# Safety-critical systems: Architecture

Systems Engineering course

(slides: István Majzik)

Budapest University of Technology and Economics
Department of Measurement and Information Systems

## Overview of the goals

# Previous topics

- What did we specify?
  - 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 $\longleftarrow$ ???
  - 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

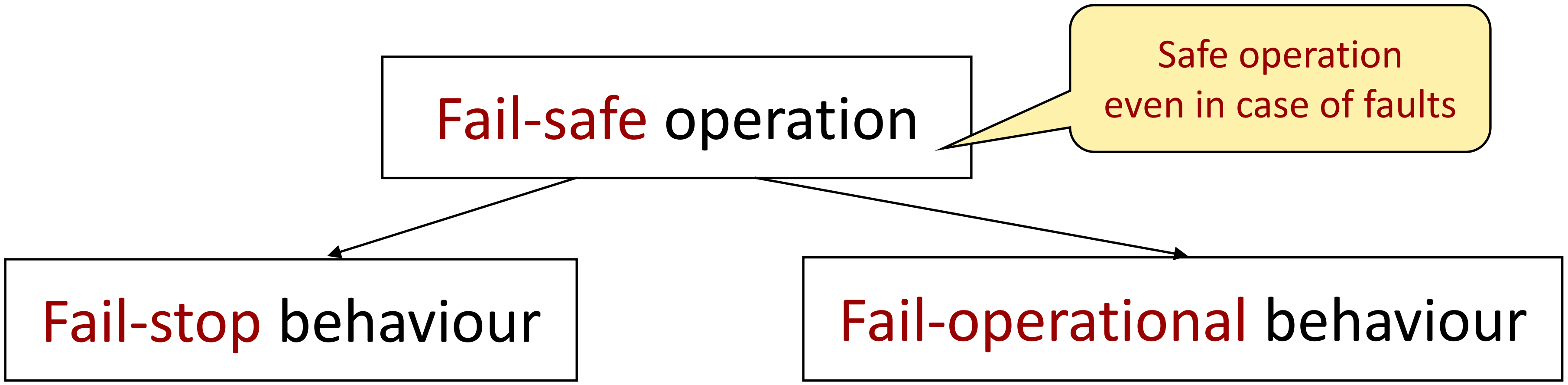
- Focus: Design of system architecture to ...
  - maintain safety
  - handle the effects of faults in hardware and software components

# 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


Fail-safe operation
Safe operation even in case of faults
Fail-stop behaviour
Fail-operational behaviour

- Stopping (switch-off) is a safe state
- In case of a detected error the system has to be stopped
- Error detection is required
- E.g.: X-ray machine

- 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
- E.g.: airplane

# 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
- E.g.: X-ray machine

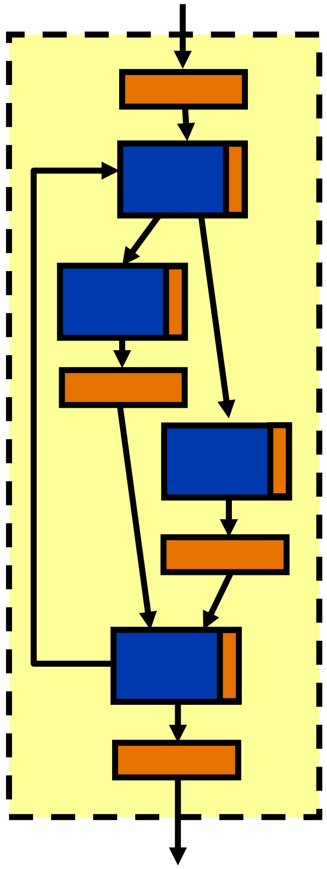
## Fail-operational behaviour

- 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
- E.g.: airplane

## 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



🟦 Function execution

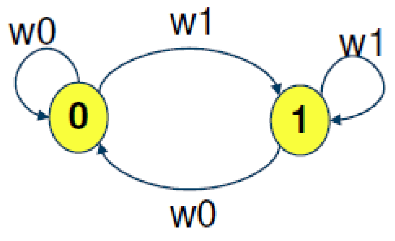
🟧 Error detection

# Implementation of on-line error detection

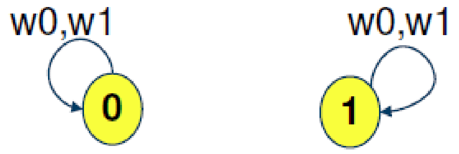
- Application dependent (ad-hoc) techniques
  - Acceptance checking (e.g.: too low, too high value)
  - Timing related checking (e.g.: too early, too late)
  - Cross-checking (e.g.: using inverse function)
  - Structure checking (e.g.: broken structure)
- Application independent (platform) mechanisms
  - 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)

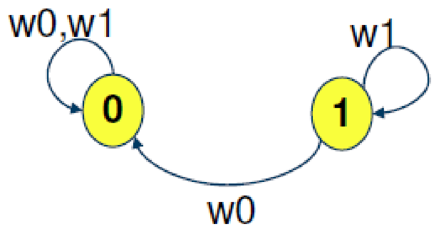
States of a correct cell to be checked:



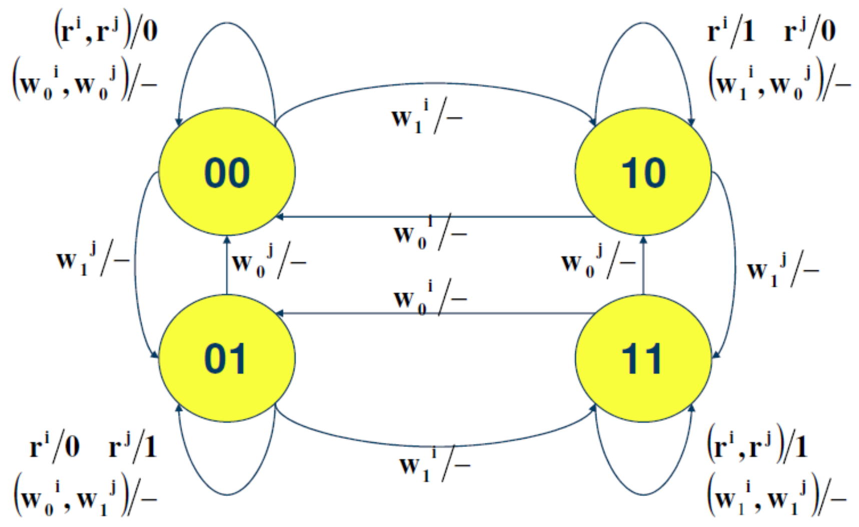
States in case of stuck-at 0/1 faults:



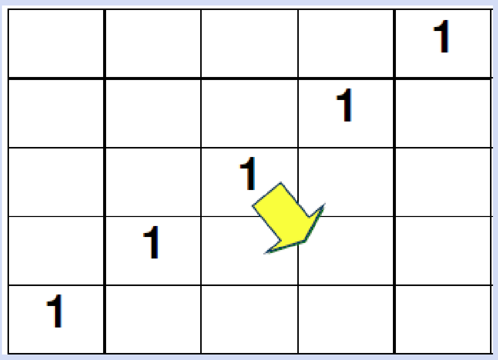
States in case of transition fault:



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



**Testing: „March” algorithms (w/r)**



| | | | | |
|---|---|---|---|---|
| | | | | 1 |
| | | | 1 | |
| | | 1 | | |
| | 1 | | | |
| 1 | | | | |

## Example: Checking execution flow (sw)

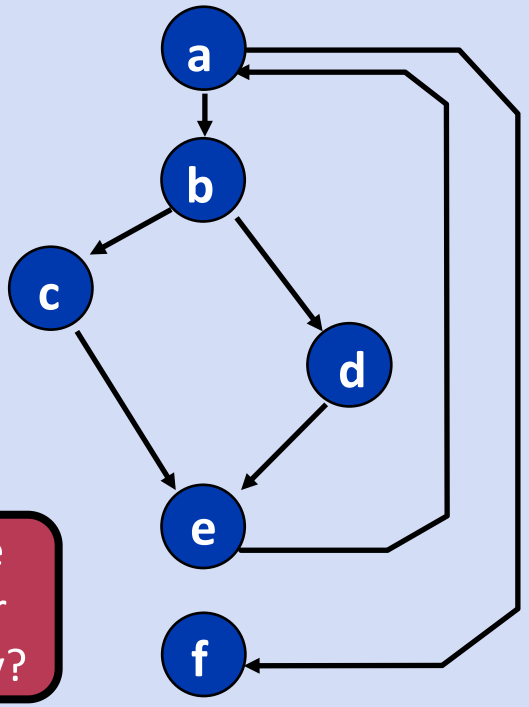
- Checking the correctness of statement sequence
  - Reference for correct behavior: Program control flow graph

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.")
```

What if the compiler or CPU is faulty?

Control flow graph:


$$\begin{tikzcd}
 & a \arrow[d] \arrow[ddddd, bend left=60] & \\
 & b \arrow[dl] \arrow[dr] & \\
c \arrow[ddr] & & \\
 & & d \arrow[dl] \\
 & e \arrow[uuuu, bend right=60] & \\
 & f &
\end{tikzcd}$$

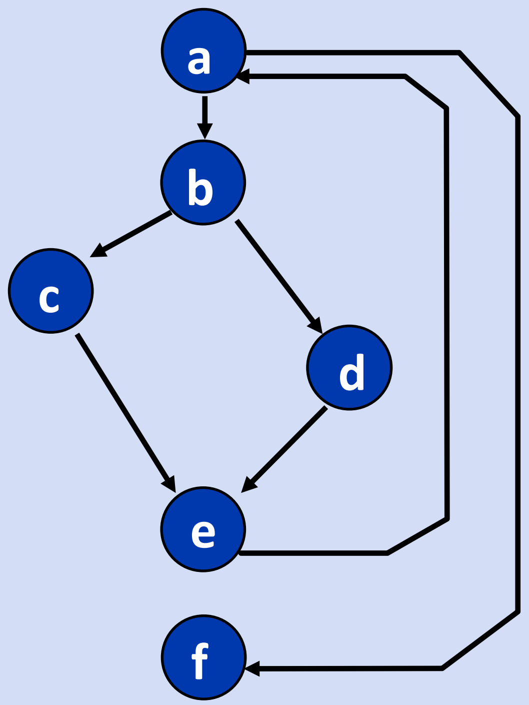
# Example: Checking execution flow (sw)

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

Instrumented source code:

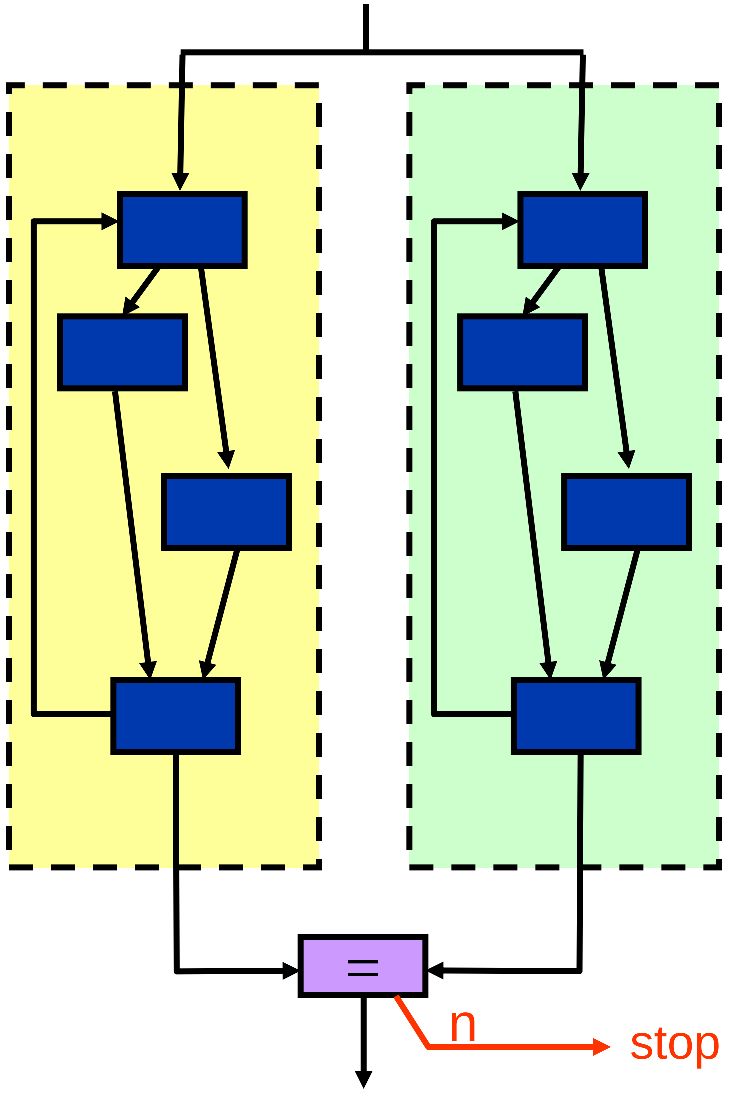
```
a:  S(a); for (i=0; i<MAX; i++) {
b:        S(b); if (i==a) {
c:              S(c); n=n-i;
          } else {
d:              S(d); m=m-i;
          }
e:        S(e); printf(“%d\n”,n);
    }
f:  S(f); printf(“Ready.”)
```

Control flow graph:

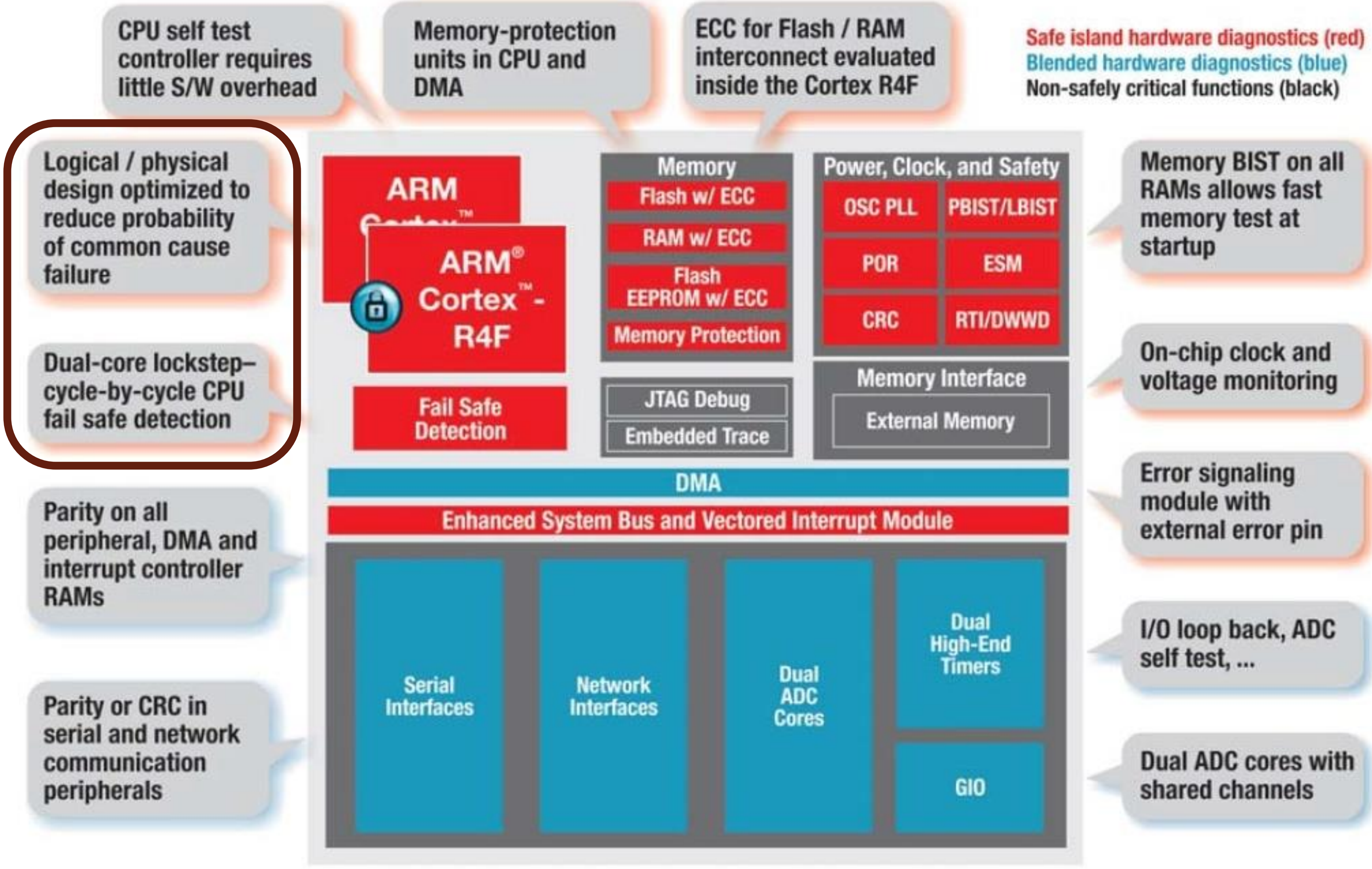


## 2. Two-channels architecture with comparison

- 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

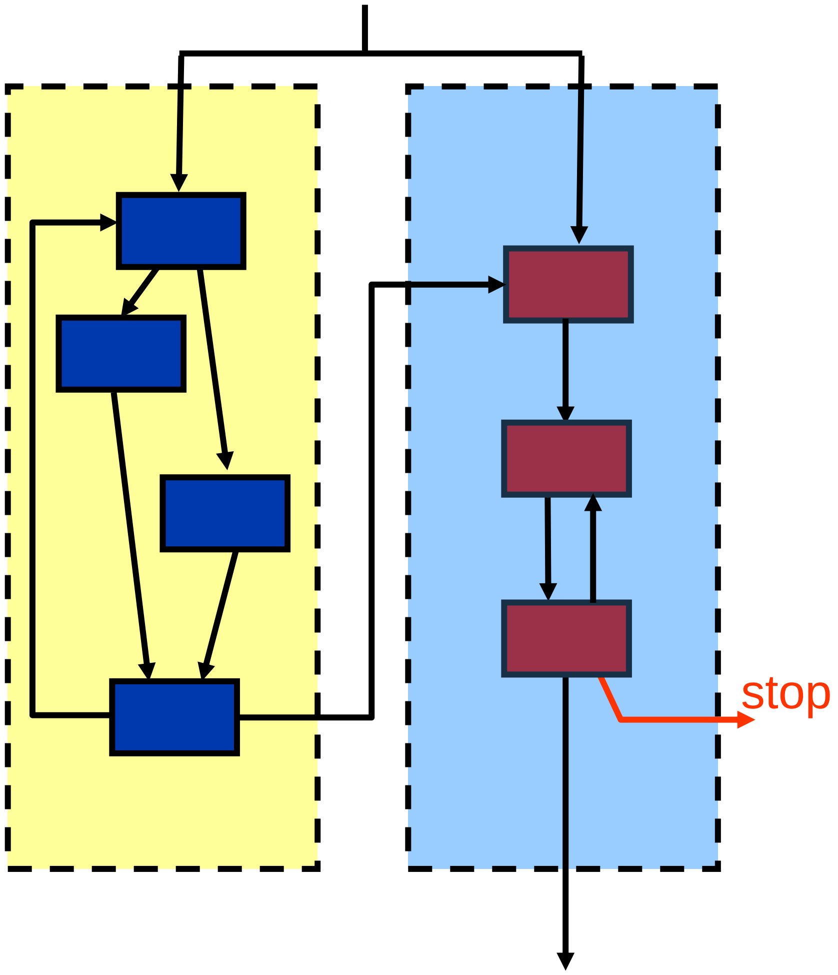


# Example: TI Hercules Safety Microcontrollers

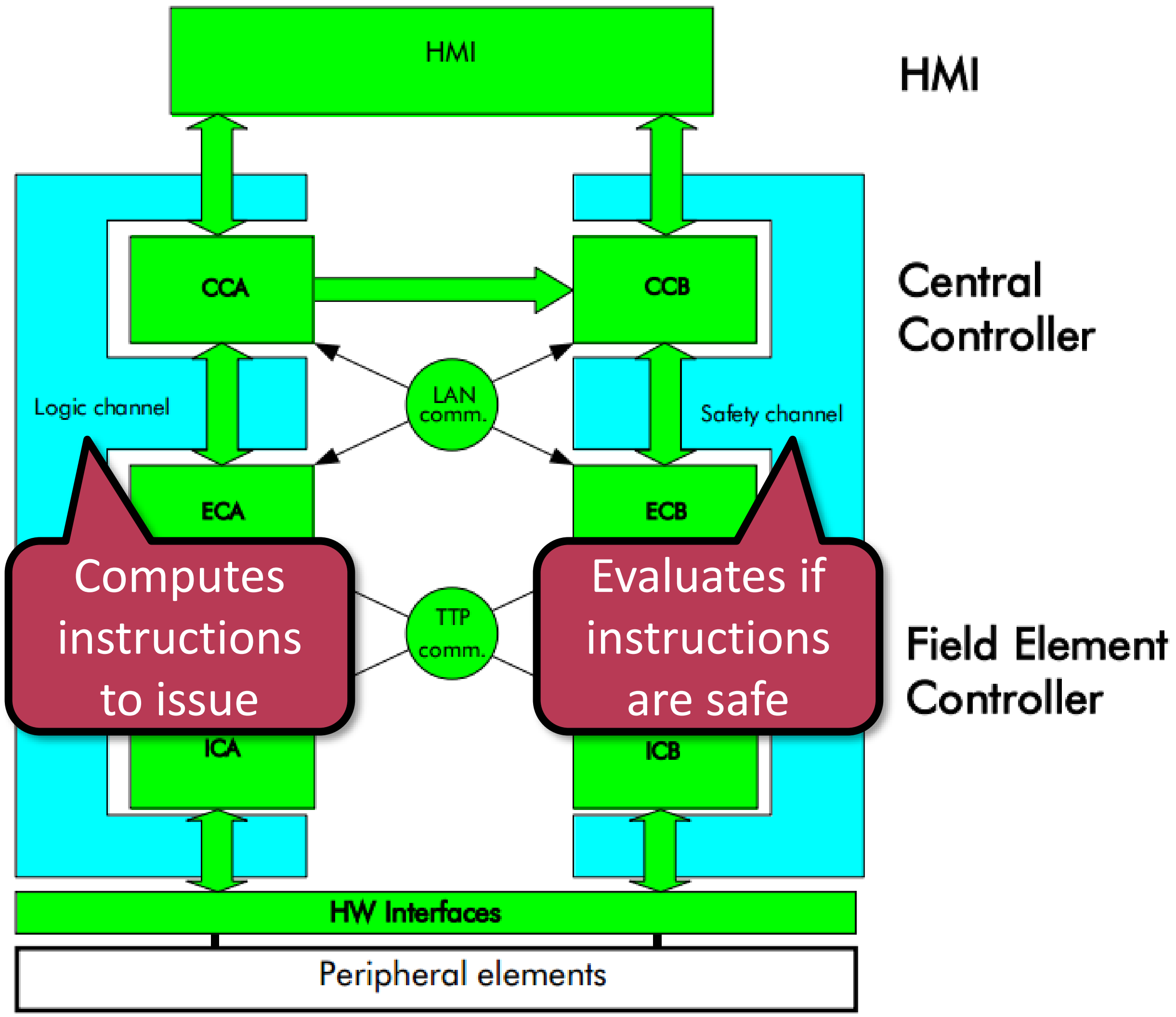

CPU self test controller requires little S/W overhead
Memory-protection units in CPU and DMA
ECC for Flash / RAM interconnect evaluated inside the Cortex R4F
Safe island hardware diagnostics (red)
Blended hardware diagnostics (blue)
Non-safely critical functions (black)
Logical / physical design optimized to reduce probability of common cause failure
Dual-core lockstep– cycle-by-cycle CPU fail safe detection
Parity on all peripheral, DMA and interrupt controller RAMs
Parity or CRC in serial and network communication peripherals
ARM Cortex™
ARM® Cortex™- R4F
Fail Safe Detection
Memory
Flash w/ ECC
RAM w/ ECC
Flash EEPROM w/ ECC
Memory Protection
JTAG Debug
Embedded Trace
Power, Clock, and Safety
OSC PLL
PBIST/LBIST
POR
ESM
CRC
RTI/DWWD
Memory Interface
External Memory
DMA
Enhanced System Bus and Vectored Interrupt Module
Serial Interfaces
Network Interfaces
Dual ADC Cores
Dual High-End Timers
GIO
Memory BIST on all RAMs allows fast memory test at startup
On-chip clock and voltage monitoring
Error signaling module with external error pin
I/O loop back, ADC self test, ...
Dual ADC cores with shared channels

## 3. Two-channels architecture with safety checking

- Independent second channel
  - Safety bag/Monitor/Shield: only safety checking
  - Diverse implementation
  - Checking the output of the primary channel
    - E.g. command is not dan-gerous at the time of issue
- Advantages
  - Explicit safety rules
  - Independence of the checker channel



# Example: Elektra interlocking system



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**

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
- E.g.: X-ray machine

**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
- E.g.: airplane

# 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

- Subjects of redundancy:
  - Hardware redundancy
    - Extra hardware components (inherent in the system or planned for fault tolerance)
  - Software redundancy
    - Extra software modules (e.g. multiple, diverse implementations)
  - Information redundancy
    - Extra information (e.g., error correcting codes)
  - Time redundancy
    - Repeated execution (to handle transient faults)
- Availability of redundant resources
  - 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 redundancy?

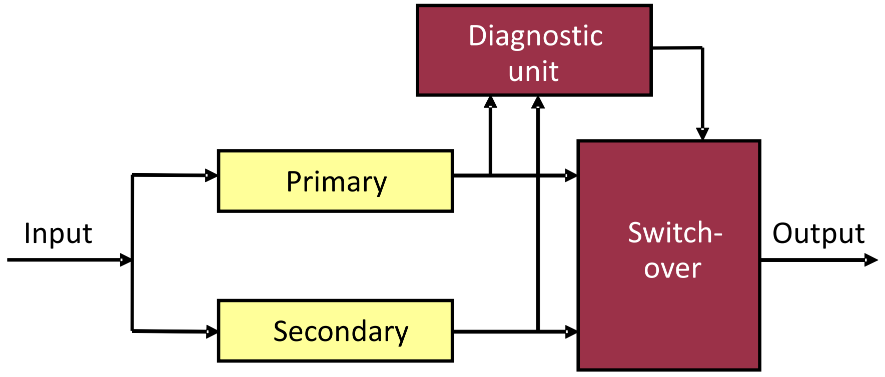
- Hardware design faults: (< 1%)
  - Hardware redundancy with design diversity
- Hardware permanent operational faults: (~ 20%)
  - Hardware redundancy (e.g.: redundant processor)
- Hardware transient operational faults: (~70-80%)
  - Time redundancy (e.g.: instruction retry)
  - Information redundancy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

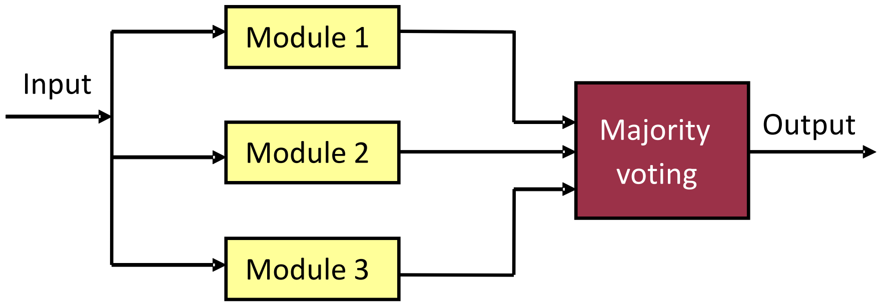
With diversity in case of considering design faults

Replication:

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



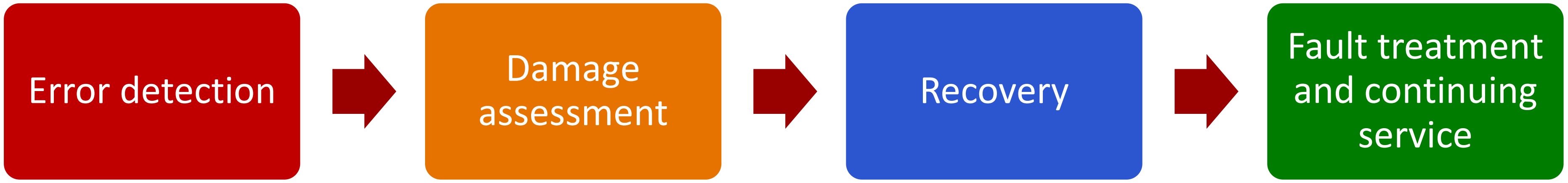
- TMR: Triple Modular Redundancy
  - Masking the failure
    by majority voting
  - Voter is a critical component
    (but simple)



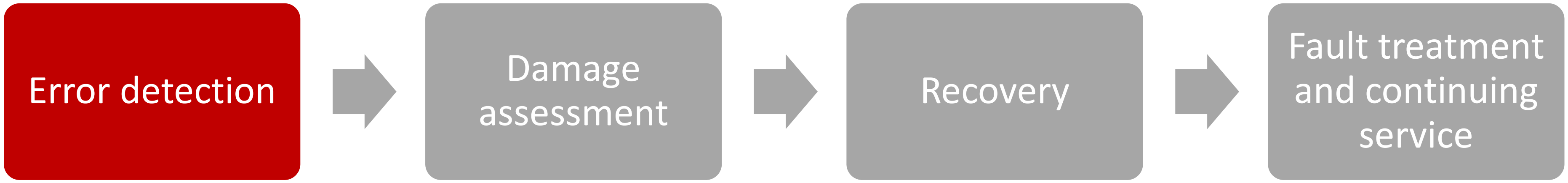
- NMR: N-modular redundancy
  - Masking the failure by majority voting
  - Mission critical systems: Surviving the mission time

## 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:



# Phase 1: Error detection

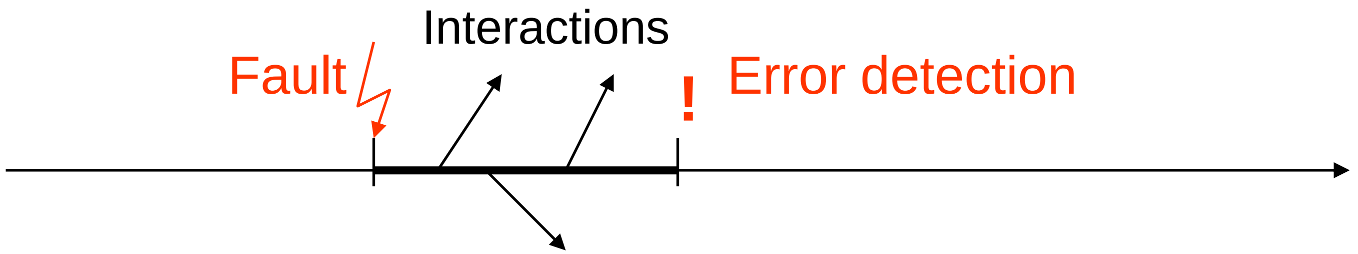


- Application independent mechanisms:
  - E.g., detecting illegal instructions at CPU level
  - E.g., detecting violation of memory access restrictions
- Application dependent techniques:
  - Acceptance checking
  - Timing related checking
  - Cross-checking
  - Structure checking
  - Diagnostic checking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

# Phase 2: Damage assessment

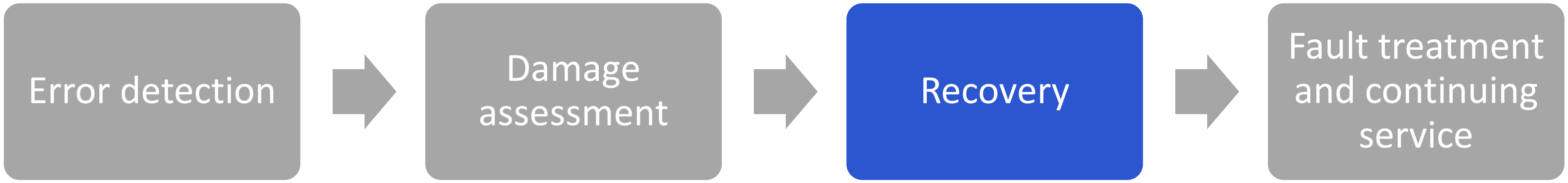
Error detection → **Damage assessment** → Recovery → Fault treatment and continuing service

- 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)

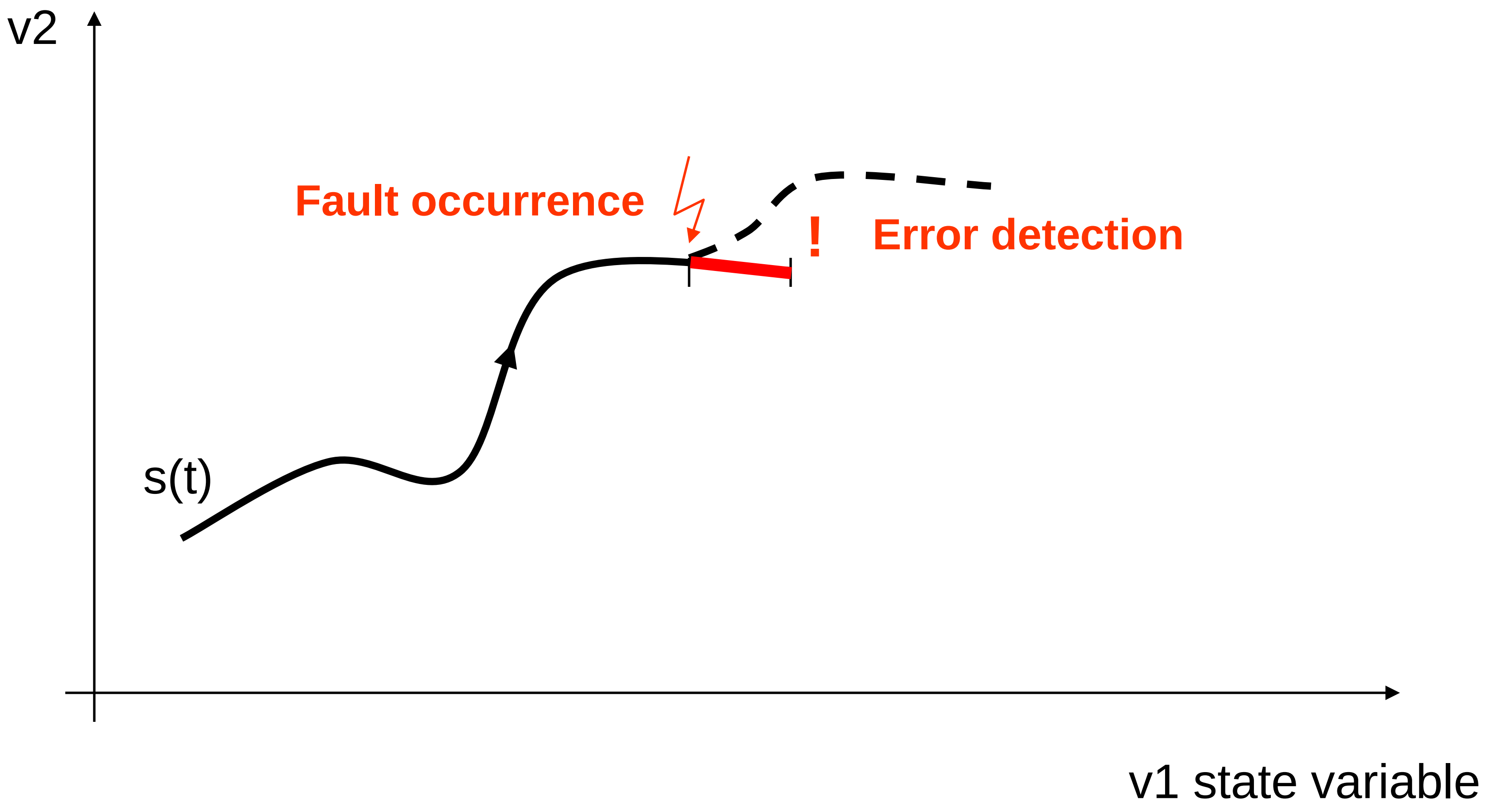
# Phase 3: Recovery



- 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

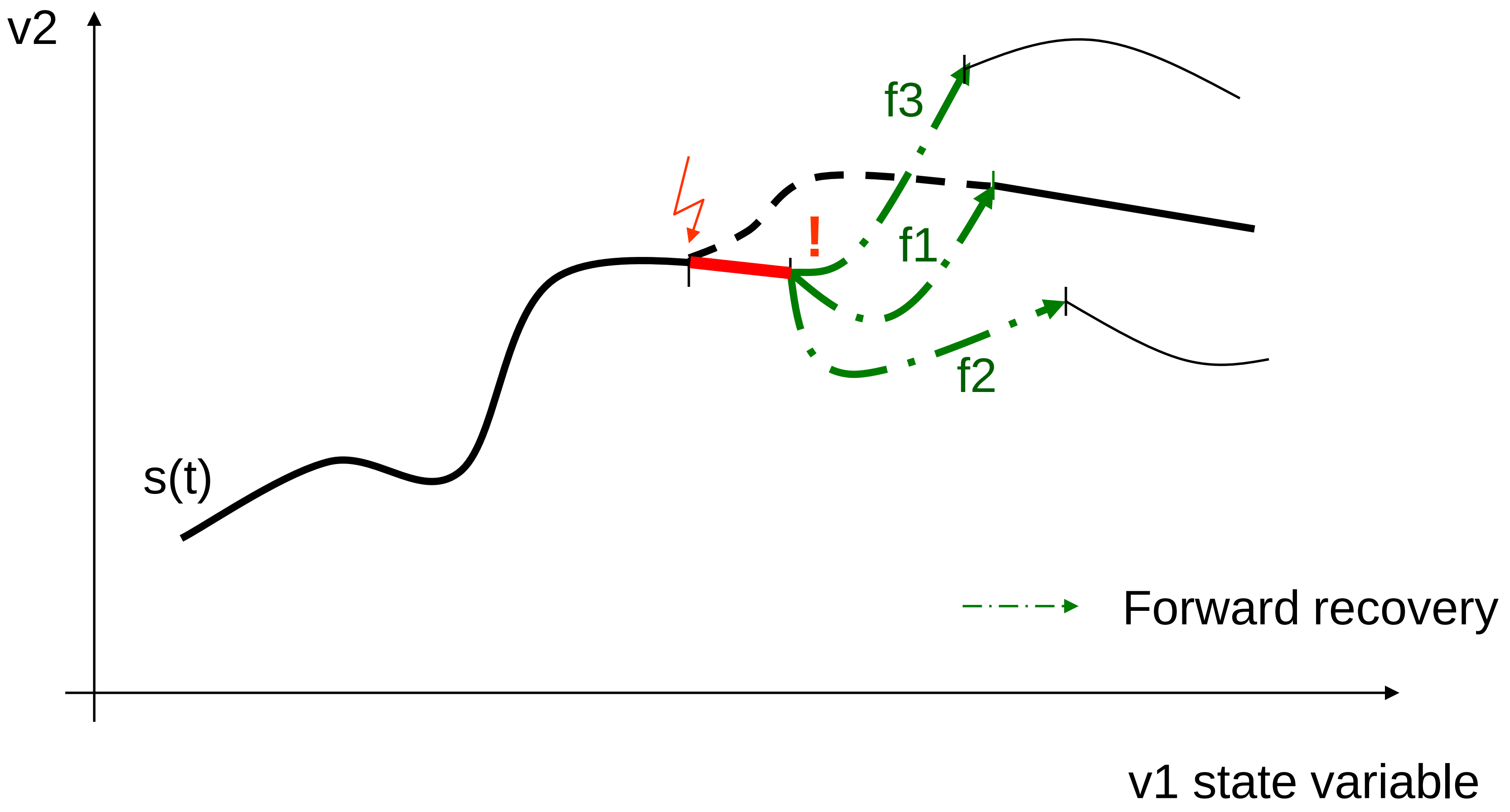
# Types of recovery

- State space of the system: Error detection



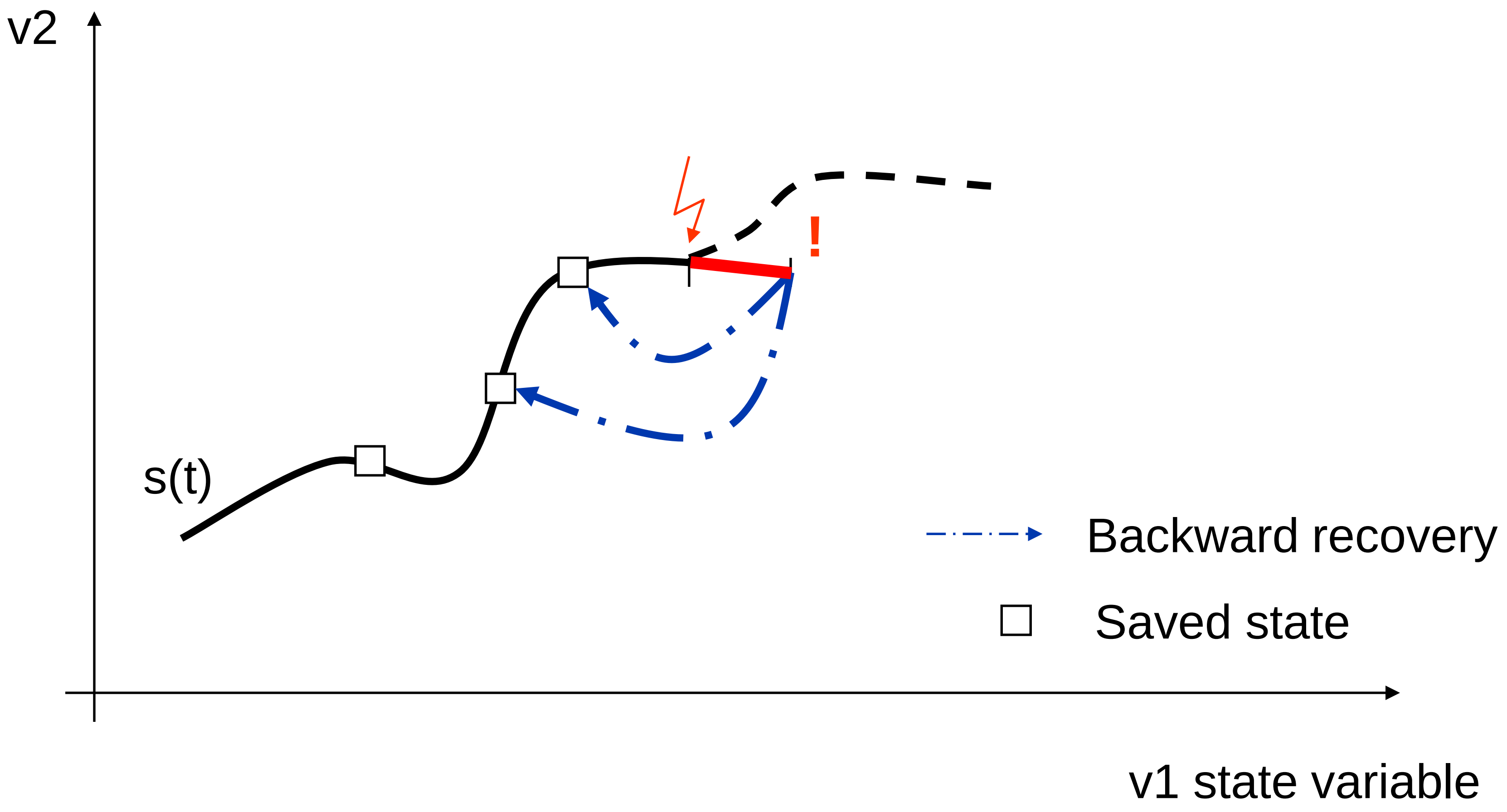
# Types of recovery

- State space of the system: Forward recovery



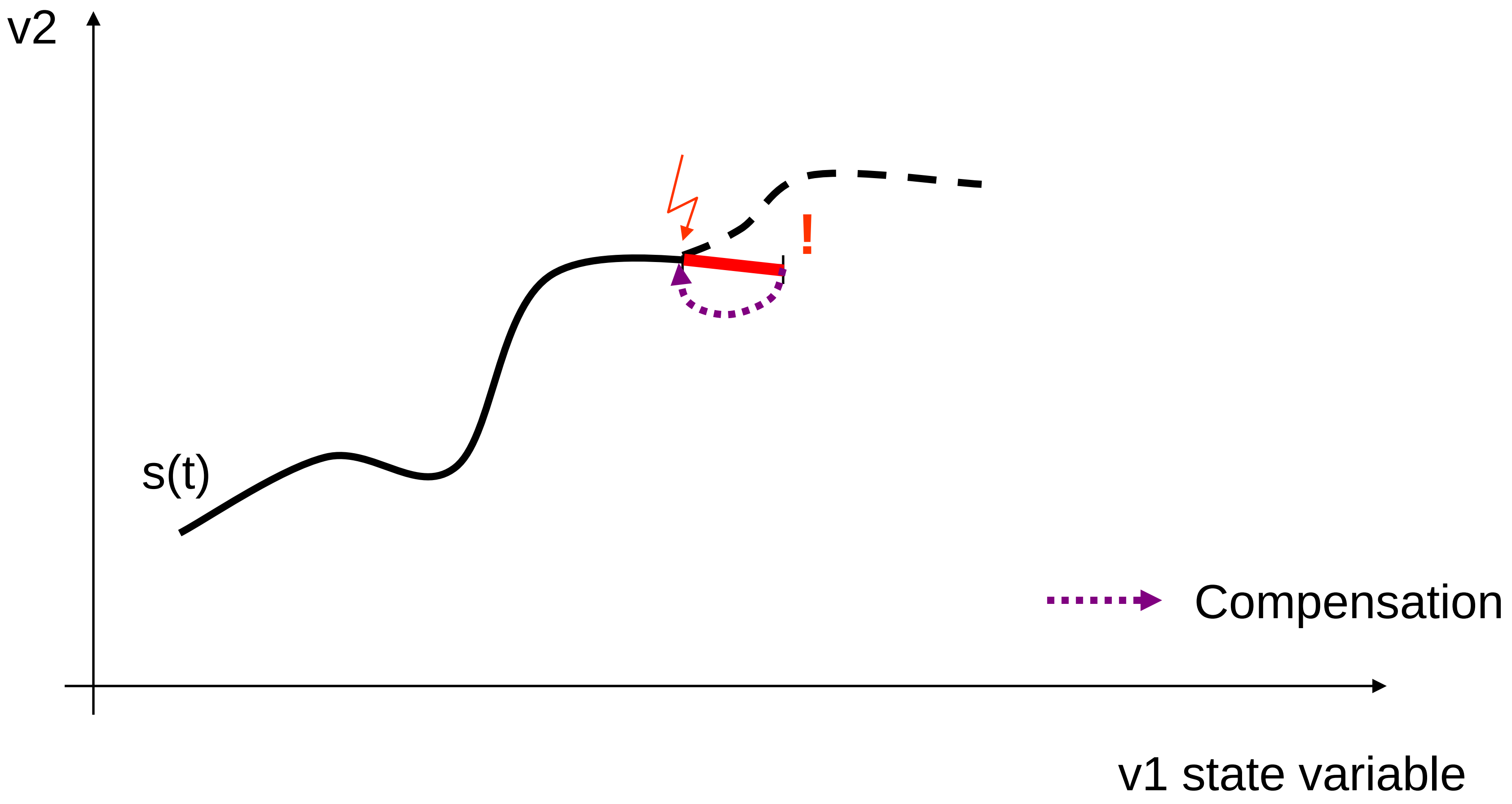
# Types of recovery

- State space of the system: Backward recovery



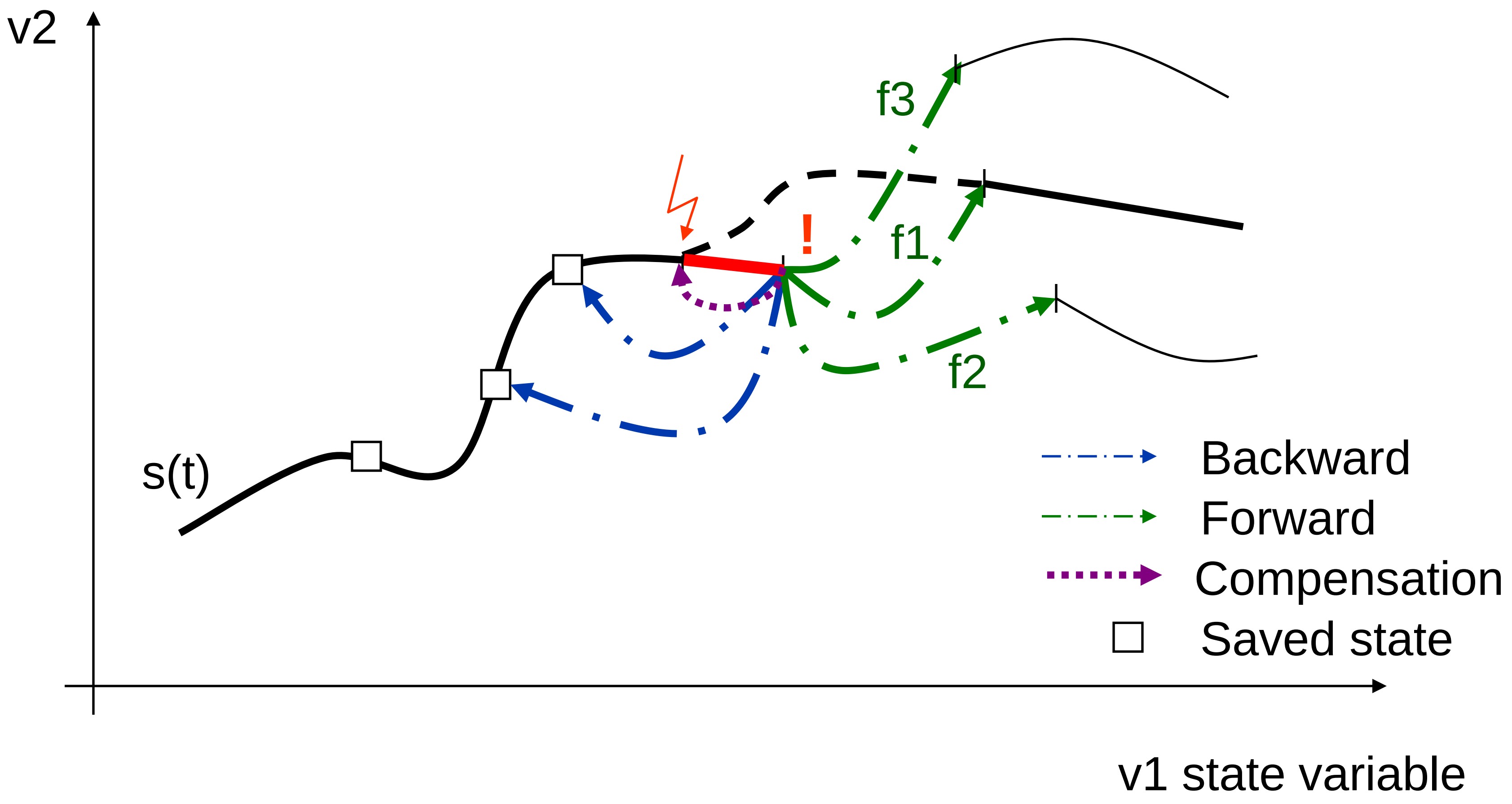
## Types of recovery

- State space of the system: Compensation



# Types of recovery

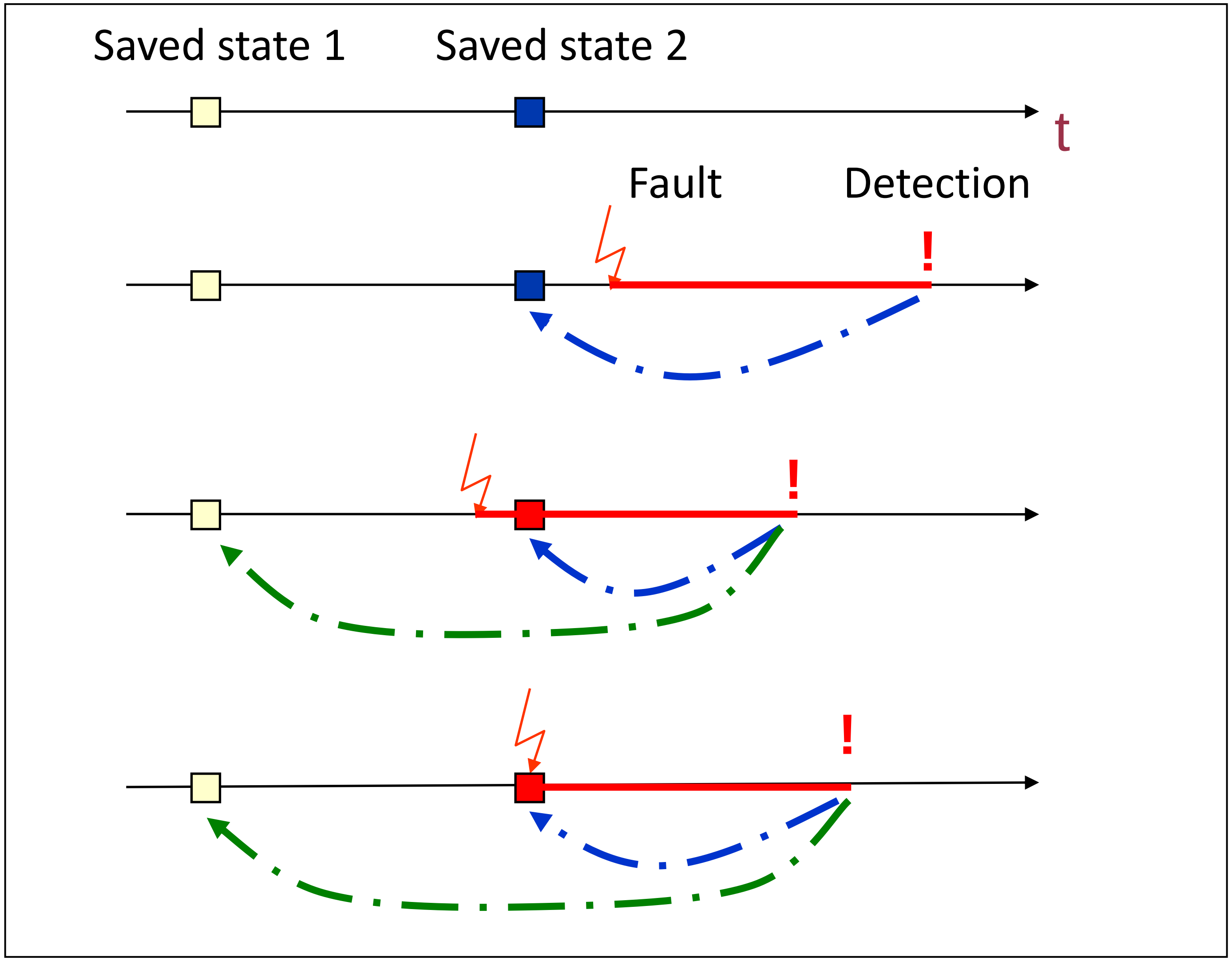
- State space of the system: Types of recovery



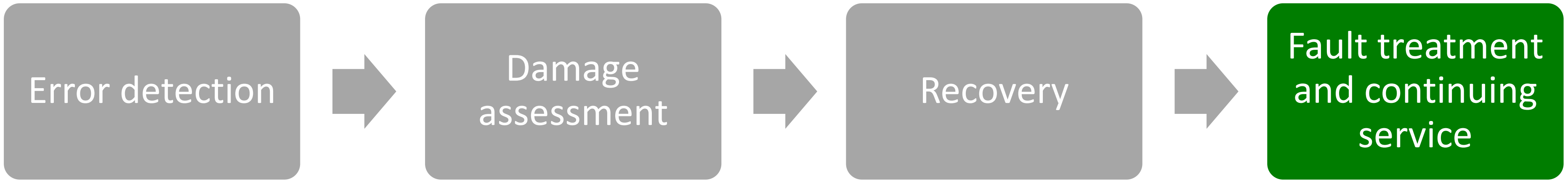
## Backward recovery

- 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

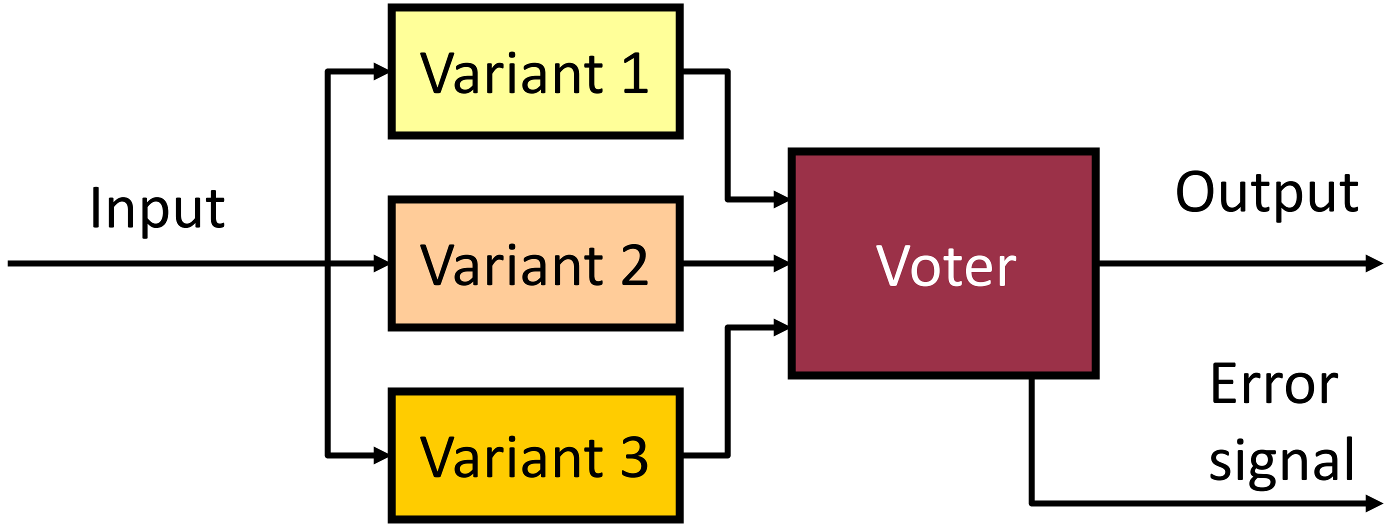
- Repeated execution is not effective for design faults!
- Redundancy with design diversity is required

  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

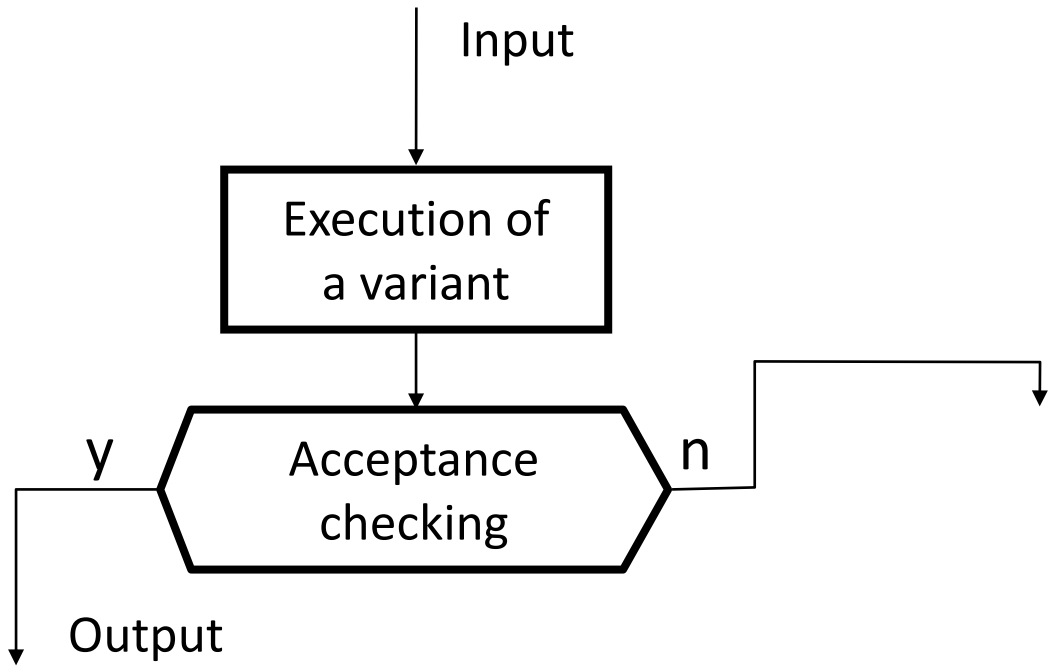
# N-version programming

- 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)



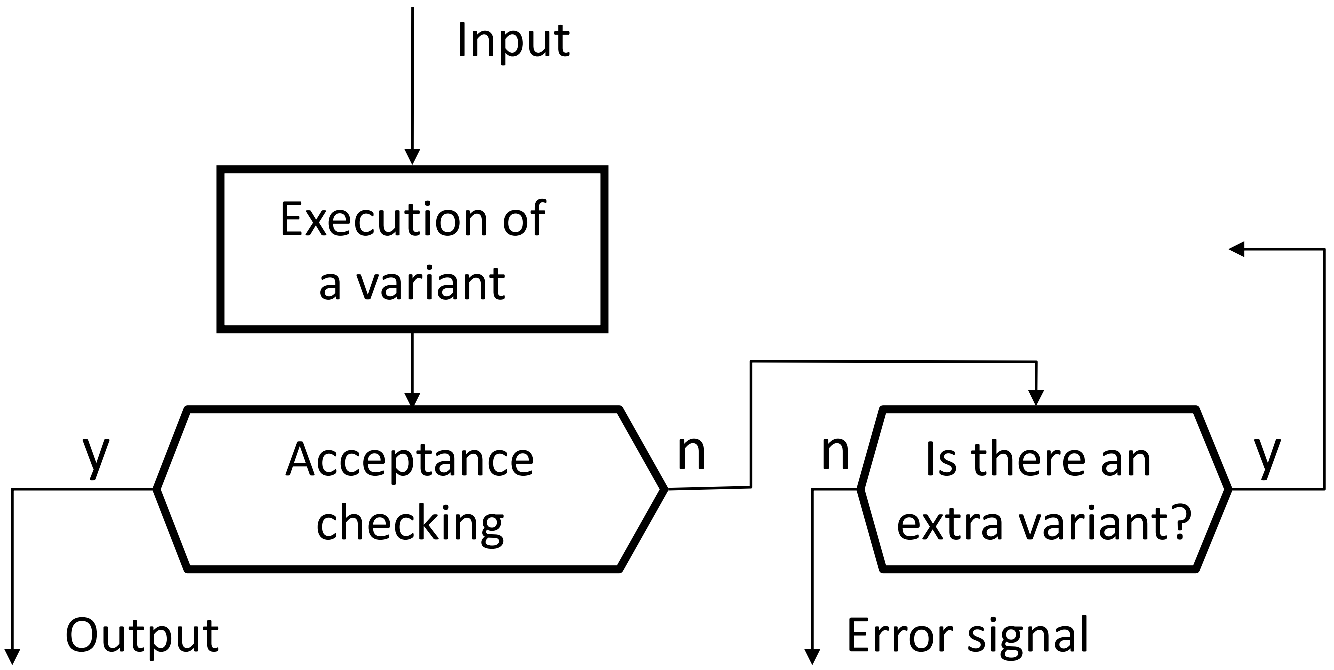
# Recovery blocks

- 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



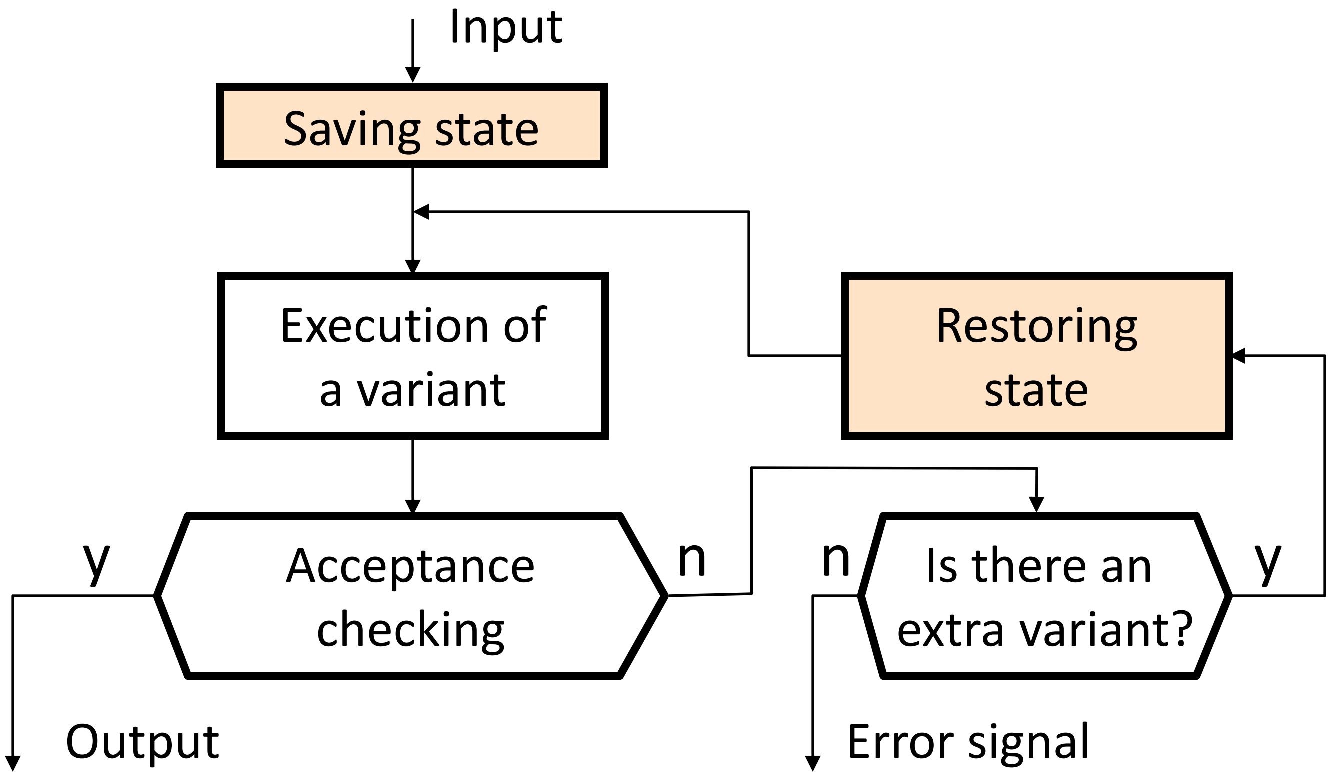
# Recovery blocks

- 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)