# Architecture in Dependable Embedded Systems

Dr. Isabella Stilkerich, Felix Bräunling, Dr. Ulrich Becker

# Overview

Systems and Software Development

Architecture Goals

Dependability and Functional Safety

Real-Time and Concurrency

# Overview

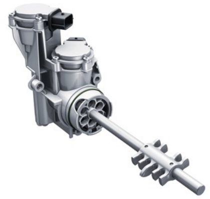
**Systems and Software Development**

Architecture Goals

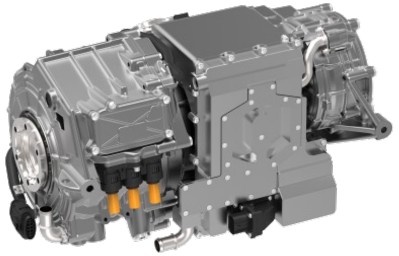
Dependability and Functional Safety

Real-Time and Concurrency

# Example: Schaeffler's Embedded Systems



Gearbox actuator



eAxle

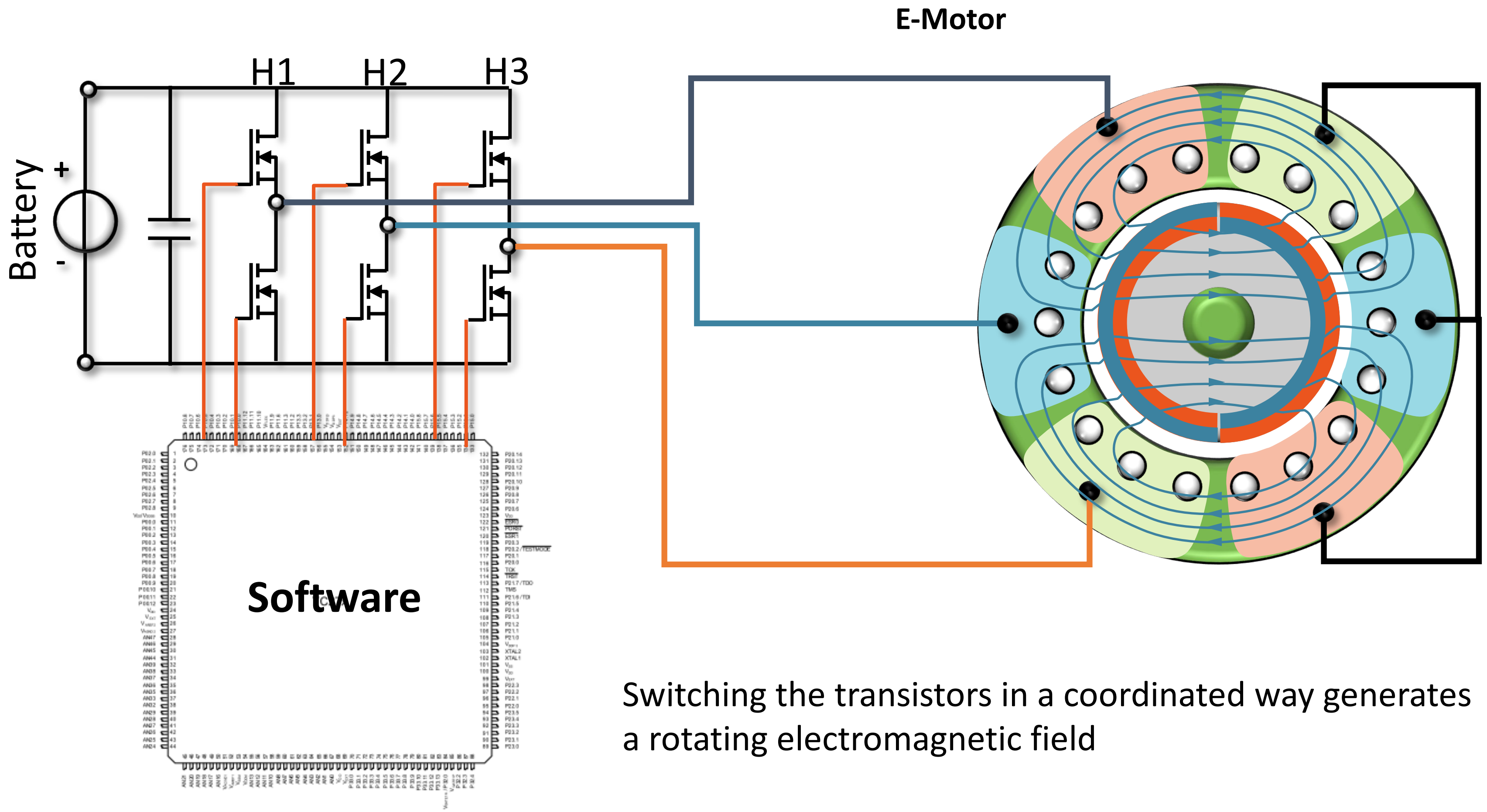
Hybrid Module

Active Roll-Stabilizer

E-Wheel Drive

A wide range of these applications use an embedded system for e-motor control

# Embedded System E-Motor Control



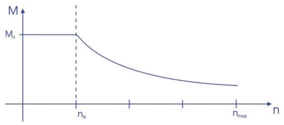

Switching the transistors in a coordinated way generates a rotating electromagnetic field

# Functional Features

## Motor types

- Permanent magnet synchronous motor
- Asynchronous induction motor

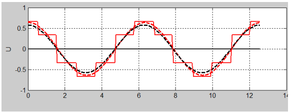
## Electric current control



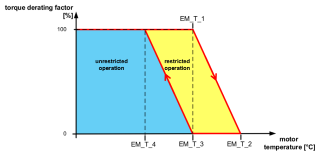

- Field oriented control
- Feed forward, magnetic saturation, reluctance
- Field weakening control
- (Over-)modulation schemes and variable switching frequencies

## Superimposed controllers

- Speed (window) control
- Jerk control

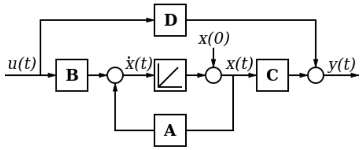


## Derating and Diagnostics



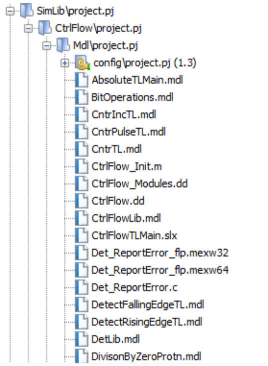

- Self protection and fault detection
- Performance derating

## Sensors and Observers




- Angle tracking observer
- Power loss and temperature estimation
- Magnetic flux in stator windings

## Libraries for various utilities




- Table lookup and interpolation
- Numerical routines
- Signal filters

# Overview

Systems and Software Development

**Architecture Goals**

Dependability and Functional Safety

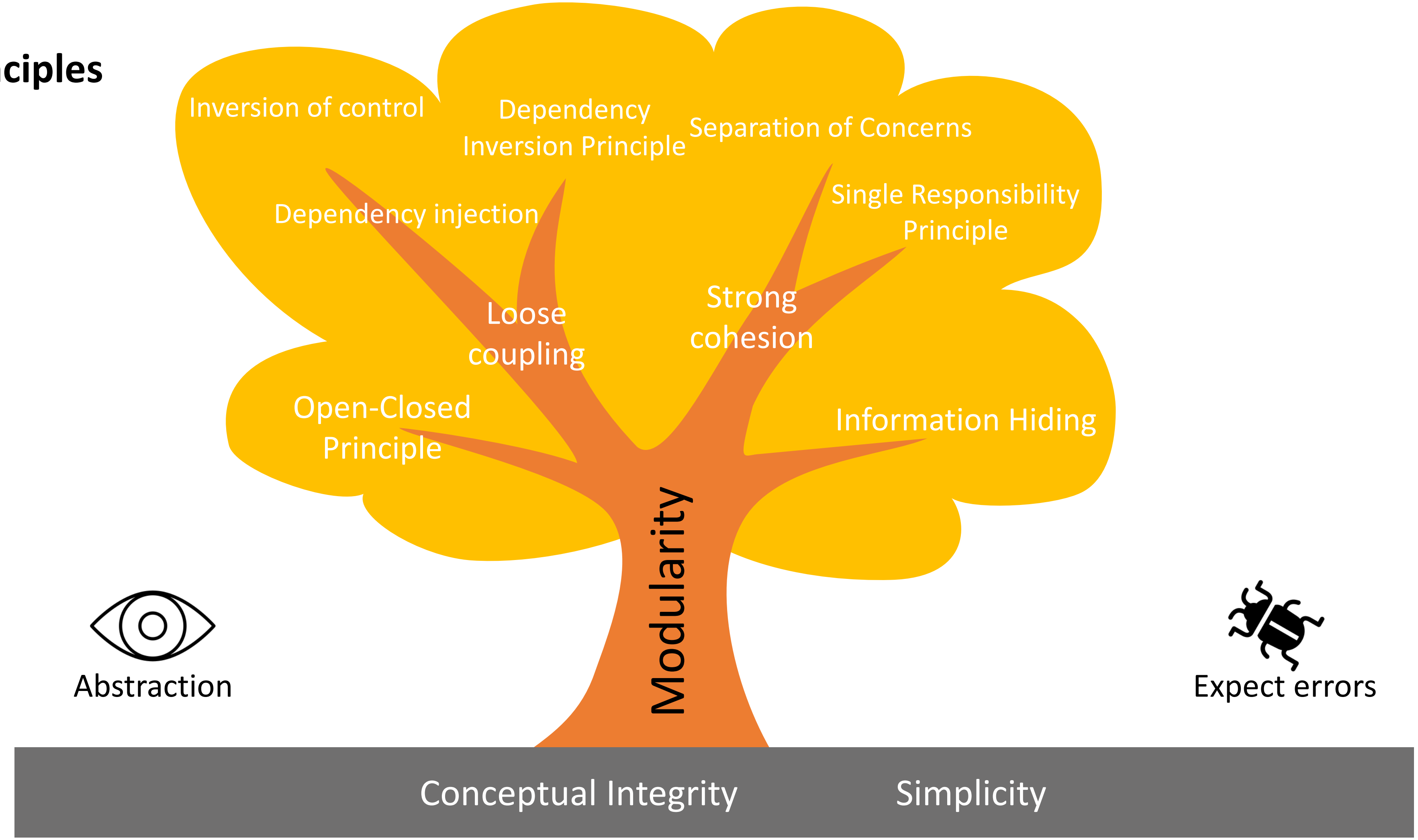
Real-Time and Concurrency

# Important Qualities: Architecture Goals

- High intelligence and complexity of the control software (selected of qualities):
  - Functional correctness: torque precision, dynamics, safety
  - Performance efficiency: time behavior, resource utilization, energy
  - Reliability: availability
  - Security
  - Portability: adaptability
  - Maintainability
- Qualities are often cross-cutting concerns
- Technical constraint: Use of AUTOSAR (Automotive Open System Architecture)

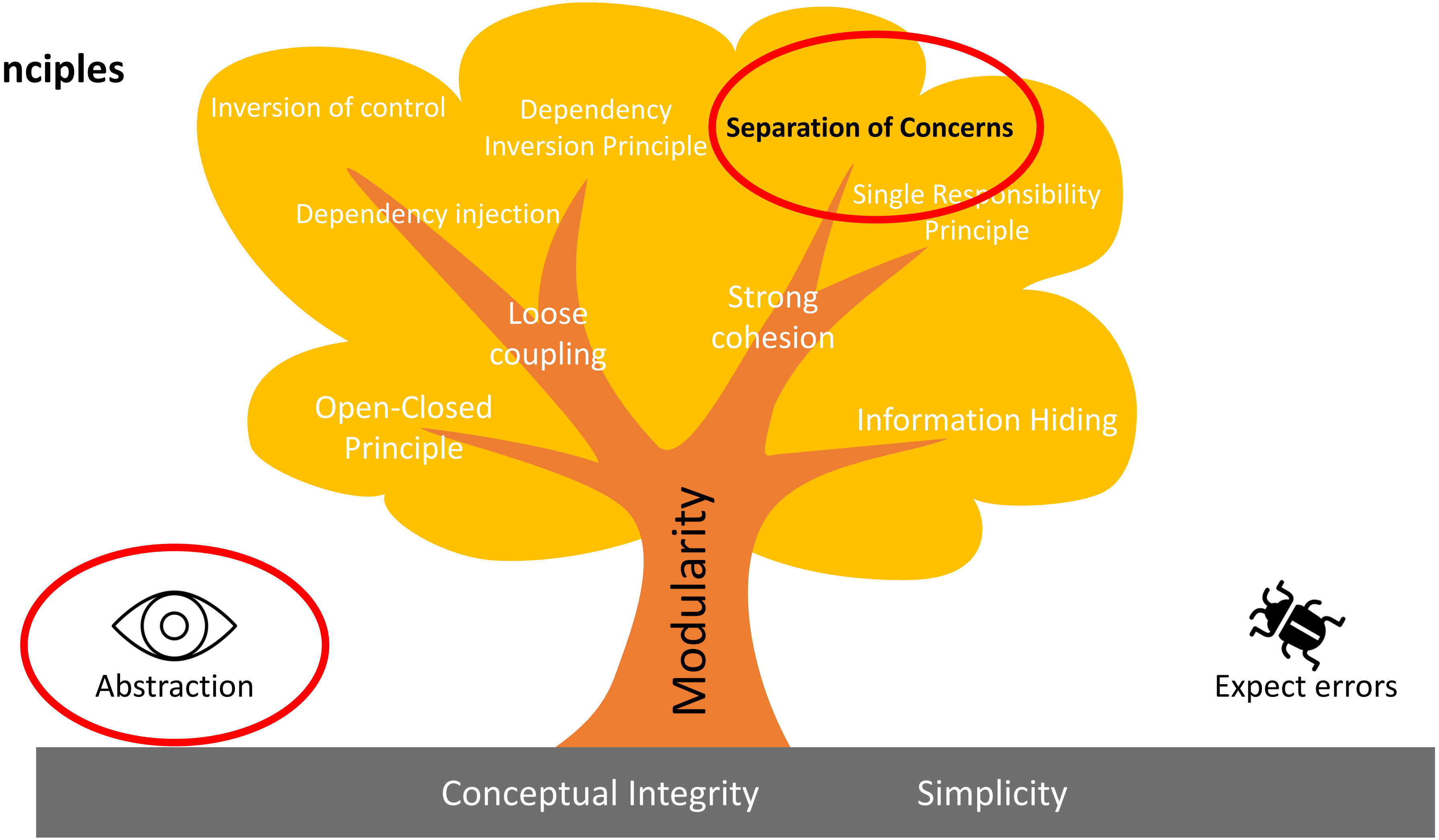
# How to address these complex topics?

**Design Principles**



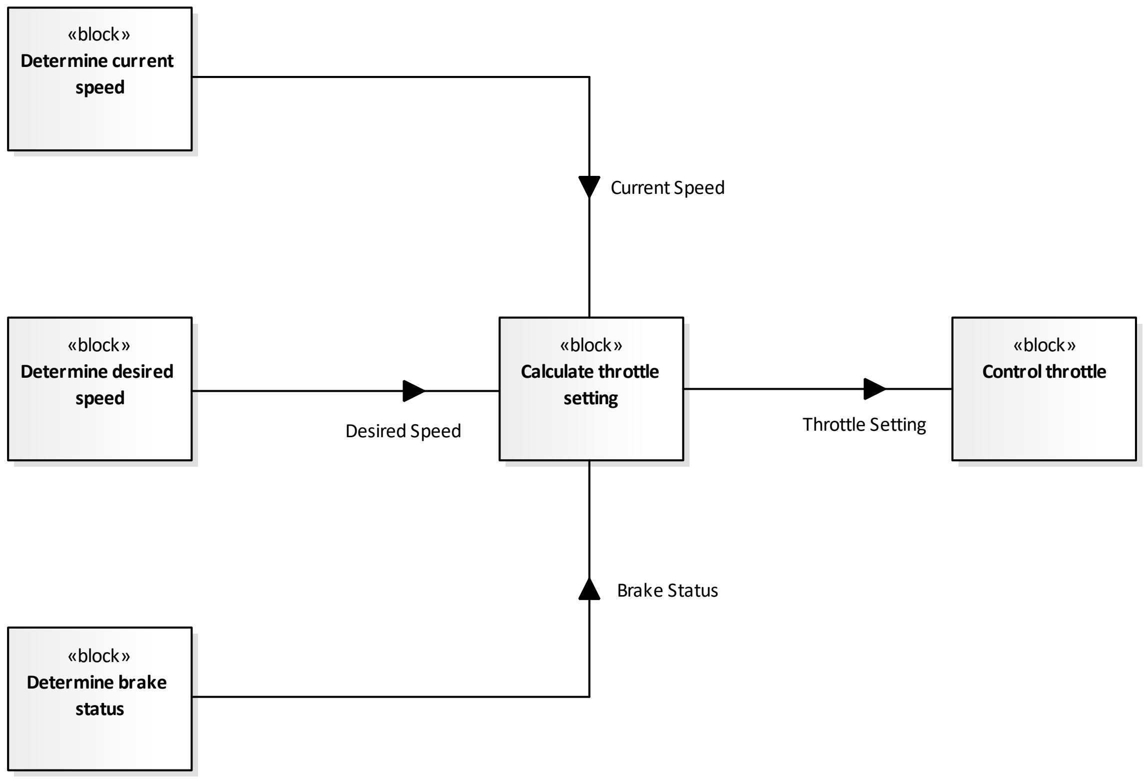
# How to address these complex topics?

**Design Principles**



# Functional Architecture (1)

## Example: Cruise Control

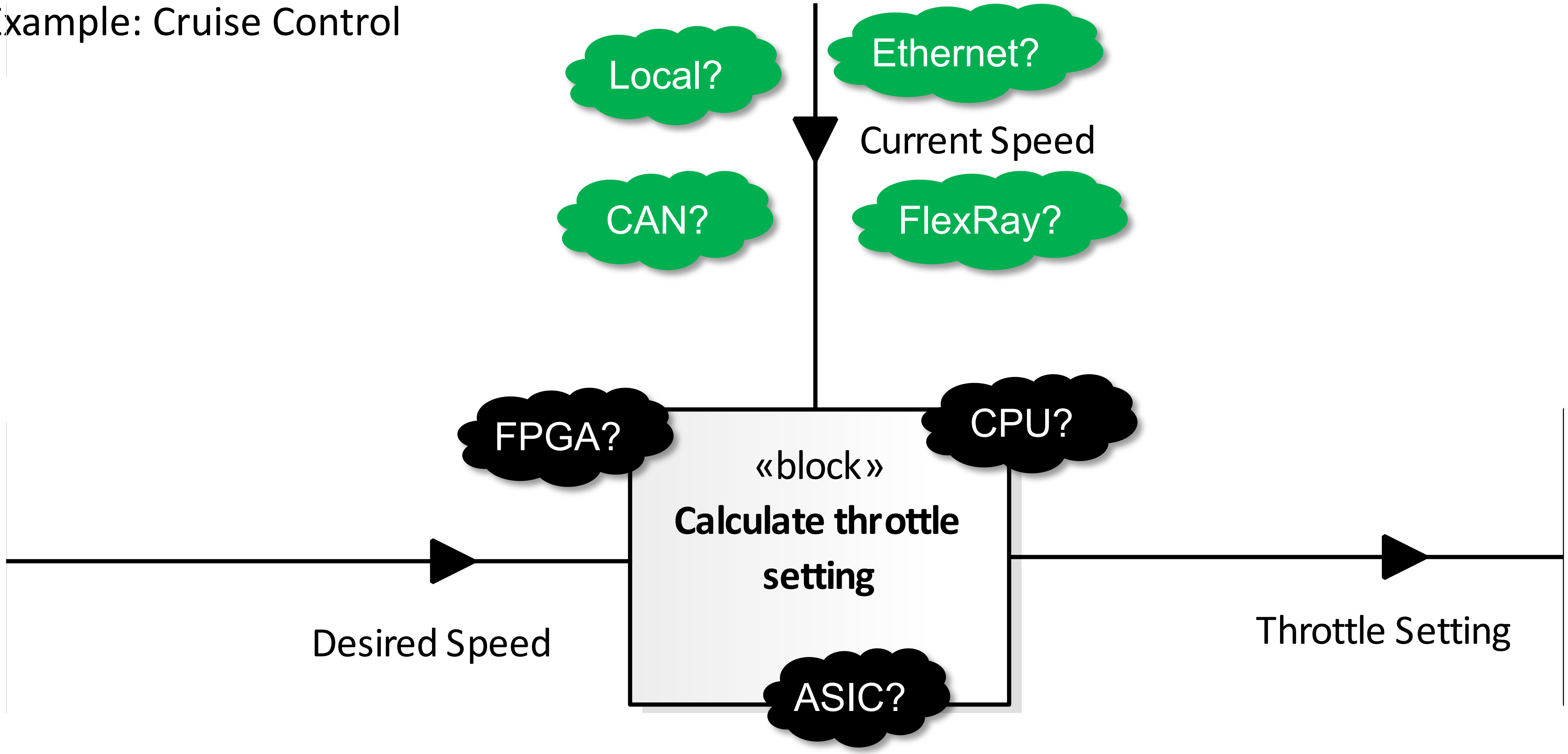


# Functional Architecture (2)

- A **functional architecture** represents knowledge about the core function logic:
  - **Central concepts** of the core functional logic, their **attributes** and **relationships**
  - Enables a better **understanding** of the function logic
  - Establishes a **common language**
  - Helps to detect **inconsistencies** and **redundancies**
  - Builds the connection to requirements engineering (cf. domain models)
- Functional architectures **abstract from technical aspects**
  - The core functional architecture is **independent of technical concerns** and has an **independent life cycle**
- Modeling includes **structure** (e.g. representation of concepts and their relationships in a class diagram) and **behavior** (e.g. modeling of interaction of structural elements in a sequence diagram)
- **Experts** for **functional architectures** are often **not software developers**, but experts for electric motors, physicians, …

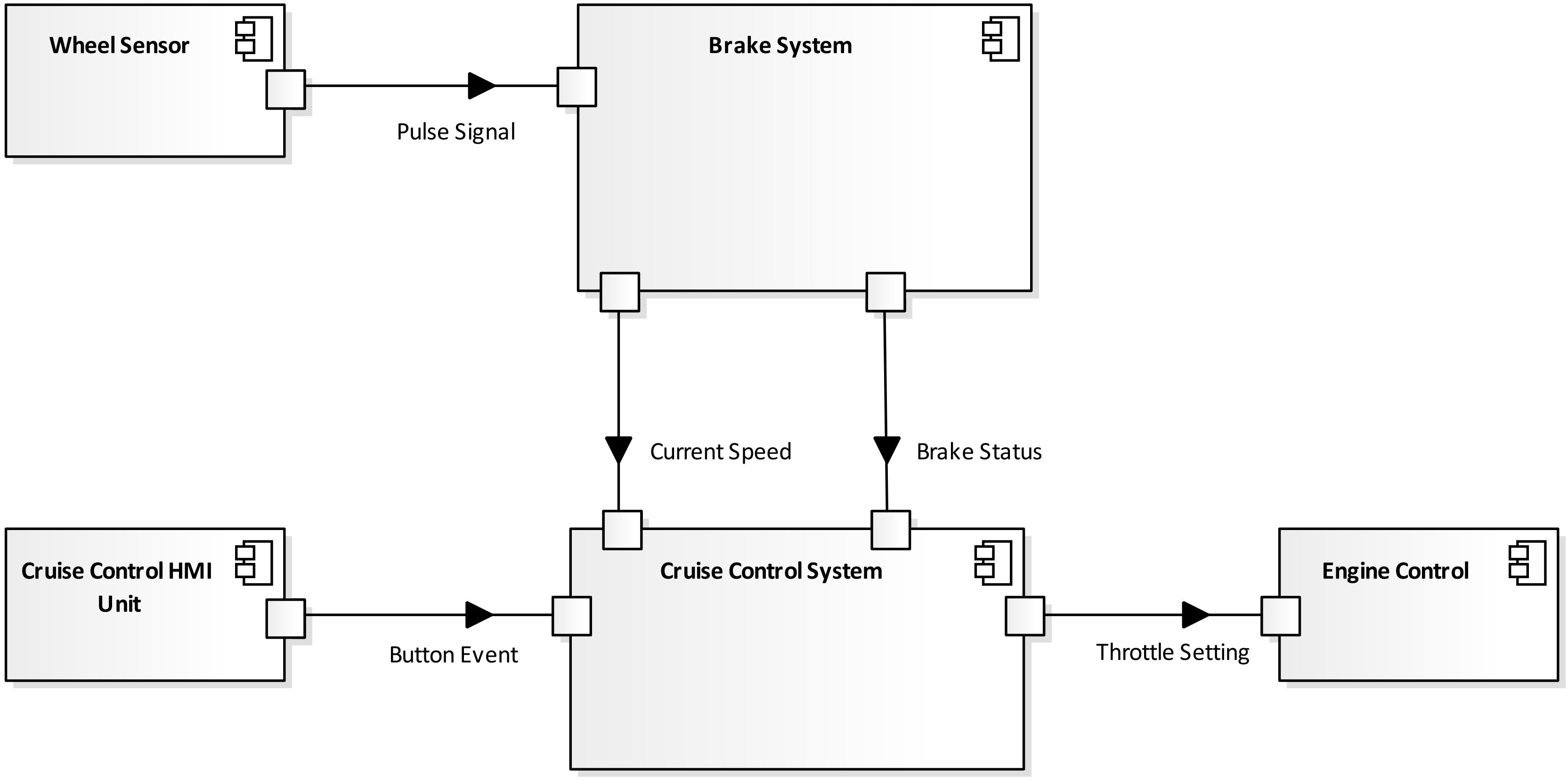
# Functional Architecture (3)

Example: Cruise Control

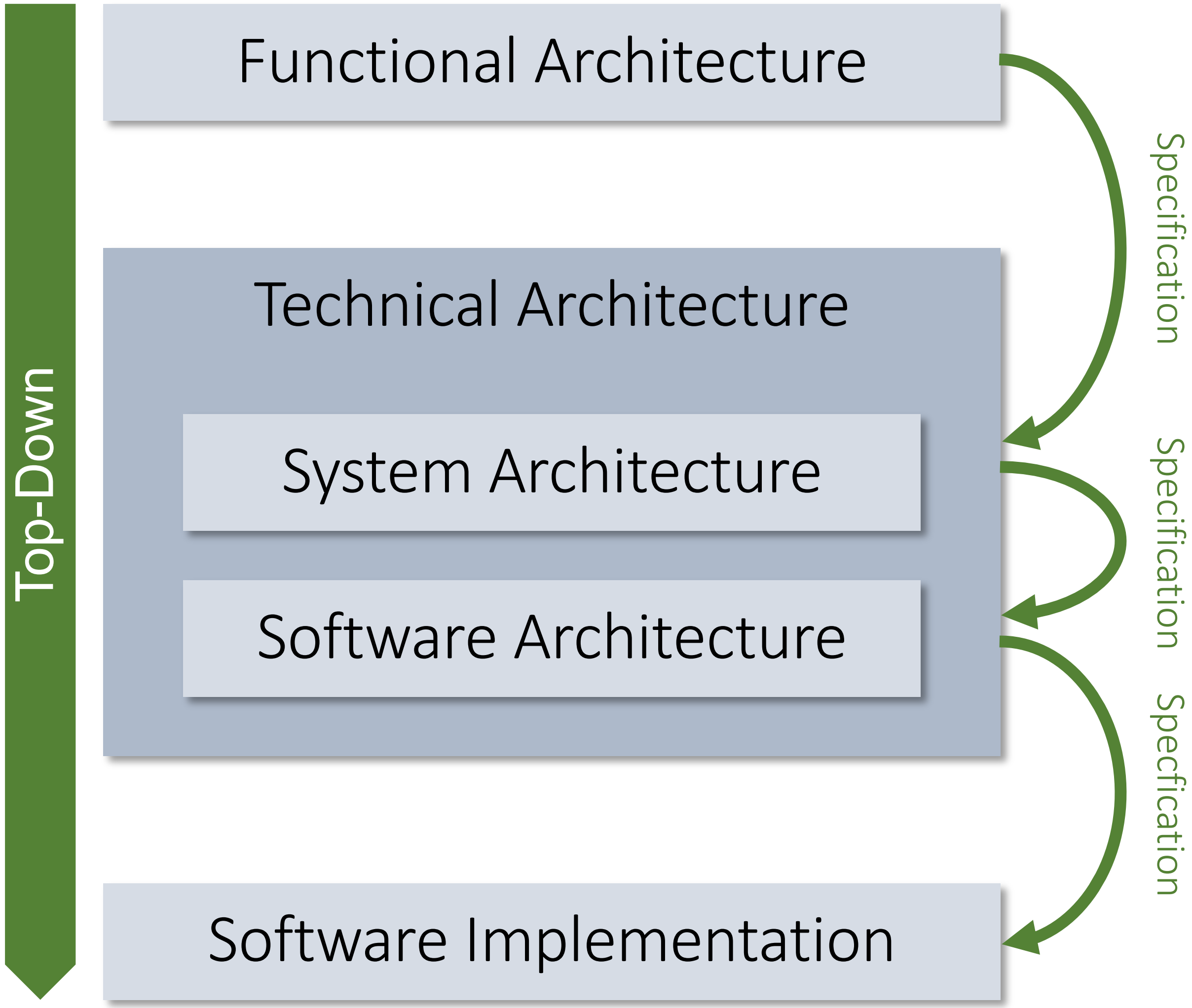


# Technical Architecture (First Sketch)

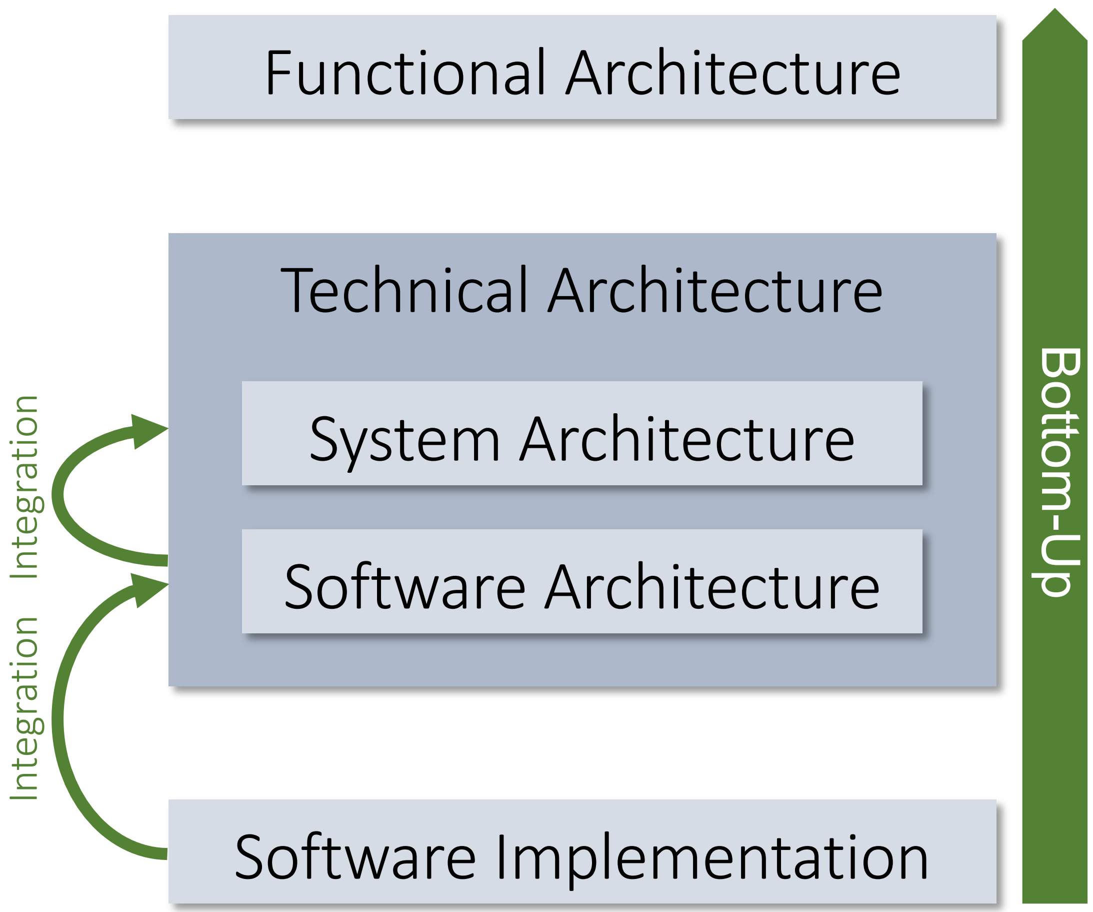
Example: Cruise Control



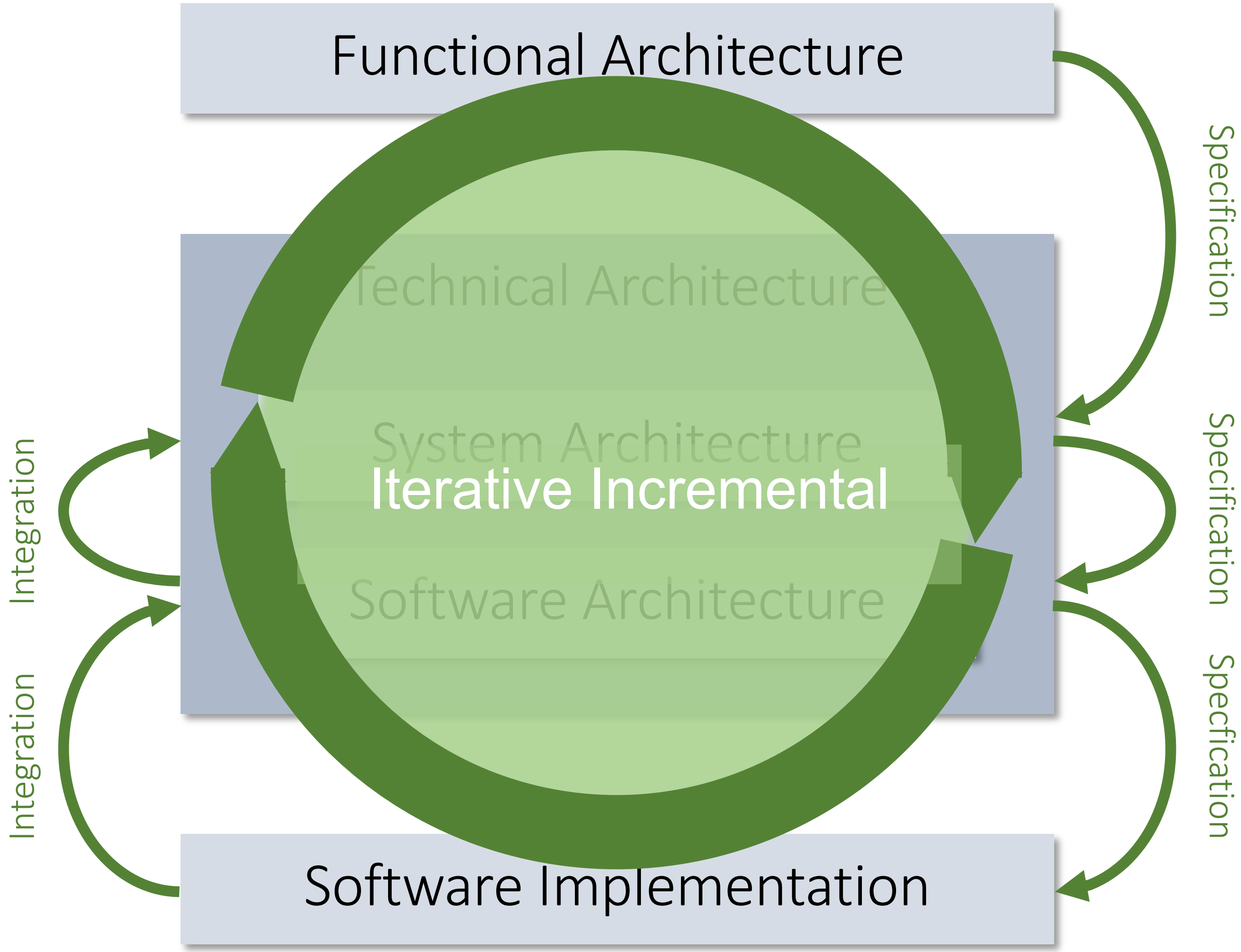
# How to Construct a Dependable Embedded System?



# How to Construct a Dependable Embedded System?



# How to Construct a Dependable Embedded System?



# Overview

Systems and Software Development

Architecture Goals

**Dependability and Functional Safety**

Real-Time and Concurrency

# Example: Architecture Goals

Functionality, safety, real-time behavior: Alignment of design goals

- Functionality often benefits from methods applied in the context of safety-relevant systems, e.g., isolation and real-time properties
- Safety mechanisms should not just be „mounted on top of functionality“

Properties such as timing, memory usage and safety are a cross-cutting system aspect

- They have to be respected at all system, hardware and software levels
- The engineering disciplines rely on each other, they are equally important
- Properties should be included in the design process just as any other functionality or relevant property

# Isolation in ISO 26262: Freedom from Interference (FFI)

From ISO26262-6, Annex D

- Software elements must not affect each other in an unintended and negative way
  - Errors in an application shall not spread to other applications
  - Errors in an application shall not spread to infrastructure services
  - Errors in an application shall not affect other system elements
- Elements subject to decomposition must be isolated from each other

Achievement of FFI

- Timing and execution: Temporal isolation: Scheduling, execution budgets, watchdogs, ...
- Memory: Spatial isolation: Semantic analysis, memory-protection unit, ...
- Safe exchange of information: Communication between isolated elements: checksums, ...

# FFI in Space and Time

**Physical isolation** of software instances (e.g., independent MCUs): **Federated architecture**

All resources (memories, CPU time, etc.) can be assigned to a specific functionality

Often, functionalities need to cooperate, they have dependencies

- Safe data exchange between components
- Waiting times / latencies have to be respected in system design, etc.

Functionalities may also be deployed on the same MCU: **Integrated architecture**

- To reduce physical weight and size as well as costs
- Complicates the provision of FFI
- In contrast to physically isolated components, sophisticated mechanisms are needed for FFI

# Overview

Systems and Software Development

Architecture Goals

Dependability and Functional Safety

**Real-Time and Concurrency**

# Real-Time Systems

DIN 44300: Standard for information processing

- Real-time operation is the operation of a computer system, whose programs for data processing are operational in a way, so that processing results are available in a specified time span.
- Depending on the use case, data can be delivered with a random temporal distribution or at determined points in time.

# Real-Time Systems

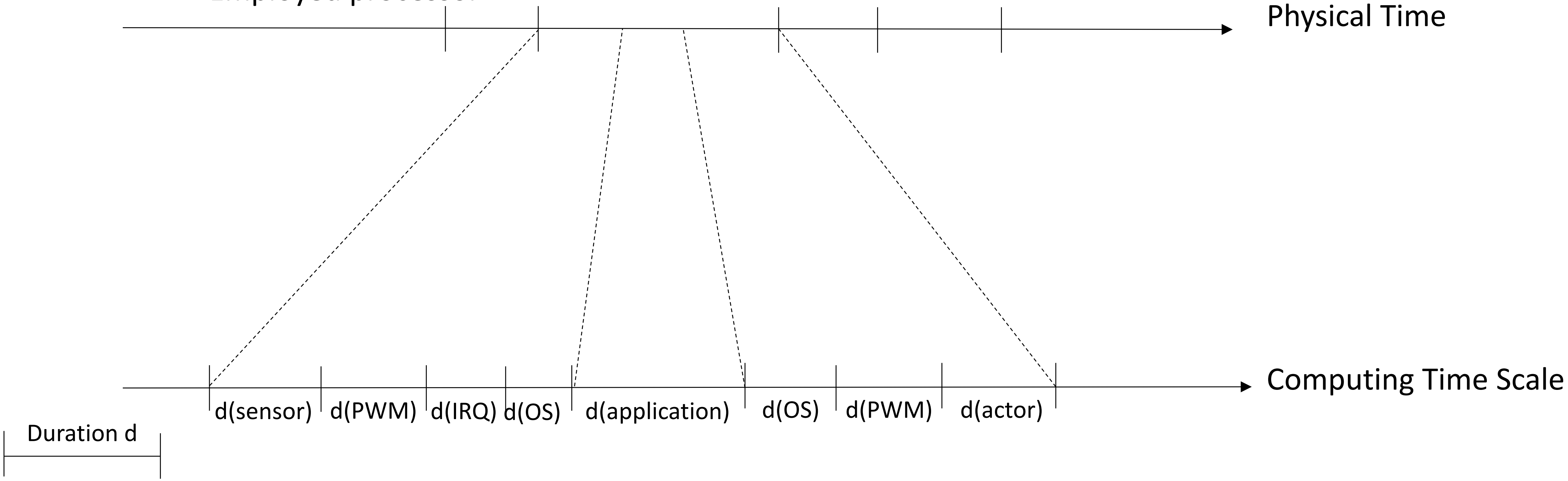
- A real-time system computes **results** in reaction to **events**
- The point in time, at which the result must be available, is called **deadline**
- Fastness does **not guarantee** the real-time capability
  - Interrupts may cause unpredictable execution variations

Time is not an internal characteristic of a computing system

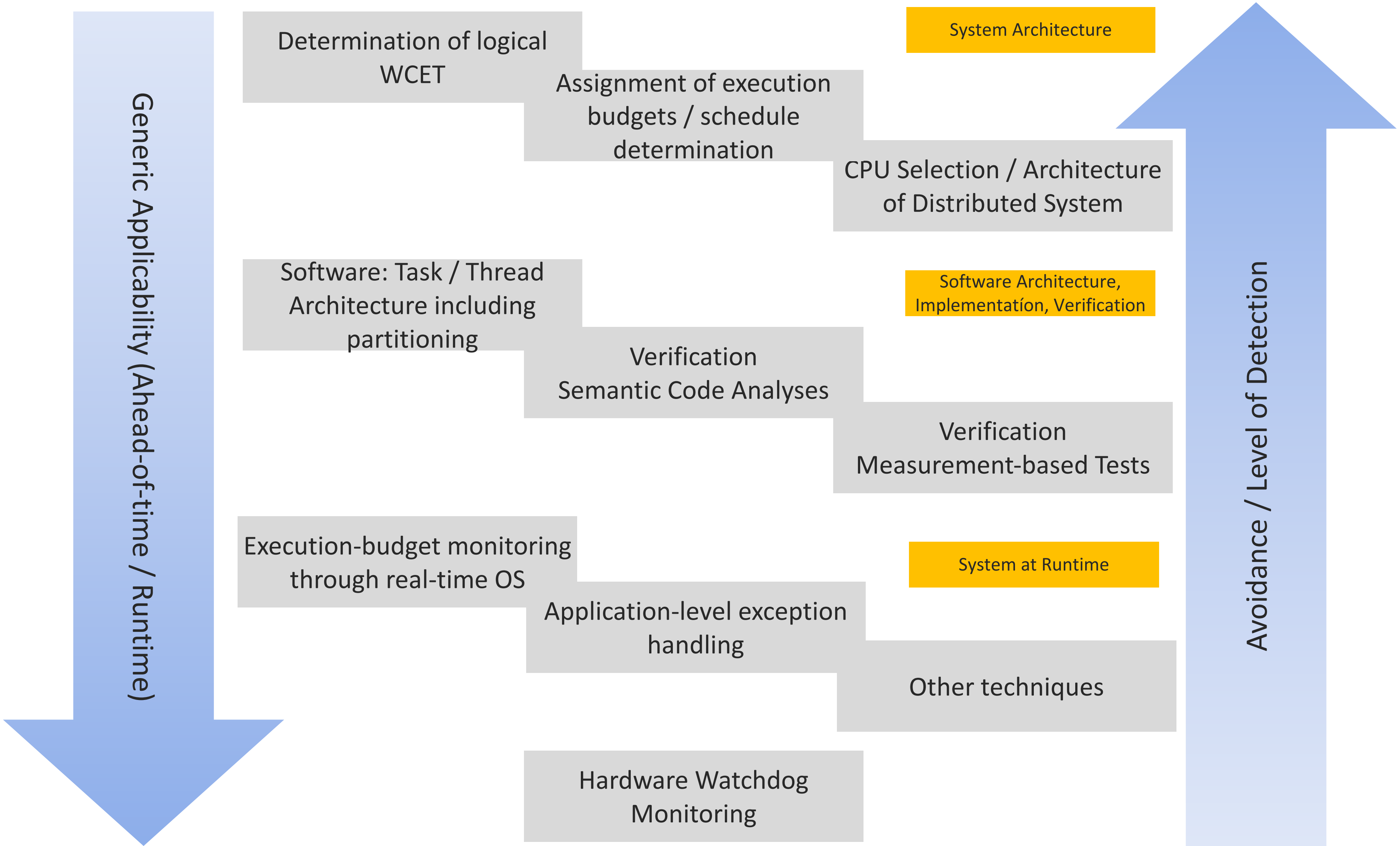
- The computing system‘s  time scale may not be identical to ist environment
- Temporal conditions of the controlled object have to be suitably mapped in the computing systems

# Controlling Real-Time System E-Motor

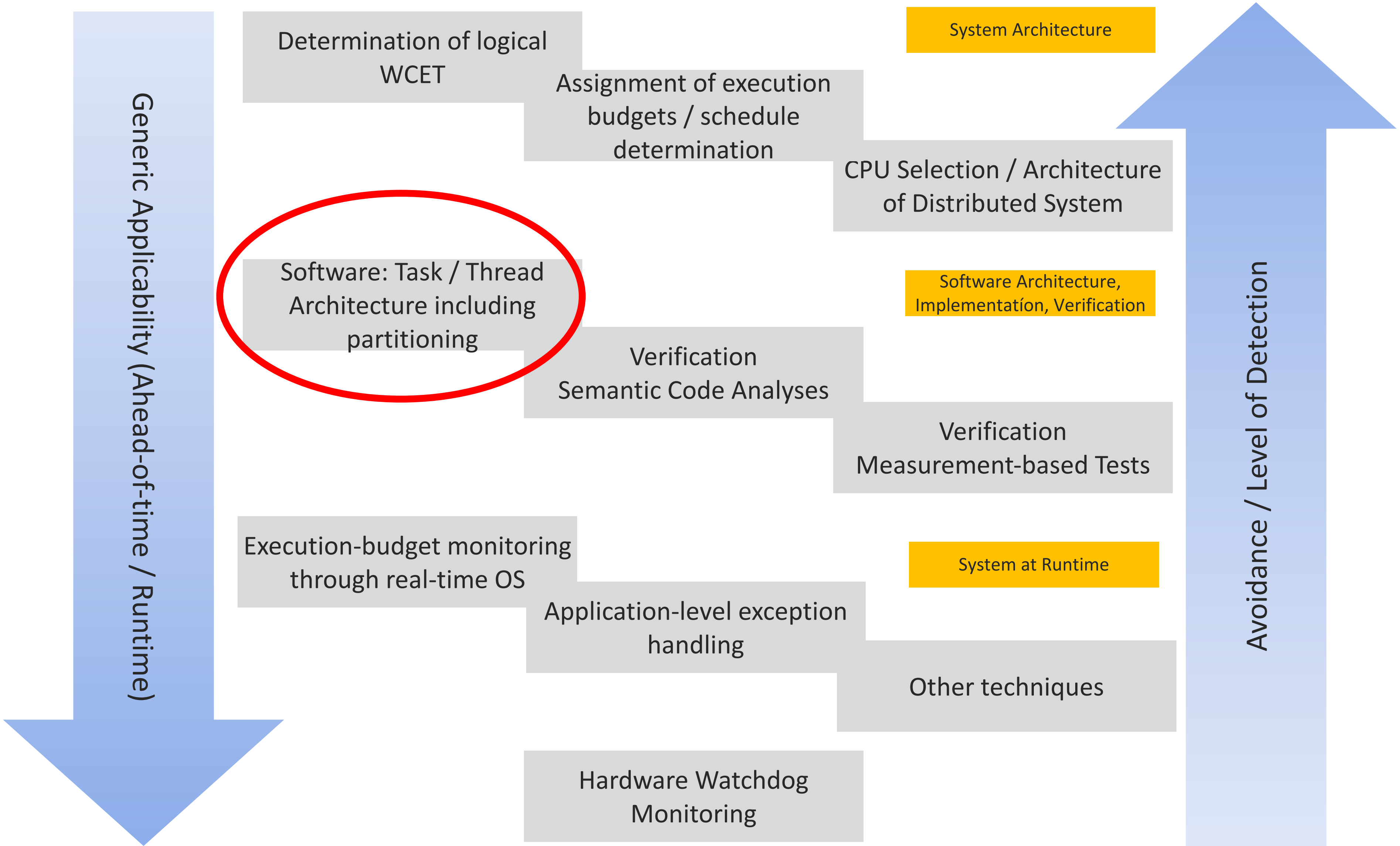
- Examinations have to be performed on various development levels
- Which elements have to be examined to ensure timely behavior?
  - Real-time (RT) application
  - Real-time operating system (RTOS) and runtime system
  - Employed processor



# Mechanisms for Providing Timely Execution



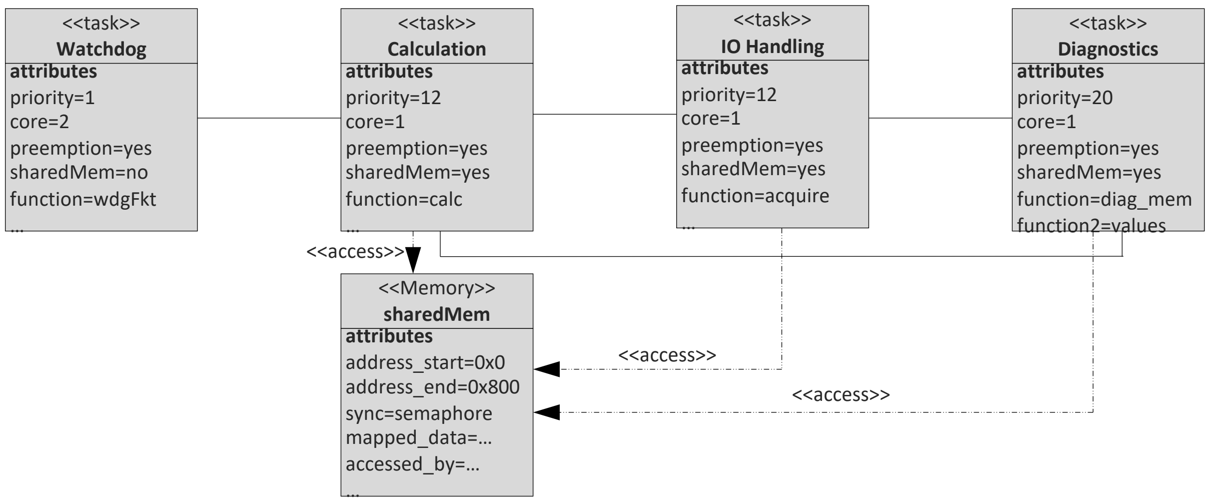
# Mechanisms for Providing Timely Execution



# Temporal and Spatial Isolation: A Software Topic Only?

CPU time and memory must be shared across components

- CPU time sharing can be achieved by the use of an RTOS scheduler
- A scheduler provides a **framework** for the construction of a real-time system
  - An unfortunate application structure may impede timely execution
  - A proper thread / task architecture has to be created
- Memory partitions and their locations have to be defined, data and code has to be assigned



# Temporal and Spatial Isolation: A Software Topic Only? No!

Scheduling and isolation are system-architectural topics:

- The temporal /spatial **partitioning** is dependent on the **system requirements / architecture**
  - Mathematical **scheduling analyses** are performed on both **functional and technical architecture, e.g., rate-monotonic analysis (RMA)**
- CPU selection
- Distributed network of MCUs, etc.
- Aspects at all system-architectural levels influence each other

**Example: Temporal Constraints, Computational Spacetime, Error Spreading**

- Undesired memory accesses may induce temporal faults
- Unspecified or faulty sensor values may induce temporal faults
- A faulty design specification may induce temporal faults
- Measures (e.g., software-based replication) meant to provide safety
  - Affect timing behavior
  - May in turn induce temporal faults

**The holistic solution has to be respected during analyses!**

# Scheduling at the Implementation Level

- Scheduling deals with the determination of **points in time** at which **work units** are executed on a **particular processor**
- Scheduling is a two-phase approach

Software Architect

1. Work units have to be assigned to threads (statically at design time)

Software Architect

2. Threads have to be assigned to processors (statically / dynamically)

Operating System

# Separation of Concerns

Planning of temporal handling and dispatching of threads

1. Scheduling is the planning **strategy**
   - Construction of a thread-execution plan, which defines the order thread processing; statically at design time or dynamically at runtime
2. Dispatching is the thread-management **mechanism**
   - Implementation of the thread-execution plan
   - Overhead depends on **thread type** (process, user-level, kernel-level, i.e., memory-protection-zone assignment) being used

# Thread of Control (1)

- An OS thread / task is an abstraction of the operating system provided to
  - programs from the application layer
  - infrastructure-software programs (e.g., drivers)
- A thread executes (parts of a) program(s) and is a modelling element in a software architecture
  - The thread-architecture view is defined by the architect
    - Thread structure (relations, dependabilities)
    - Assignment of properties: priority, preemption, events
    - Assignment to memory-protection zones (address spaces)

# Thread of Control (2)

This approach is a realization of the separation-of-concerns principle

- Separate what (code) from how (execution)
- An OS partially encapsulates the architecture goal *timing behavior* in a software architecture
- Supports code **reusability** and **extensibility** (in contrast to (manually applied) Cyclic Executive Pattern)

The thread-management overhead of the OS depends on the thread-architecture

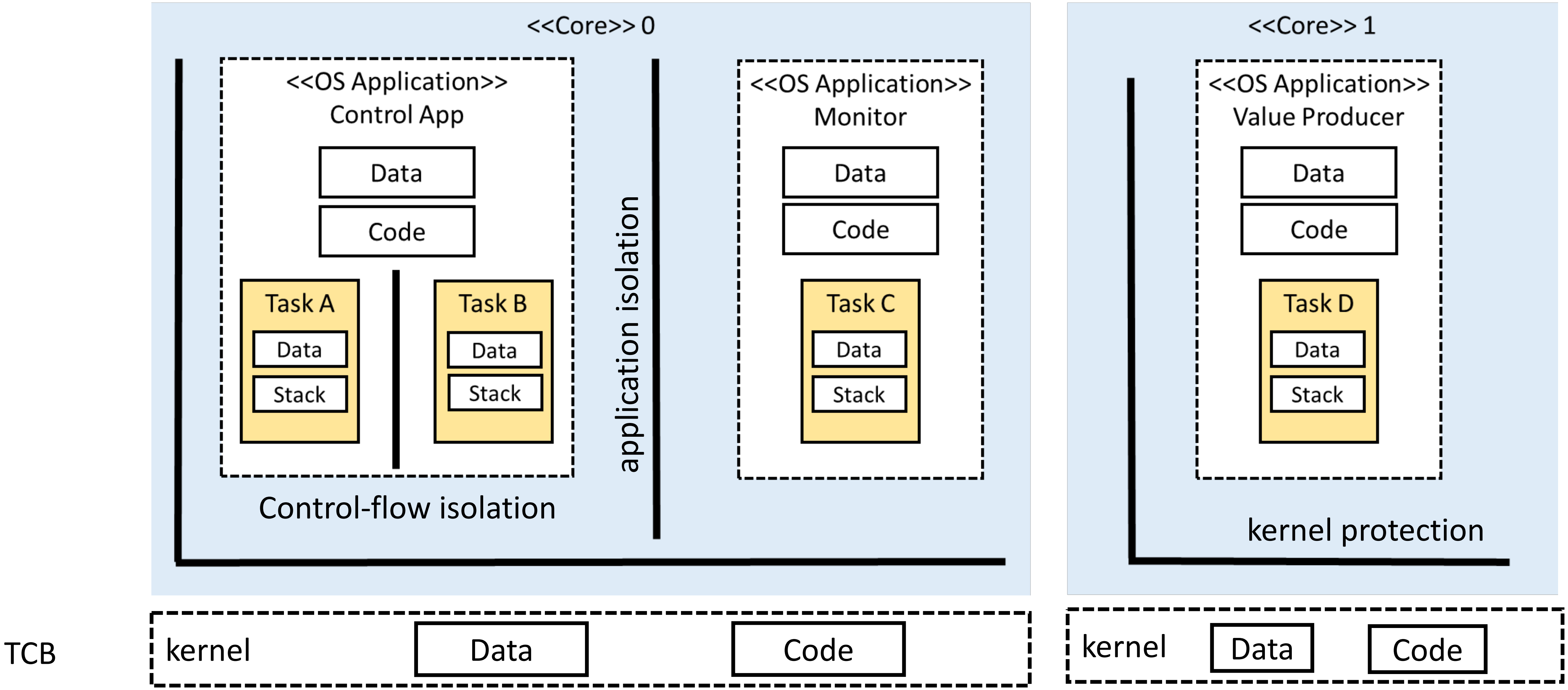
- Single-threaded program
- Multi-threaded program
  - Single address space
  - Isolated OS kernel
  - Multiple isolated address spaces

# Multi-Threading

When using multi-threading, new architectural issues need to be solved, e.g.,

- Verification of the scheduling decisions on the implementation level
- **Design of memory-protection zones / address spaces**
- **Handling of concurrency situations**

# AUTOSAR OS for FFI: Memory-Protection Zones



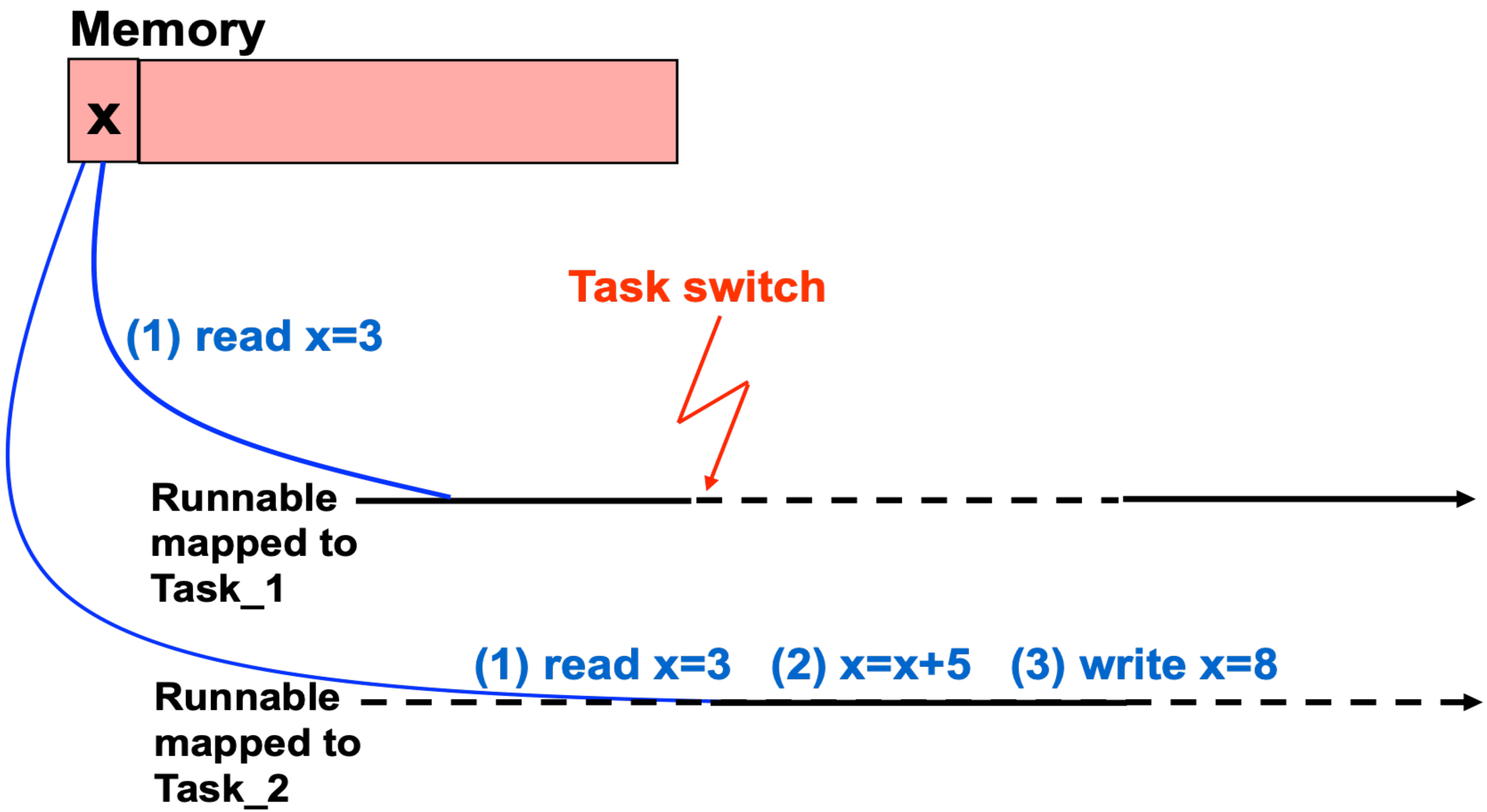
# Overhead of Thread Management (Unicore)

Logical Thread Types

1. Single Thread: Lowest overhead
   - O(activation / function invocation)
   - Managing the activation record (e.g. pushing parameters to the stack)
2. Multiple Threads
   - 2.1: Single address space
     - O(Thread switch) + O(1.); update of registers
   - 2.2: Separate address space for the operating system
     - O(system call) + O(2.1); trap handling
   - 2.3: Isolated address spaces for threads
     - O(address space switch) + O(2.2.); Update MMU / MPU caches

This has to be taken into account during timing analyses!

# Lost-Update Problem



# How is this problem solved by the OS?

Synchronization of data can be achieved in several ways, e.g., by

- Priority Ceiling Protocol
- Spinlocks
- Suppression of interrupts
- Constructively by systematic scheduling

# CPSA Training: Dependable Embedded Systems

Interested in more details of dependable embedded systems design?

- Visit the iSAQB training
- Details on the curriculum can be found here:

https://isaqb-org.github.io/curriculum-embedded/curriculum-embedded-en.pdf