Dependable Systems

Definitions and Metrics (IV)

Dr. Peter Tröger

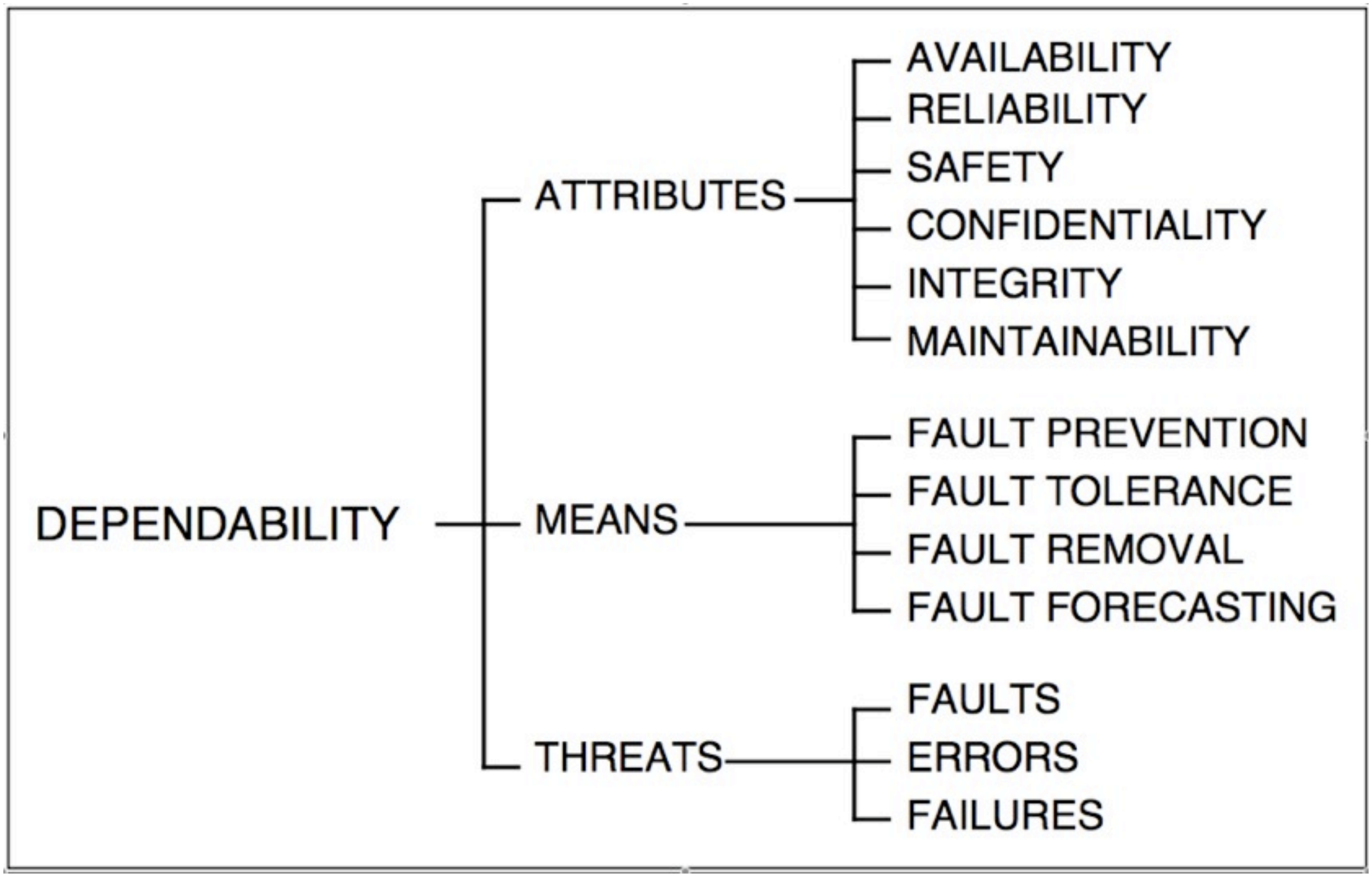
Sources:

J.C. Laprie. Dependability: Basic Concepts and Terminology

Echtle, Klaus: Fehlertoleranzverfahren. Heidelberg, Germany : Springer Verlag, 1990.

Pfister, Gregory F.: High Availability. In: In Search of Clusters. , S. 379-452

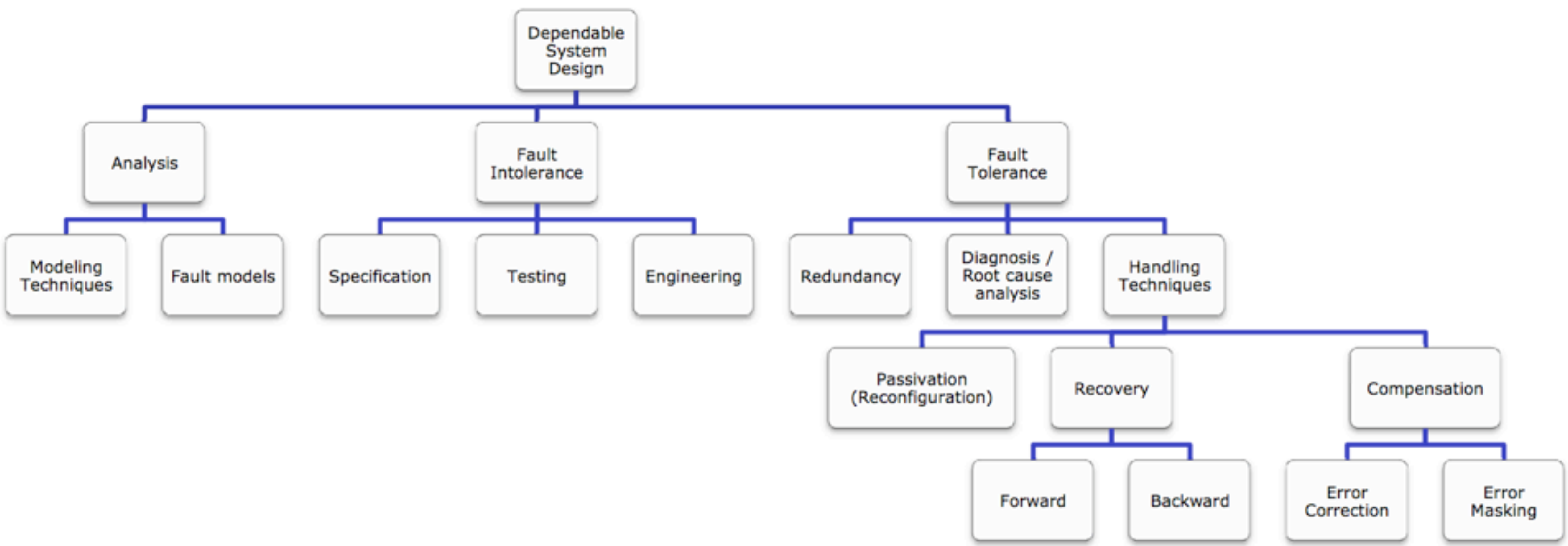
Dependability Tree (Laprie)



Dependability Means (Laprie)

- Offline / online techniques
  - Fault intolerance techniques
    - **Fault prevention** - Prevent fault occurrence or introduction
    - **Fault removal** - Reduce the presence of faults
    - 100% fault-free servicing for the whole life time is not possible
  - Fault tolerance techniques
    - **Fault forecasting** - Estimate the present number, future incidence, and the consequences of faults
    - **Fault tolerance** - Provide service complying with specification in spite of faults
- Problems with **coverage** and **validation of the validator**

Dependable System Design (Echtle)



Fault Prevention

- Specific approaches for avoiding faults
  - Specialized specification formalisms and techniques
  - Specialized development / manufacturing process to prevent design faults
  - Shielding
  - Only use ultra-reliable components
- General engineering approaches
  - Software engineering procedures
  - Quality management regulations and enforcement
  - Training and organization of maintenance departments

Fault Removal

- Make faults disappear before fault tolerance becomes relevant
- Step 1: Verification
  - Check if the system adheres to **verification conditions**; if not, take next steps
  - **Static verification**: Static analysis, data flow analysis, compiler checks
  - **Dynamic verification**: Symbolic execution or verification testing
- Step 2: Diagnosis
  - Find the faults that influenced the verification conditions
- Step 3: Correction
  - Fix the problem, repeat the steps (**regression**)
- Fault removal during operation: **Corrective maintenance (curative / preventive)**

Testing

- Selecting test inputs is driven from different view points
  - Testing purpose: **conformance testing**, **fault-finding testing**
  - System model: **functional testing** (with functional model) or **structural testing**
  - Fault model: enables **fault-based testing**
- Deterministic testing vs. random testing
- Structural testing of hardware is fault-finding, fault-based, structural testing
- Structural testing of software is fault-finding, non-fault-based, structural testing
- **Golden unit**: Reference system for comparison of output for a given input

Fault Tolerance

- Fault tolerance is the ability of a system to operate correctly in presence of faults.

  or

- A system S is called **k-fault-tolerant** with respect to a set of algorithms $\{A_1, A_2, \dots, A_p\}$ and a set of faults $\{F_1, F_2, \dots, F_p\}$ if for every k-fault F in S, A_i is executable by a subsystem of system S with k-faults. (Hayes, 9/76)

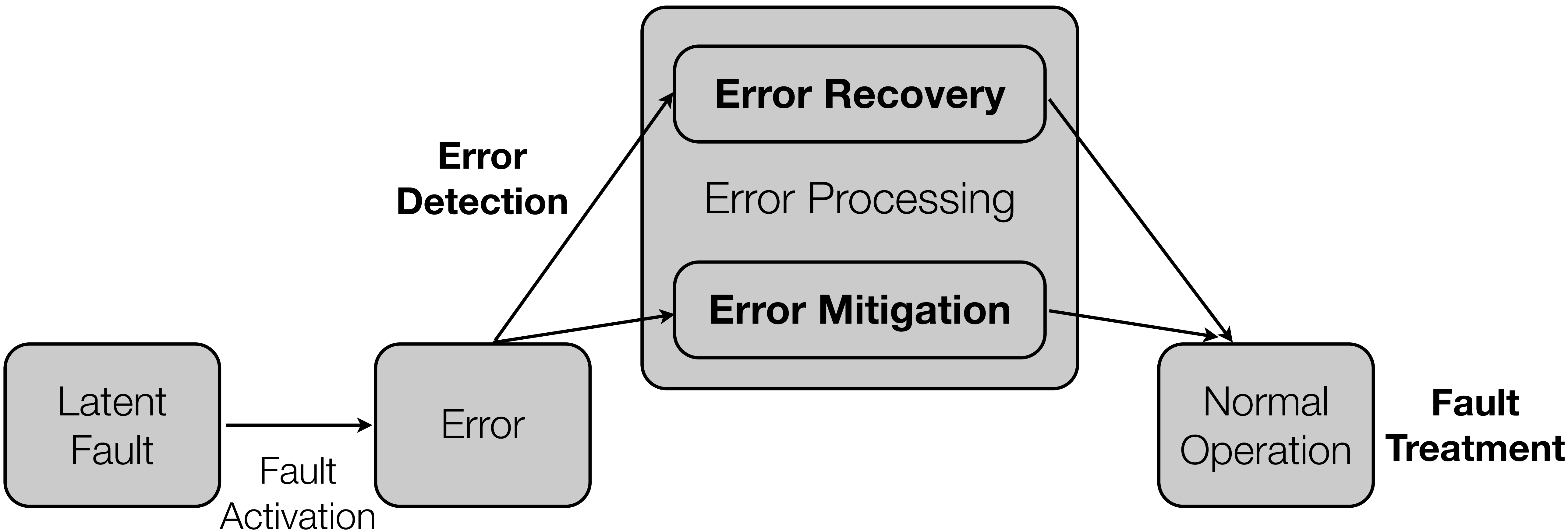
  or

- Fault tolerance is the use of **redundancy** (time or space) to achieve the desired level of system dependability - costs !
- Accepts that an implemented system will not be fault-free
- Implements automatic recovery from errors
- Is a recursive concept (voter replication, self-checking checkers, stable memory)

Fault Tolerance

- Typical design methodology in many technical and biological systems
  - Spare wheel in cars, redundant organs, ...
- Fault tolerance mechanisms need to be evaluated by dependability attributes
  - Minimum, maximum, average reliability and availability
  - Easy to formulate and understand, hard to prove - failure rate remains unknown
  - Quantitative limits based on fault model (which faults in which components)
- Typically ‚one-fault-at-a-time‘ assumption
- Different attributes of fault tolerance implementation to be checked
  - Functional verification, sensitivity analysis, minimum amount of resource resp. computational overhead, implementation performance, transparency, portability

Phases of Fault Tolerance (Hanmer)



Decomposition of Fault Tolerance (Lee & Anderson)

- **Error detection**
  - Presence of fault is deducted by detecting an error in some subsystem
  - Implies failure of the according component
- **Damage confinement**
  - Delimit damage caused due to the component failure
- **Error processing - recovery / compensation**
  - System recovers from the effect of an error
- **Fault treatment**
  - Ensure that fault does not cause again failures

Fault Tolerance - Error Detection

- **Replication check**
  - Output of replicated components is compared / voted
  - Independent failures, physical causes -> many replicas possible (e.g. HW)
  - Finds also design faults, if replicated components are from different vendors
- **Timing checks** (‚watchdog timers‘)
  - Timing violation often implies that component output is also incorrect
  - Typical solution for node failure detection in a distributed system
- **Reasonableness checks** - Run-time range checks, assertions
- Structural and coding checks, diagnostics checks, algorithmic checks
- Ideal: **Self-checking component** with clear **error confinement areas**

Fault Tolerance - Error Detection

- Replication checks are powerful and expensive, examples:
    - Execute identical copies on different hardware (component failures)
    - Execute separate and different versions (assumes independent design faults)
    - Execute same copies different times (transient faults)
    - Replicate only portion of the system
    - Works for both hardware and software
- Signaling aspect in the error detection task
    - Typical software model are exceptions, a way for implementing forward recovery
- Combination fault detection and fault location

Fault Tolerance - Damage Confinement (Taylor)

- **System decomposition**
  - Every communication link might enable damage spreading
  - Introduce mutual suspicion
  - Hardware-based separation of software components
  - OS-based separation (processes, runtime monitors, special shells)
- **Law-governed architecture**
  - Externalize contrains on interaction by runtime rules
- **Strongly-typed language**
  - Language guarantees the absence of unintended control flows

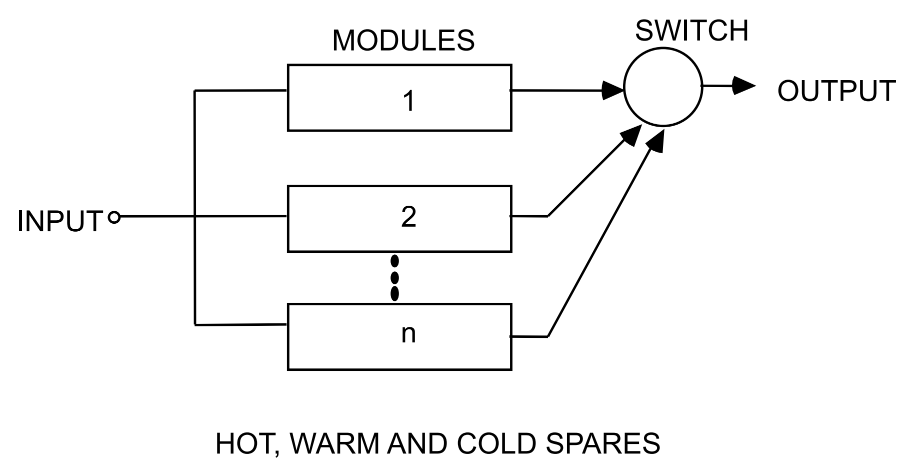
Preventing Error Propagation

- Especially relevant when single components communicate their data
  - **Single-source information** - local clock, sensor data, transaction status ...
  - Non-failed component must find an **agreement** how to treat received information
    - Special topic in distributed systems
    - Atomic broadcast, clock synchronization, membership protocols

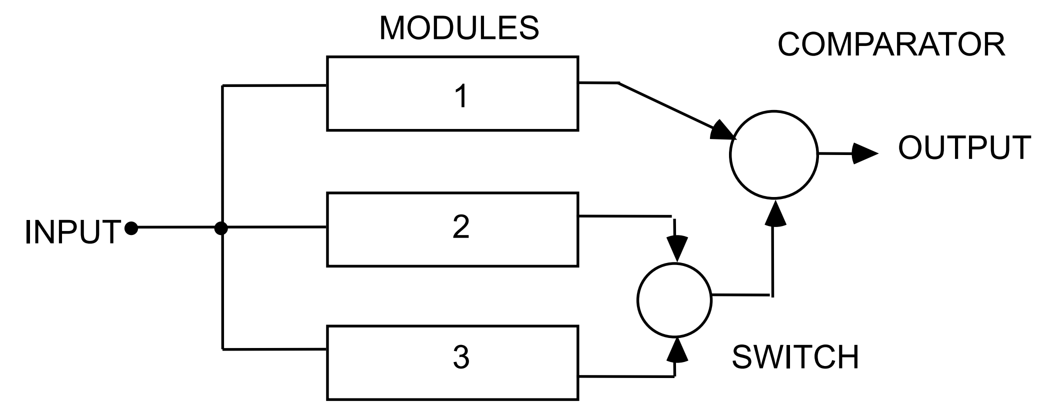
Fault Tolerance - Error Processing Through Recovery

- **Forward error recovery**
  - Error is masked to reach again a consistent state (**fault compensation**)
  - Corrective actions need detailled knowledge (**damage assessment**)
  - New state is typically computed in another way
    - Examples: error correcting codes, non-journaling file system check, advanced exception handlers, (voters)
- **Backward error recovery**
  - Roll back to previous consistent state (**recovery point** / **checkpoint**)
  - Very suitable for transient faults
  - Computation can be re-done with same components (**retry**), with alternate components (**reconfigure**), or can be ignored (**skip frame**)

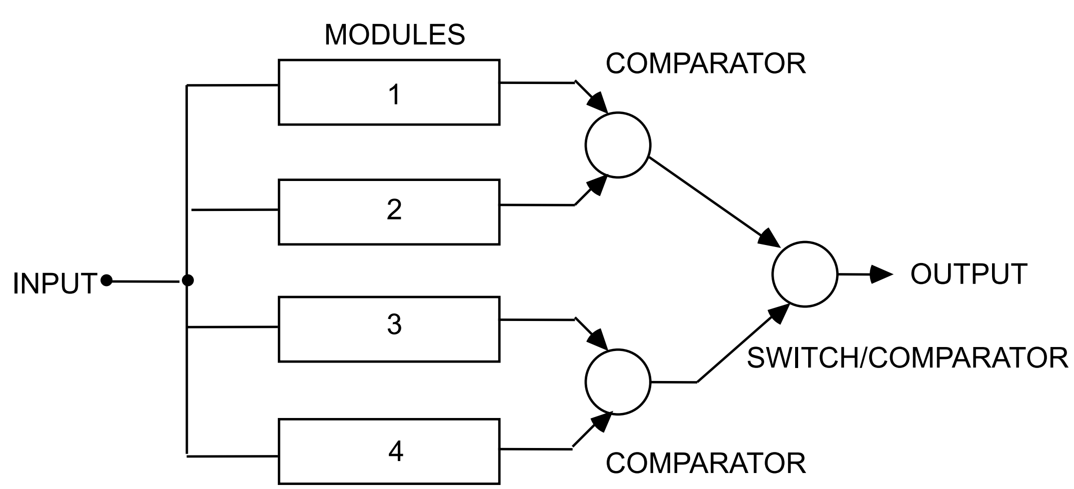
Forward Recovery Through Redundancy (Malek)



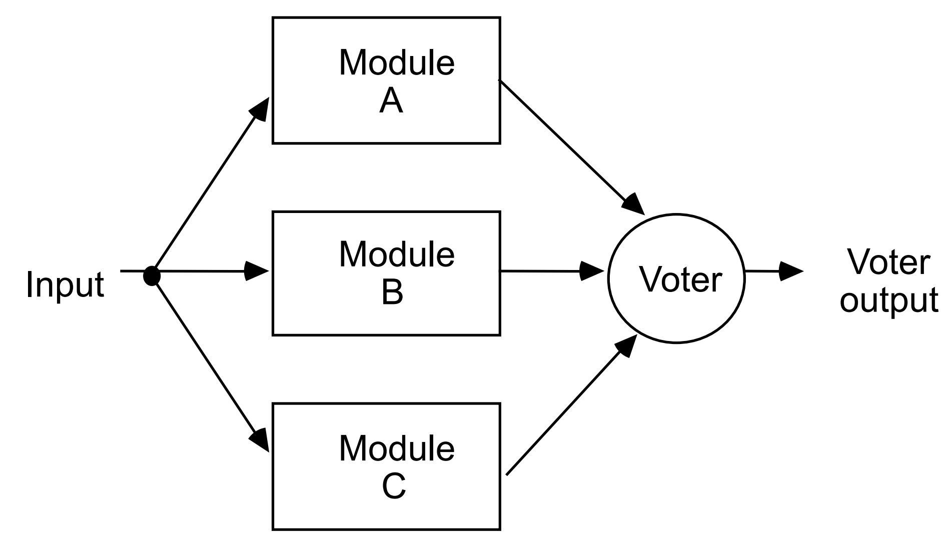
Backup Sparing



Duplex and Spare



Pair and Spare



Triple Modular Redundancy (TMR)

Fault Tolerance - Fault Treatment

- **Fault diagnosis** - determine error cause‘s location and nature
- **Fault passivation** - (remove faulty component &) reconfigure system
  - Error processing might already remove the fault - ‚**soft fault**‘
  - Typical example are temporary faults
- Fault tolerance manager
  - Careful diagnosis with hardware support
  - Damage assessment by disabling faulty components automatically
  - Example: IBM mainframe architecture
- Software rejuvenation
  - Gracefully terminating an application and immediately restarting it at a clean internal state

Fault Tolerant Mindset (Hanmer)

- What can go wrong in any given situation ?
  - Mindset to be applied in all development stages
- „Every problem in computer science boils down to tradeoffs“ [Henschen]
  - How much MTTF do you need for the MTTR ?
  - Fault prevention vs. fault tolerance vs. failure severity
- KISS principles, leave out „bells and whistles“
- Incremental additions of reliability - long-term products
- Defensive Programming
  - Simple error handling; fix root cause, not symptoms; make data auditble; make code maintainable;

Fault Tolerant Design Methodology (Hanmer)

- Assess things that can go wrong with the system (e.g. fault trees).
  - Find potential risks and according system failures.
- Define strategies to mitigate the identified risks.
  - Failure avoidance options, prevent faults from activation
- Create a mental model of the system design with redundancy.
- Design error detection and error processing capabilities.
- Design in the failure mitigation capabilities.
- Design human-computer interactions and modes of management.

Dependable Design Strategies (Malek)

- Decompose the system
  - Identify fault classes, fault latency and fault impact for the components
  - Identify “weak spots” and assess potential damage
  - Integrate partial recovery / reintegration / restart
- Determine qualitative and quantitative specs for fault tolerance and evaluate your design in specific environment
- Develop / utilize fault and error detection techniques and algorithms
- Develop / utilize fault isolation techniques and algorithms
- Refine fault tolerance, iterate for improvement
- Re-use proven components, but be aware of integration issues