Architecture of Safety Critical Systems

Design and Integration of Embedded Systems

István Majzik

Department of Measurement and Information Systems

Goals

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

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

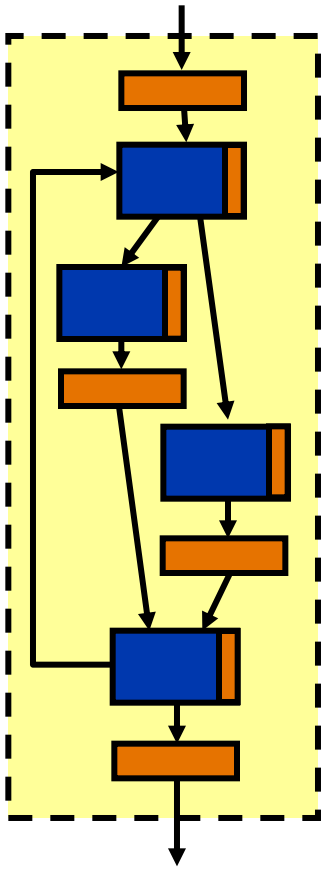
Fail-operational behaviour

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

Typical architectures for fail-stop operation

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

- Single processing flow with error detection
- Scheduled hardware self-tests
  - After switch-on: Detailed self-test
  - In run-time: Periodic on-line tests
- Online software error detection
  - 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

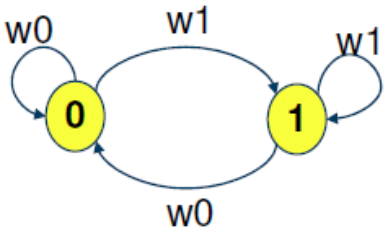


Implementation of on-line error detection

- 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 data structure)
- Application independent (platform) mechanisms
  - Hardware supported on-line checking
    - CPU level: Invalid instruction, user/supervisor modes etc.
    - MPU level: Protection of memory ranges
  - OS level checking
    - Invalid parameters of system calls
    - OS level protection of resources (locking, authorization etc.)

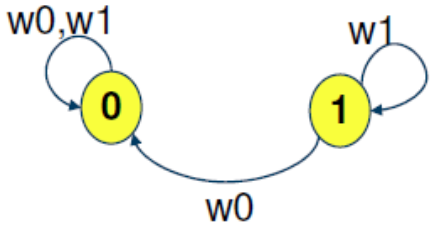
Example: Testing memory cells (hw)

States of a correct cell to be checked:

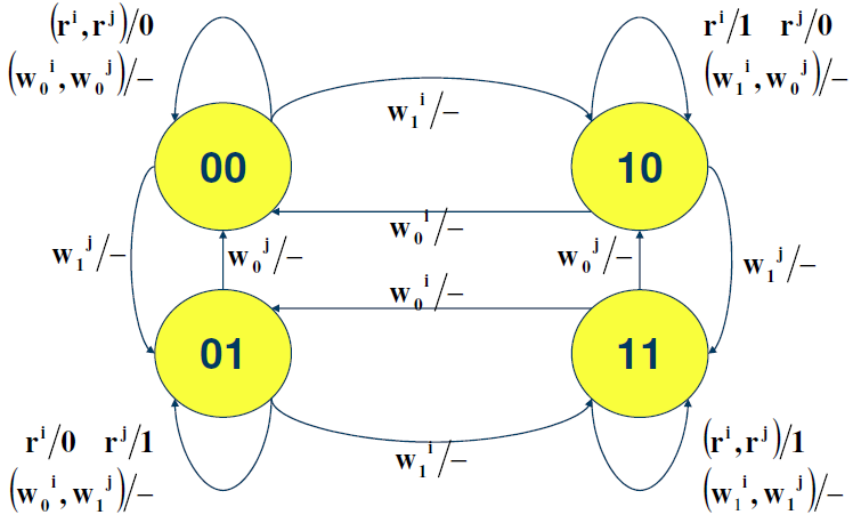


- Observed in case of stuck-at 0/1 faults:

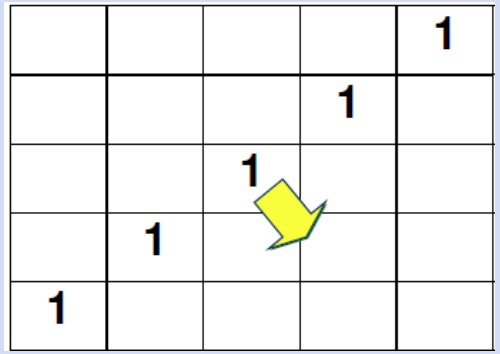
- Observed if w1 transition fault:



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



Testing by „marching" algorithms (w/r)



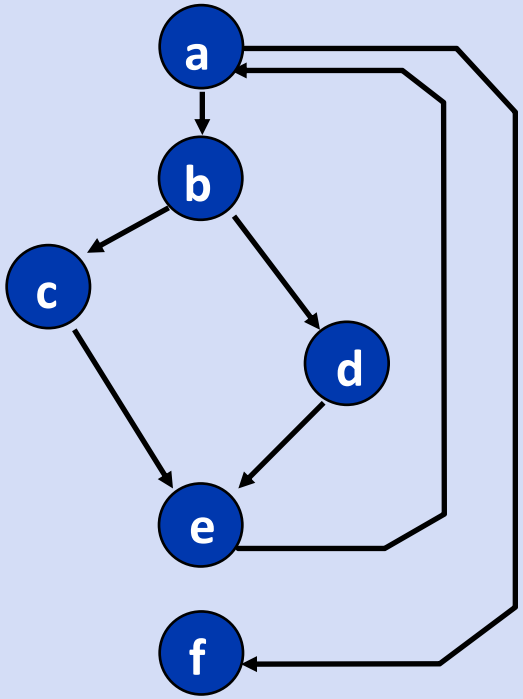
Example: Checking software execution

- Checking the correctness of control flow
  - 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:



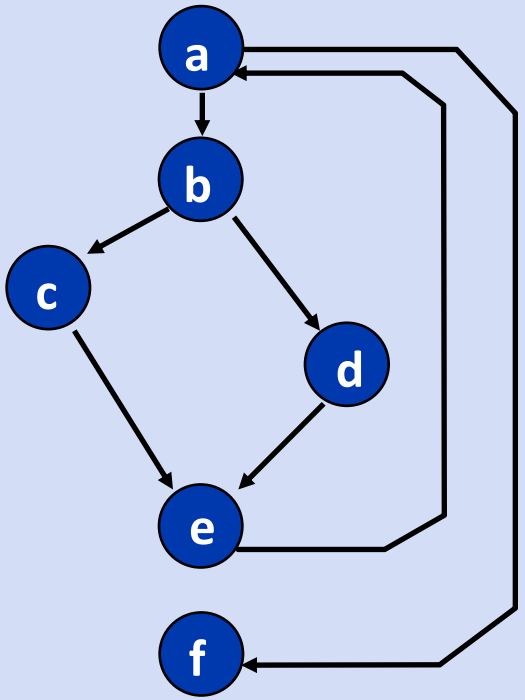
Example: Checking software execution

- Checking the correctness of control flow
  - 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:



Example: SAFEDMI development

Driver

DMI

EVC

EVC: European Vital Computer (on board)

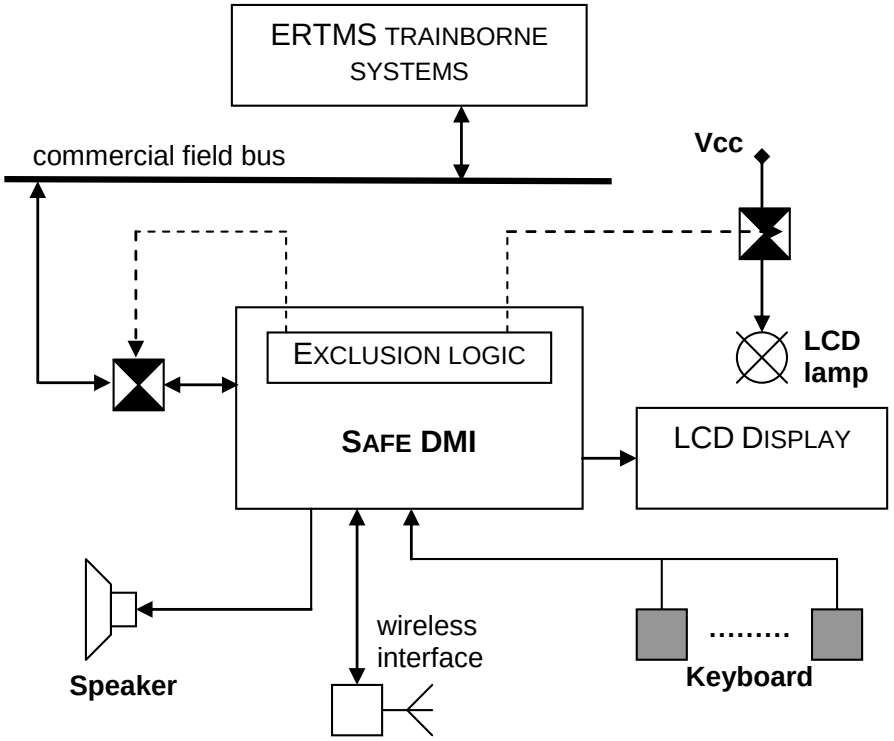
Maintenance center

Characteristics:

- Safety-critical functions
  - Information visualization
  - Processing driver commands
  - Data transfer to EVC
- Safe wireless communication
  - System configuration
  - Diagnostics
  - Software update

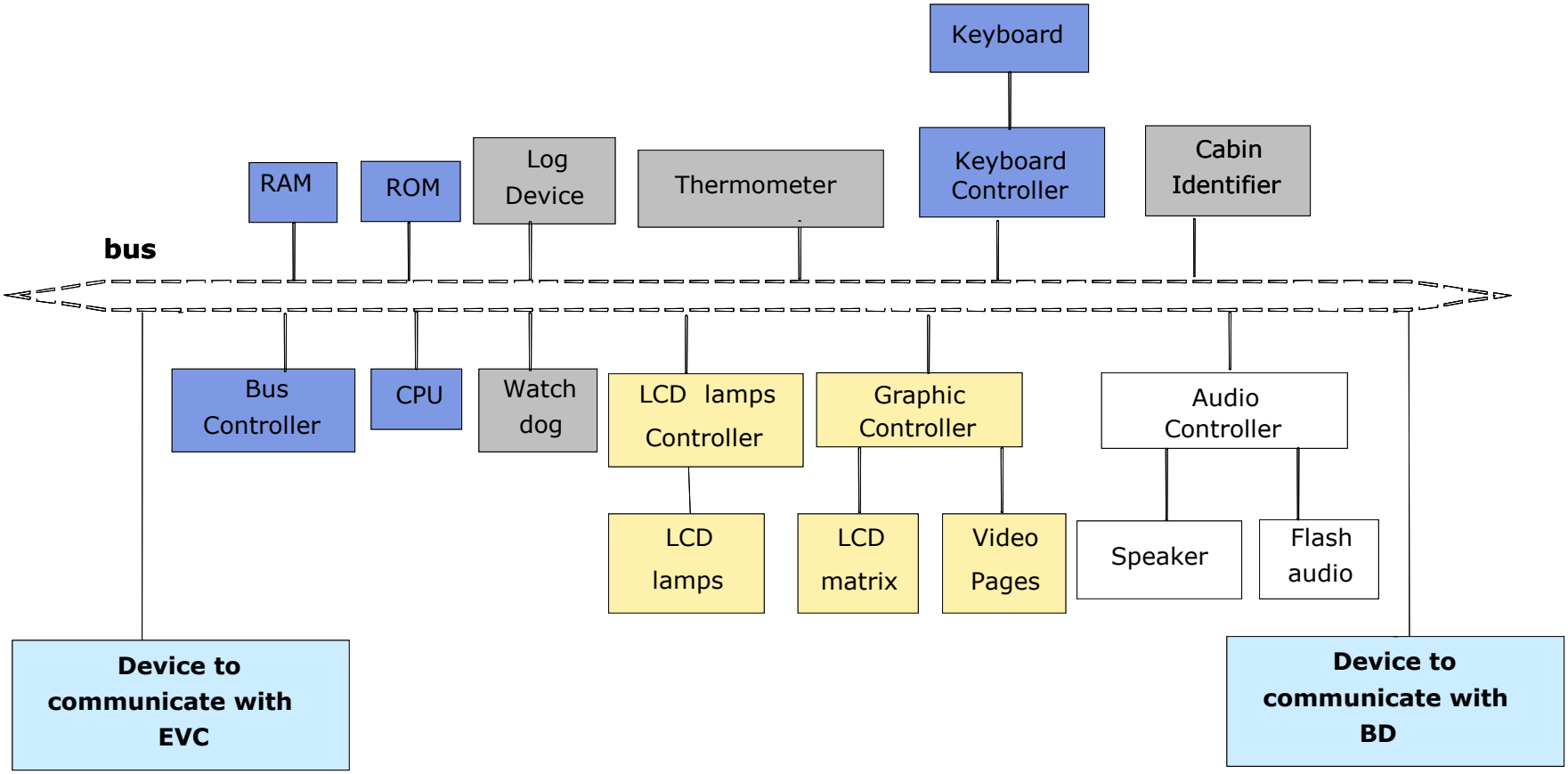
Example: SAFEDMI architectural concept

- Single-channel electronic structure based on reactive fail-safety (error detection and error handling)
- Generic (off-the-shelf) hardware components are used
- Most of the safety mechanisms implemented in software



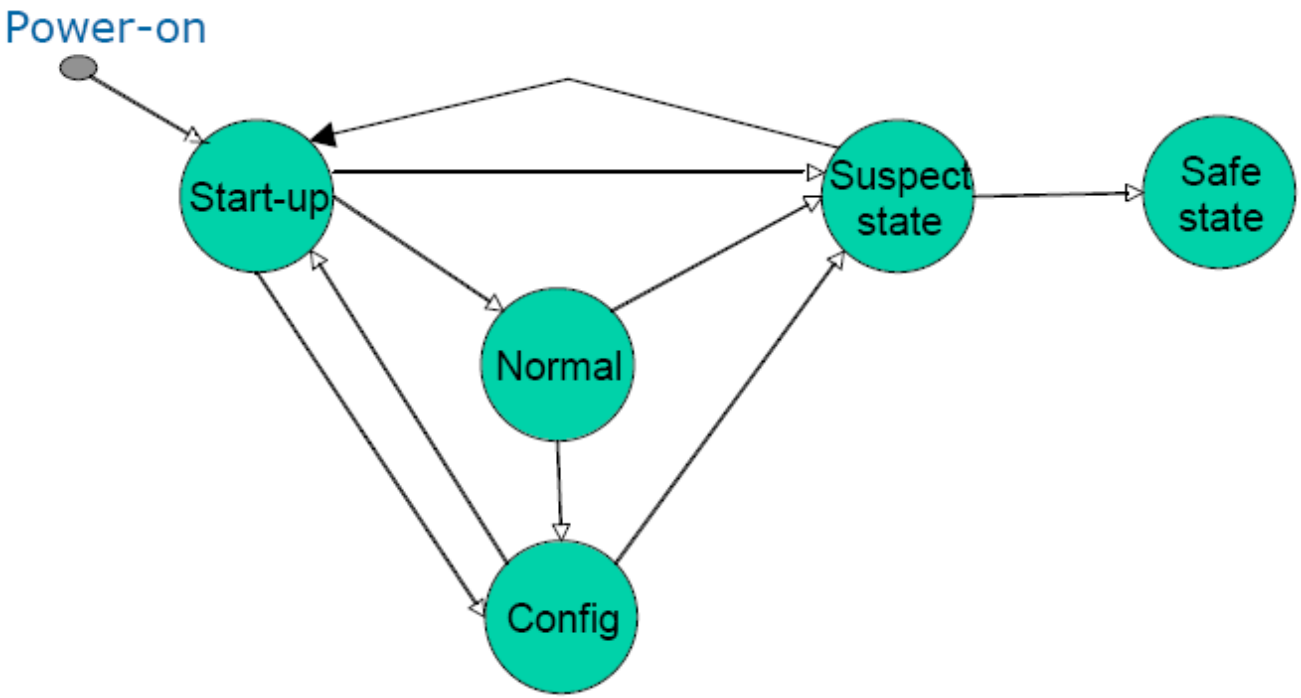
Example: SAFEDMI hardware architecture

Components:



Example: SAFEDMI operating modes

- Operating modes:
  - Startup, Normal, Configuration, Safe state
- Error processing: Suspect state
  - Intermediate state to distinguish transient and permanent faults
  - The fault is permanent if it occurs again when restart is tried → safe state

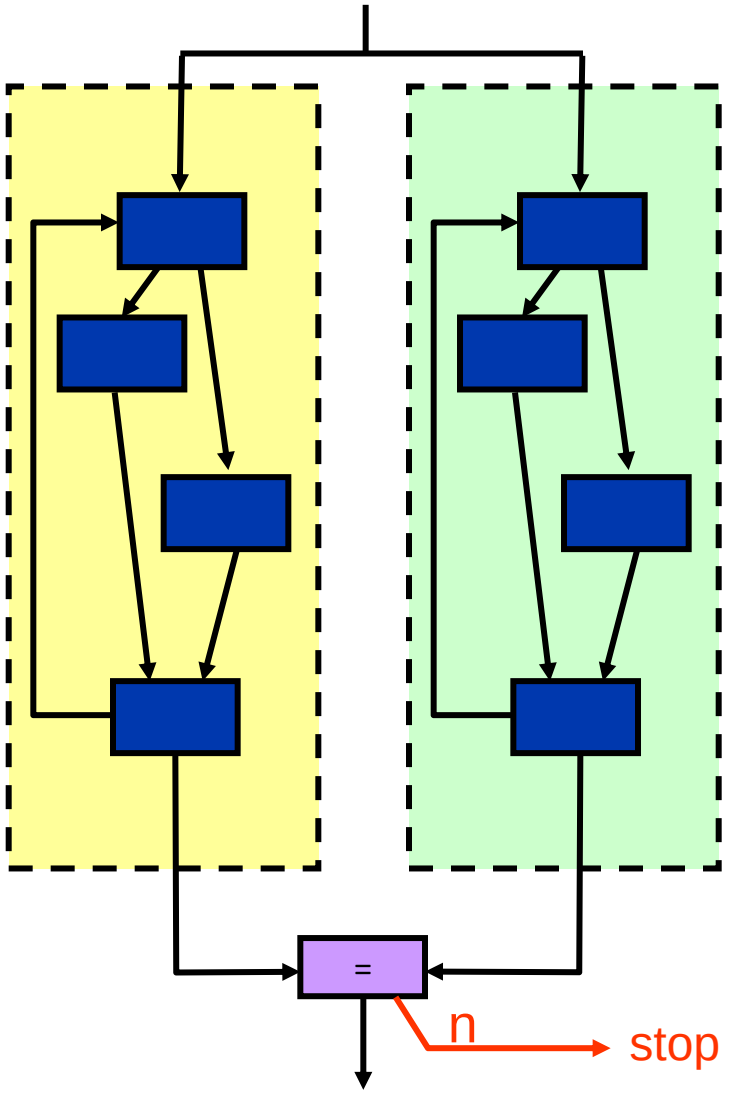


Example: SAFEDMI error detection techniques

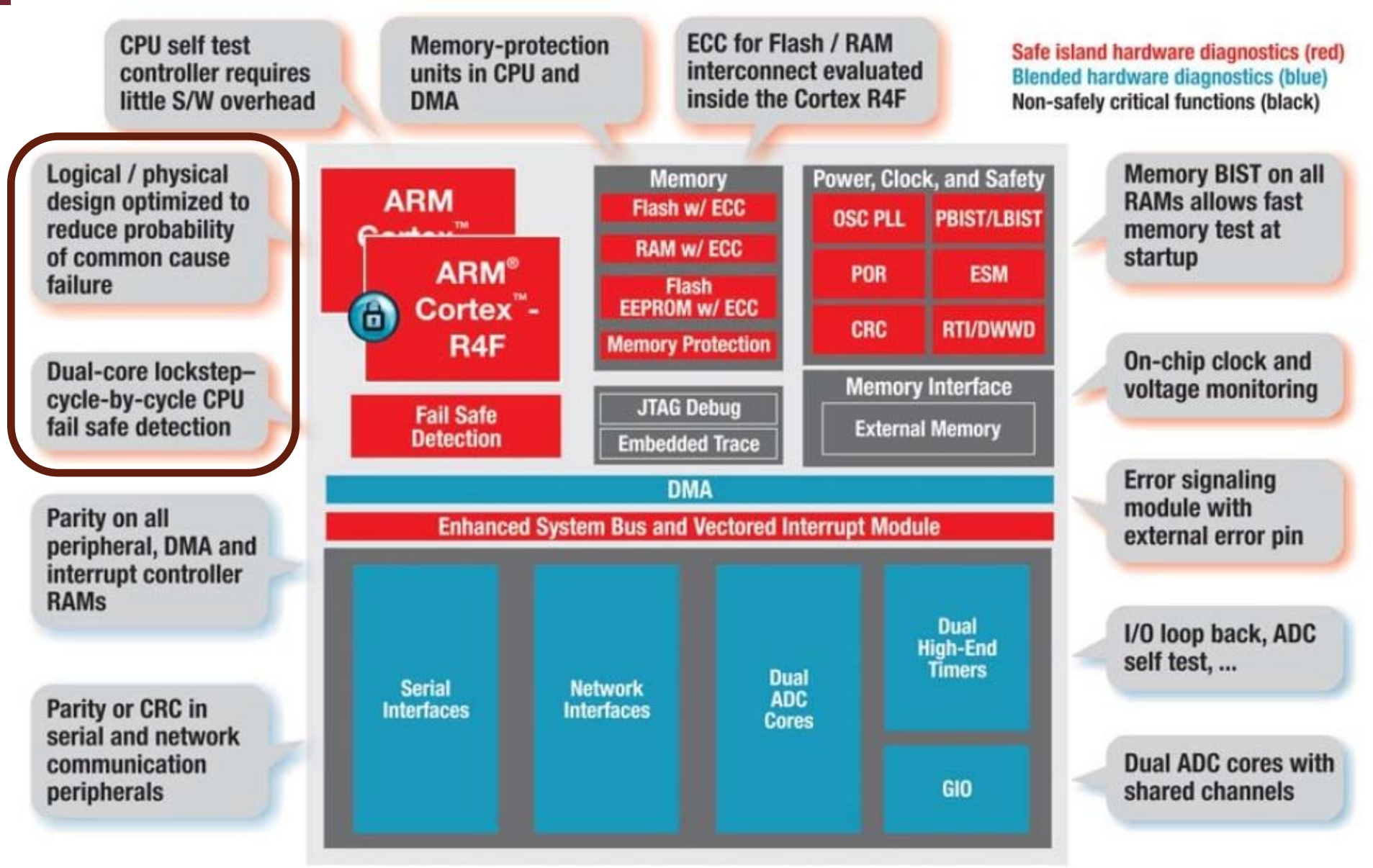
- Startup: Detection of permanent hardware faults
  - CPU testing with the help of an external watchdog circuit
  - Memory testing with marching algorithms
  - EPROM integrity checking with error detection codes
  - Device (peripherals) testing with the help of the driver
- Normal/Configuration: Periodic and online checking
  - Scheduled self-tests for hardware
  - Data integrity in communication and configuration functions: Data acceptance / credibility checks, error detection codes
  - Control related functions (e.g., changing operating modes): Control flow monitoring, time-out checking, acknowledgements
  - Data related functions (e.g., constructing bitmap for the display): Duplicated computation and comparison of the results

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 remain undetected
  - Long detection latency

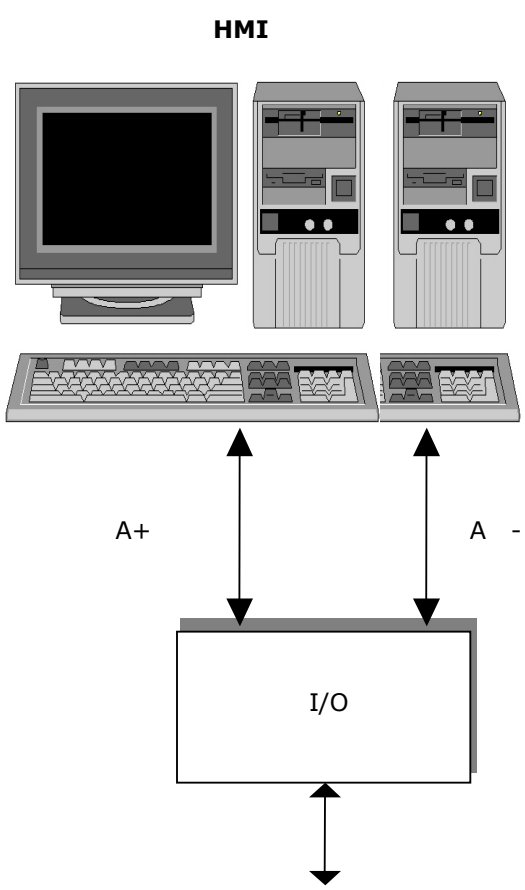


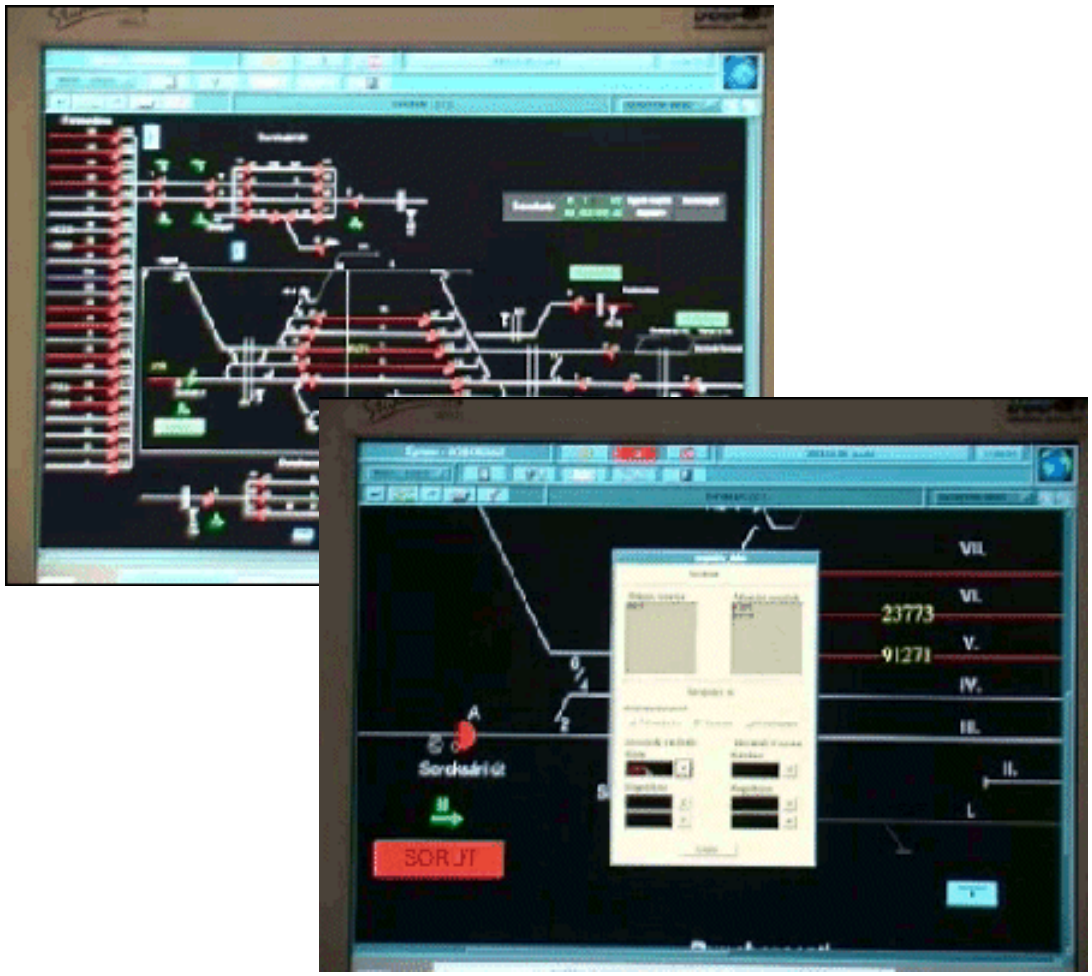
Example: Safety Microcontrollers



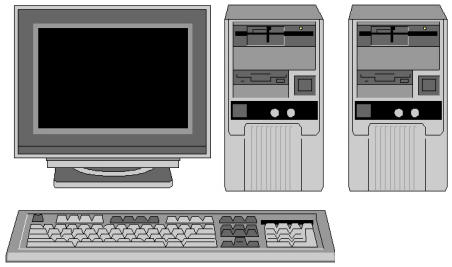
Example: SCADA system

- Supervisory Control and Data Acquisition system

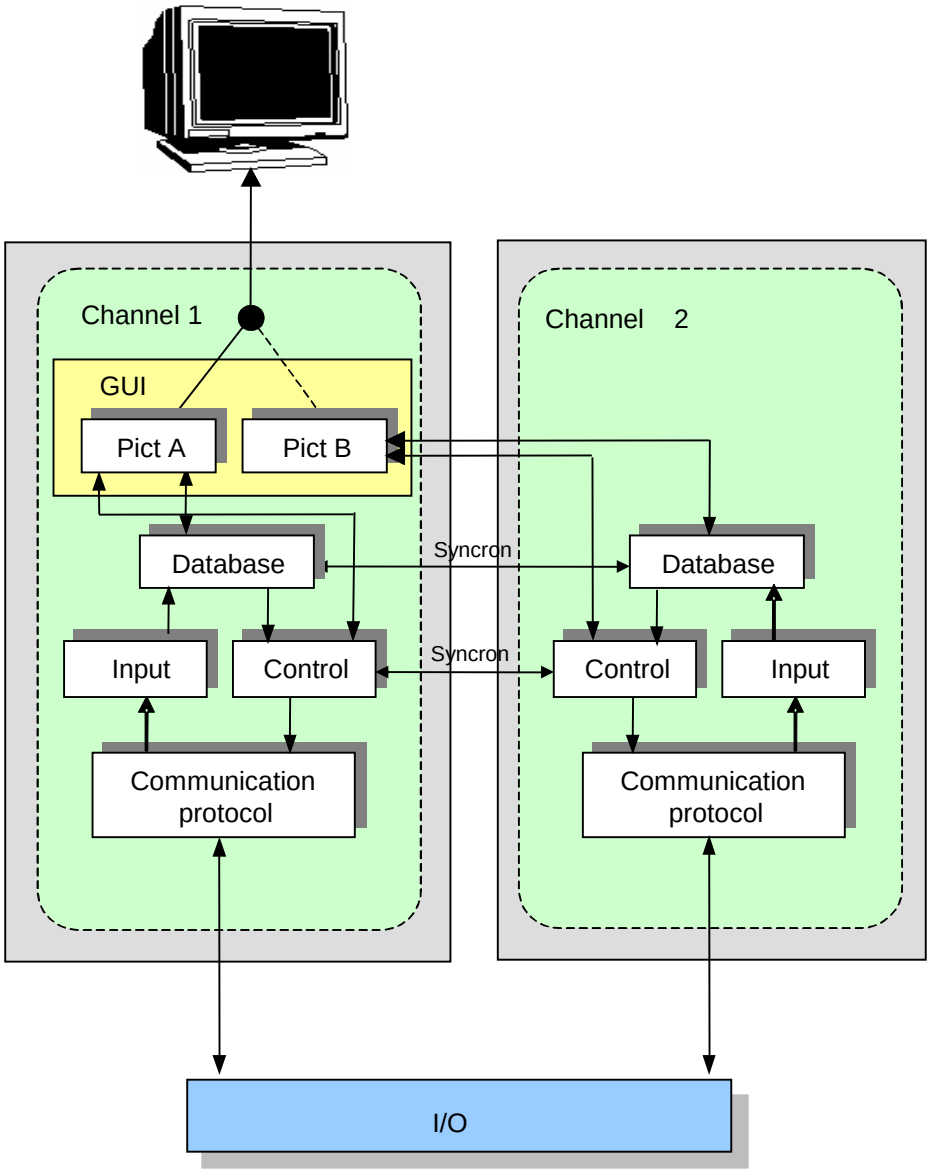




Example: SCADA system architecture

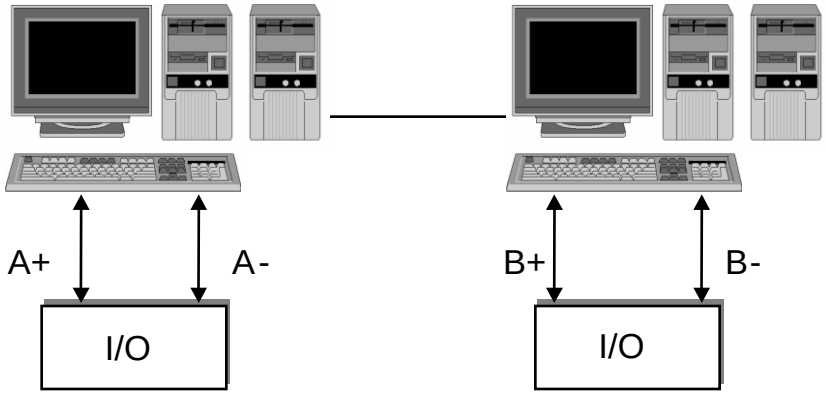


- Two channels
- Display: Periodically switching between bitmaps provided by the two channels: Comparison by the operator (stable or not)
- Synchronization: Detection of internal errors before the effects reach the outputs



Example: SCADA deployment options

- Two channels on the same server
  - Statically linked software modules
  - Independent execution in memory, disk and time
  - Diverse data representation
    - Binary data (signals): Two representations (original/negated)
    - Diverse indexing in the technology database
- Two channels on two servers
  - Synchronization on dedicated network
- Increasing availability by redundancy:
  - Two „2-out-of-2” scheme: Switch-over when primary pair detects a permanent fault



Example: SCADA error detection techniques

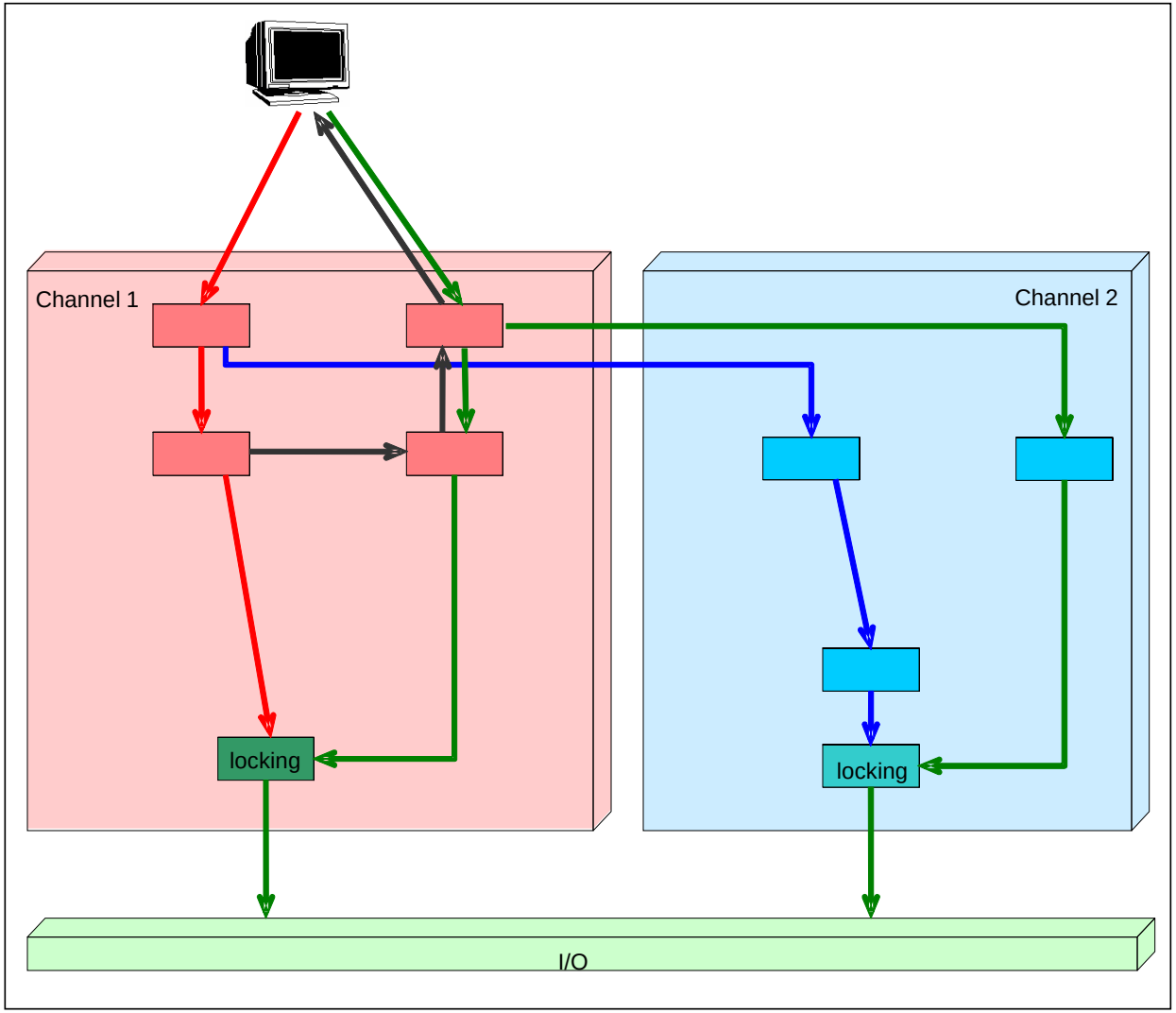
For random hardware faults during operation:

- Comparison of channels: Operator and I/O circuits
  - Heartbeat: Blinking RGB-BGR symbols indicate the regular update of the bitmap on the screen
- Watchdog process
  - Checking the operation of the processes (heartbeats)
- Regular comparison of the content of the technology database
  - Detecting latent errors

For unintended control by the operator:

- Three-phased control of outputs:
  - Preparation of output (but without effect; locking their activation)
  - Reading back the prepared output using independent software modules
  - Acknowledgement by the operator (using diverse GUI operations)

Example: SCADA three phases of control



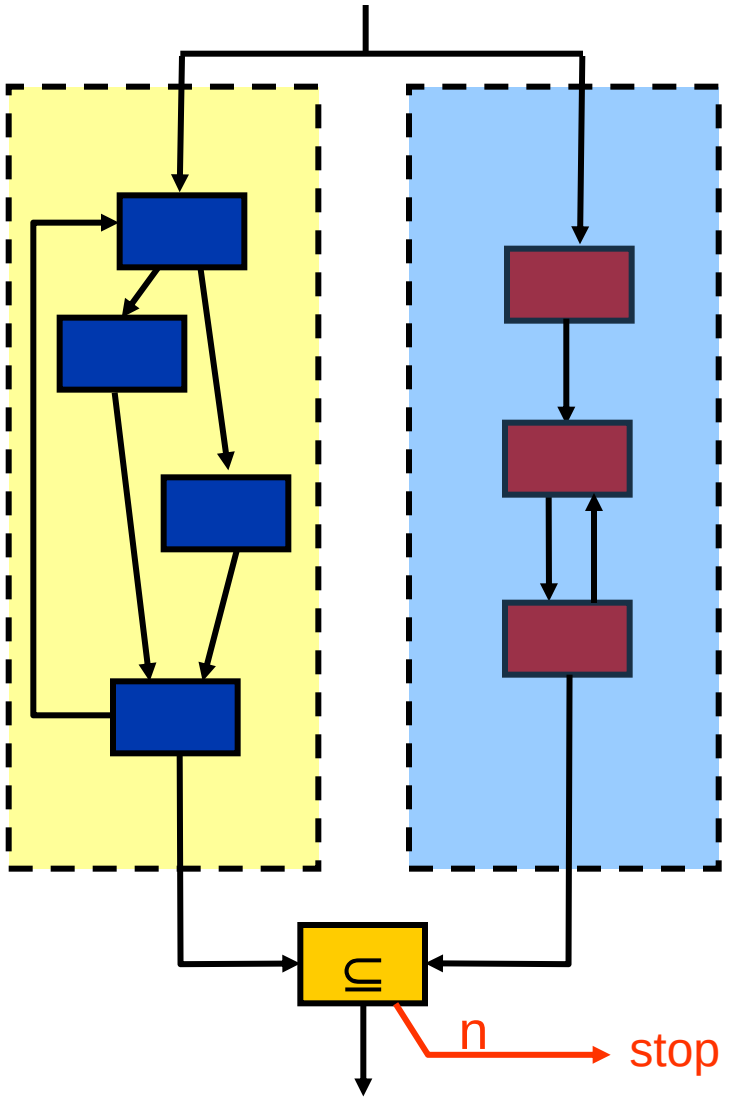
1 1

2

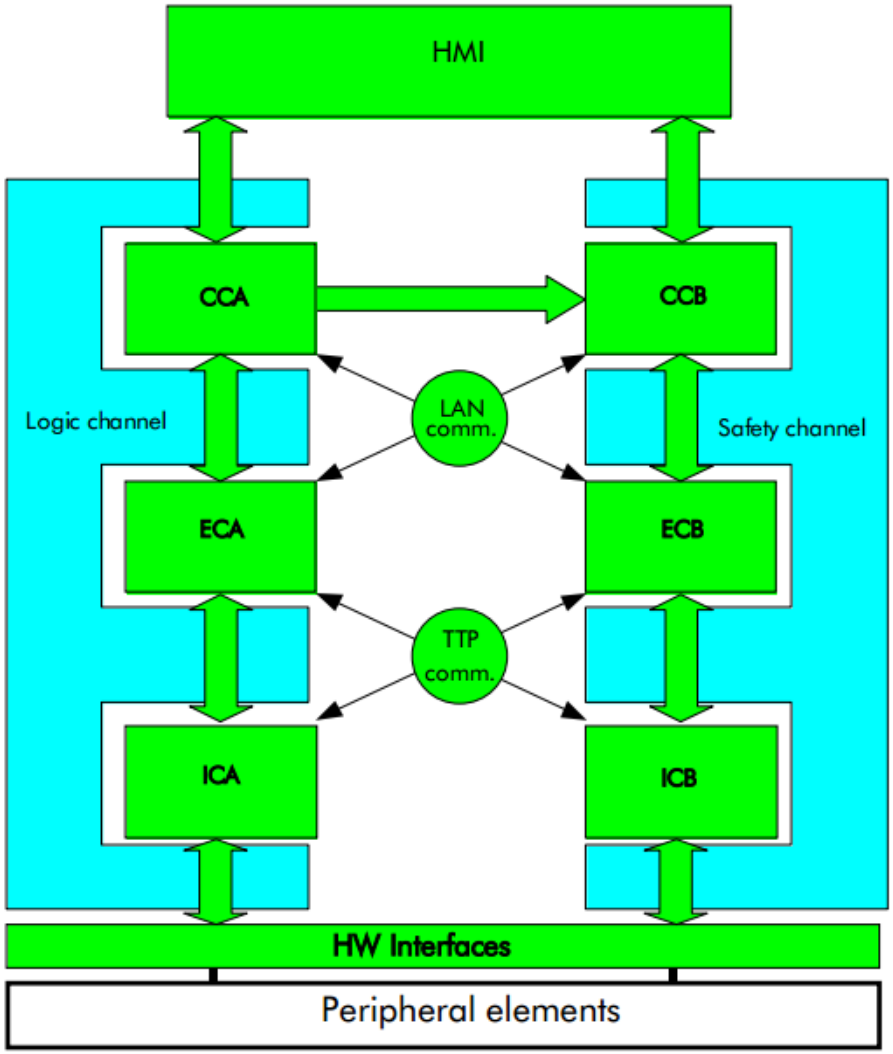
3

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

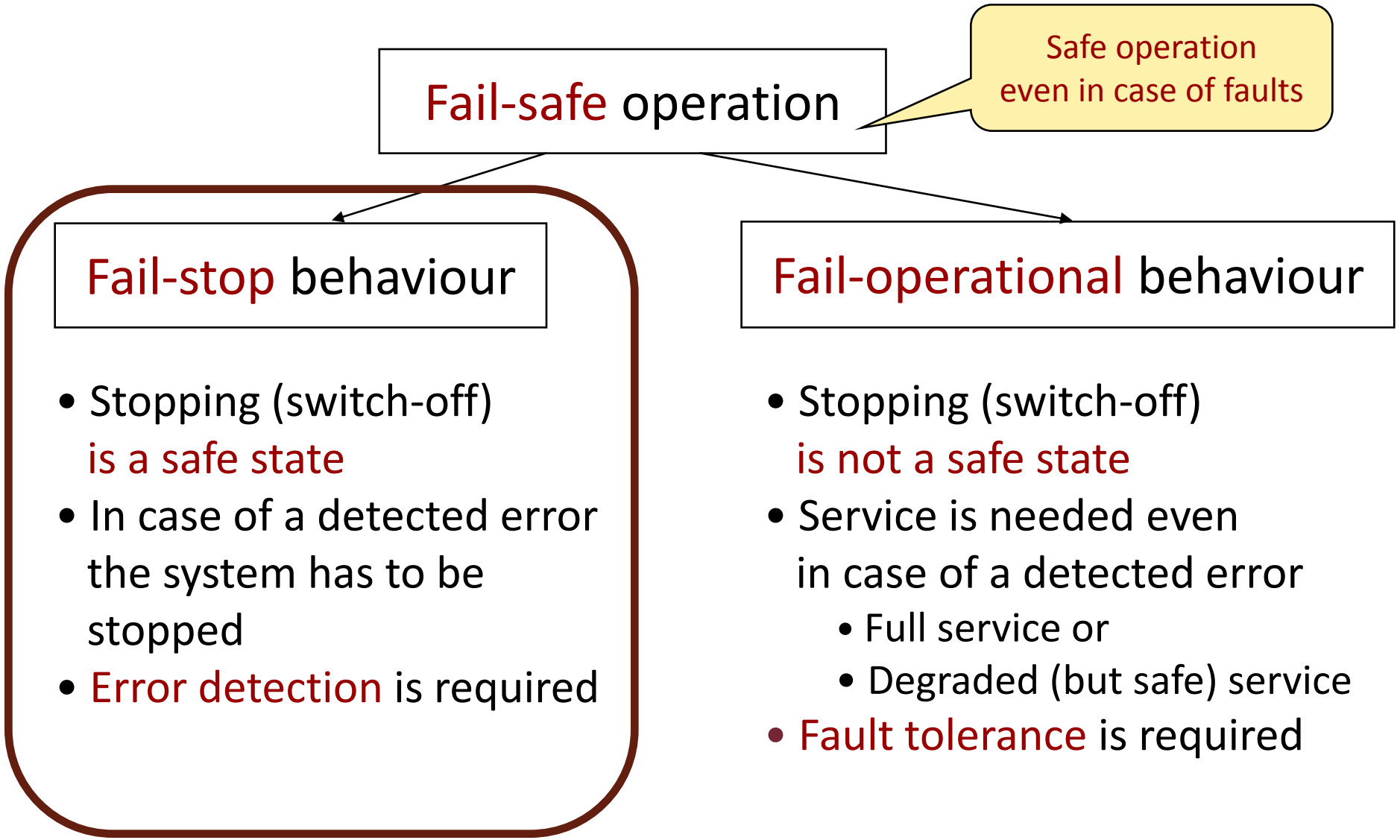
Central Controller

Field Element Controller

Two channels:

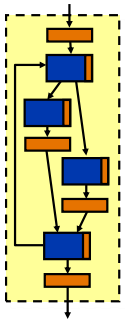
- Logic channel: CHILL (CCITT High Level Language) procedural programming language
- Safety channel: PAMELA (Pattern Matching Expert System Language) rule-based programing language

Summary: Objectives of architecture design

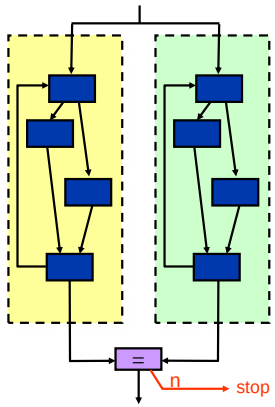


Summary: Solutions for fail-stop behavior

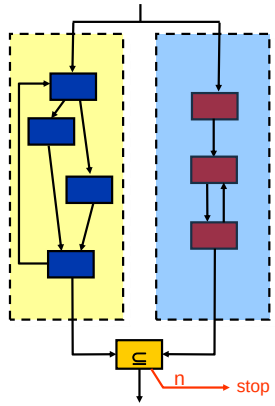
1. Single channel with built-in self-test
   - Hardware: Power-on self-test (POST) and built-in self-test (BIST)
   - Software: Online self-checking



2. Two-channels architecture with comparator
   - Replicated processing channels with shared input (problem: common failures)
   - Comparison of the channels' output



3. Two-channels architecture with safety checking
   - Independent, diverse checker channel
   - Checking the output of the primary channel



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

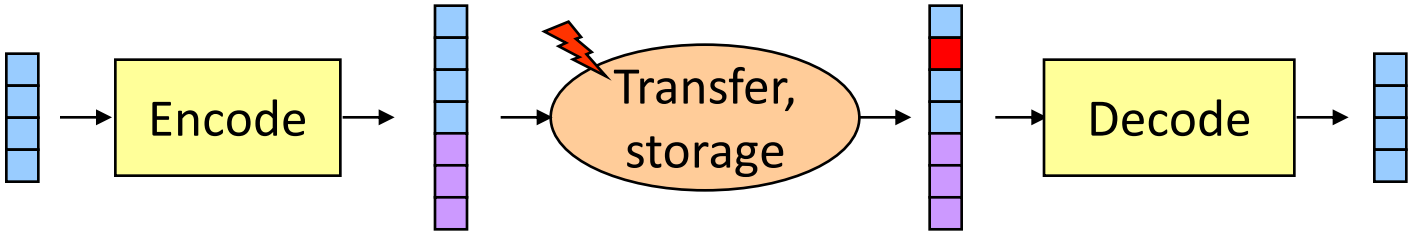
- Fault tolerance: Providing (safe) service in case of faults
  - Intervening into the fault → error → failure chain
    - Detecting the error and assessing the damage
    - Involving extra resources to perform corrections / recovery
    - Providing correct service without failure
    - (Providing degraded 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 is active but has reduced load
  - Hot: The redundant component is active in fault-free case

Example: Error detecting and correcting codes



- Error detecting codes (EDC): Only detection of errors
  - Parity bit: Increasing the Hamming-distance, 1 bit error can be detected
  - Checksum: Using in case of files, messages
- Error correcting codes (ECC): Identifying and correcting errors
  - Higher Hamming distance: Errors can be corrected
    - E.g.: (7, 4) bit Hamming code: 1 bit error corrected, 1 or 2 bit errors detected
  - Information blocks: More difficult codes are used
    - E.g.: (255, 223) byte Reed-Solomon code: 16 byte errors can be corrected
- Limited error correction capability
  - Information storage: In long time, more errors can accumulate than the number of errors that can be corrected by the applied codes
  - Basic idea: Periodic reading, correcting and writing back the information

4 data bits, 3 redundant bits

Overview: How to use the redundancy?

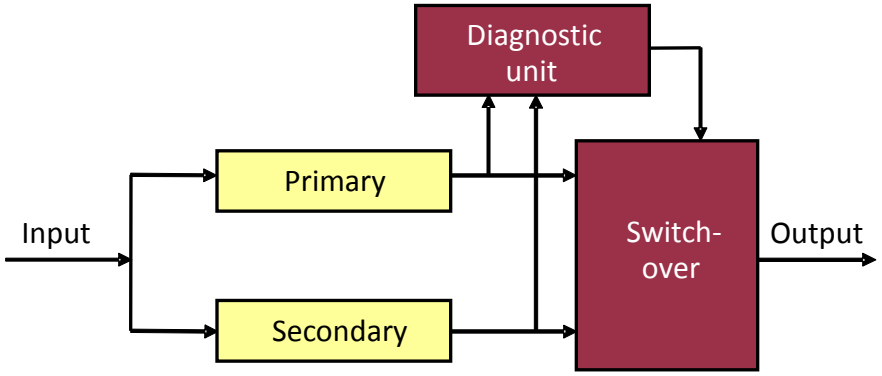
- Hardware design faults: (< 1%)
  - Hardware redundancy with design diversity
- Hardware permanent operational faults: (~ 20%)
  - Hardware redundancy (e.g.: redundant processor)
- Hardware transient operational faults: (~70-80%)
  - Time redundancy (e.g.: instruction retry)
  - Information redundancy (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

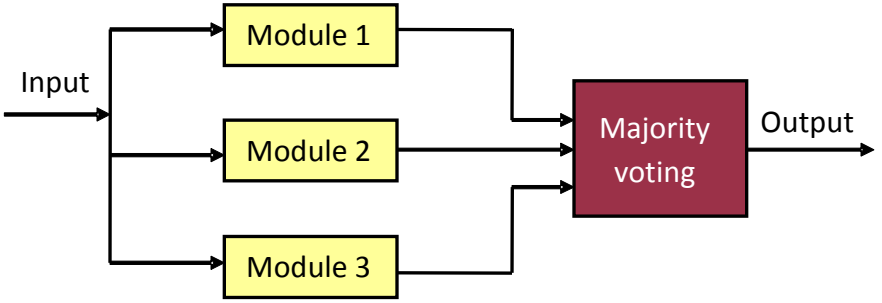
With diversity in case of considering design faults

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)



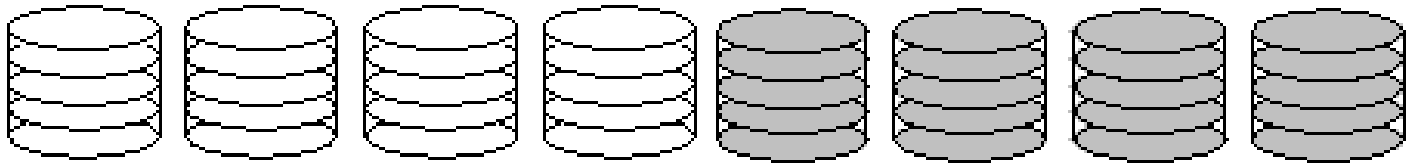
- NMR: N-modular redundancy
  - Masking the failure by majority voting
  - Mission critical systems: Goal is to survive the mission time

Implementation of the replication

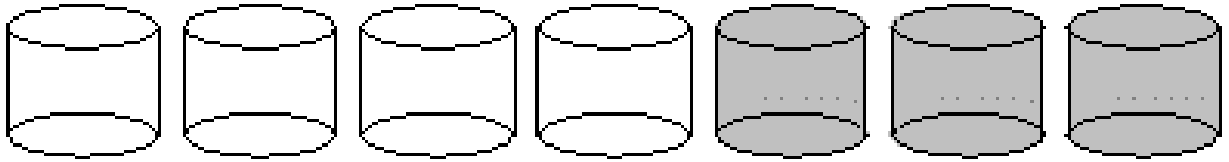
- Equipment/server level:
  - Servers: High availability server clusters
    - E.g., Linux HA Clustering, Windows Server Failover Clustering
  - Software support: Failover and failback
- Board level:
  - Run-time reconfiguration: “Hot-swap”
    - E.g., CompactPCI, HDD, power supply
  - Software support: monitoring, reconfiguration
- Component level:
  - Replication of components: TMR
  - Self-checking circuits (processing encoded information)

Example: RAID disk configura-tions

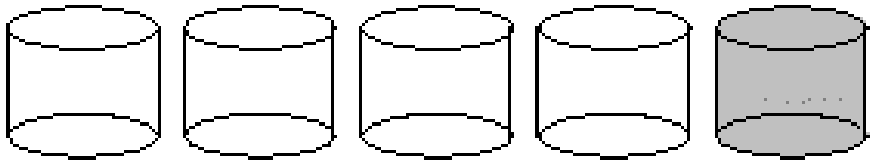
RAID: Redundant Array of Independent Disks



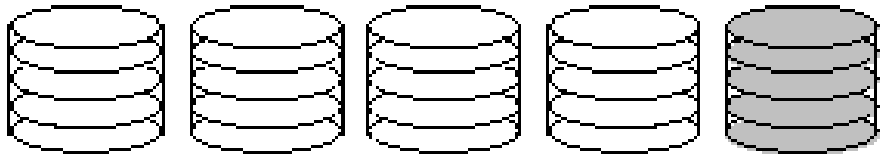
RAID-1: Mirroring (duplicated disks)



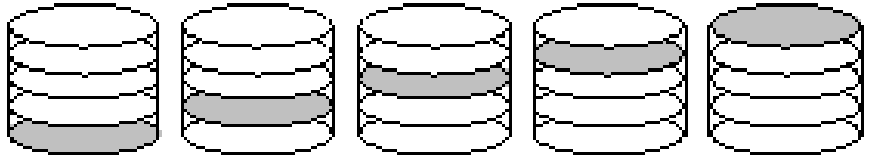
RAID-2: Bit-level ECC (error correcting codes)



RAID-3: Bit-level parity (assumption: faulty disk can be identified)



RAID-4: Block-level parity (to improve performance)



RAID-5: Block-level parity (to avoid bottleneck of the parity disk)

2. Fault tolerance for transient hardware faults

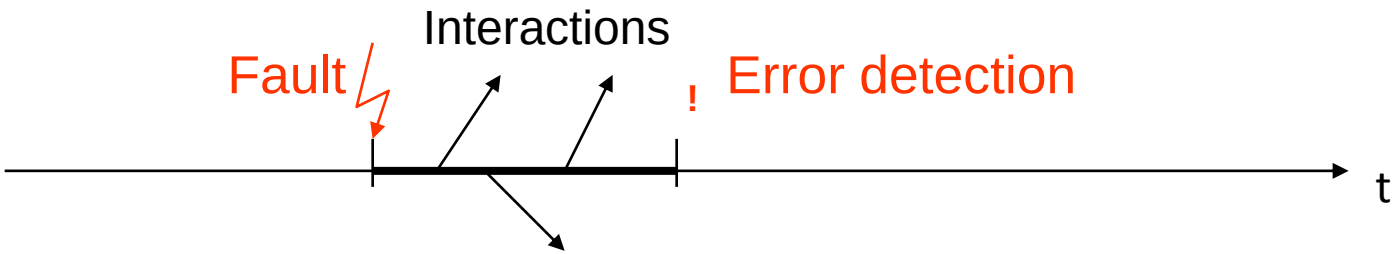
- Approach: Fault tolerance implemented by software
  - Detecting the error
  - Setting a fault-free state by handling the fault effects
  - Continuing the execution from that state (assuming that transient faults will not occur again)
- Four phases of operation:
  1) Error detection
  2) Damage assessment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



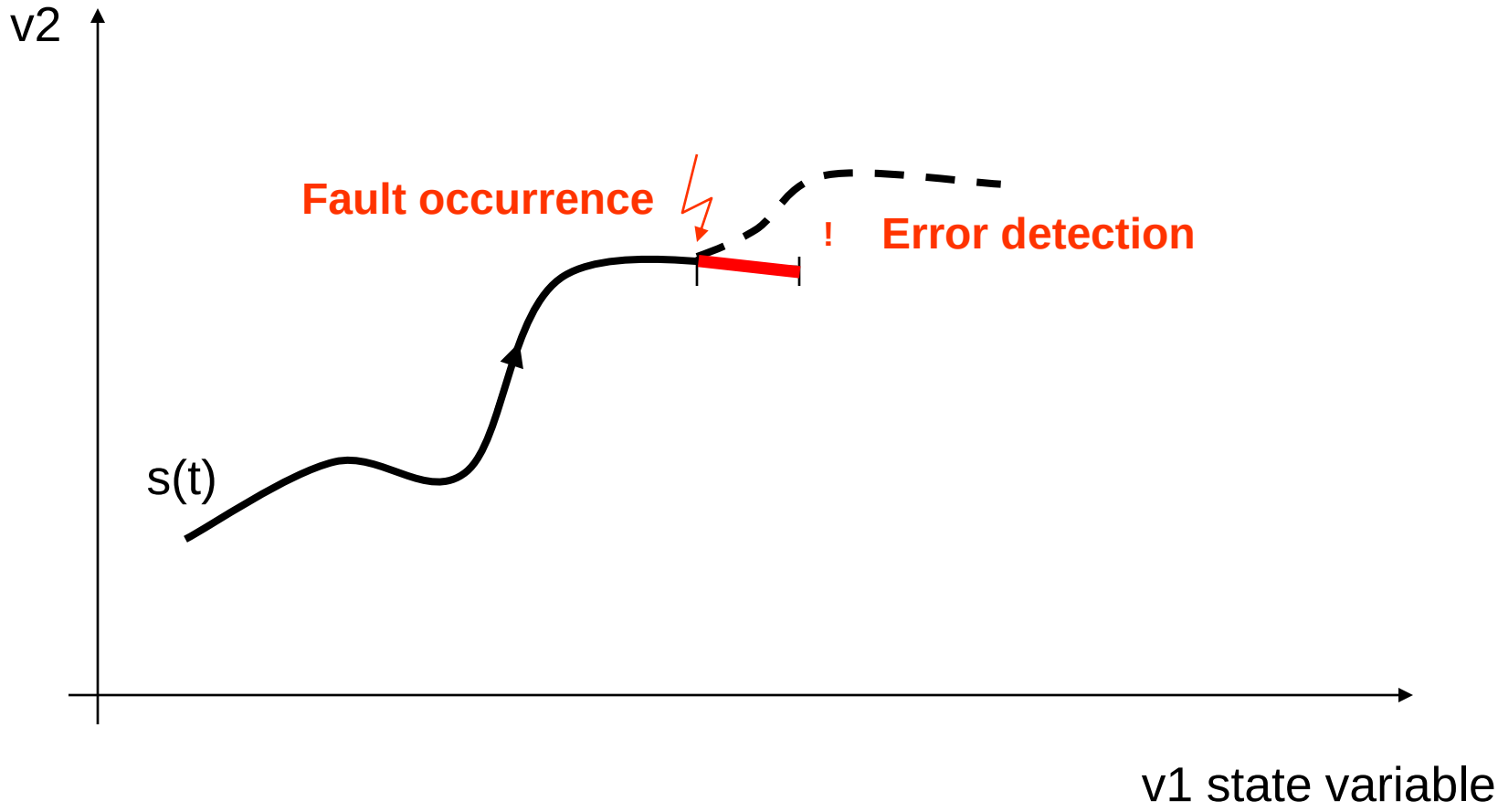
- Limiting error propagation: Checking interactions
  - Input acceptance checking (to detect external errors)
  - Output credibility checking (to provide „fail-silent” operation)
- Estimation of components affected by a detected error
  - On the basis of logged resource accesses and communication
  - Analysis of interactions (that happened 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 inherent redundant information

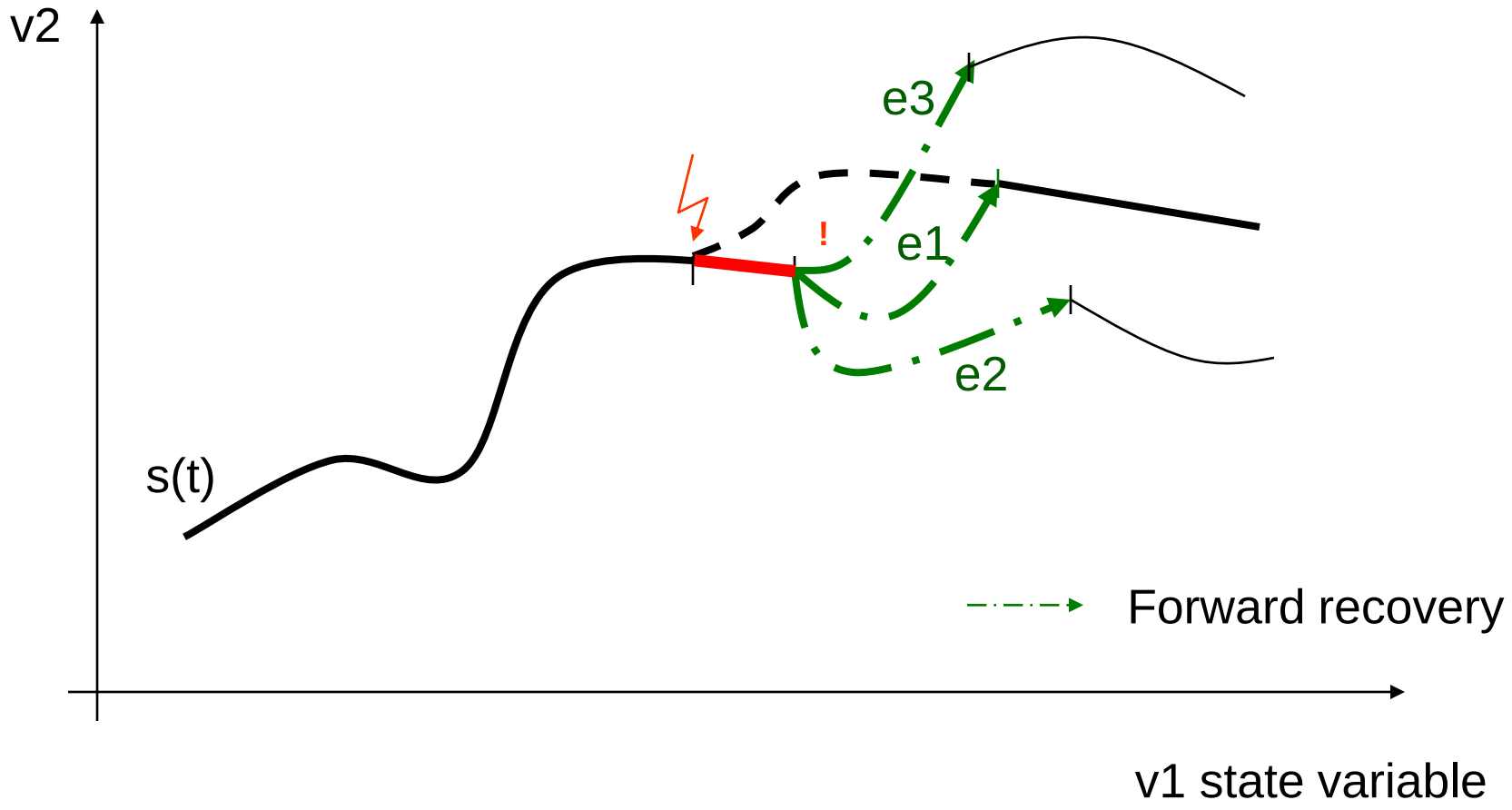
Types of recovery

- State space of the system: Error detection



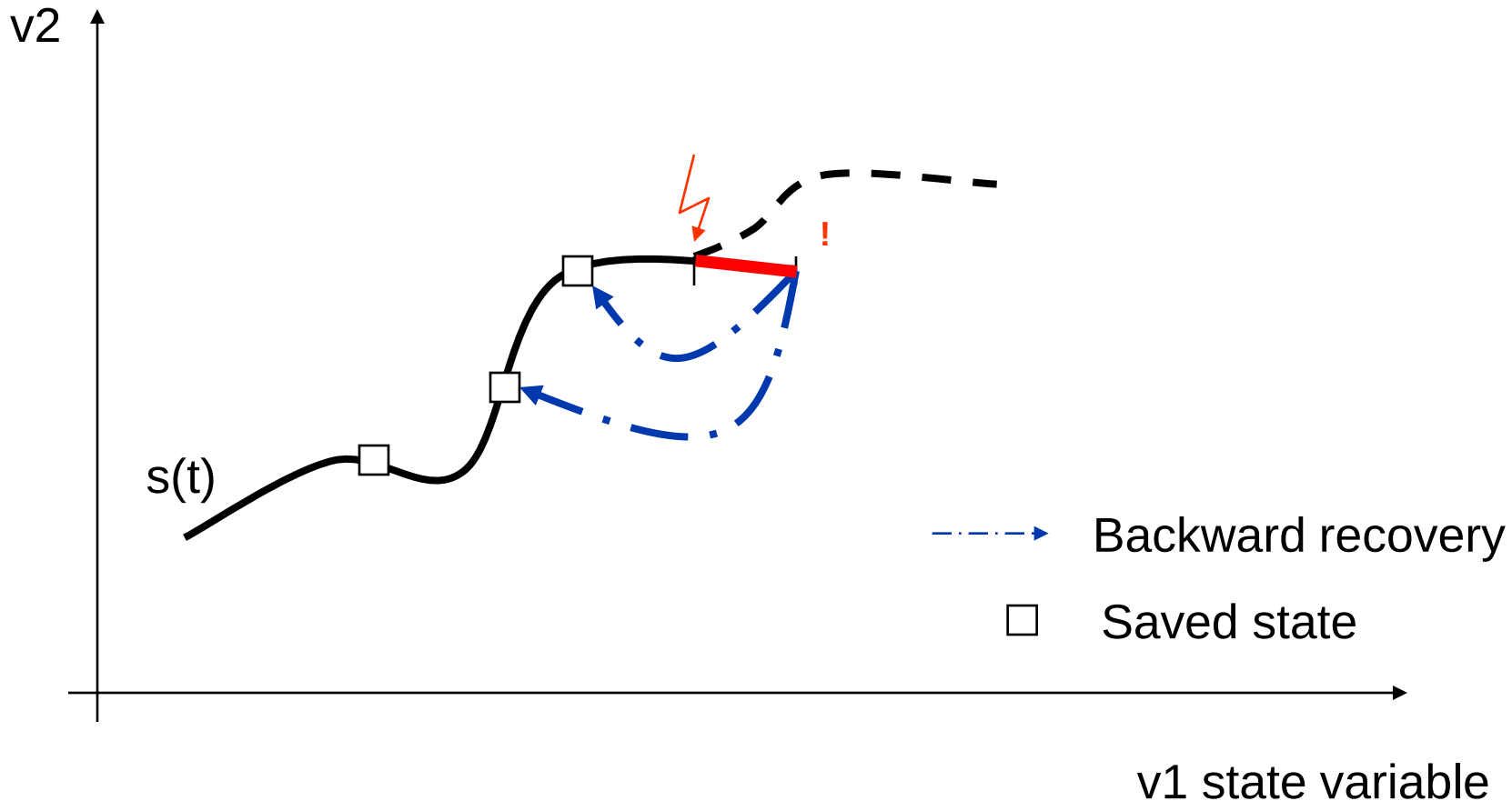
Types of recovery

- State space of the system: Forward recovery



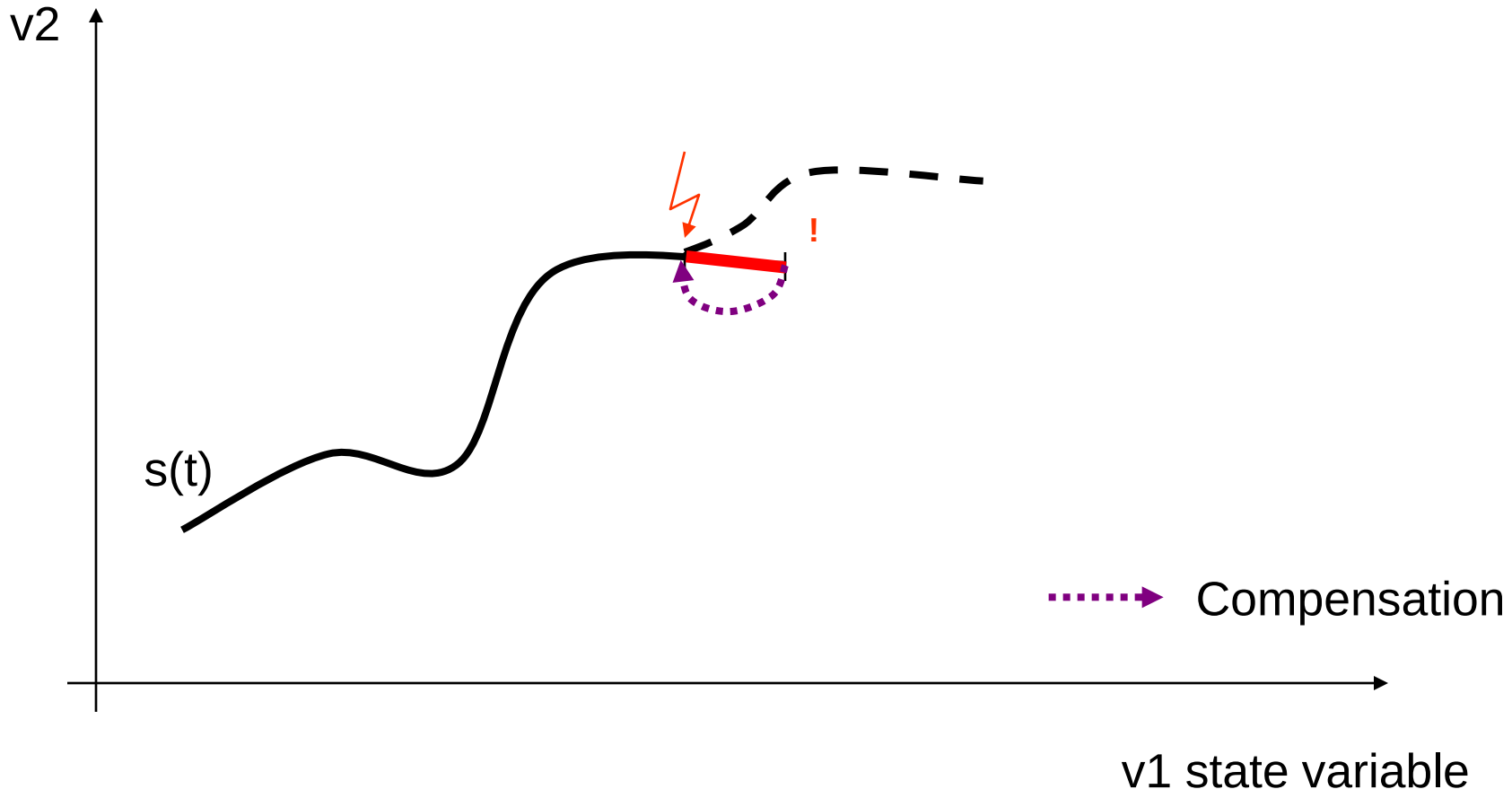
Types of recovery

- State space of the system: Backward recovery



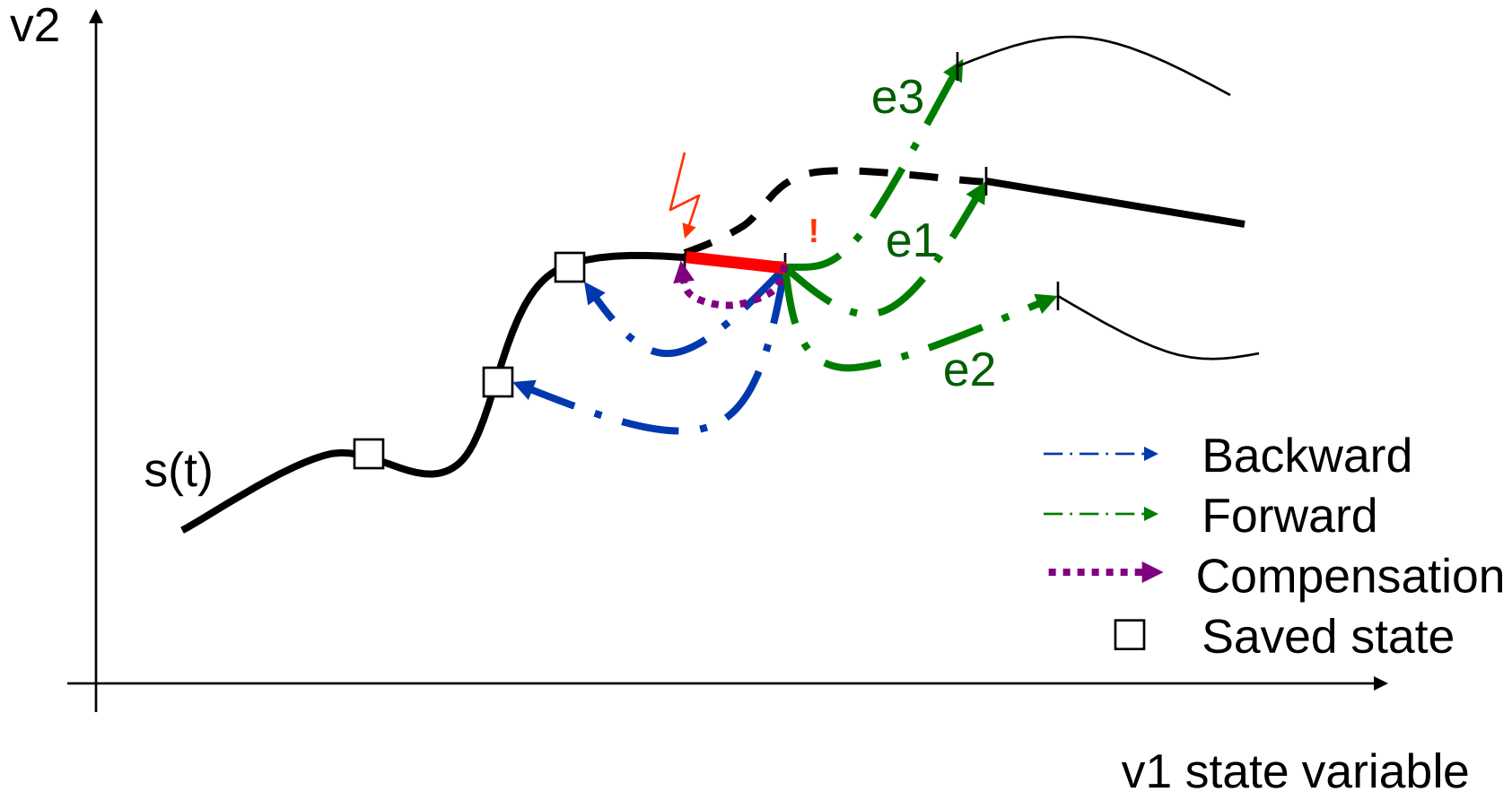
Types of recovery

- State space of the system: Compensation



Types of recovery

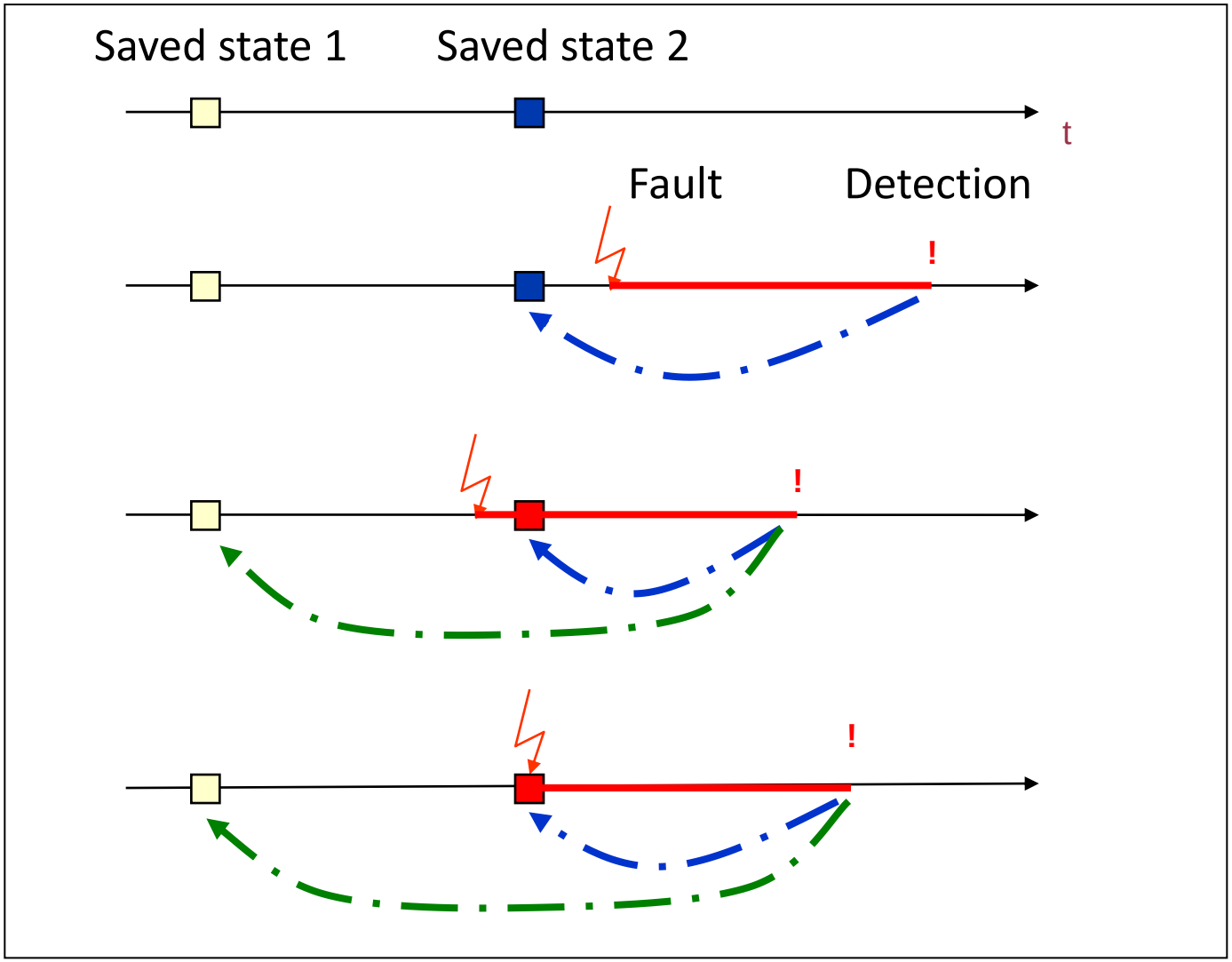
- State space of the system: Types of recovery



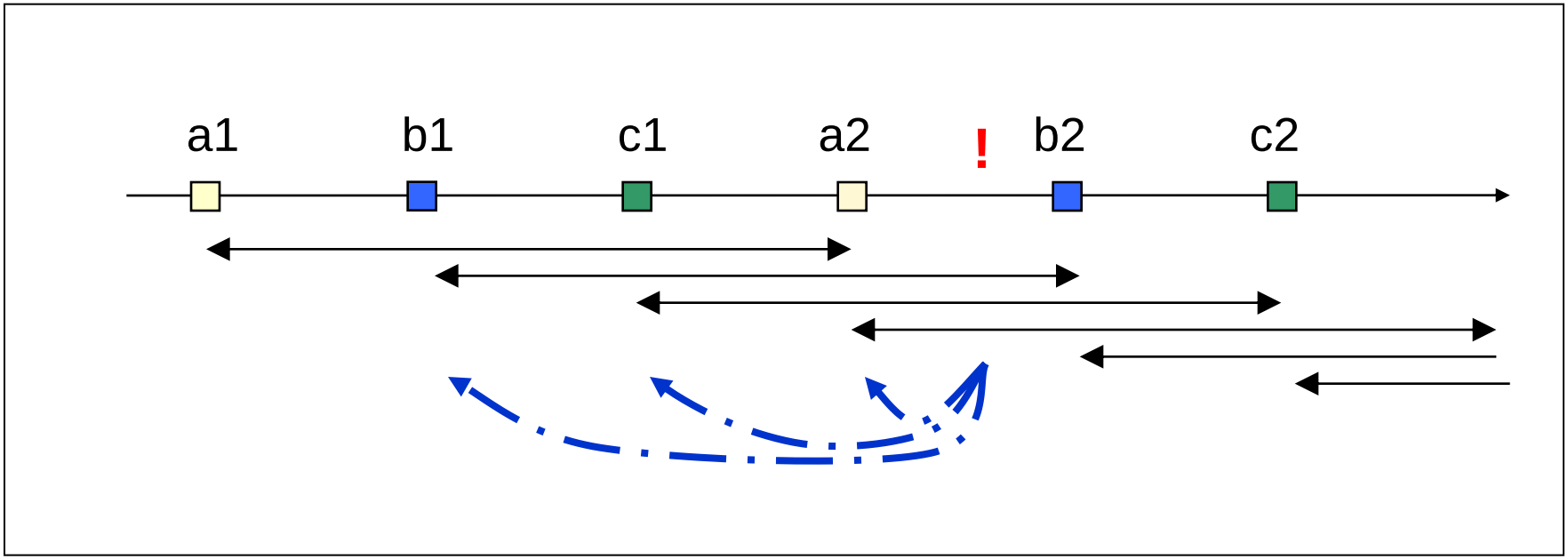
Backward recovery

- 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”
- Backward recovery based on operation logs
  - Limited scope: Errors due to unintended operations
  - Recovery is performed by the withdrawal of operations (by executing inverse operation, revoking the effects etc.)
  - Analogy: ”undo”

Scenarios of backward recovery



Checkpoint intervals



Aspects of optimizing checkpoint intervals:

- Stable storage is slow (→ overhead) and has limited capacity
- Computation is lost after the last checkpoint
- Long error detection latency increases the chance of damaged checkpoints

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 or replacement

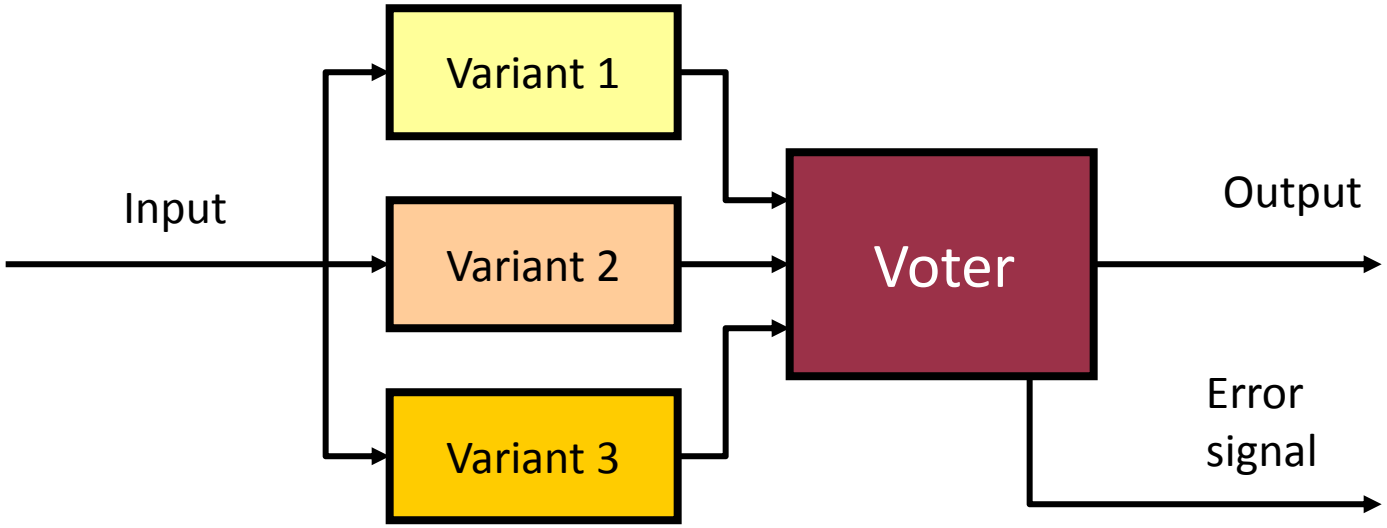
4. Fault tolerance for software faults

- 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
  - N-self-checking programming
  - Recovery blocks

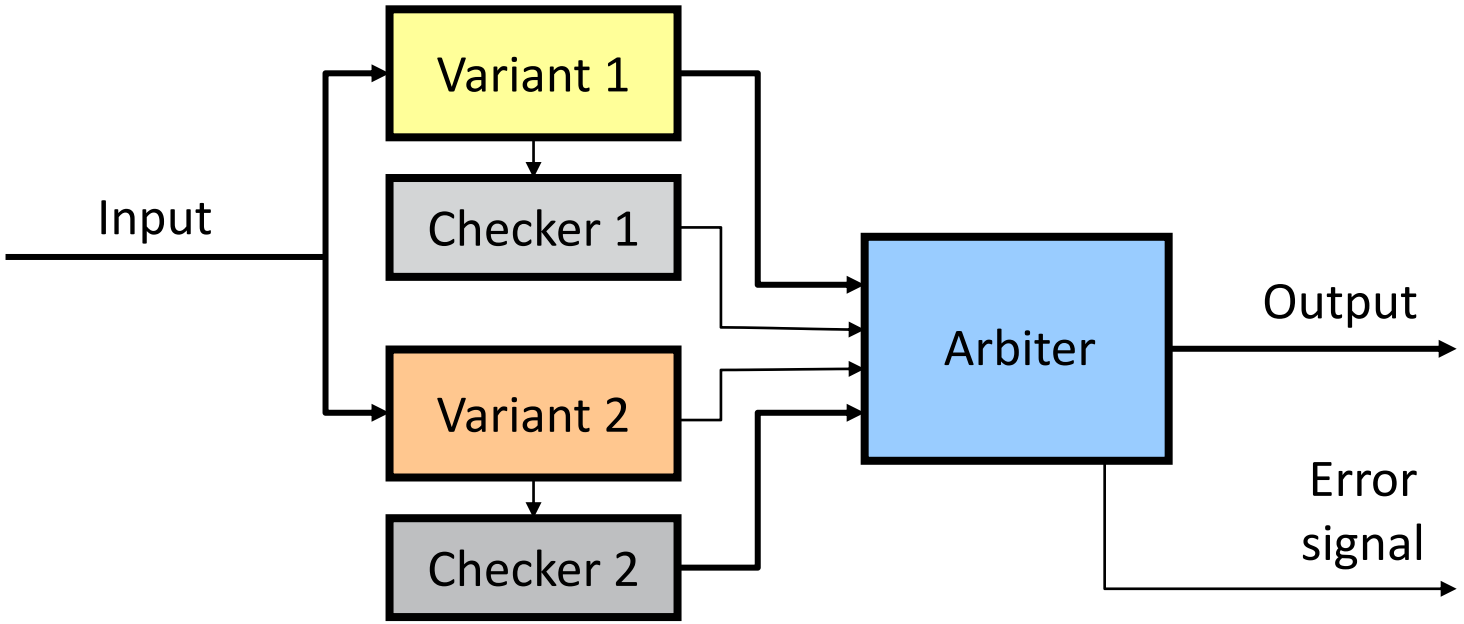
N-version programming

- Active redundancy:
  Each variant is executed (in parallel or serially)
  - 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)



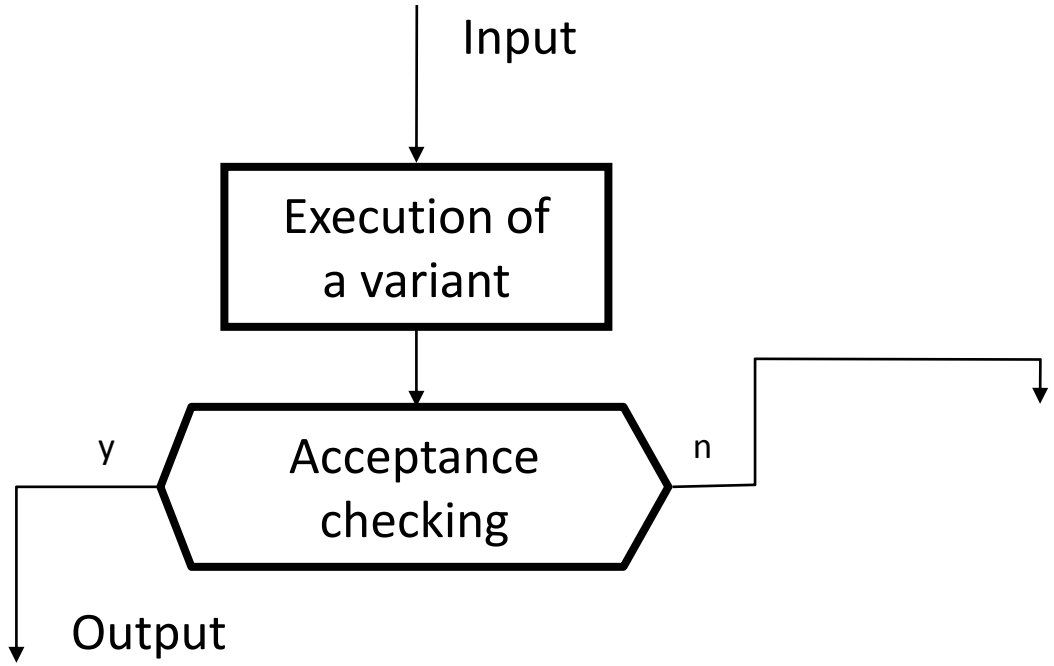
N-self-checking programming

- Active dynamic redundancy
  - N self-checking components: Variant + checker
  - In case of detected fault: Switching from the primary component to the redundant one



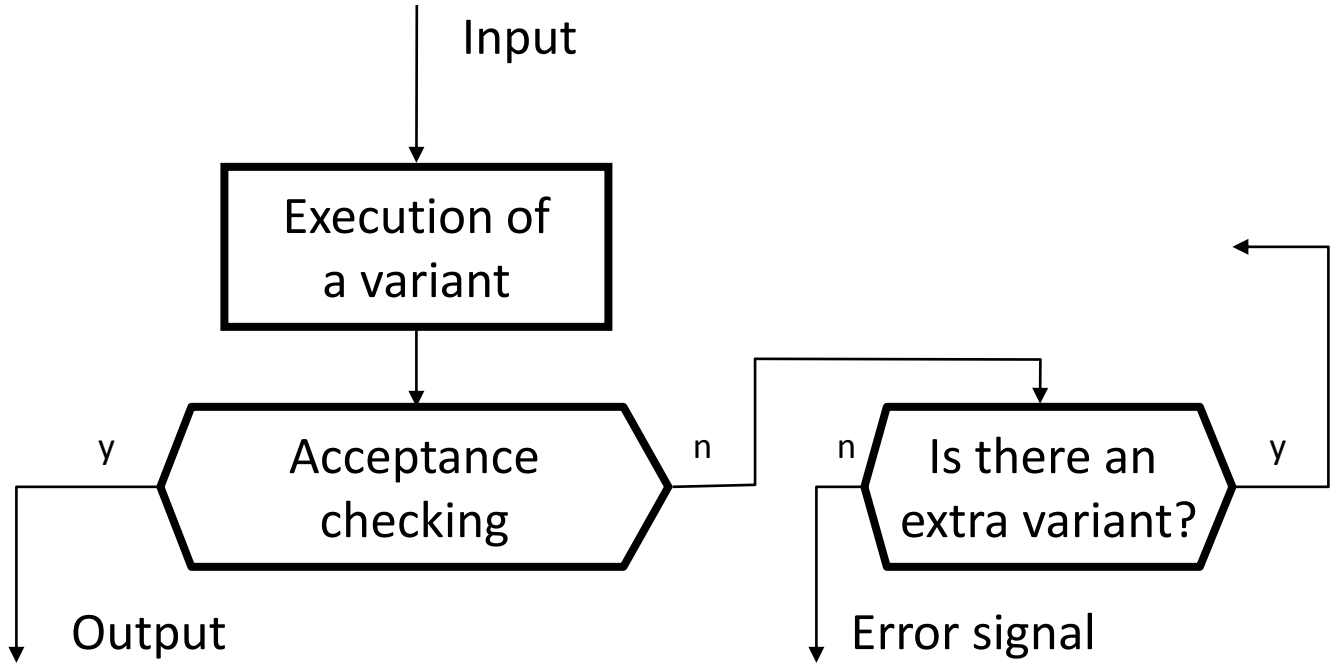
Recovery blocks

- 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



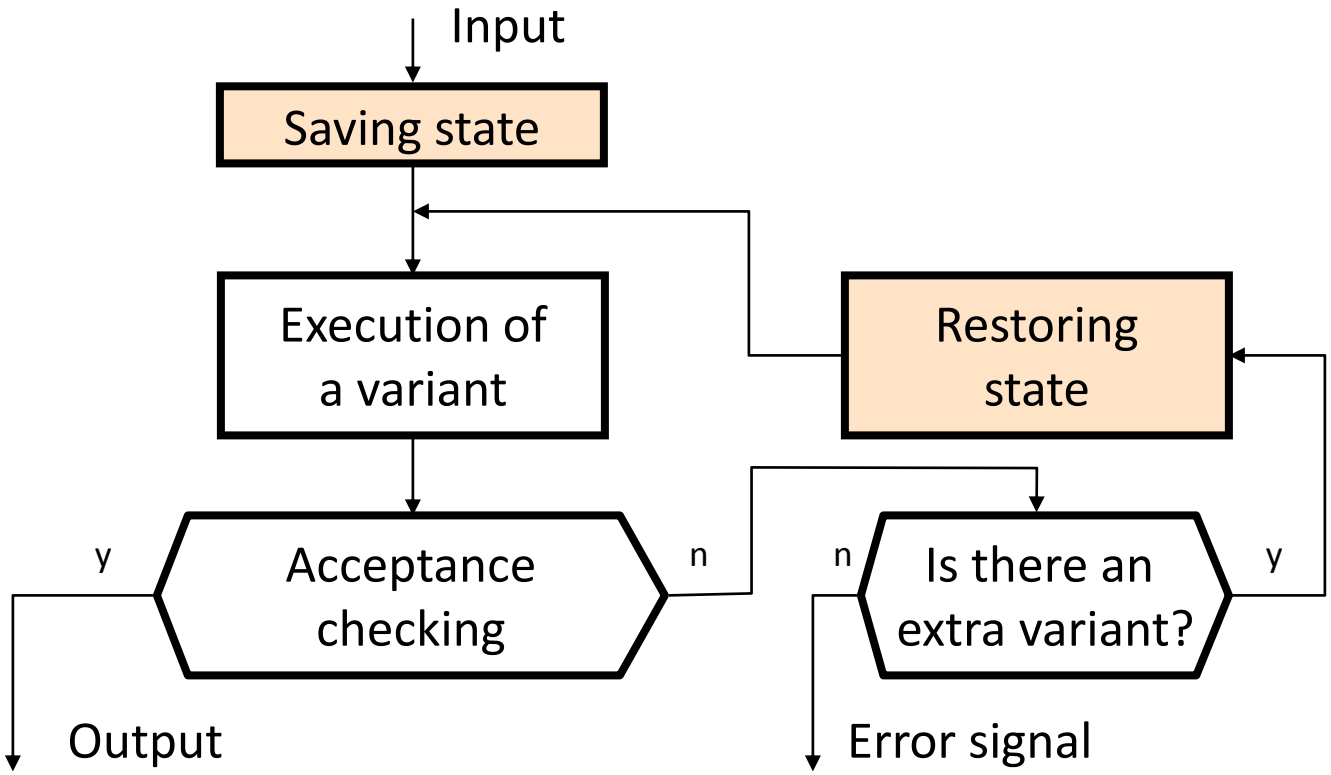
Recovery blocks

- 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 (typically)	Serial only
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

k

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)

Software architecture design in standards

- IEC 61508:
  Functional safety in electrical / electronic / programmable electronic safety-related systems
- Measures for software architecture design

Table A.2 – Software design and development: software architecture design (see 7.4.3)

	Technique/Measure*	Ref	SIL1	SIL2	SIL3	SIL4
1	Fault detection and diagnosis	C.3.1	---	R	HR	HR
2	Error detecting and correcting codes	C.3.2	R	R	R	HR
3a	Failure assertion programming	C.3.3	R	R	R	HR
3b	Safety bag techniques	C.3.4	---	R	R	R
3c	Diverse programming	C.3.5	R	R	R	HR
3d	Recovery block	C.3.6	R	R	R	R
3e	Backward recovery	C.3.7	R	R	R	R
3f	Forward recovery	C.3.8	R	R	R	R
3g	Re-try fault recovery mechanisms	C.3.9	R	R	R	HR
3h	Memorising executed cases	C.3.10	---	R	R	HR
4	Graceful degradation	C.3.11	R	R	HR	HR
5	Artificial intelligence - fault correction	C.3.12	---	NR	NR	NR
6	Dynamic reconfiguration	C.3.13	---	NR	NR	NR
7a	Structured methods including for example, JSD, MASCOT, SADT and Yourdon.	C.2.1	HR	HR	HR	HR
7b	Semi-formal methods	Table B.7	R	R	HR	HR
7c	Formal methods including for example, CCS, CSP, HOL, LOTOS, OBJ, temporal logic, VDM and Z	C.2.4	---	R	R	HR
8	Computer-aided specification tools	B.2.4	R	R	HR	HR

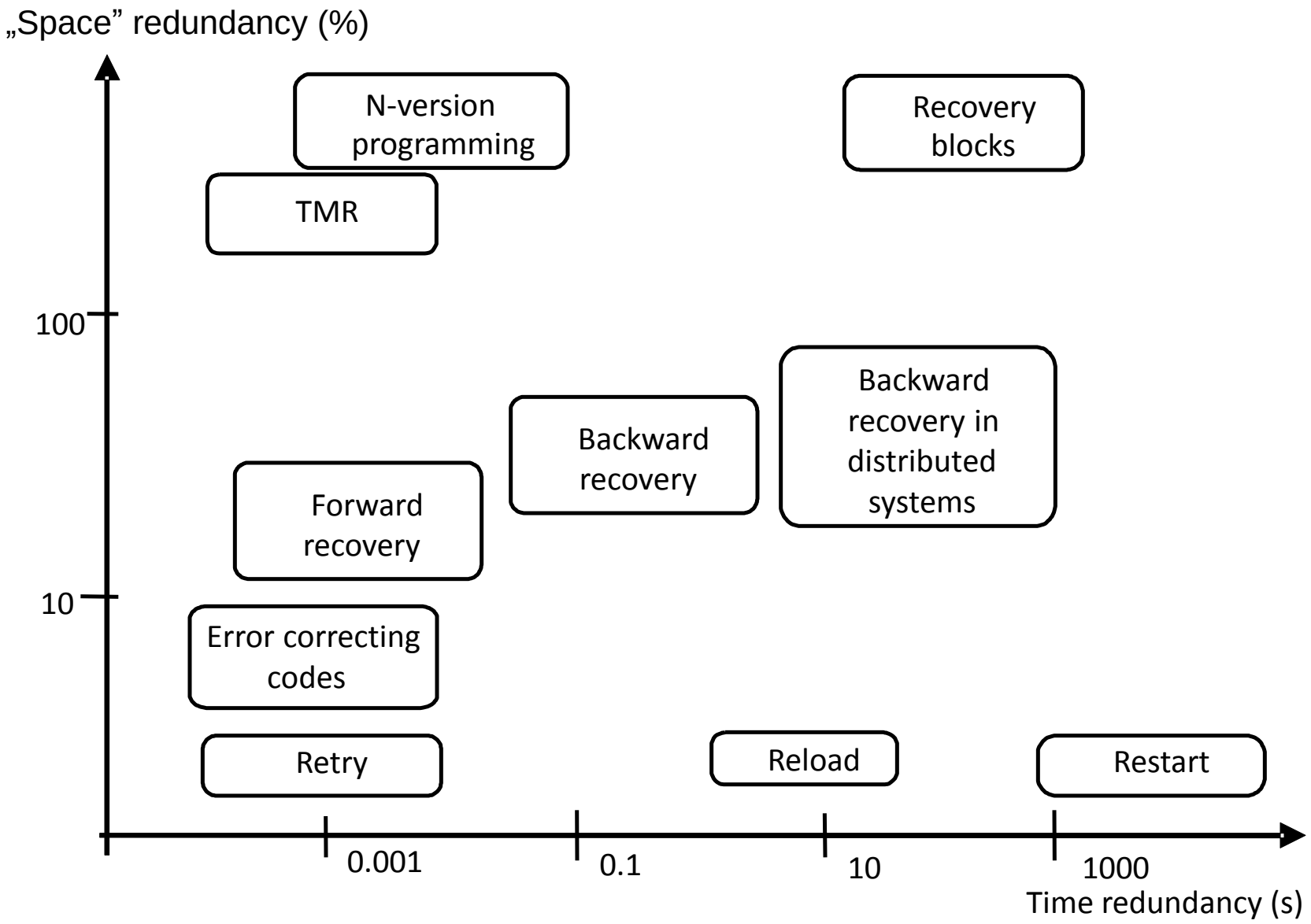
NOTE – The measures in this table concerning fault tolerance (control of failures) should be considered with the requirements for architecture and control of failures for the hardware of the programmable electronics in IEC 61508-2.

* Appropriate techniques/measures shall be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following the number. Only one of the alternate or equivalent techniques/measures has to be satisfied.

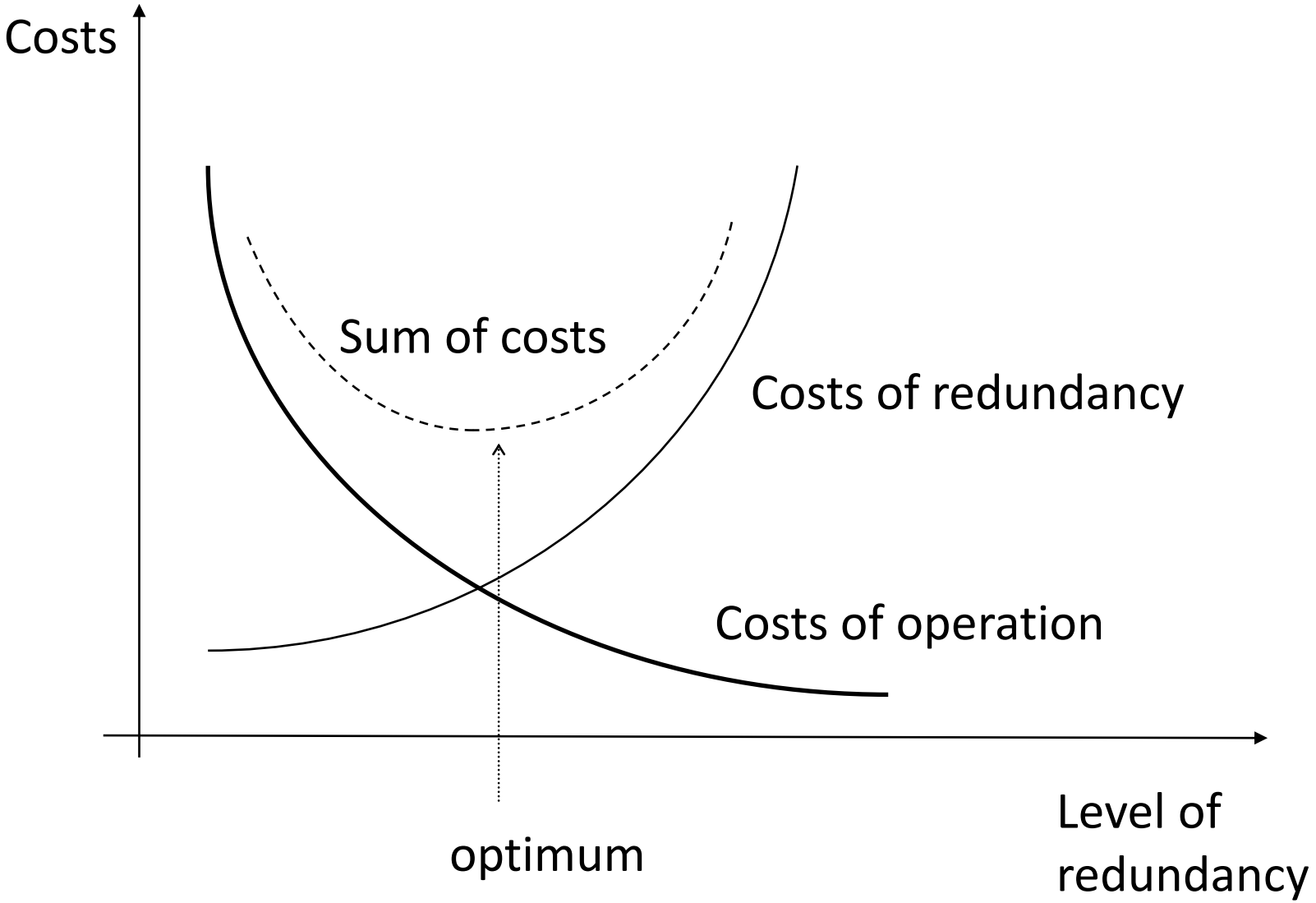
Summary: Time needed for redundancy

- Pure time redundancy: Retry
  - Low-level hardware: processor micro-instruction
  - Higher level: Function, task repeated execution
  - Effective in case of transient faults
- Time overhead: Side effect of other redundancy
  - Hard real-time systems: design aspect to guarantee the execution time of fault handling / tolerance
  - Preferred solutions:
    - Permanent hardware faults: masking, warm redundancy
    - Transient hardware faults: forward error recovery
    - Software (design) faults: N-version programming

Redundancy in space (resources) and time



Costs of redundancy and faults



Testing fault tolerance

- Inducing faults: Fault injection
  - Hardware:
    - Generating “real” faults:
      stuck-at bus signals, power failures,
      particle radiation, temperature shock
    - Hardware dependent, slow
  - Software:
    - Generating fault effects (changing the system state):
      setting registers, memory bits
    - More flexible, faster
    - Questionable whether real faults lead to these effects
  - Hybrid
- Monitoring the effects (in operation)

Summary: Safety architectures

- Fail-stop solutions
  - Single channel with built-in self-checks
  - Dual channel with comparison
  - Dual channel with independent checker
- Fail-operation (fault-tolerance) solutions
  - Hardware design faults: Diverse redundant hardware components
  - Hardware permanent operational faults: Replicated hardware components
  - Hardware transient operational faults:
    - Software implemented redundancy: Error detection and recovery
    - Information redundancy: Error correcting codes
    - Time redundancy: Retrying execution
  - Software design faults: Diverse redundant sw components