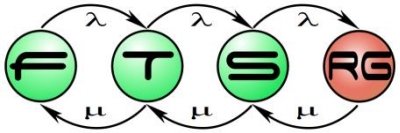
Safety-critical systems: Requirements & Architecture

Systems Engineering course

András Vörös
(slides: István Majzik)



Previous topics: Requirements

Functional vs Extra-functional

Functional

- Specific to a component of the system
- Core technical functionality

Extra-functional

- Fulfilled by the system as a whole
- Performance
- Reliability
- Safety
- Security

Previous topics: Requirements

Examples

Functional:

- The operator shall be able to change the direction of turnouts
- Train equipments shall periodically log sensor data with a timestamp

Safety:

- The system shall ensure safe traffic within a zone
- The system shall stop two trains if they are closer than a minimal distance
- No single faults shall result in system failure

Performance:

- The system should allow five trains per every 10 minutes

Reliability:

- The allowed downtime of the system should be less than 1 hour per year
- The system shall continue normal operation within 10 minutes after a failure

Supportability:

- The system shall allow remote access for maintenance

Security:

- The system shall provide remote access only to authorized personnel

Usability:

- The user interface should contain only 3 alerts at a time

Safety requirements

Introduction

- Safety-critical systems
  - Informal definition: Malfunction may cause injury of people
- Safety-critical computer-based systems
  - E/E/PE: Electrical, electronic, programmable electronic systems
  - Control, protection, or monitoring
  - EUC: Equipment under control

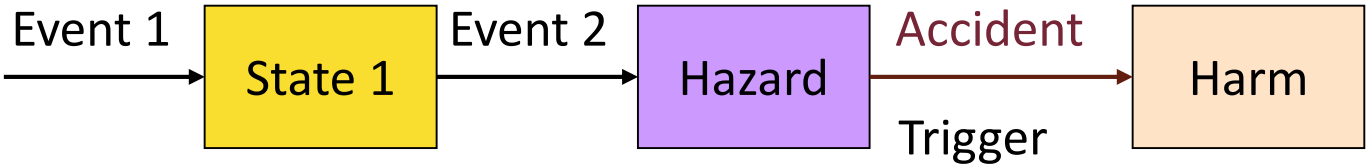
Railway signaling, x-by-wire, interlocking, emergency stopping, engine control, …

Accident examples

- Toyota car accident in San Diego, August 2009
- Hazard: Stuck accelerator (full power)
  - Floor mat problem
- Hazard control: What about…
  - Braking?
  - Shutting off the engine?
  - Putting the vehicle into neutral? (gearbox: D, P, N)

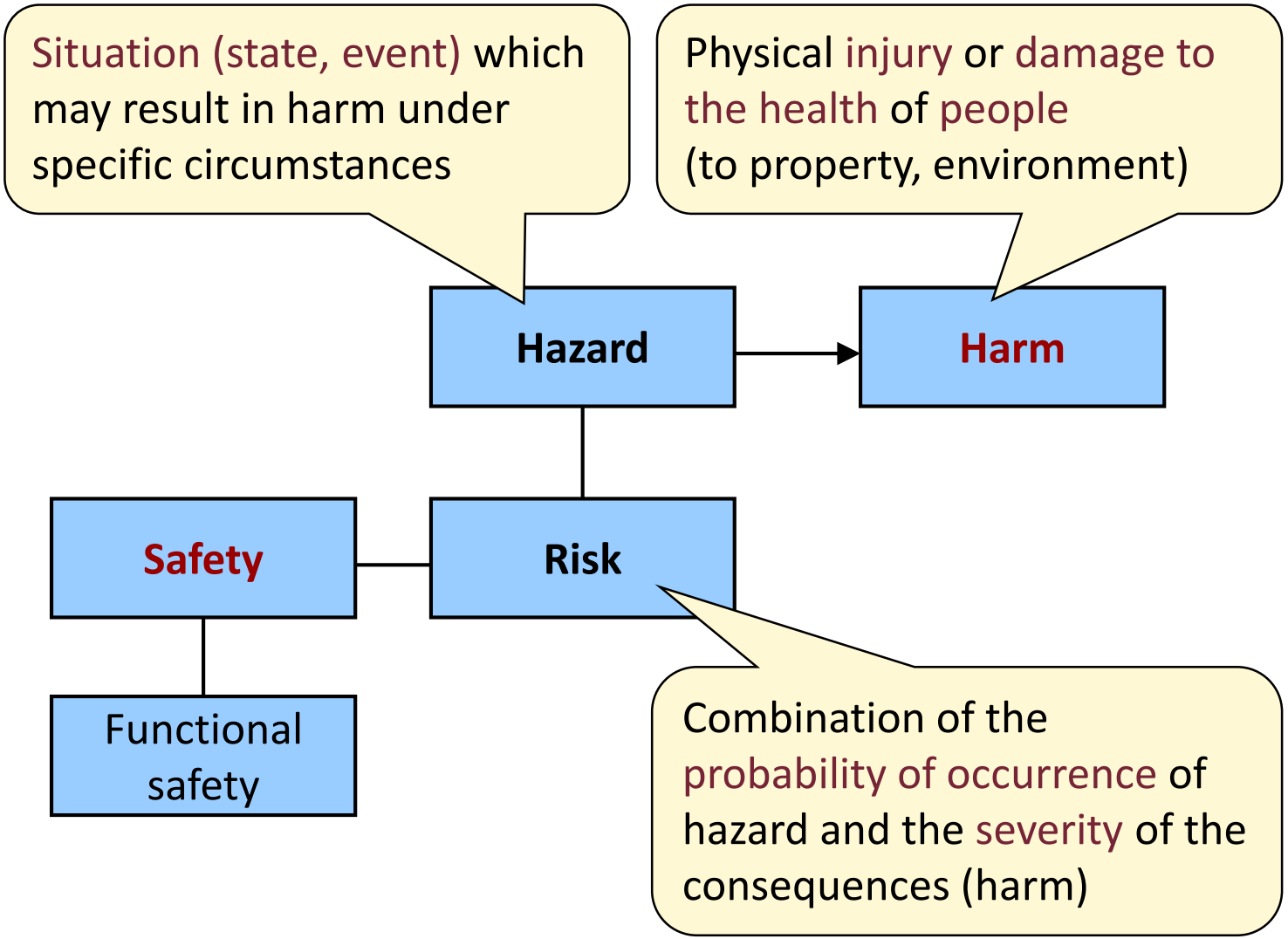
Conclusions from accident examples

- Harm is typically a result of a complex scenario
  - (Temporal) combination of failure(s) and/or normal event(s)
  - Hazards may not result in accidents

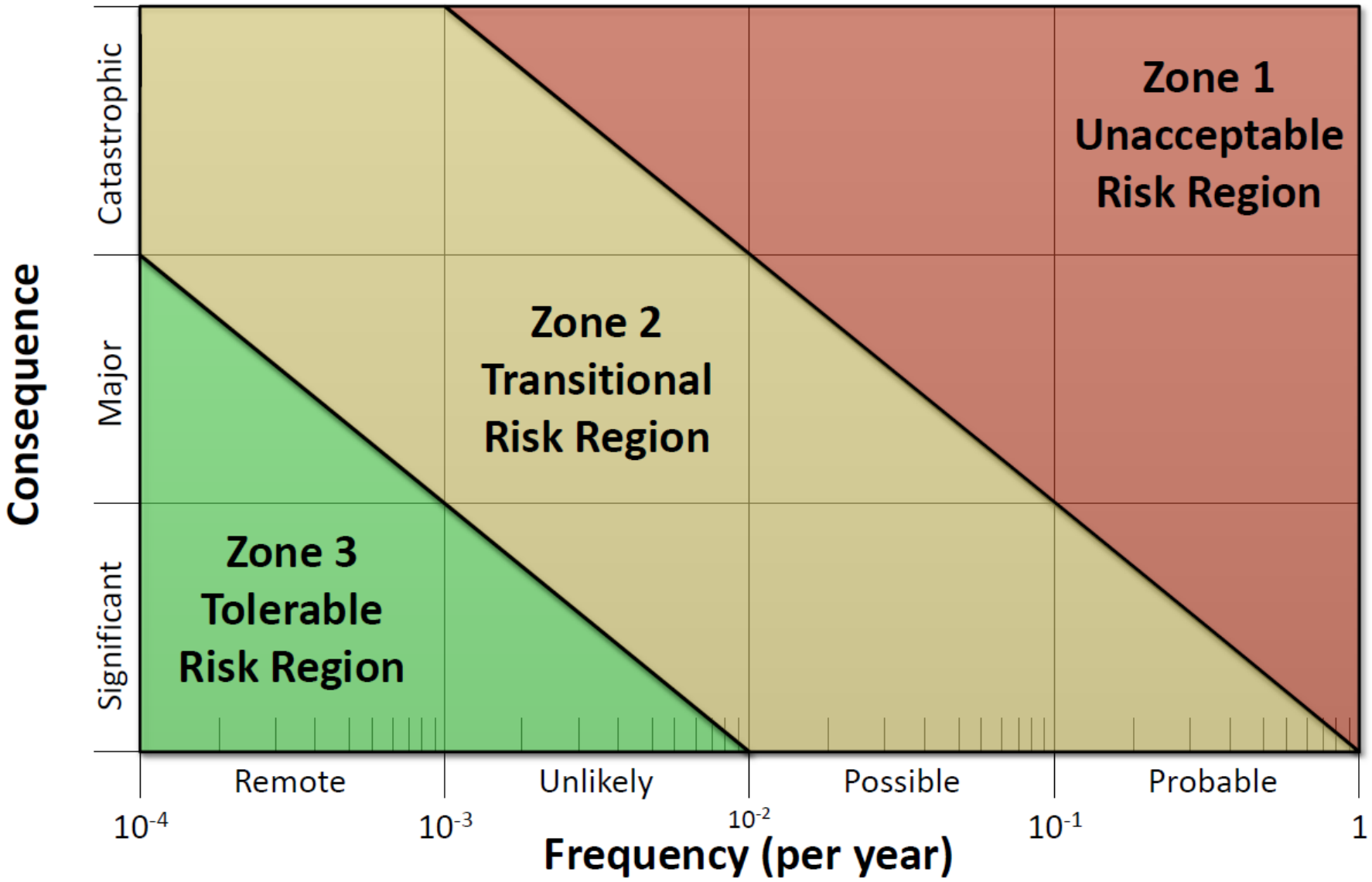


- Hazard ≠ failure
  - Undetected (and unhandled) error is a typical cause of hazards
  - But hazard may also be caused by (unexpected) combination of normal events (correct operation)
- Central problems in safety-critical systems:
  - Analysis of situations that may lead to hazard: Risk analysis
  - Assignment of functions to avoid hazards → accidents → harms
  - Specification of (extra-functional) safety requirements

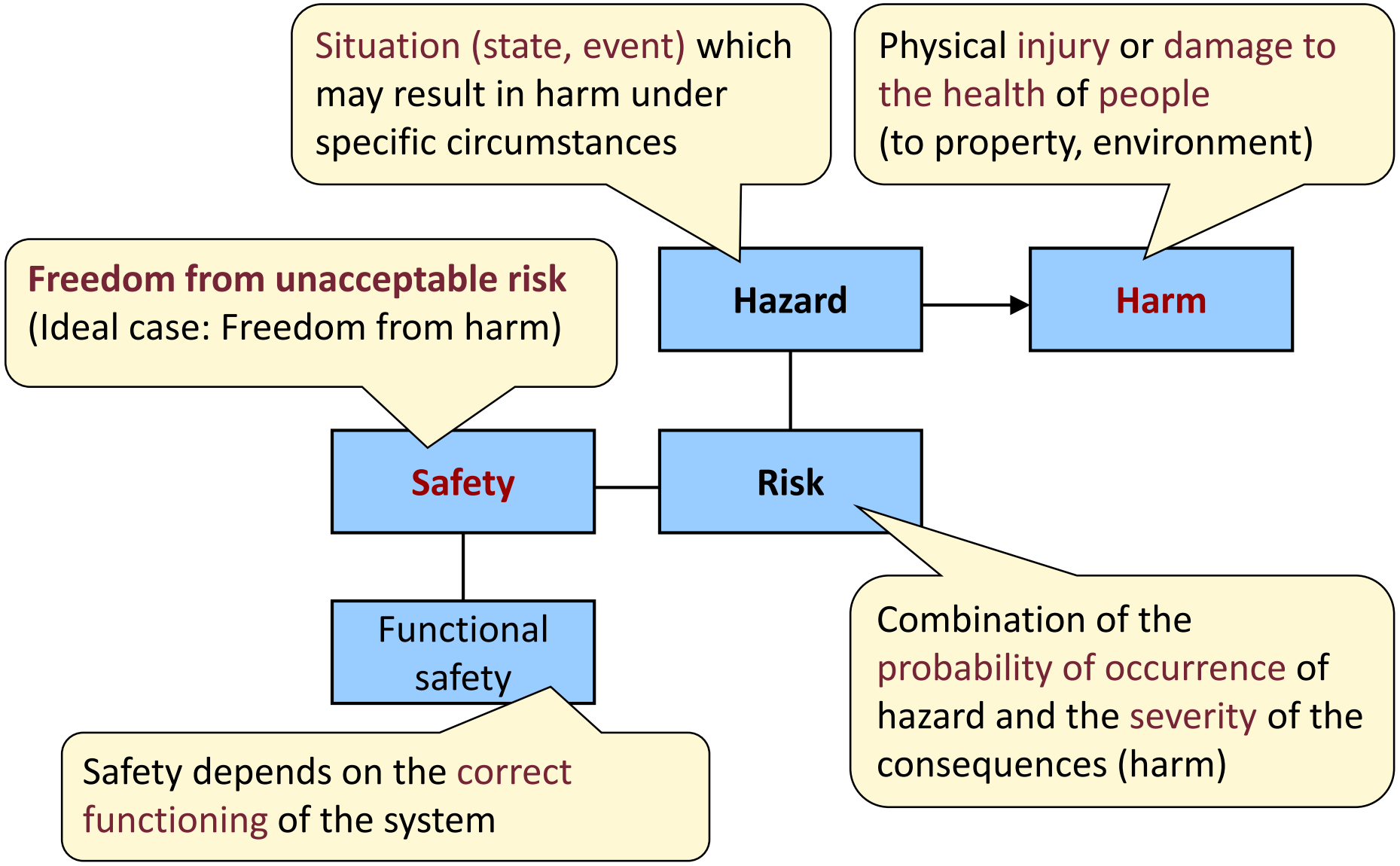
Terminology in the requirements



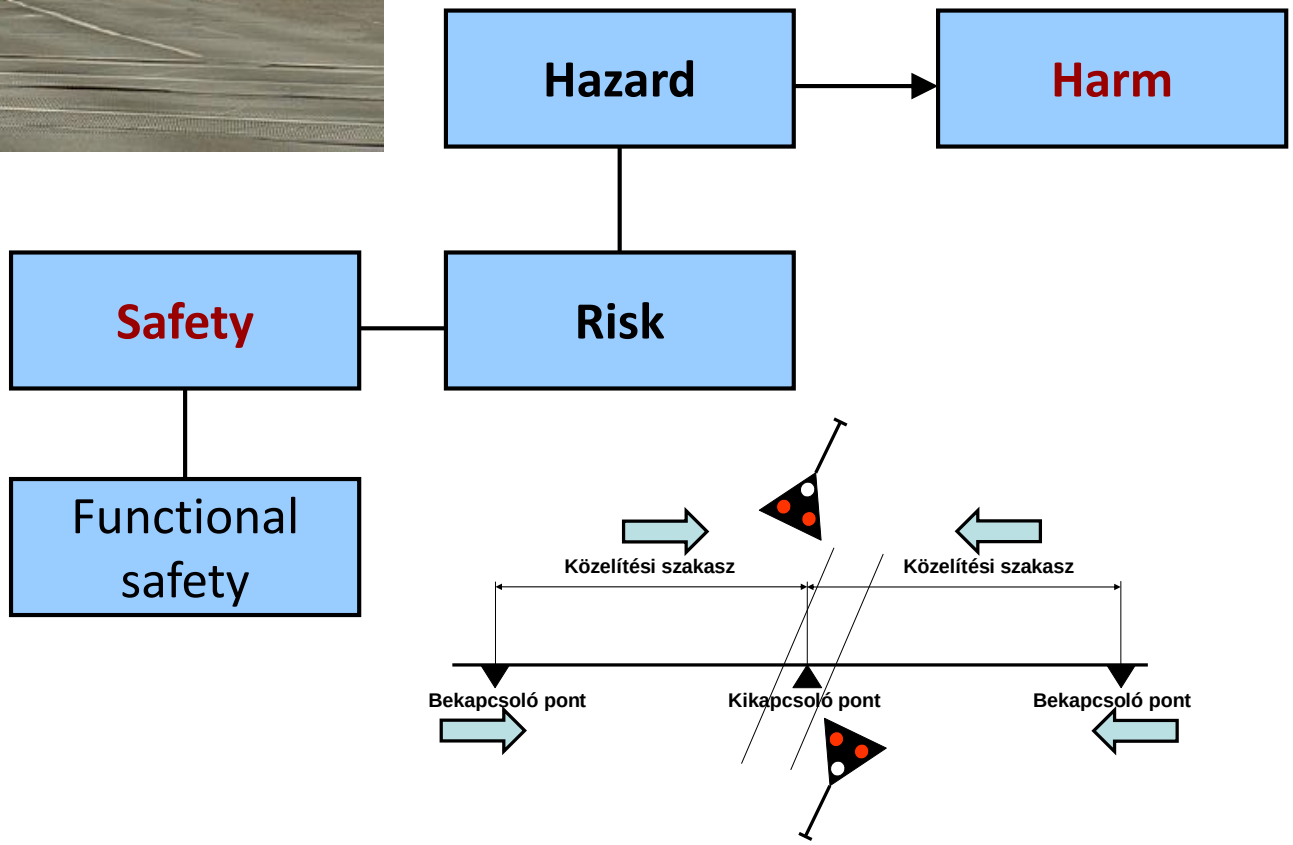
Risk categories



Terminology in the requirements



Example: Application of the terminology



What we have to specify?

Safety function requirements

- Function which is intended to achieve or maintain a safe state for the EUC
  - In other words: What the system shall do in order to avoid / control the hazard
- (Part of the) functional requirements specification

Safety integrity requirements

- Probability that the safety-related system satisfactorily performs the required safety functions (without failure)
- Probabilistic approach to safety
  - Example 1: Buildings are designed to survive earthquake that occurs with probability >10% in 50 years
  - Example 2: Dams are designed to withhold the highest water measured in the last 100 years

Safety integrity requirements

- Integrity depending on the mode of operation
  - Low demand mode: Average probability of failure to perform the desired function on demand
  - High demand (continuous) mode: Average rate of failure to perform the desired function
    (rate: failure per hour)
- High demand mode: Tolerable Hazard Rate (THR)

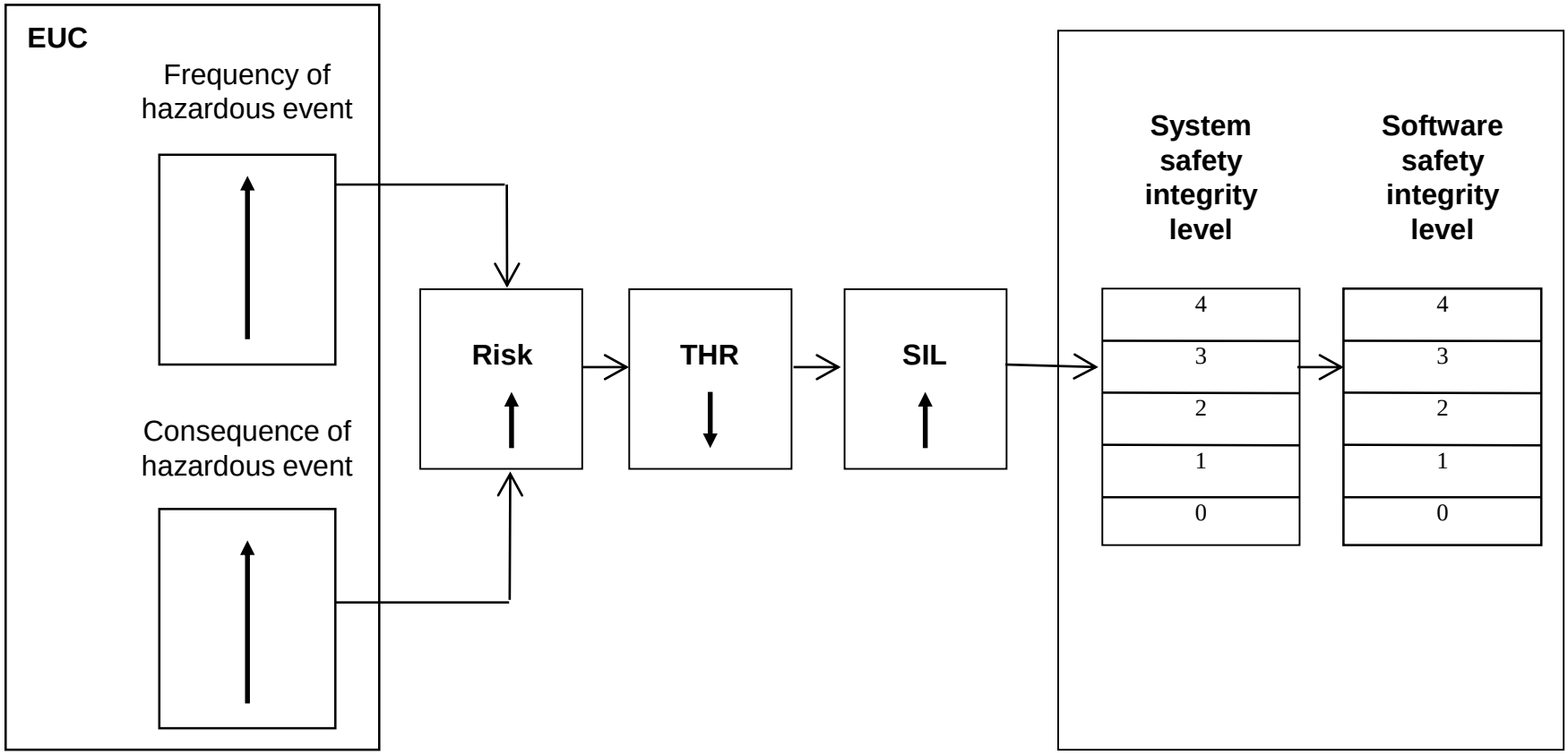
If the lifetime is 15 years then 1 equipment will fail out of the 750 equipments

SIL	Failure of a safety function per hour
1	$10^{-6} \leq THR < 10^{-5}$
2	$10^{-7} \leq THR < 10^{-6}$
3	$10^{-8} \leq THR < 10^{-7}$
4	$10^{-9} \leq THR < 10^{-8}$

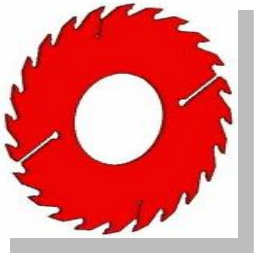
Operation without failures in more than 11.000 years??

Determining SIL: Overview

- Hazard identification and risk analysis -> Target failure measure



Example: Safety requirements

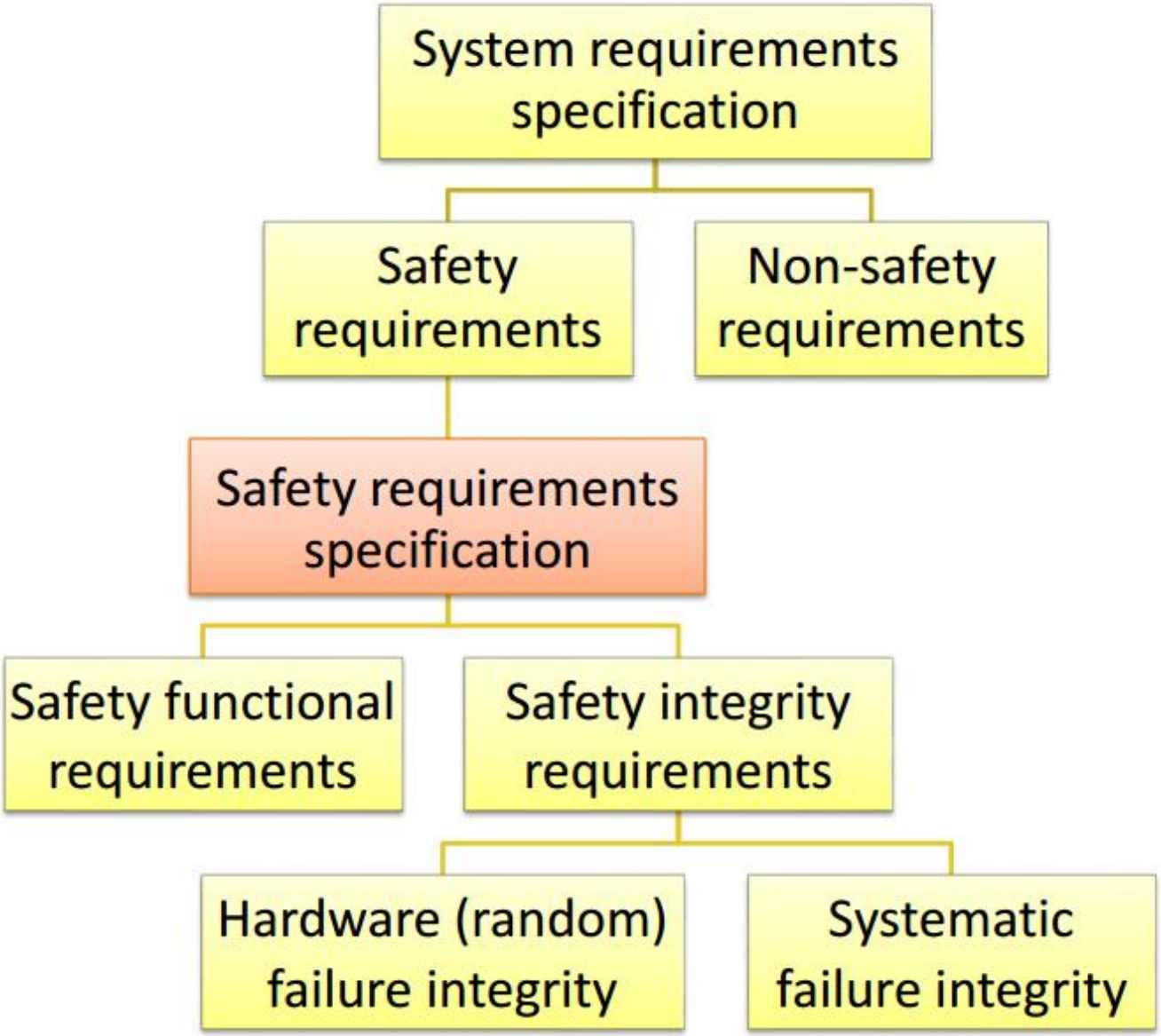


- Machine with a rotating blade and a solid cover
  - Cleaning of the blade: Lifting of the cover is needed
- Risk analysis: Injury of the operator
  when cleaning the blade while the motor is rotating
  - Hazard: If the cover is lifted more than 50 mm and the motor does not stop in 1 sec
  - There are 20 machines, during the lifetime 500 cleaning is needed for each machine; it is tolerable only once that the motor is not stopped
- Safety function: Interlocking
  - Safety function requirement: When the cover is lifted to 15 mm, the motor shall be stopped and braked in 0.8 sec
- Safety integrity requirement:
  - The probability of failure of the interlocking (safety function) shall be less than 10^{-4} (one failure in 10.000 operation)

Satisfying safety integrity requirements

- Failures that influence safety integrity:
  - Random (hardware) failures: Occur accidentally at a random time due to degradation mechanisms
  - Systematic (software) failures: Occur in a deterministic way due to design / manufacturing / operating flaws
- Achieving safety integrity:
  - Random failure integrity: Selection of components (considering failure parameters) and the system architecture
  - Systematic failure integrity: Rigor in the development
    - Development life cycle: Well-defined phases
    - Techniques and measures: Verification, testing, measuring, …
    - Documentation: Development and operation
    - Independence of persons: Developer, verifier, assessor, …
- Safety case:
  - Documented demonstration that the product complies with the specified safety requirements

Summary: Structure of requirements



Dependability related requirements

(When safety is not enough)

Characterizing the system services

- Typical extra-functional characteristics
  - Reliability, availability, integrity, ...
  - These depend on the faults occurring during the use of the services
- Composite characteristic: Dependability

Definition: Ability to provide service in which reliance can justifiably be placed

  - Justifiably: based on analysis, evaluation, measurements
  - Reliance: the service satisfies the needs
  - Basic question: How to avoid or handle the faults affecting the services?

Threats to dependability

Development process → **Product in operation**

- Design faults
- Implementation faults

- Hardware faults
- Configuration faults
- Operator faults

Verification during the development

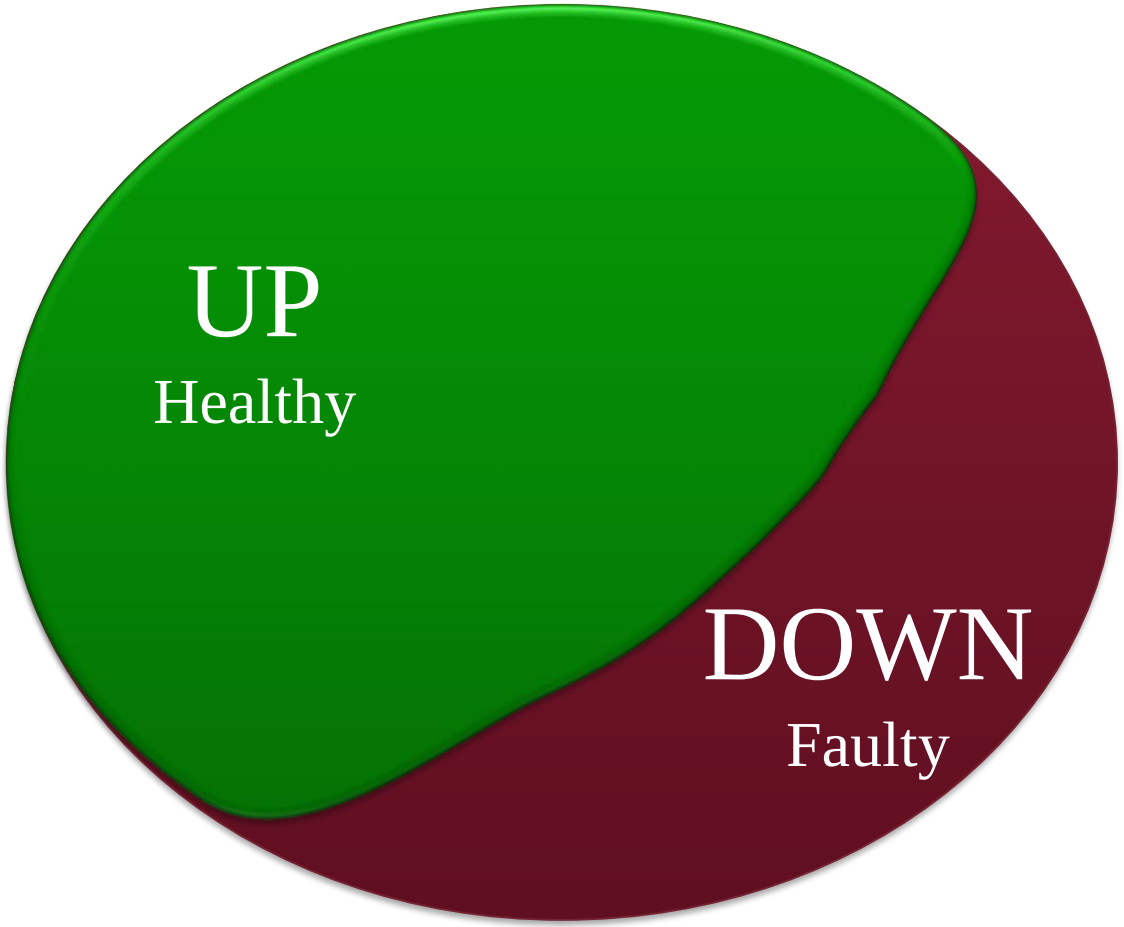
Fault tolerance during operation

Attributes of dependability

Attribute	Definition
Availability	Probability of correct service (considering repairs and maintenance) "Availability of the web service shall be 95%"
Reliability	Probability of continuous correct service (until the first failure) "After departure the onboard control system shall function correctly for 12 hours"
Safety	Freedom from unacceptable risk of harm
Integrity	Avoidance of erroneous changes or alterations
Maintainability	Possibility of repairs and improvements

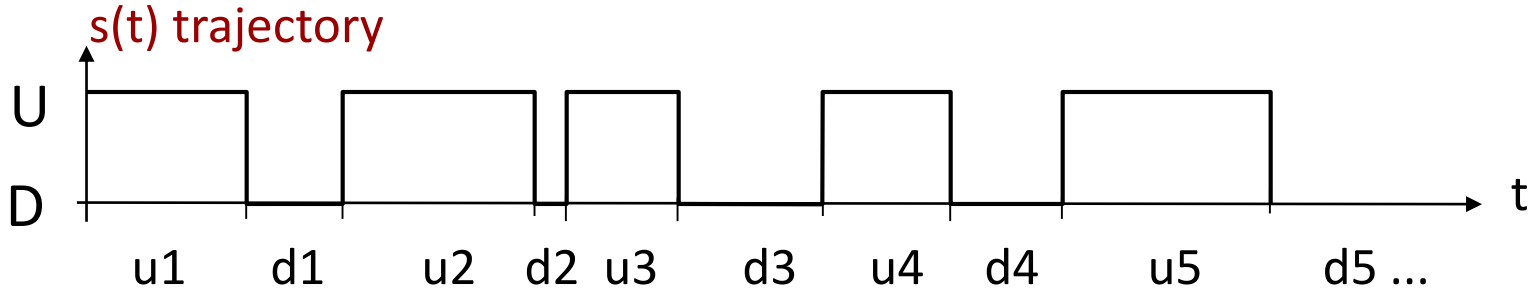
State partitions

- S: state space of the system



Dependability metrics: Mean values

- Basis: Partitioning the states of the system
  - Correct (U, up) and incorrect (D, down) state partitions



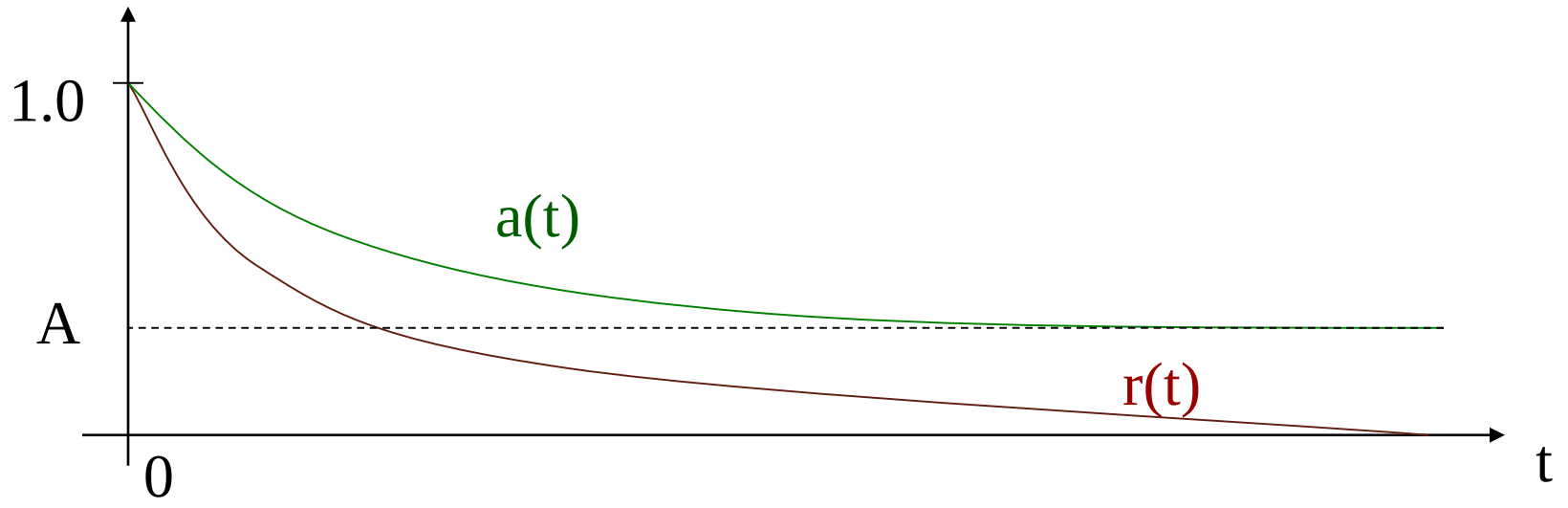
- Mean values:
  - Mean Time to First Failure: MTFF = E{u1}
  - Mean Up Time: MUT = MTTF = E{ui}
    (Mean Time To Failure)
  - Mean Down Time: MDT = MTTR = E{di}
    (Mean Time To Repair)
  - Mean Time Between Failures: MTBF = MUT + MDT

Dependability metrics: Probability functions

- Availability: $a(t) = P\{s(t) \in U\}$
- Asymptotic availability: $A = \lim_{t \to \infty} a(t)$

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

- Reliability: $r(t) = P\{s(t') \in U, \forall t' < t\}$



Availability related requirements

Availability	Failure period per year
99%	~ 3,5 days
99,9%	~ 9 hours
99,99% („4 nines”)	~ 1 hour
99,999% („5 nines”)	~ 5 minutes
99,9999% („6 nines”)	~ 32 sec
99,99999%	~ 3 sec

Availability of a system built up from components,
where the availability of single a component is 95%,
and all components are needed to perform the system function:

- Availability of a system built from 2 components: 90%
- Availability of a system built from 5 components : 77%
- Availability of a system built from 10 components : 60%

Attributes of components

- Fault rate: $\lambda(t)$

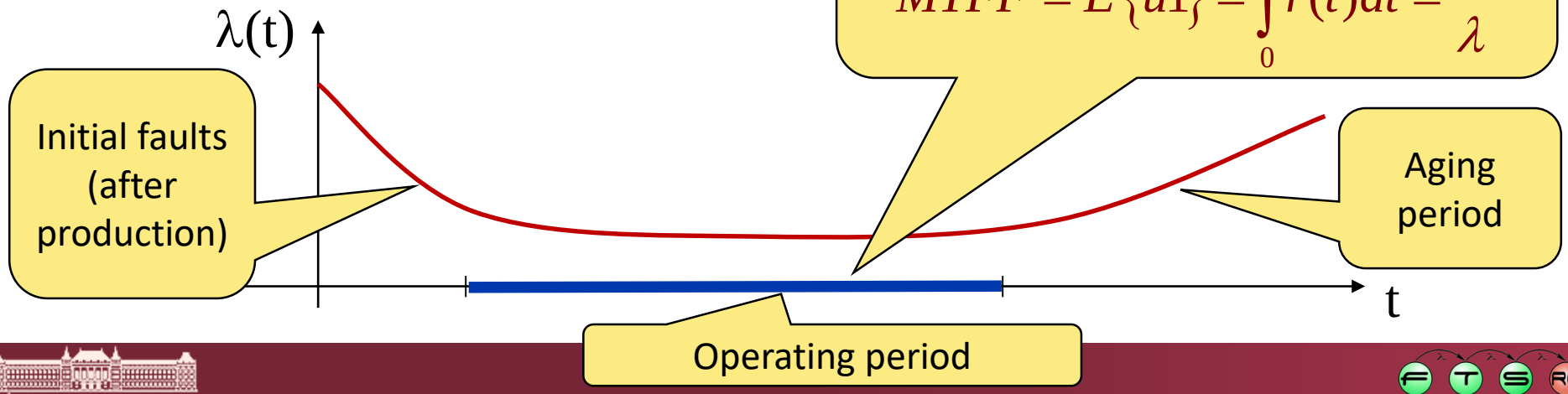
  Probability that the component will fail at time point t
  given that it has been correct until t

$$\lambda(t)\Delta t = P\{s(t+\Delta t) \in D \mid s(t) \in U\} \text{ while } \Delta t \to 0$$

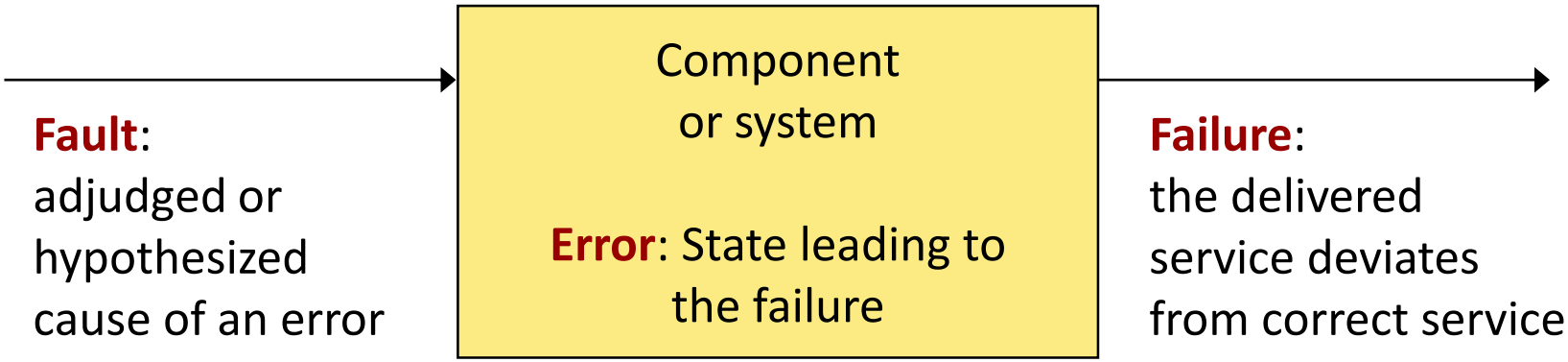
- Reliability of a component on the basis of this definition:

$$r(t) = e^{-\int_0^t \lambda(t)dt} \quad \Leftrightarrow \lambda(t) = -r'(t) /$$

- For electronic components:



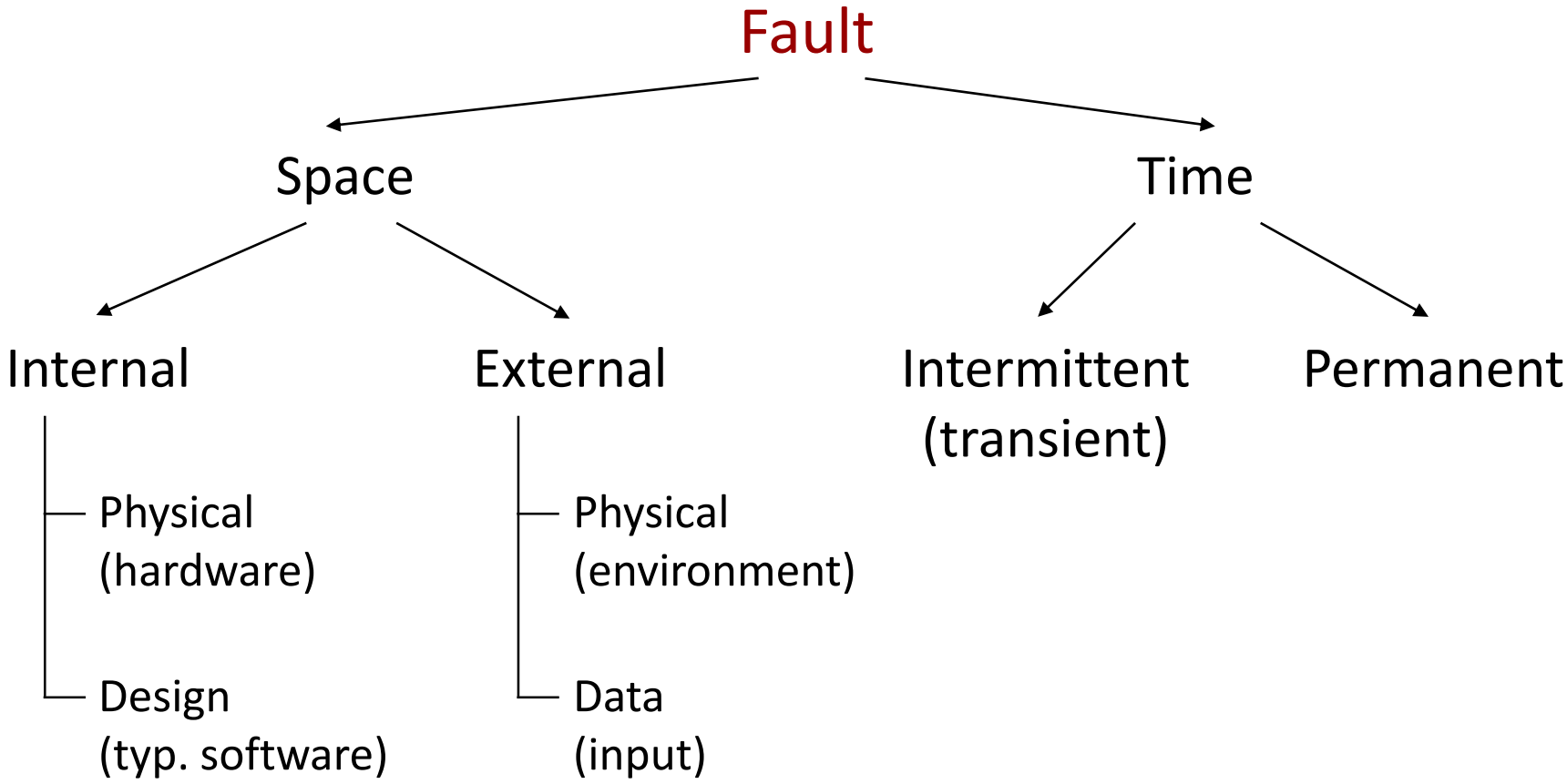
Threats to dependability



Fault → Error → Failure examples:

Fault	Error	Failure
Bit flip in the memory due to a cosmic particle →	Reading the faulty memory cell will result in incorrect value →	The robot arm collides with the wall
The programmer increases a variable instead of decreasing →	The faulty statement is executed and the value of the variable will be incorrect →	The final result of the computation will be incorrect

The characteristics of faults



Software fault:

- Permanent design fault (systematic)
- Activation of the fault depends on the operational profile (inputs)

Means to improve dependability

- Fault prevention:
  - Physical faults: Good components, shielding, ...
  - Design faults: Good design methodology
- Fault removal:
  - Design phase: Verification and corrections
  - Prototype phase: Testing, diagnostics, repair
- Fault tolerance: Avoiding service failures
  - Operational phase: Fault handling, reconfiguration
- Fault forecasting: Estimating faults and their effects
  - Measurements and prediction
    E.g., Self-Monitoring, Analysis and Reporting Technology (SMART)

Summary

- Safety requirements
  - Basic concepts: Hazard, risk, safety
  - Safety integrity
- Dependability requirements
  - Attributes of dependability
  - Quantitative attributes (definitions): reliability and availability
  - The fault – error – failure chain
  - Means to improve dependability: fault prevention, fault removal, fault tolerance, fault forecasting

Safety architecture

Previous topics

- What we specified?
  - Safety function requirements: Function which is intended to achieve or maintain a safe state
  - Safety integrity requirements: Probability of a safety-related system satisfactorily performing the required safety functions (i.e., without failure)
- Safety Integrity Level and component fault rates
  - SIL 4: 10^{-8} ...10^{-9} faults per hour
  - Typical electronic components: 10^{-5}…10^{-6} faults/hour
  - Typical software: 1..10 faults per 1000 line of code

Goals

- Safety critical systems study block
  1. Requirements in critical systems: Safety, dependability
  2. Architecture design (patterns) in critical systems
  3. Evaluation of system architecture

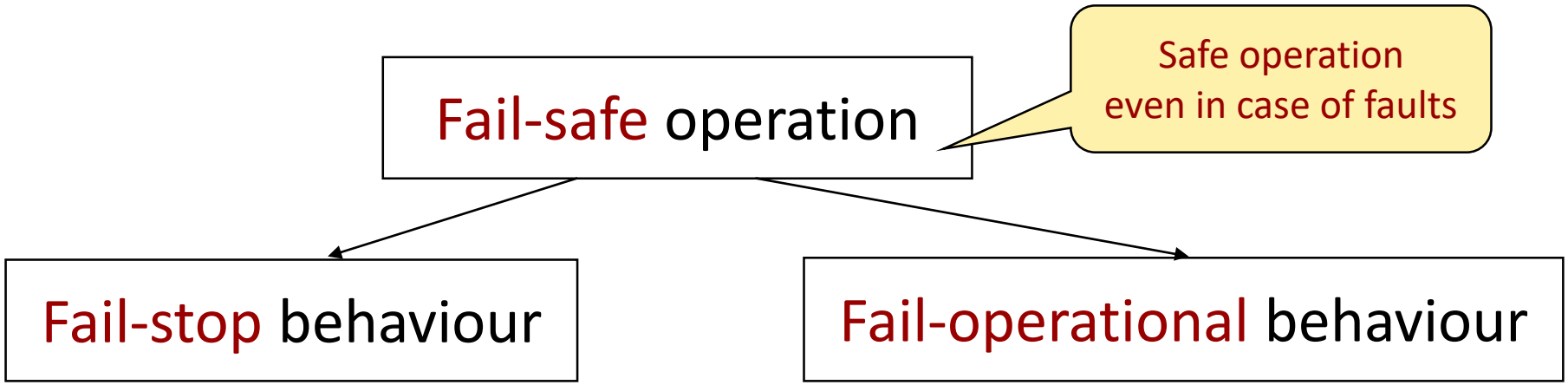
- Focus: Design of system architecture to ...
  - maintain safety
  - handle the effects of faults in hardware and software components

Learning objectives

Architecture design in safety critical systems

- Understand the role of architecture
- Know the typical architecture level solutions for error detection in case of fail-stop behavior
- Propose solutions for fault tolerance in case of
  - Permanent hardware faults
  - Transient hardware faults
  - Software faults
- Understand the time and resource overhead of the different architecture patterns

Objectives of architecture design



- Stopping (switch-off) is a safe state
- In case of a detected error the system has to be stopped
- Error detection is required

- Stopping (switch-off) is not a safe state
- Service is needed even in case of a detected error
  - full service
  - degraded (but safe) service
- Fault tolerance is required

Objectives of architecture design

Fail-safe operation

Safe operation
even in case of faults

Fail-stop behaviour

- Stopping (switch-off) is a safe state
- In case of a detected error the system has to be stopped
- Error detection is required

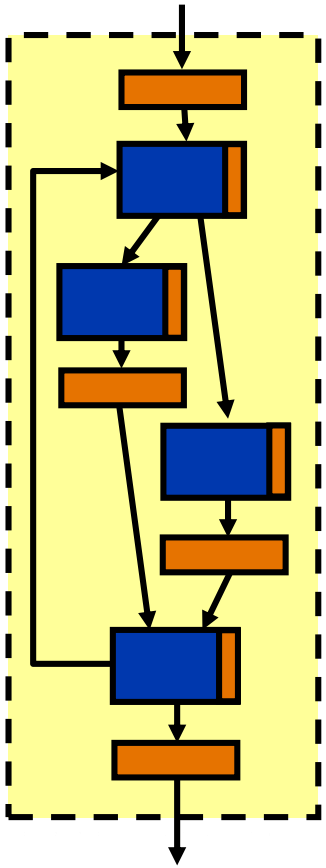
Fail-operational behaviour

- Stopping (switch-off) is not a safe state
- Service is needed even in case of a detected error
  - full service
  - degraded (but safe) service
- Fault tolerance is required

Typical architectures for fail-stop operation

1. Single channel architecture with built-in self-test

- Single processing flow with error detection
- Scheduled hardware self-tests
  - After switch-on: Detailed self-test
  - In run-time: On-line tests
- Online software self-checking
  - Typically application dependent techniques
  - Checking the control flow, data acceptance rules, timeliness properties
- Disadvantages
  - Fault coverage of the self-tests is limited
  - Fault handling (e.g., switch-off) shall be performed by the checked channel

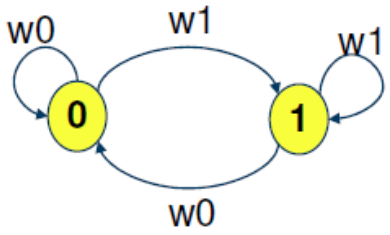


Implementation of on-line error detection

- Application dependent (ad-hoc) techniques
  - Acceptance checking (e.g.: too low, too high value)
  - Timing related checking (e.g.: too early, too late)
  - Cross-checking (e.g.: using inverse function)
  - Structure checking (e.g.: broken structure)
- Application independent (platform) mechanisms
  - Hardware supported on-line checking
    - CPU level: Invalid instruction, user/supervisor modes etc.
    - MMU level: Protection of memory ranges
  - OS level checking
    - Invalid parameters of system calls
    - OS level protection of resources

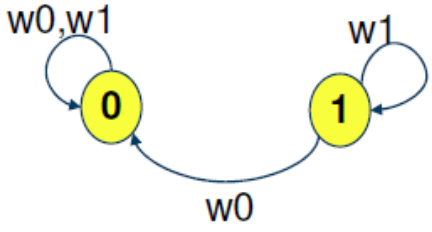
Example: Testing memory cells (hw)

States of a correct cell to be checked:

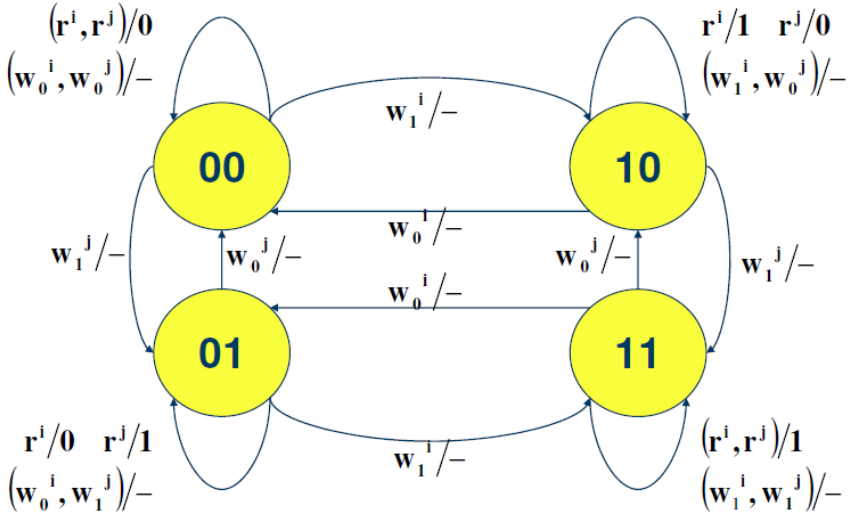


States in case of stuck-at 0/1 faults:

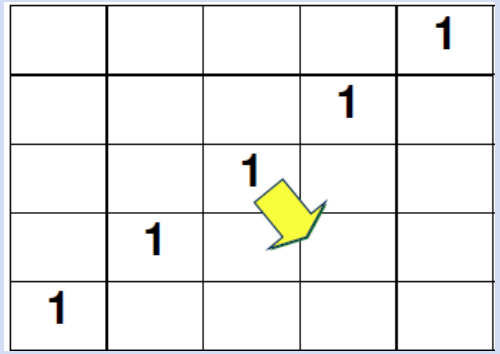
States in case of transition fault:



States of two correct (adjacent) cells to be checked:



Testing: „March" algorithms (w/r)



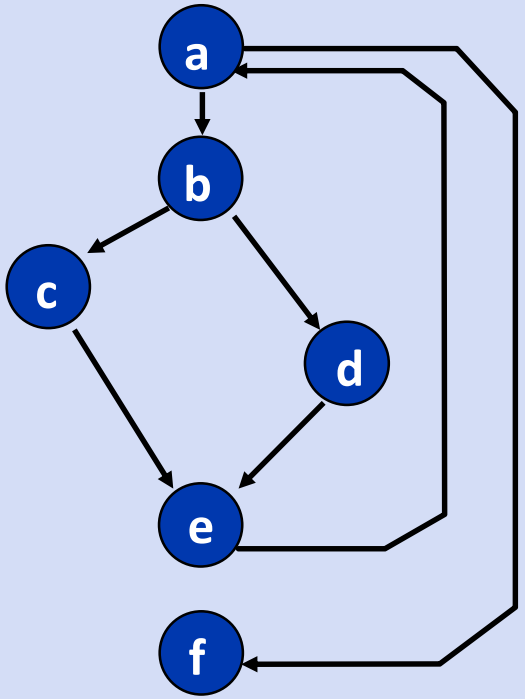
Example: Checking execution flow (sw)

- Checking the correctness of statement sequence
  - Reference for correct behavior: Program control flow graph

Source code:

```
a:  for (i=0; i<MAX; i++) {
b:      if (i==a) {
c:          n=n-i;
        } else {
d:          m=m-i;
        }
e:      printf("%d\n",n);
    }
f:  printf("Ready.")
```

Control flow graph:



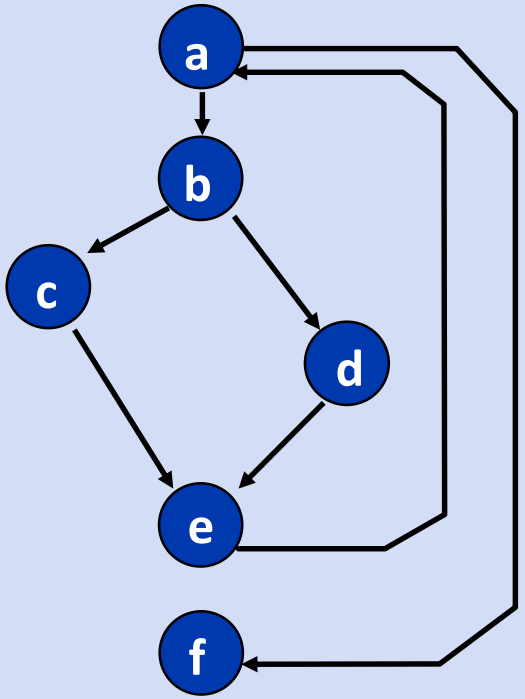
Example: Checking execution flow (sw)

- Checking the correctness of statement sequence
  - Reference for correct behavior: Program control flow graph
  - Instrumentation: Signatures to be checked in runtime

Instrumented source code:

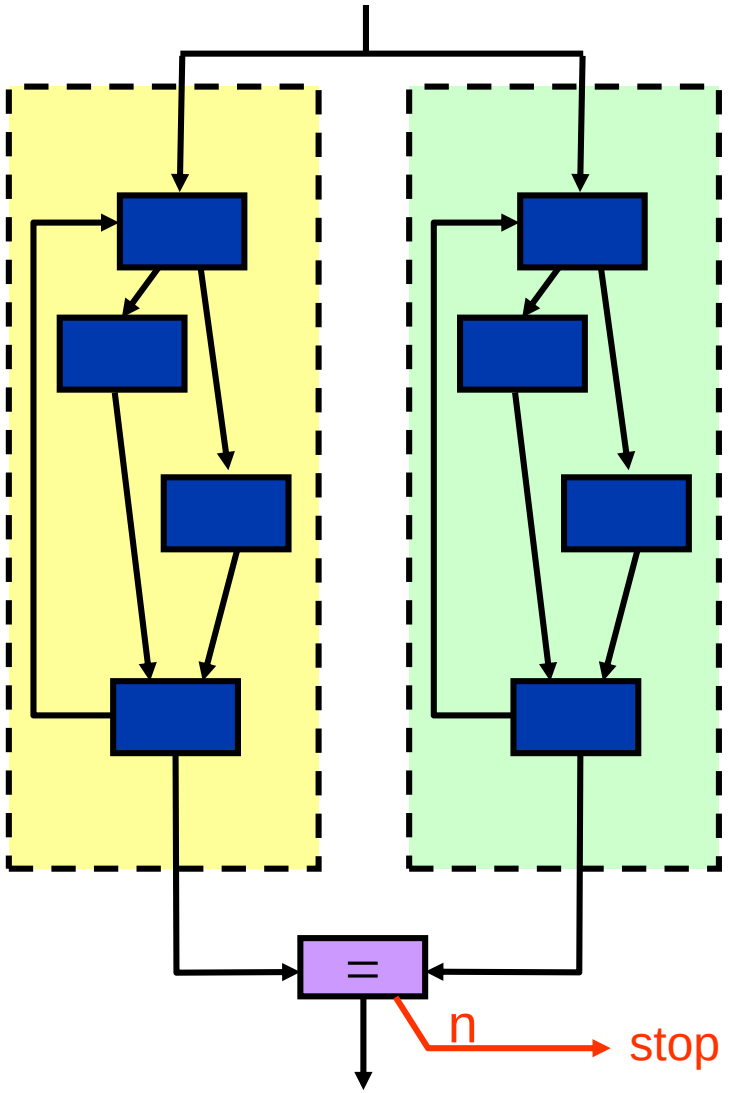
```
a:  S(a); for (i=0; i<MAX; i++) {
b:      S(b); if (i==a) {
c:          S(c); n=n-i;
        } else {
d:          S(d); m=m-i;
        }
e:      S(e); printf("%d\n",n);
    }
f:  S(f); printf("Ready.")
```

Control flow graph:

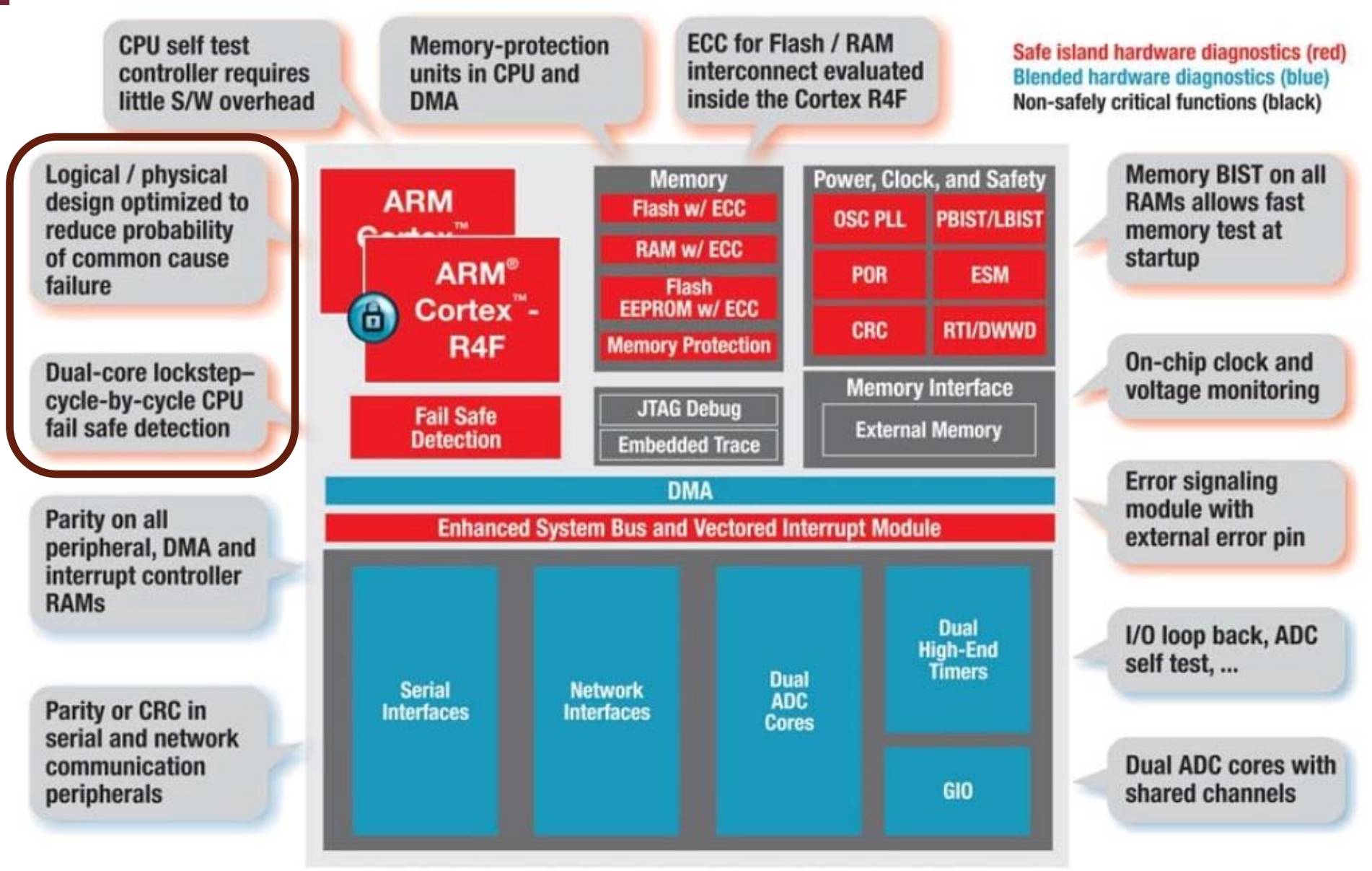


2. Two-channels architecture with comparison

- Two or more processing channels
  - Shared input
  - Comparison of outputs
  - Stopping in case of deviation
- High error detection coverage
  - The comparator is a critical component (but simple)
- Disadvantages:
  - Common mode faults
  - Long detection latency

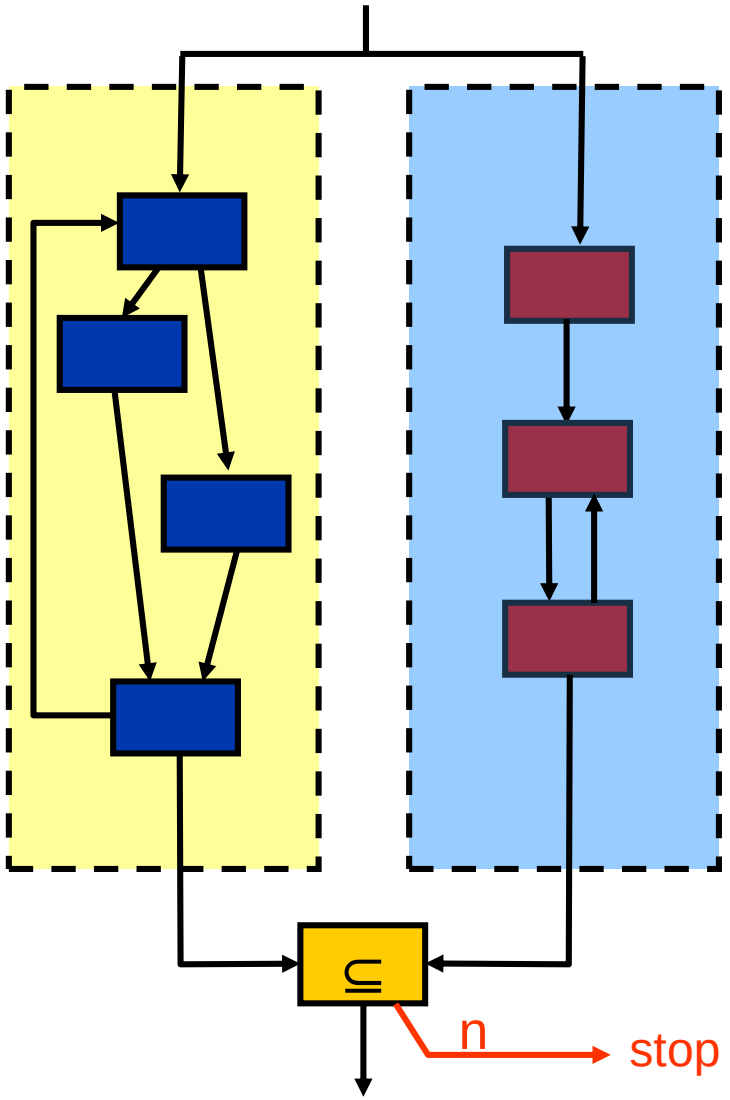


Example: TI Hercules Safety Microcontrollers

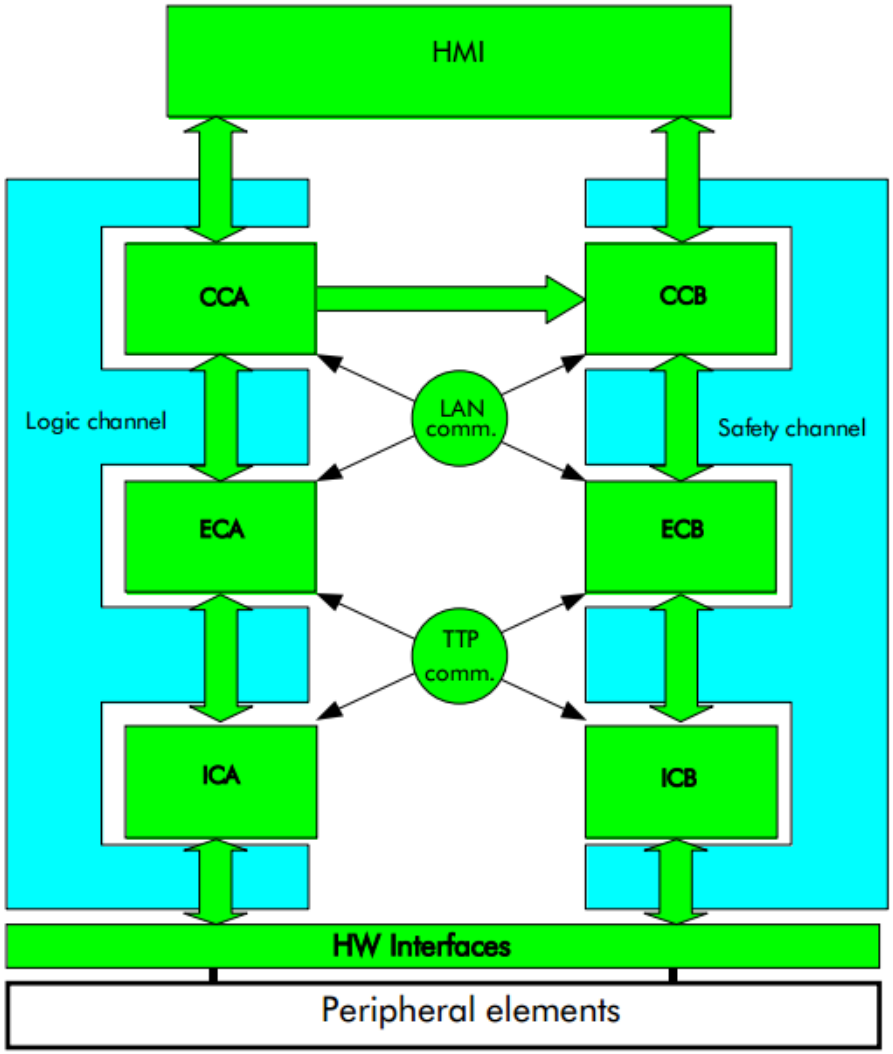


3. Two-channels architecture with safety checking

- Independent second channel
  - Safety bag: only safety checking
  - Diverse implementation
  - Checking the output of the primary channel
- Advantages
  - Explicit safety rules
  - Independence of the checker channel



Example: Elektra interlocking system



HMI

Central Controller

Field Element Controller

Two channels:

- Logic channel: CHILL (CCITT High Level Language) procedure-oriented programming language
- Safety channel: PAMELA (Pattern Matching Expert System Language) rule-based language

Typical architectures for fault-tolerant systems

Objectives of architecture design

Fail-safe operation

Fail-stop behaviour

- Stopping (switch-off) is a safe state
- In case of a detected error the system has to be stopped
- Error detection is required

Fail-operational behaviour

- Stopping (switch-off) is not a safe state
- Service is needed even in case of a detected error
  - full service
  - degraded (but safe) service
- Fault tolerance is required

Fault tolerant systems

- Fault tolerance: Providing (safe) service in case of faults
  - Intervening into the fault → error → failure chain
    - Detecting the error and assessing the damage
    - Involving extra resources to perform corrections / recovery
    - Providing correct service without failure
    - (Providing degraded service in case of insufficient resources)
- Extra resources: Redundancy
  - Hardware
  - Software
  - Information
  - Time

} resources (sometimes together)

Categories of redundancy

- Forms of redundancy:
  - Hardware redundancy
    - Extra hardware components (inherent in the system or planned for fault tolerance)
  - Software redundancy
    - Extra software modules
  - Information redundancy
    - Extra information (e.g., error correcting codes)
  - Time redundancy
    - Repeated execution (to handle transient faults)
- Types of redundancy
  - Cold: The redundant component is inactive in fault-free case
  - Warm: The redundant component has reduced load
  - Hot: The redundant component is active in fault-free case

Overview: How to use the redundancy?

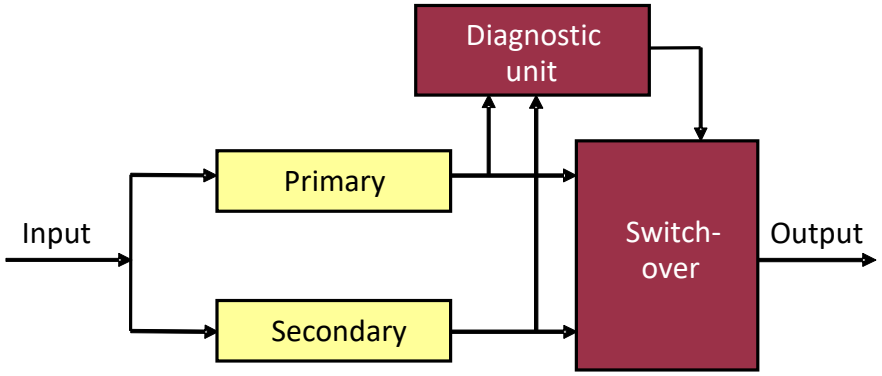
- Hardware design faults: (< 1%)
  - Hardware redundancy with design diversity
- Hardware permanent operational faults: (~ 20%)
  - Hardware redundancy (e.g.: redundant processor)
- Hardware transient operational faults: (~70-80%)
  - Time redundancy (e.g.: instruction retry)
  - Information redundancy (e.g.: error correcting codes)
  - Software redundancy (e.g.: recovery from saved state)
- Software design faults: (~ 10%)
  - Software redundancy with design diversity

1. Fault tolerance for hardware permanent faults

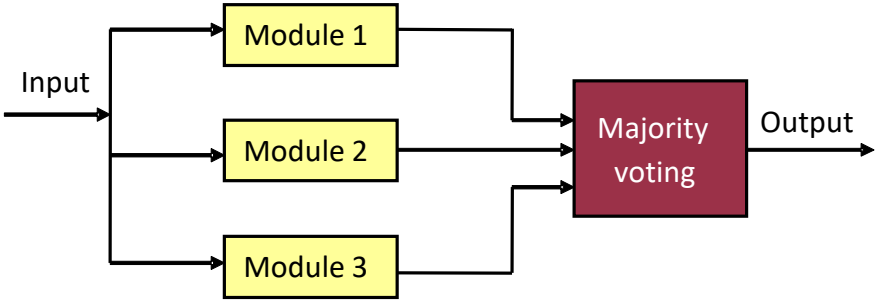
With diversity in case of considering design faults

Replication:

- Duplication with diagnostics:
  - Error detection by comparison
  - With diagnostic unit:
    Fault tolerance by switch-over



- TMR: Triple Modular Redundancy
  - Masking the failure by majority voting
  - Voter is a critical component (but simple)



- NMR: N-modular redundancy
  - Masking the failure by majority voting
  - Mission critical systems: Surviving the mission time

2. Fault tolerance for transient hardware faults

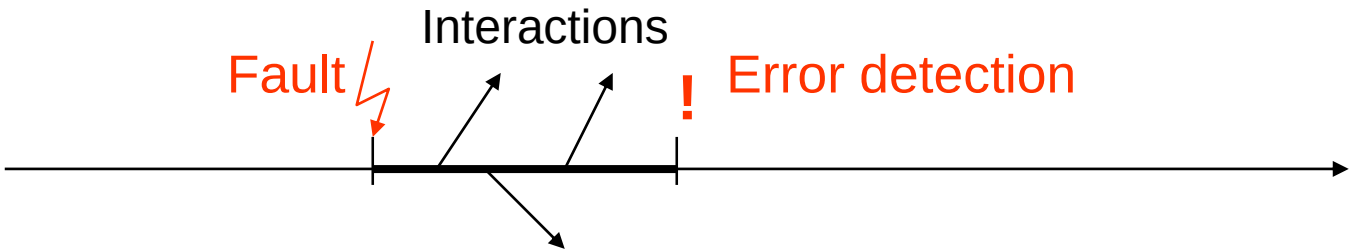
- Approach: Fault tolerance implemented by software
  - Detecting the error
  - Setting a fault-free state by handling the fault effects
  - Continuing the execution from that state (assuming that transient faults will not occur again)
- Four phases of operation:
  1) Error detection
  2) Damage assessment
  3) Recovery
  4) Fault treatment and continuing service

Phase 1: Error detection

- Application independent mechanisms:
  - E.g., detecting illegal instructions at CPU level
  - E.g., detecting violation of memory access restrictions
- Application dependent techniques:
  - Acceptance checking
  - Timing related checking
  - Cross-checking
  - Structure checking
  - Diagnostic checking
  - …

Phase 2: Damage assessment

- Motivation: Errors can propagate among the components between the occurrence and detection of errors



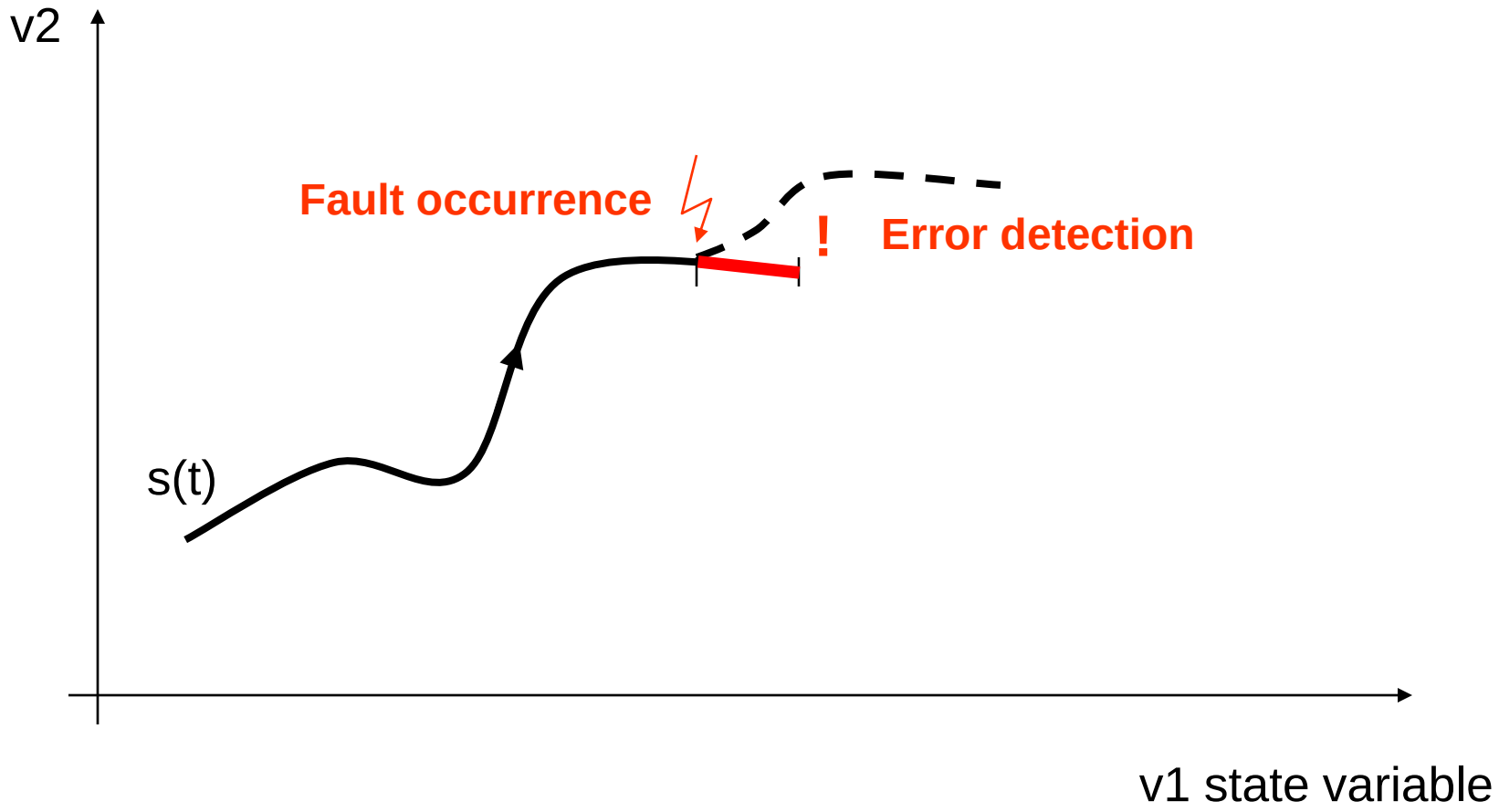
- Limiting error propagation: Checking interactions
  - Input acceptance checking (to detect external errors)
  - Output credibility checking (to provide „fail-silent” operation)
- Estimation of components affected by a detected error
  - Logging resource accesses and communication
  - Analysis of interactions (before error detection)

Phase 3: Recovery

- Forward recovery:
  - Setting an error-free state by selective correction
  - Dependent on the detected error and estimated damage
  - Used in case of anticipated faults
- Backward recovery:
  - Restoring a prior error-free state (that was saved earlier)
  - Independent of the detected error and estimated damage
  - State shall be saved and restored for each component
- Compensation:
  - The error can be handled by using redundant information

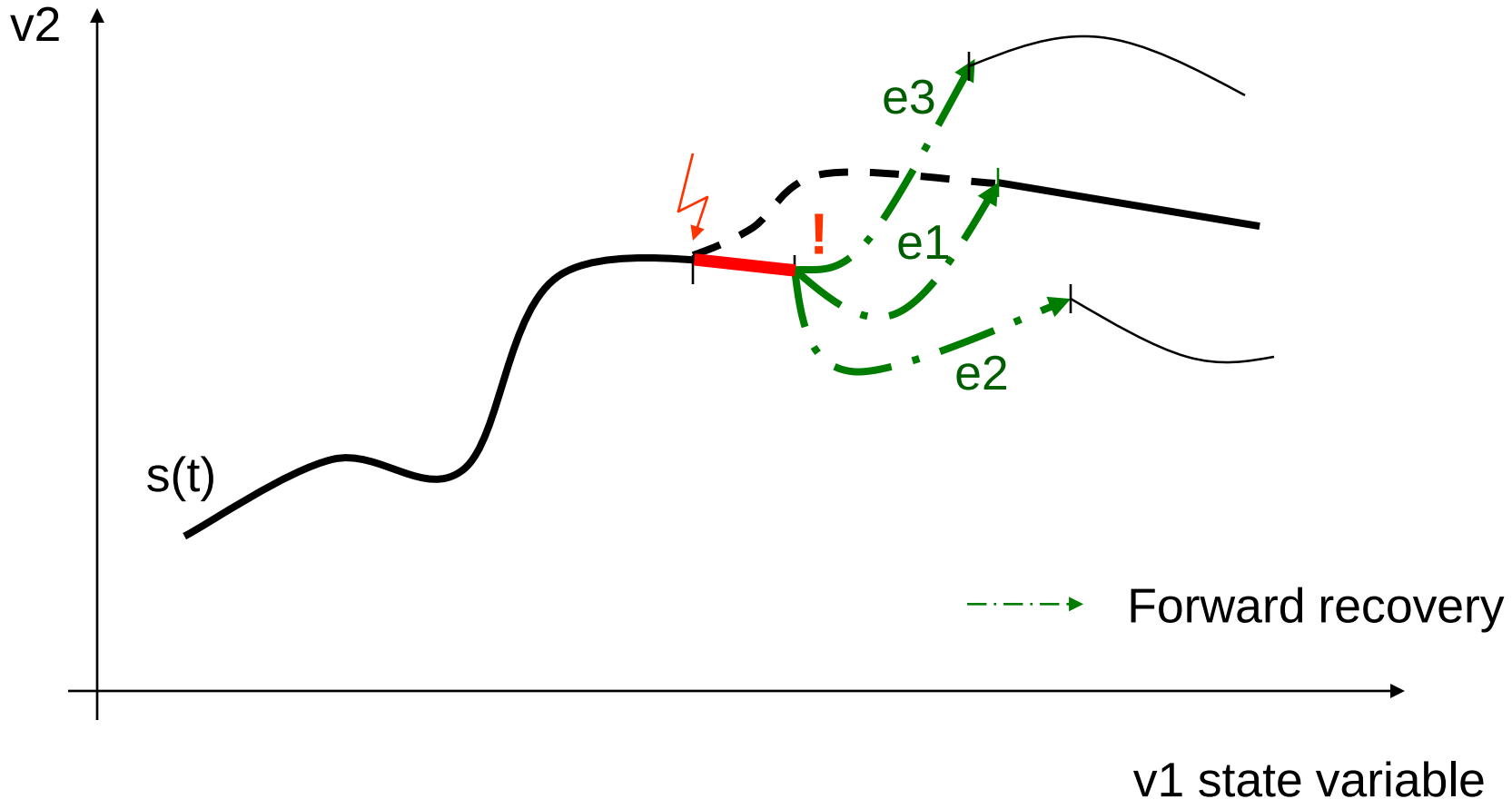
Types of recovery

- State space of the system: Error detection



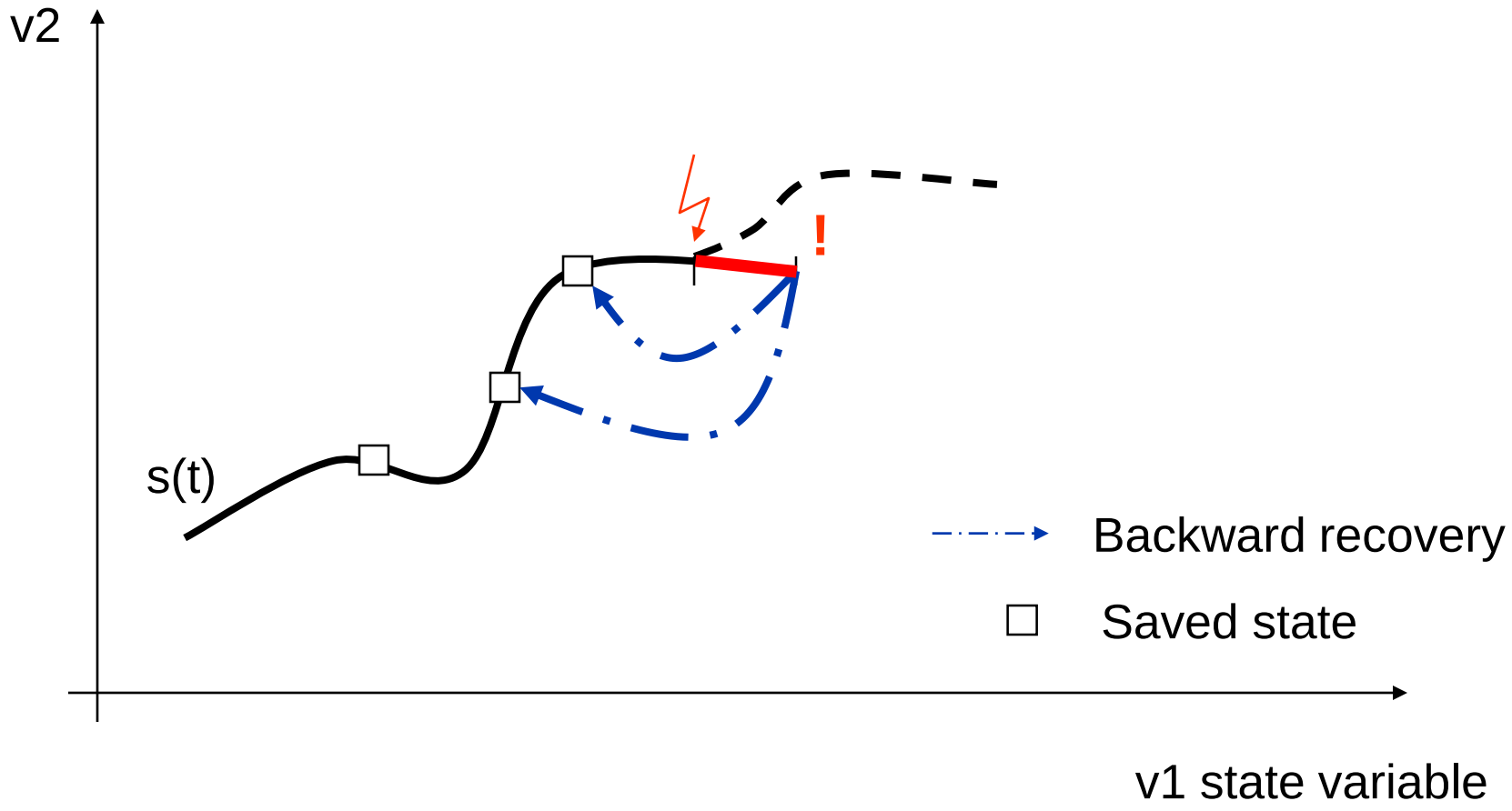
Types of recovery

- State space of the system: Forward recovery



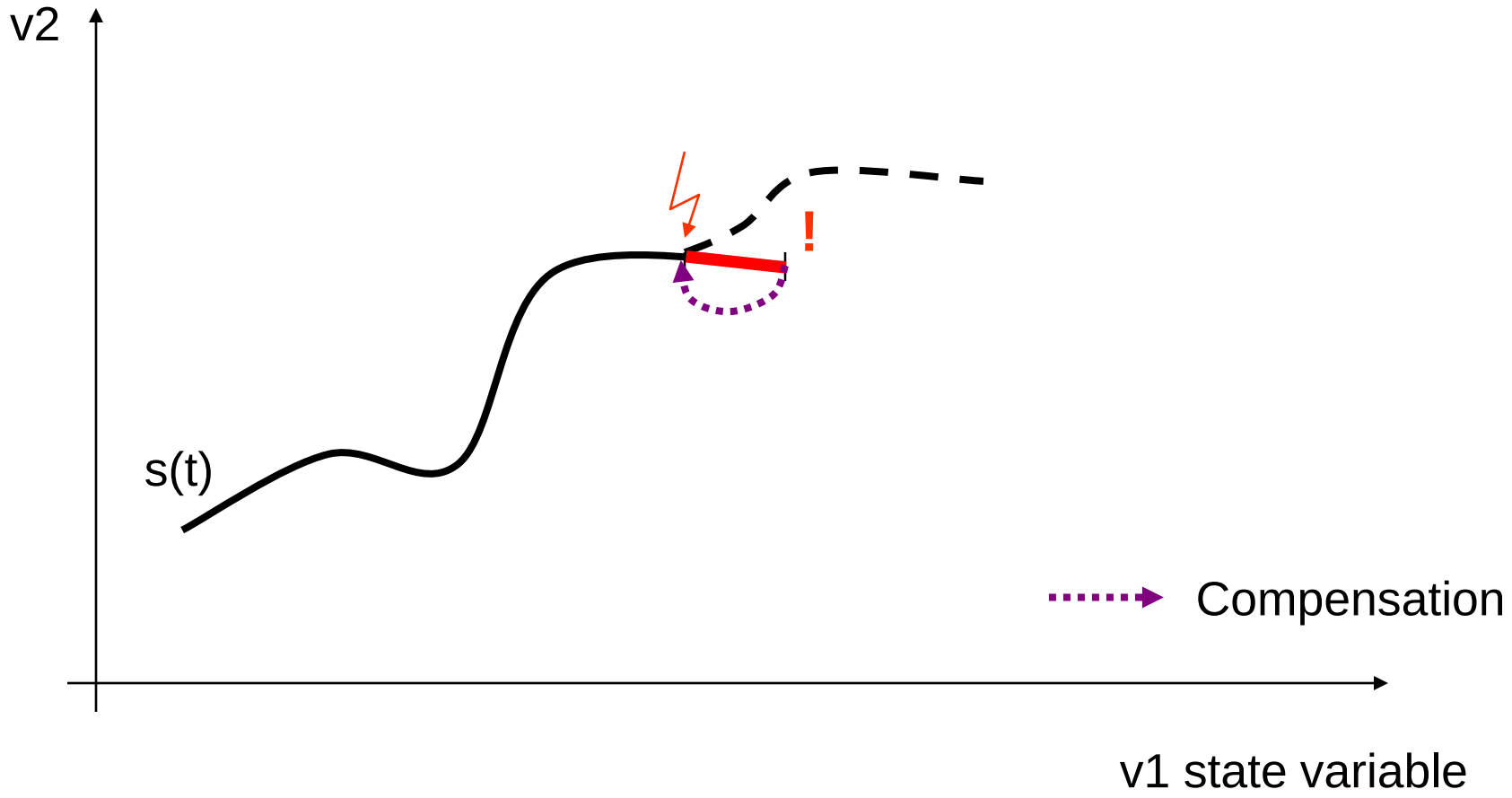
Types of recovery

- State space of the system: Backward recovery



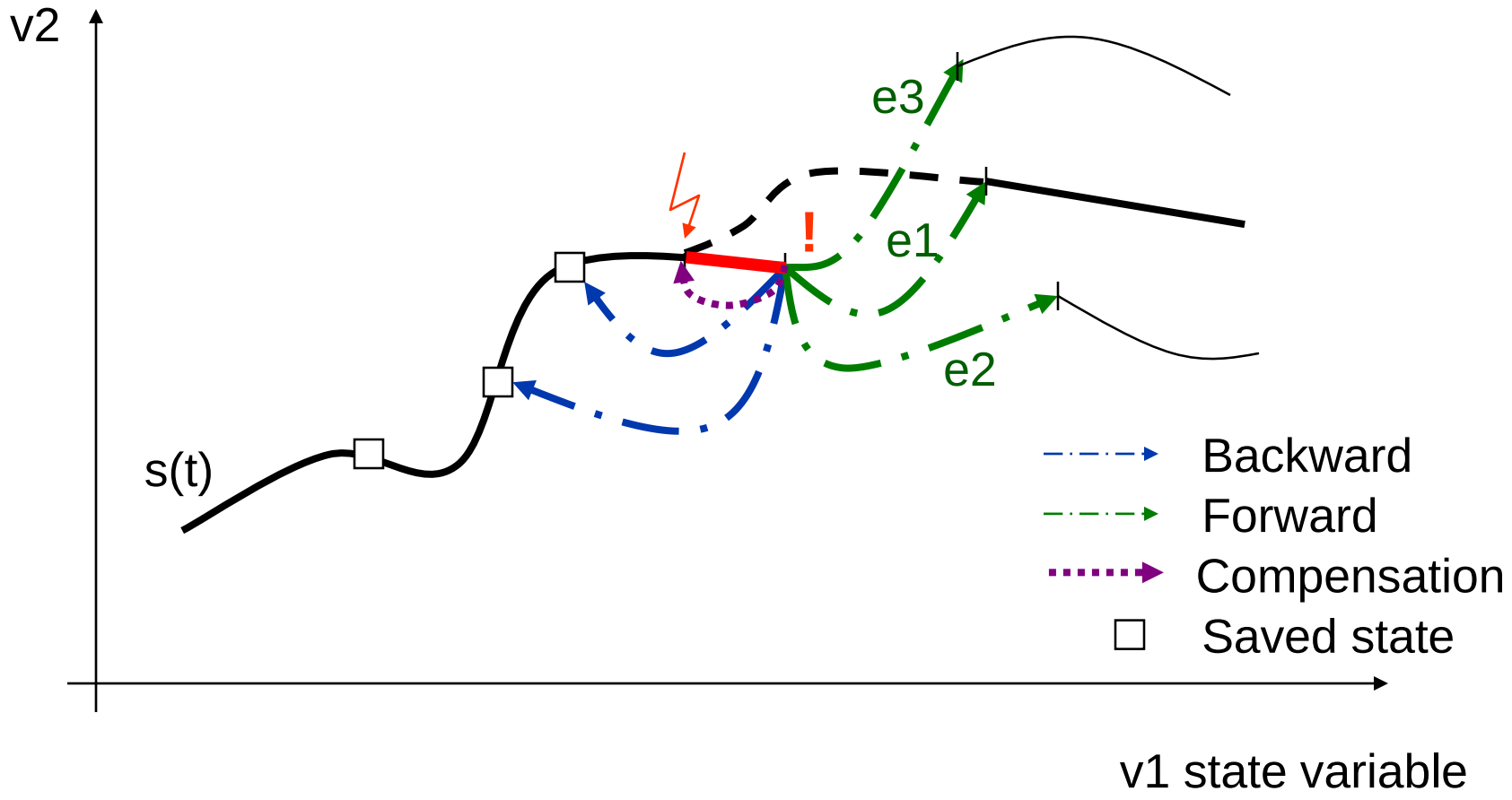
Types of recovery

- State space of the system: Compensation



Types of recovery

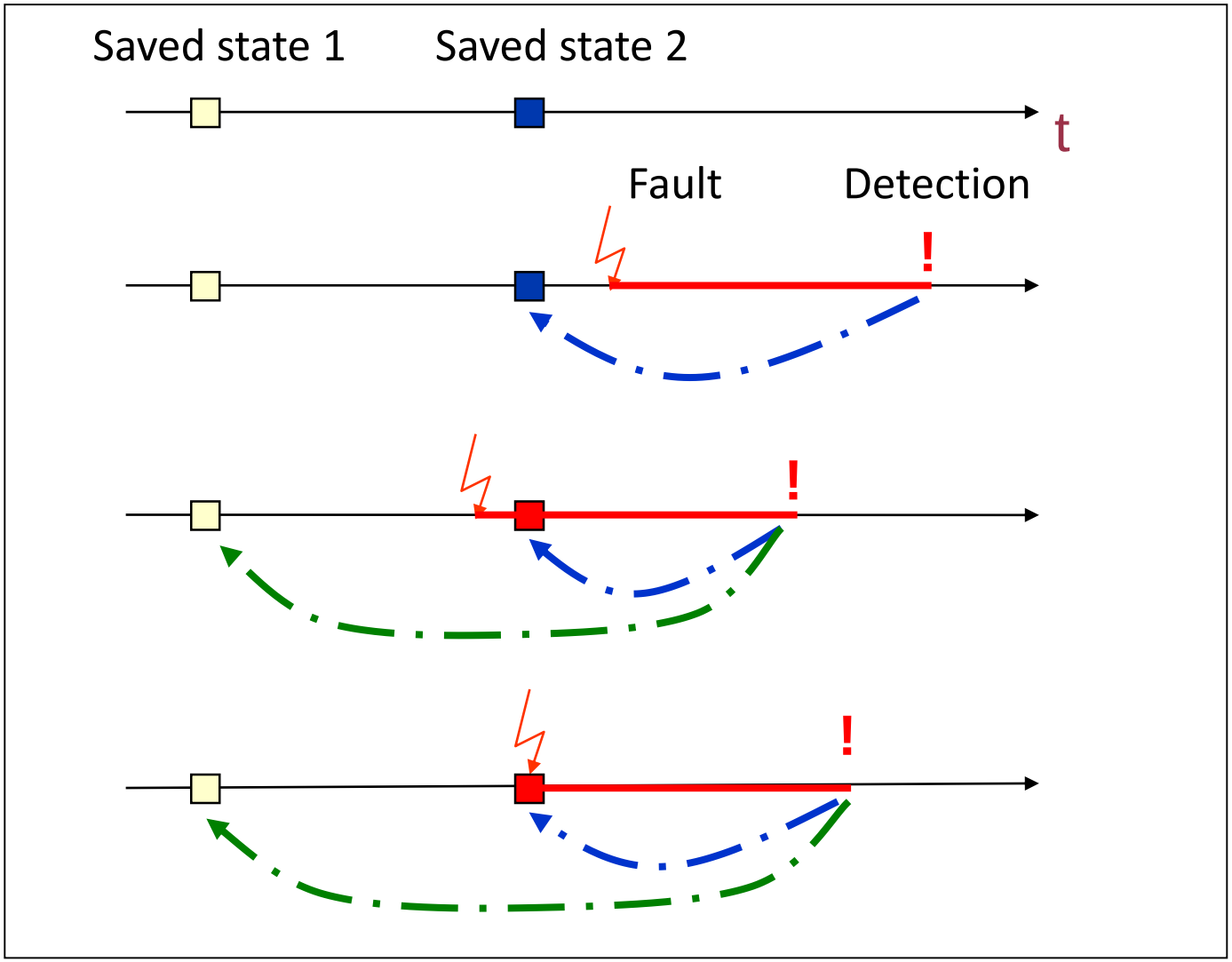
- State space of the system: Types of recovery



Backward recovery

- Backward recovery based on saved state
  - Checkpoint: The saved state
  - Checkpoint operations:
    - Save: copying the state periodically into stable storage
    - Recovery: restoring the state from the stable storage
    - Discard: deleting saved state after having more recent one(s)
  - Analogy: “autosave”
- Limited applicability: Based on operation logs
  - Error to be handled: unintended operation
  - Recovery is performed by the withdrawal of operations
  - Analogy: ”undo”

Scenarios of backward recovery



Phase 4: Fault treatment and continuing service

- For transient faults:
  - Handled by the forward or backward recovery
- For permanent faults:
  - Recovery is unsuccessful (the error is detected again)
  - The faulty component shall be localized and handled

Approach:

- Diagnostic checks to localize the fault
- Reconfiguration
  - Replacing the faulty component using redundancy
  - Degraded operation: Continuing only the critical services
- Repair and substitution

4. Fault tolerance for software faults

- Repeated execution is not effective for design faults!
- Redundancy with design diversity is required

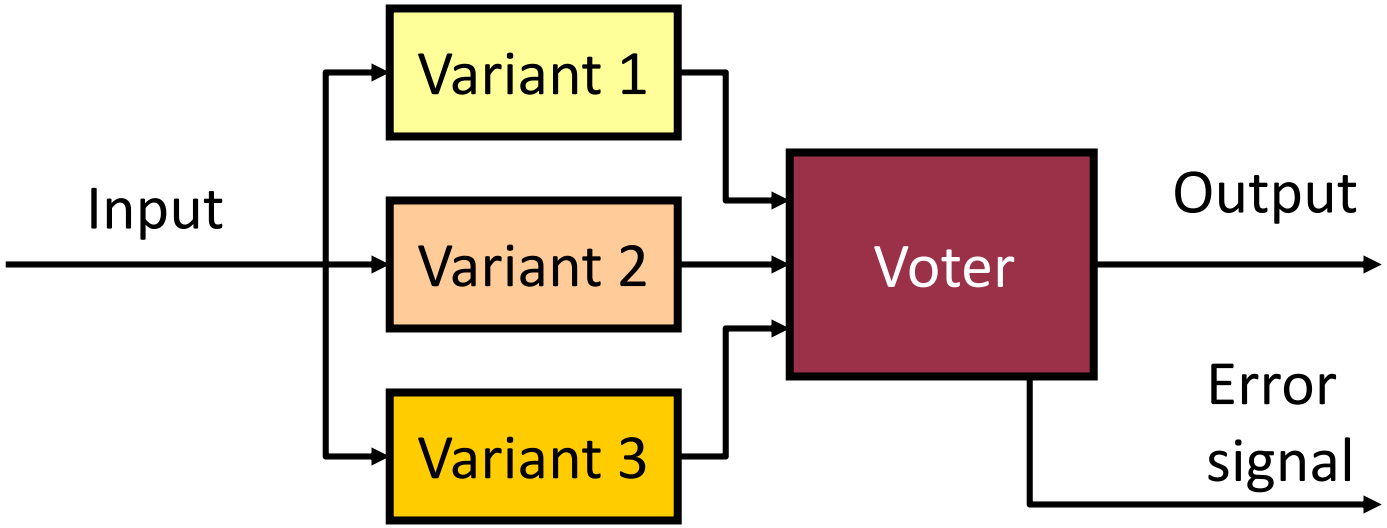
  Variants: Redundant software modules with
  - diverse algorithms and data structures,
  - different programming languages and development tools,
  - separated development teams

  in order to reduce the probability of common faults
- Execution of variants:
  - N-version programming
  - Recovery blocks

N-version programming

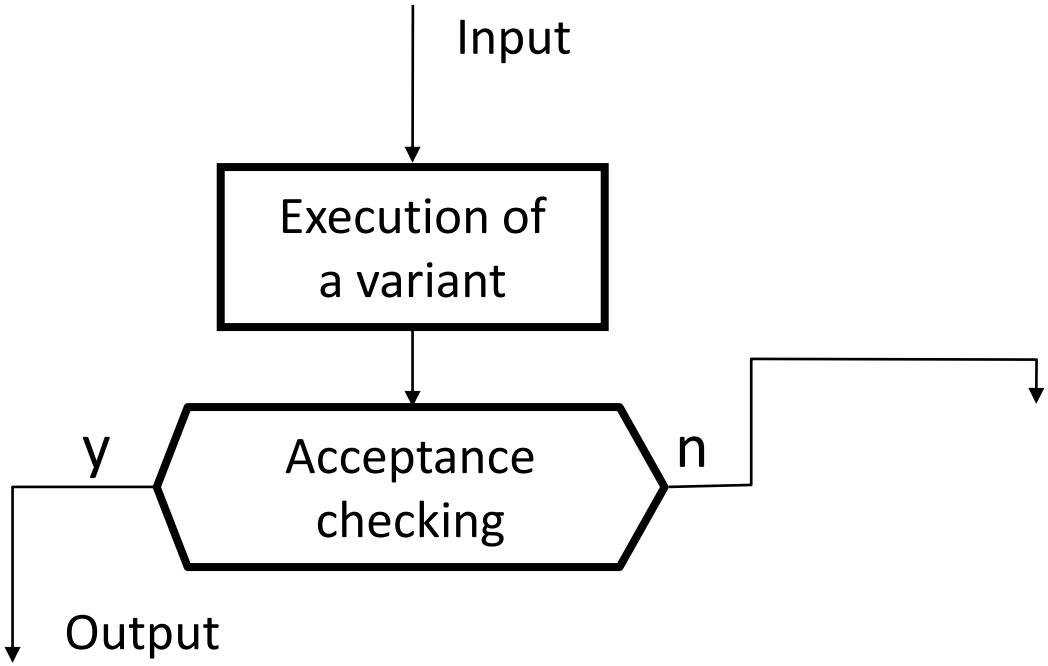
- Active redundancy:

  Each variant is executed (in parallel)
  - The same inputs are used
  - Majority voting is performed on the output
    - Acceptable range of difference shall be specified
    - The voter is a critical component (but simple)



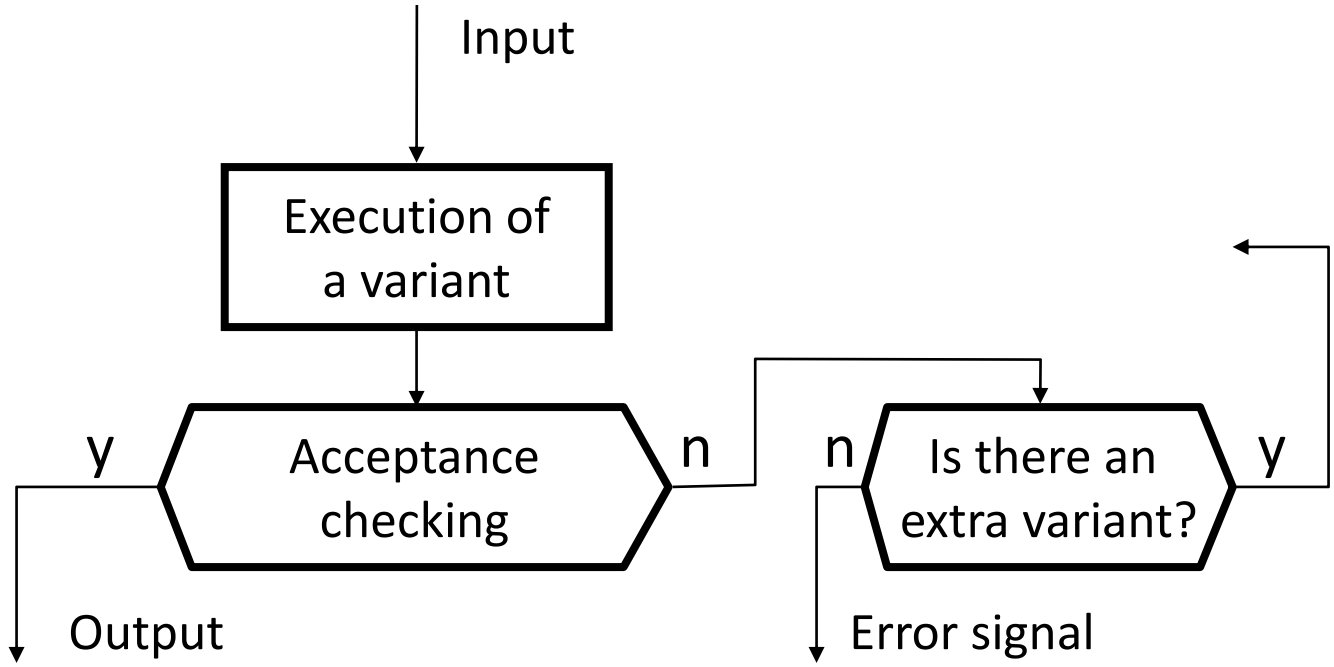
Recovery blocks

- Passive redundancy: Activation only in case of faults
  - The primary variant is executed first
  - Acceptance checking on the output of the variants
  - In case of a detected error another variant is executed



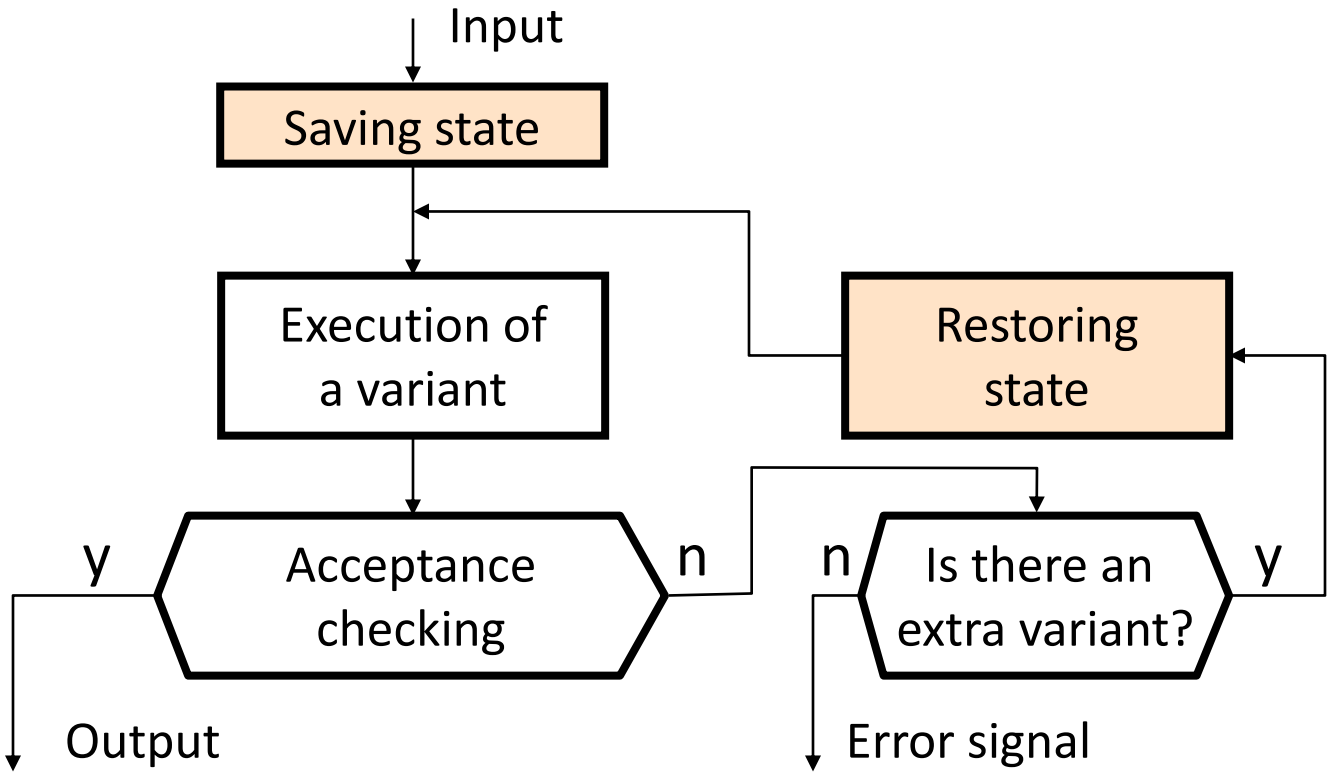
Recovery blocks

- Passive redundancy: Activation only in case of faults
  - The primary variant is executed first
  - Acceptance checking on the output of the variants
  - In case of a detected error another variant is executed



Recovery blocks

- Passive redundancy: Activation only in case of faults
  - The primary variant is executed first
  - Acceptance checking on the output of the variants
  - In case of a detected error another variant is executed



Comparison of the techniques

Property/Type	N-version programming	Recovery blocks
Error detection	Majority voting, relative	Acceptance checking, absolute
Execution of variants	Parallel	Serial
Execution time	Slowest variant (or time-out)	Depending on the number of faults
Activation of redundancy	Always (active)	Only in case of fault (passive)
Number of tolerated faults	[(N-1)/2]	N-1

Summary

Summary: Techniques of fault tolerance

1. Hardware design faults
   - Diverse redundant components
2. Hardware permanent operational faults
   - Replicated components: TMR, NMR
3. Hardware transient operational faults
   - Fault tolerance implemented by software
     1. Error detection
     2. Damage assessment
     3. Recovery: Forward or backward recovery (or compensation)
     4. Fault treatment
   - Information redundancy: Error correcting codes
   - Time redundancy: Repeated execution (retry, reload, restart)
4. Software design faults
   - Variants as diverse redundant components (NVP, RB)