks?
  - Süsteemi loomisel kasutatud meetodite kirjeldus
  - Kuidas on lahendatud veakindluse (fault tolerance) ja töökindlusega (reliability) seotud probleemid?
  - Millised meetmed on võetud kasutusele süsteemi ohutuse tagamiseks?
  - Kuidas on tagatud süsteemi korrektsus?
  - Kas süsteemi loomisel on jälgitud mingeid standardeid või sertifitseerimismeetodeid? Milliseid ja mis määral?
  - Kokkuvõte

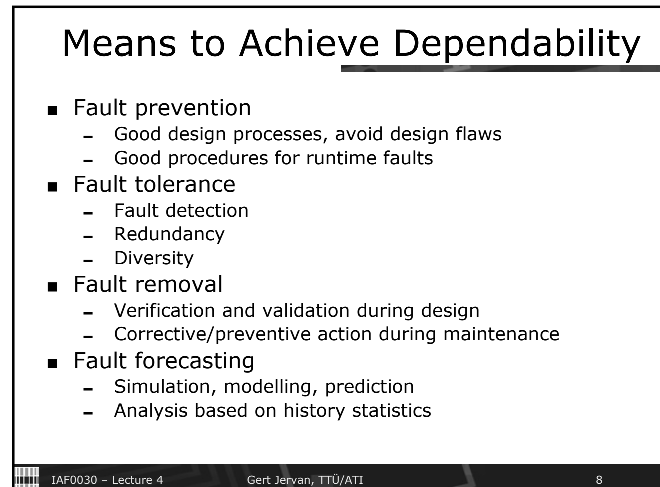
Basics

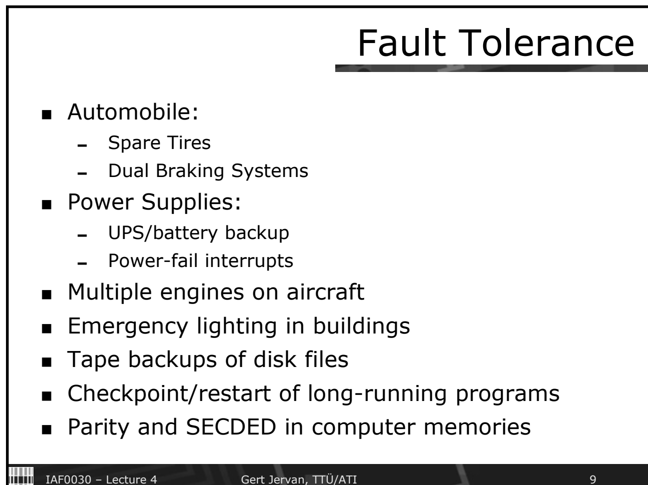
- Computing systems are characterized by five fundamental properties:
  - functionality
  - usability
  - performance
  - cost
  - dependability

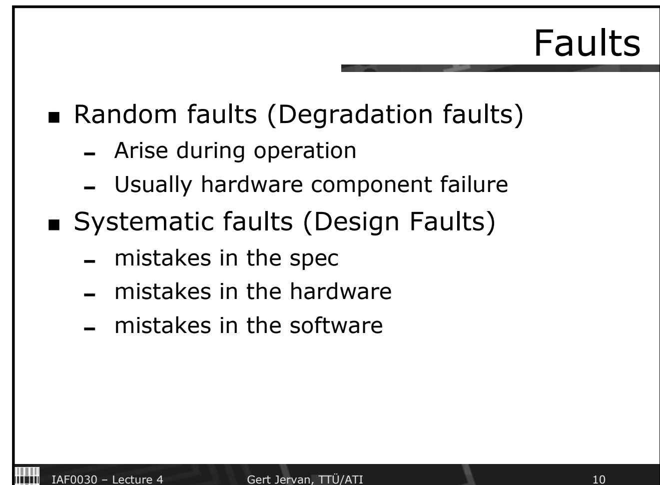
Faults

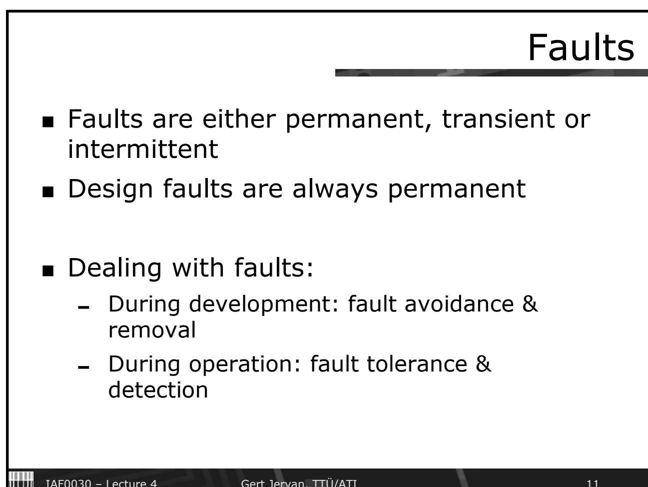
- Faults are there!
- Either prevent, **tolerate**, remove or forecast
- We need redundancy
  - System that is more complex than needed for performing the required task

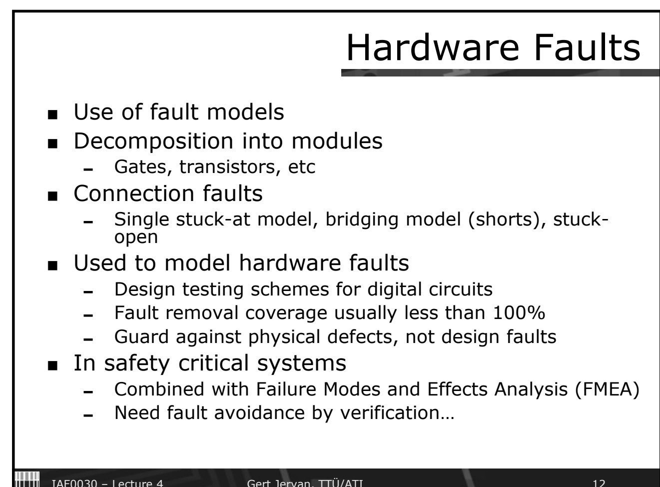

Dependability
DEPENDABILITY
THREATS
FAULTS
ERRORS
FAILURES
ATTRIBUTES
AVAILABILITY
RELIABILITY
SAFETY
CONFIDENTIALITY
INTEGRITY
MAINTAINABILITY
MEANS
FAULT PREVENTION
FAULT TOLERANCE
FAULT REMOVAL
FAULT FORECASTING
IAF0030 – Lecture 4
Gert Jervan, TTÜ/ATI
7


Means to Achieve Dependability
Fault prevention
Good design processes, avoid design flaws
Good procedures for runtime faults
Fault tolerance
Fault detection
Redundancy
Diversity
Fault removal
Verification and validation during design
Corrective/preventive action during maintenance
Fault forecasting
Simulation, modelling, prediction
Analysis based on history statistics
IAF0030 – Lecture 4
Gert Jervan, TTÜ/ATI
8


Fault Tolerance
Automobile:
Spare Tires
Dual Braking Systems
Power Supplies:
UPS/battery backup
Power-fail interrupts
Multiple engines on aircraft
Emergency lighting in buildings
Tape backups of disk files
Checkpoint/restart of long-running programs
Parity and SECDED in computer memories
IAF0030 – Lecture 4
Gert Jervan, TTÜ/ATI
9


Faults
Random faults (Degradation faults)
Arise during operation
Usually hardware component failure
Systematic faults (Design Faults)
mistakes in the spec
mistakes in the hardware
mistakes in the software
IAF0030 – Lecture 4
Gert Jervan, TTÜ/ATI
10


Faults
Faults are either permanent, transient or intermittent
Design faults are always permanent
Dealing with faults:
During development: fault avoidance & removal
During operation: fault tolerance & detection
IAF0030 – Lecture 4
Gert Jervan, TTÜ/ATI
11


Hardware Faults
Use of fault models
Decomposition into modules
Gates, transistors, etc
Connection faults
Single stuck-at model, bridging model (shorts), stuck-open
Used to model hardware faults
Design testing schemes for digital circuits
Fault removal coverage usually less than 100%
Guard against physical defects, not design faults
In safety critical systems
Combined with Failure Modes and Effects Analysis (FMEA)
Need fault avoidance by verification…
IAF0030 – Lecture 4
Gert Jervan, TTÜ/ATI
12

Other Faults

- Hardware design and specification faults
  - Few fault models available
  - Many faults cannot be modelled
  - System must meet the spec, but spec might be incorrect as well
  - Spec errors may manifest as either hardware or software failures
  - Use of formal methods (formal spec. languages, automata theory, formal verification, model checking, etc.)

Software Faults

- Bugs:
  - Software spec faults
  - Coding faults
  - Logical errors within calculations
  - Stack overflows or underflows
  - Uninitialized variables
- No random failures and it does not degrade with age
- Always systematic
- Exhaustive testing almost impossible
- Must be tolerated

SW Testing - i.e. Verification

- Verification:
  - SW testing
  - formal verification
- Functional and structural testing
- Path testing, transaction flow testing, data-flow testing, domain testing, mutation testing etc.

Fault Detection Techniques

- Functionality checking
  - march test
- Consistency checking
  - range checking, overflow
- Signal comparison
- Information redundancy
  - checksums, cyclic redundancy codes, error correcting codes
- Monitoring techniques
  - Loopback testing
  - Power supply monitoring

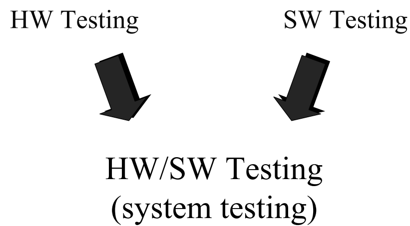
Watchdog Timer

- An inexpensive method of error detection
- Process being watched must reset the timer before the timer expires, otherwise the watched process is assumed as faulty
- Watchdog timers only detect errors which manifest themselves as a control-flow error such that the system does not continue to reset the timer
- Only processes with relatively deterministic runtimes can be checked, since the error detection is based entirely on the time between timer resets

Heartbeats

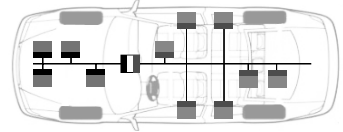
- A common approach to detecting process and node failures in a distributed (networked) computing environment.
- Periodically, a monitoring entity sends a message (a heartbeat) to a monitored node or process and waits for a reply.
- If the monitored node does not respond within a predefined timeout interval, the node is declared as failed and appropriate recovery action is initiated.
- Adaptive or smart

System Testing



Testing Distributed RT-Systems

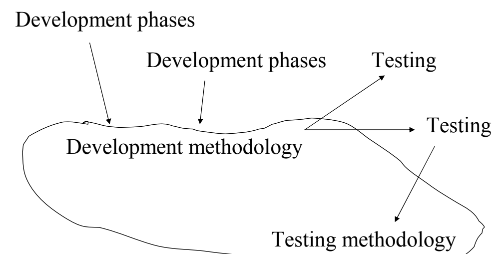
- *RT-System* – system, which is required to adhere not only functional but also tempoal requirements ("timing constraints" or "deadlines")
- RT-systems:
  - Hard RT-systems
  - Soft RT-systems



Testing Distributed RT-Systems

- Problems with distributed systems:
  - Increased complexity
  - The difficulties of observing and monitoring
  - Non-reproducible behaviour of the system
  - The lack of synchronized global clock and, consequently, the difficulties of unambiguously defining a "global state"

Testing Distributed RT-Systems



Testing Distributed RT-Systems

- Observability
  - What?
  - How?
  - When?
- Auxiliary outputs, interactive debuggers
- "Probe effect" - bad, bad, bad...
  - Can be ignored, minimized or avoided

Testing Distributed RT-Systems

- **Reproducibility**
  - *Regression testing* – retesting after errors have been corrected
    - errors truely corrected
    - no new errors
  - A distributed system may be non-reproducible due to nondeteminism in it's hardware, software or operating system

Testing Distributed RT-Systems

- **Obtaining reproducibility**
  - Language-based approach
    - Enforcing the identified scenarios during execution
    - All solutions rely on source code transformations
  - Implementation based approach
    - Collecting all missing information during an execution of the system
    - Event histories or traces

Testing Distributed RT-Systems

- Disadvantages of implementation based approach:
  - Special dedicated HW (to monitor)
  - Large amount of information
  - Can we guarantee the correctnes of reply?
  - Modified programs. What happens with event histories. Are they still valid?
  - Event histories can be used only on target systems

Testing Distributed RT-Systems

- Interdependence of Obsevability and Reproducibility
  - Not independent!
  - Probe effect

Testing Distributed RT-Systems

- **The host/target approach**
  - Host - development
  - Target - execution
- Testing on the host system is used for (functional) unit testing and preliminary integration testing (as much as possible)
- Testing on the target system involves completing the integration test and performing the system test. Also performance, timing, etc.

Testing Distributed RT-Systems

- **Environment simulation** (for target system test)
  - Simulated v. real environment:
    - Safety and/or cost considerations.
    - "rare event" situations
    - More control over simulated environment
    - Easier to obtain responses and test results
  - On-line v. off-line test data generation:
    - Need to generate large amounts of input data
    - Runs cost-effectively

Testing Distributed RT-Systems

- **Representativity**
  - Only small number of real-world scenarios can be anticipated and taken into account.
  - Only a fraction of those anticipated real-world scenarios can be tested due to the *combinatorial explosion* of possible event and input combinations.
- *Test coverage* - how many of the anticipated real-time scenarios can be or have been covered by corresponding test scenarios.

Self-checking distributed systems

- Run-time checking of the effects of faults on system behaviors needs to be carried out continuously.
- Reliability – the key to distributed SW quality

Self-checking distributed systems

- Aspects to design correct SW:
  - Reliability with which the SW specifications are adequately described and correctly implemented in the actual implementation.
  - Run-time checking

Self-checking distributed systems

- *Fault-secure systems* are systems, where faults may be enforced not to propagate.
  - Faults are not visible or have no effect
  - Faults are visible, but it's easy to notice that an error exists
- *Self-testing* – System is self testing when there exists testing behavior, occurring during the run-time behavior of the system, such that this fault will be propagated to the output and it's easy to notice, that there is a fault (out of predefined set of values)
- System is *self-checking* for a set of faults, if whatever a fault belonging to this set, it is fault-secure and self-testing.

Self-checking distributed systems

- Worker-observer
  - the *worker* is a classical implementation of the system behavior
  - the observer is a given redundant implementation whose outputs are comparable with the outputs of the worker.
- To obtain observing behavior:
  - Redundancy
  - Reference
  - Visibility
    - Worker cooperates with the observer
    - Worker behavior can be spied by the observer

Self-checking distributed systems

- A *formal observer* is a subsystem designed to check distributed behaviors where:
  - Its sw is independent of the specific protocols to be checked in the considered system;
  - Its data are defined by the protocols to be checked and this data can be formally specified and verified.

Self-checking distributed systems

- Design of the system
  - write a description of the beavior of the system to be implemented;
  - Implement the system itself, i.e., the worker;
  - From the description of the worker, select (based on experience) that part of the behavior which should be observed and write a formal model of it.

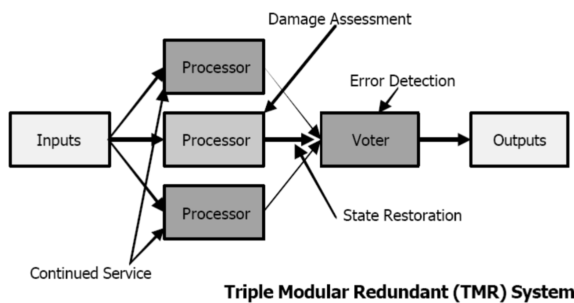
Self-checking distributed systems

- The system is *quasi self-checking* if
  - It is an observer-worker system
  - The observer is a formal observer.
- For "real-life" only part of the system will be modelled.
- Formal model must be able to
  - Express simplified specifications of distributed systems
  - Support verification procedures
  - Be able to act as a basis for implementing the observer.

Phases of Fault Tolerace

- Error detection:
  - You must know there is a problem in order to deal with it
- Damage assessment:
  - You must know or at least estimate the damage so as to know how bad the situation is
- State restoration:
  - A consistent state is needed to continue
- Continued service:
  - Do something useful with what is left

Fault Tolerant System Example



Triple Modular Redundant (TMR) System

Risk of single-point failure

Forms of Redundancy

- Hardware redundancy
- Software redundancy
- Information redundancy
- Temporal (time) redundancy

- Design diversity, for hardware/software
  - Develop different implementations of the same hardware/software component
  - Called N-version programming
  - Then apply static or dynamic redundancy

Hardware Fault Tolerance

- Static redundancy
  - Component (at least) triplicated
    - Triple Modular Redundancy (TMR), N-Modular Redundancy (NMR)
  - Voting element used to remove effects of single failure
  - Loss Of Unit Implies:
    - Removal Or Containment
    - Service Provided By Those That Remain
- Dynamic redundancy
  - Component has a mirror that is invoked when fault occurs
  - Cold or Hot Standby, spares
  - Loss Of Unit Implies:
    - Removal Or Containment
    - Introducing Standby Unit

Hybrid Redundancy

- N-S modular redundancy with "S" spares
- As members of the N-S fail, spares switched in
- Able to tolerate up to N-2 failures
- Spares may be unpowered:
  - Saves power
  - Unpowered units much more reliable than powered
  - Attention required to infant mortality
- Clearly applicable to:
  - Long-duration systems
  - Systems with no repair opportunity

Space Shuttle Computer System

- Uses combination of
  - Redundancy, fault detection and design diversity
- Hardware voting on sensors and actuators
- Five identical computers
  - During critical stages, four computers work in NMR with voting for fault detection
  - Fifth computer performs non-critical functions, e.g. communications
- Fault tolerance
  - Tolerates failure of two computers
  - In case of third failure, crew/ground control decide which computer wins
  - Fifth computer can take over control, uses different software

Design Faults

- Design faults are much more difficult to deal with than degradation faults because:
  - They are hard to anticipate
  - Their effects are hard to predict
  - Component failure semantics tend to be undefined
- This makes all forms difficult to tolerate, especially software faults

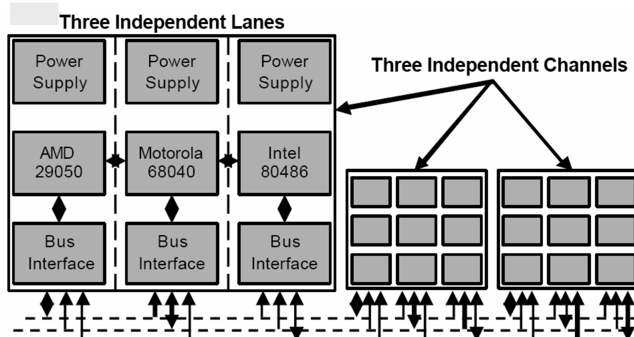
Common Design Faults

- All forms of software:
  - System software
  - Application software
  - Embedded software (firmware)
- All forms of computing hardware:
  - Hardware design faults now dominate
  - Degradation faults used to dominate
- Power supply systems
- Component interconnection wiring

Design Diversity

- Idea:
  - Design faults are "aspects" of design
  - Different designs, different faults
  - Produce multiple ***designs***—independent level.
  - Operate in parallel at execution time
- Applies to all types of design fault
- Can be configured using many system architectures that we have already seen:
  - Dual, switched dual, NMR, TMR, etc.

B777 Primary Flight Computer Architecture



B777 PFC CPUs

- Problem:
  - Processors often (essentially always) contain design faults, need to deal with them
  - 777 channel is a TMR system
- Three manufacturers, three designs
- Are these designs ***different?***
- How would you ***measure*** the difference?
- What ***metric*** is there for design diversity?

N Modular Redundancy

- Independent development of modules
- This is what Boeing did with N = 3 for processors
- Operation:
  - Parallel—***forward*** error recovery
  - Serial—***backward*** error recovery
- In software with forward error recovery, referred to as *N-version programming*
- In software with backward error recovery, referred to as *recovery block*

N-Version Programming

- NMR for software
- Practical issues:
  - Cost of development, team separation
  - Resources during execution
  - Different execution times for different versions
  - Different but similar output values
  - Different but valid output values (multiple correct solutions)

N-Version Programming

- Performance:
  - Assumed statistical independence
  - If not independent, then ***no lower bound***
  - Common specification defects
  - Common implementation (***design***) faults
- Problem compounded by ***comparison checking*** during testing

Questions?

Tallinn University of Technology

Gert Jervan

Department Of Computer Engineering
Tallinn University of Technology
Estonia