Constructing Safer Systems by Design

@Automotive ISO 26262: Functional Safety Adaptation and Integration

Isabella Stilkerich

ID 01IS16025

SCHAEFFLER

Agenda

Motivation: Functionality and Safety

System and System Failures

Faults in Hardware and Software

Coping with Faults: Avoidance, Detection, Handling

- Example: Coping with Faults in Timing (Temporal Faults)

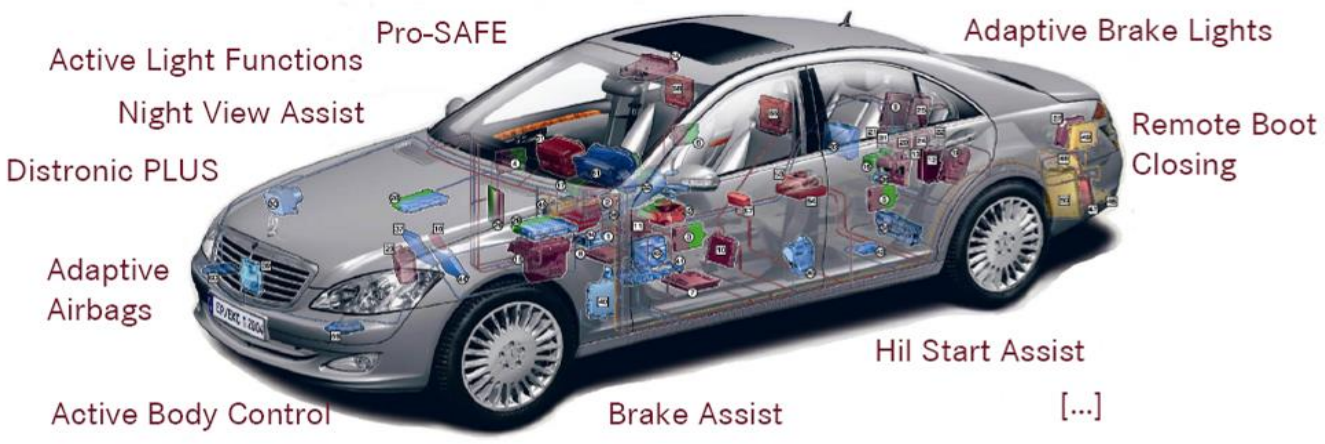
Résumé

Motivation: Rising Complexity

DAIMLER Functional Safety

Current Situation
Trends in Automotive Electric/Electronics (E/E)

- Increasing functionality and complexity of software-based car functions
- Increasing risks from systematic faults and random hardware faults
- Most of the new car functions are safety-related



© Courtesy of Daimler; Presentation given at Automotive Electronics and Electrical Systems Forum 2008, May 6, 2008, Stuttgart, Germany

Motivation: Functionality and Safety

Functionality and safety: Alignment of both design goals

- Functionality often benefits from methods applied in the context of safety-relevant systems
- Safety mechanisms should not just be „mounted on top of functionality“

Safety is a cross-cutting system aspect

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

Design Goal: Safe-by-Construction

System and System Functions

What is a system?

- In general, a compound of building blocks (elements, components)
- The collaboration of components achieves one or several specified **system function(s)**
- A system function has specified functionality and properties

What is a system in the context of ISO 26262?

- A compound of **hardware** and **software**
- Minimum: Sensor, actor, microcontroller (MCU)

A system is executed correctly if the system implements the intended system function as specified.

System Failure

A failure causes the system to deviate from its specified behavior (e.g.,faulty output values). The failure can be caused by an error, that is, a discrepancy in the system's internal state and an error (e.g., a corrupted memory address) is caused by a fault (e.g., bit flip in memory caused by EMI).

System Failures Caused by Hardware (1)

Hardware can fail: Microcontrollers, sensors, actors

- Hardware can have systematic faults: Design faults
- Random faults are common: Hardware has a innate failure rate (aging, technology, etc.)

Mechanisms at the hardware level can be taken to address both systematic and random faults

- Application of **design methods** to avoid systematic and random faults
- Application of **mechanisms** to detect and handle systematic and random faults, e.g.
  - Use of ECC-protected memories to detect errors in memory cells
  - Use of a hardware watchdog to detect timing anomalies, etc.

System Failures Caused by Hardware (2)

Mechanisms at the hardware level may be supported by software (technical reasons, costs etc.)

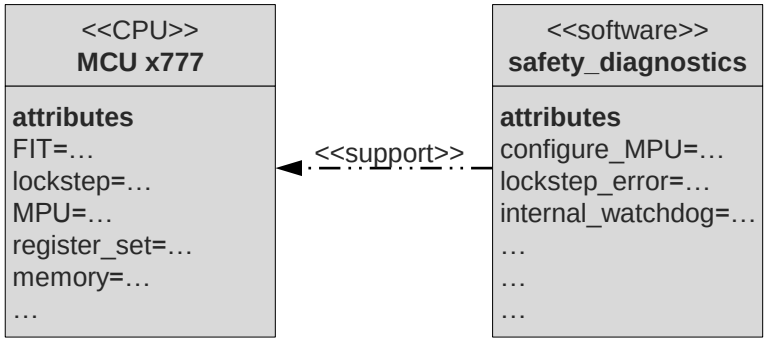
But: Software can fail, too!

- Caused by systematic faults in software, e.g. data races, overruns, etc.
- Caused by hardware faults, e.g. data in corrupted memory area, etc.
- Caused by specification/design faults, etc.

System Failures and Safety Software

Additional mechanisms at the **software level** are necessary

- Design methods to avoid systematic faults in system design ahead of time, e.g. partitioning, element allocation in distributed network
- Design methods to avoid systematic faults in software ahead of time
- Measures to detect and handle random and systematic hardware/(software) faults at runtime

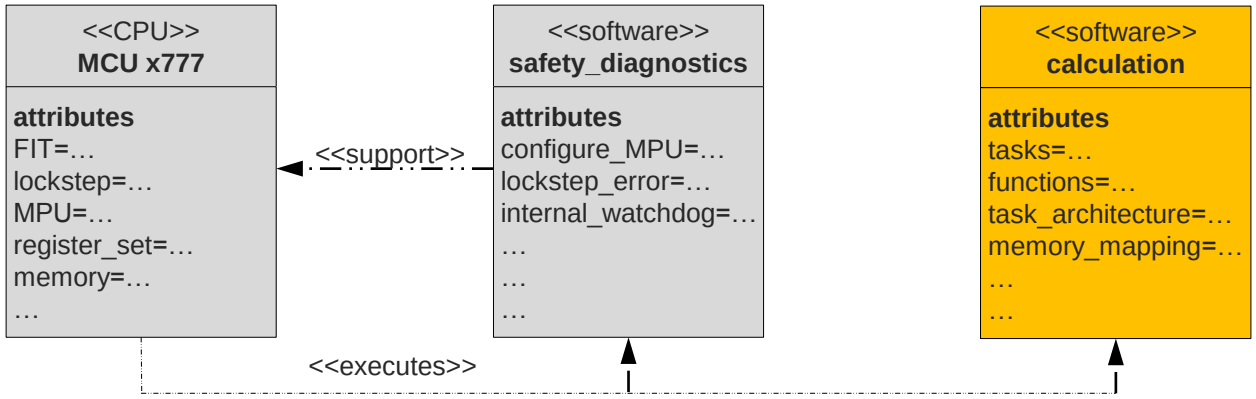


System Failures Caused by Hardware/Software

Software implementing the functionality

- Use of methods and techniques to avoid and handle systematic faults in software
- Supports Safe-by-Construction Design Goal

Objective for hardware-software compound: **Avoid faults by design**



Faults in Hardware and Software

Faults may arise at different architectural levels

- They should be found as early as possible (ahead-of-time, at design time)
- At runtime, they faults should be detected / handled in a timely manner (real-time capability)
- Early detection of faults supports design goals
  - **Quality-of-Service** of **real-time** systems, **availability**
  - **Functional safety**: Fail-safe and fail-operational systems

Topics complement each other, but I focus on **timing** in the following.

Combining Quality, Availability and Safety

Timely execution and **memory handling**: Aspects of **computational spacetime**

Properties, also termed **quality attributes**

- Timing and memory-handling faults affect functionality
- Such faults may result in or contribute to system failures and safety-goal violation

Faults in computational spacetime shall be **avoided, detected** and **handled**

Partitioning supports the establishment of logical protection realms

- Violation of protection boundaries can be avoided, detected and handled
- Definition of boundaries as systems-engineering duty
- Coarse-grained techniques: e.g., memory-protection unit, hardware watchdog
- More fine-grained techniques: e.g., semantic code analyses, scheduling theory
- Containment of faults: **Freedom from interference**

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**
- Memory: Spatial isolation
- Safe exchange of information: Communication between isolated elements

FFI in Timing

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

- Example: A functionality waits for a CAN message to activate a certain behavior
- Waiting times / latencies have to be respected in system design, etc.

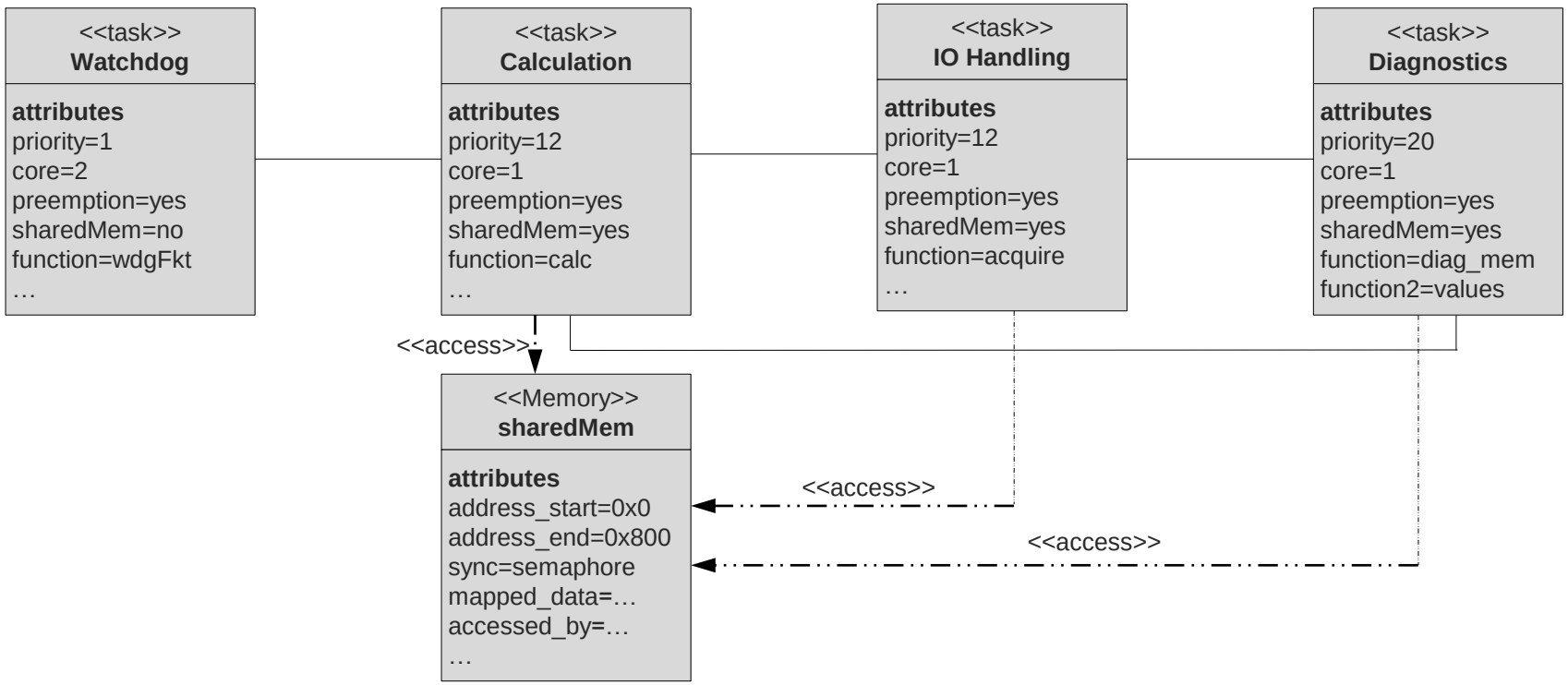
Functionalities may also be deployed on the same MCU: **Integrated architecture**

- 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

Temporal Isolation: A Software Topic Only?

CPU time 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



Temporal Isolation: A Software Topic Only? No!

Scheduling is a system-architectural topic:

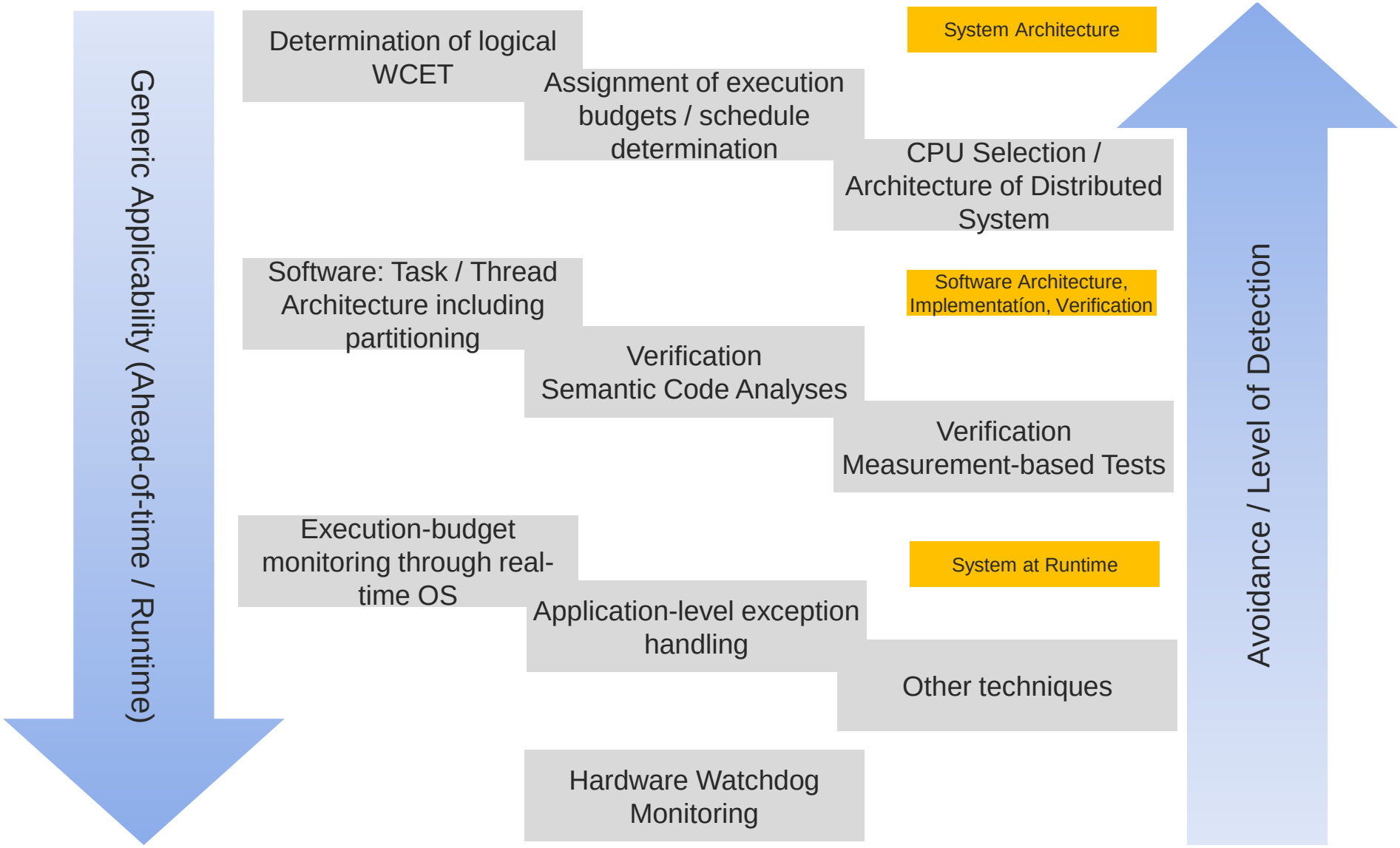
- The temporal **partitioning** is dependent on the **system requirements** and **architecture**
- CPU selection
- Distributed network of MCUs, etc.
- Aspects at all system-architectural levels influence each other

Example: Temporal Constraints, Computational Spacetime, Error Spreading

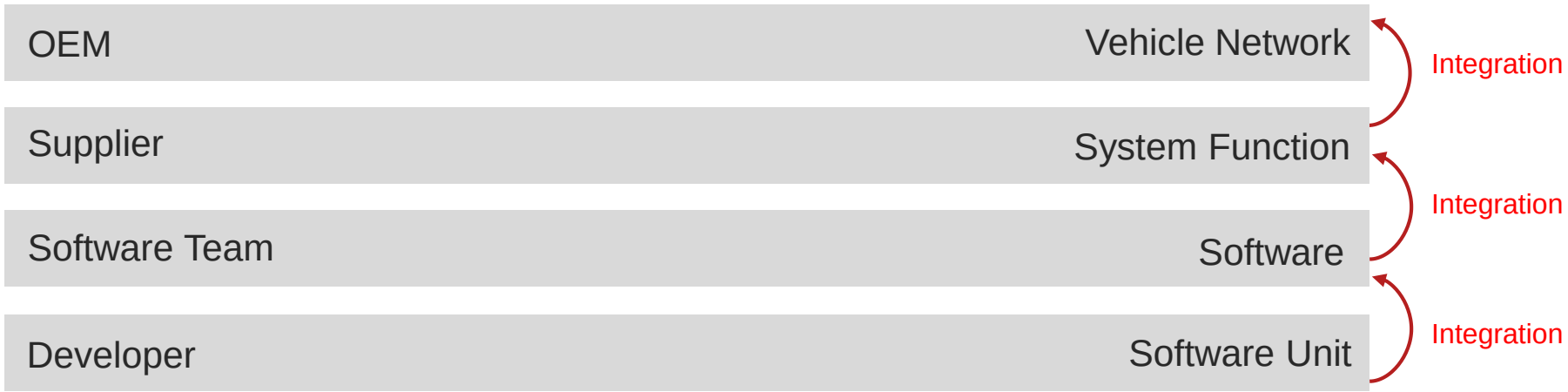
- Undesired memory accesses may induce temporal faults
- Unspecified or faulty sensor values may induce temporal faults
- A faulty design specification may induce temporal faults
- Measures (e.g., software-based replication) meant to provide safety
  - Affect timing behavior
  - May in turn induce temporal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

The holistic solution has to be respected during analyses

Techniques for Providing Timely Execution



Example: Temporal Requirements – A Bottom-Up Perspective



- Software units are assembled to construct software components
- Components realize tasks (i.e., work units) in applications
- Tasks are scheduled (i.e., planned)

Integration is a Challenge (1)

Bottom-up approach is pursued in lots of projects, but it is problematic!

Scheduling: not considered during system design and is the final step during system integration

The adherence to timing constraints is strongly dependent on provided components

- Units/components: contain implementation details influencing worst-case execution times
- Application: Mapping of components to tasks and jobs (e.g., runnables) to OS threads restrict scheduling possibilities

Distributed development and buying software components aggravate the problems imposed by bottom-up approach

Integration is a Challenge (2)

Subsequent changes in software units, components and applications are very expensive

Correction influences execution-time behavior

- Components' worst-case execution times change
- Changes in thread mapping aggrevate the problem

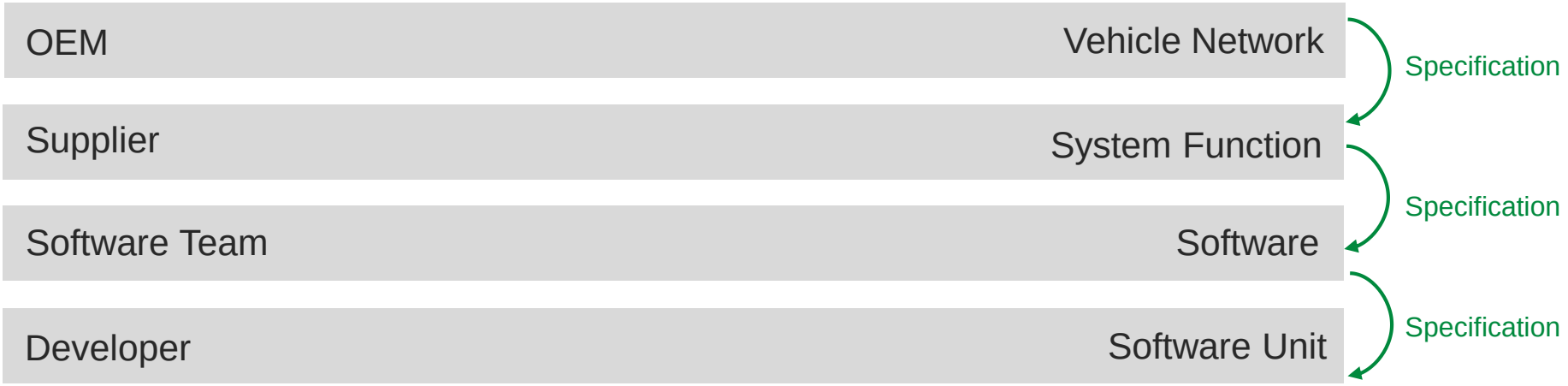
Rework may be necessary if a components needs to much CPU time and scheduling fails

- Inefficient coding
- Inapt application structure

Temporal requirements on software components must be specified

Requirements have to be derived from temporal constraints at the system level

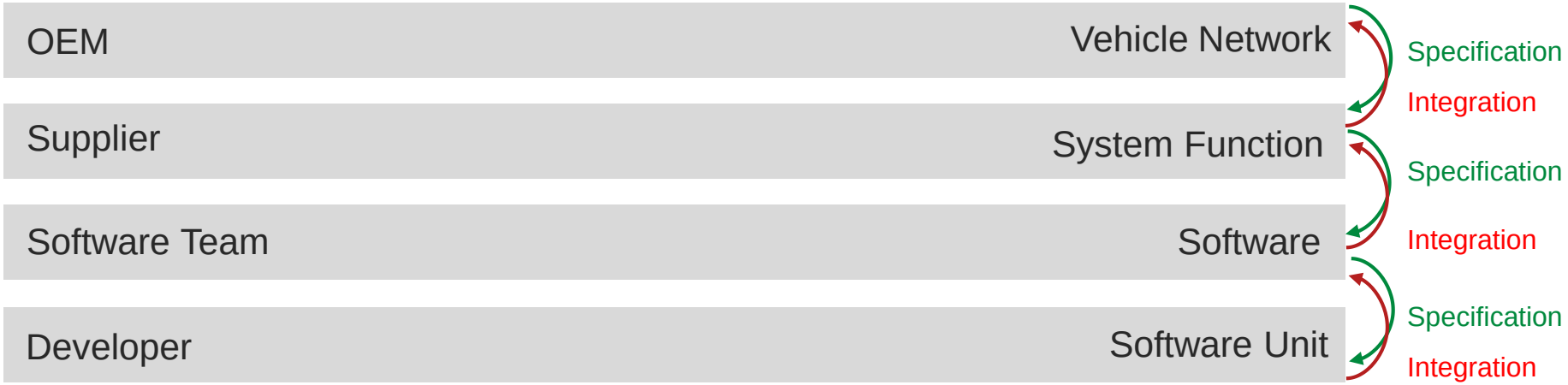
Specification of Temporal Requirements



Top-Down Specification

- OEM has knowledge about the entire system
- Applications are provided with executions budgets
- Units and components have to use the budgets wisely and have to be analyzed
- Framework of temporal constraints defines scope of actions

Iterative-Incremental Approach (1)



Problem is that global scheduling requires knowledge

- Based on experience from prior projects
- Prototype development

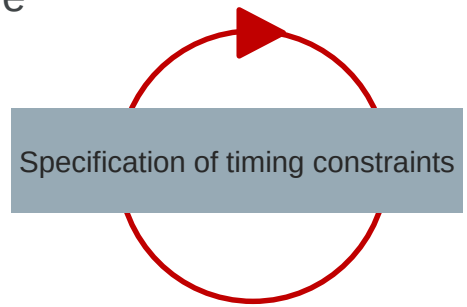
OEM requirements are often unknown

- **Sensible presumptions** have to be made
- **Design goals** for **timing** properties have to stated

Iterative-Incremental Approach (2)

An **iterative-incremental** approach is sensible (particularly in advanced engineering)

1. Start with definition of physical timing constraints, develop logical schedule
2. Choose / develop hardware, prototypes
3. Develop task architecture and implement / integrate software, prototypes
4. Evaluate timing behavior, e.g.
   - Semantic analysis on machine code and hardware description
   - Dynamic analysis, that is, measurement-based test through longest program path
   - Hybrid analyses: Combine semantic and measurement-based analyses
   - Rework results from evaluation in requirements and architecture



Iterative approach

Résumé

Construction of hard real-time systems

- In general, a hardware-software co-design approach is sensible
- A close cooperation between all disciplines is recommendable
- Timing properties affect both functionality and safety
- Techniques supporting timely execution complement each other

Iterative approach requires close cooperation between development groups and safety

Further reading:

Avoiding systematic faults in timing at the system-architectural level:

Evaluation of Architecture Variants for Hard Real-Time Systems

Real-Time Systems Lecture at Chair of Operating Systems, Computer Science Department at University of Erlangen-Nuremberg

ARAMiS II Research Project