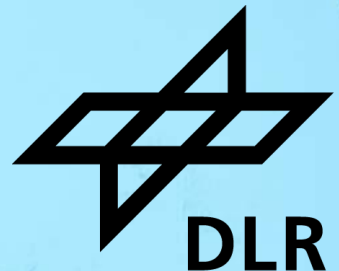


CONTRACT-BASED DESIGN IN MODEL-BASED SYSTEMS ENGINEERING

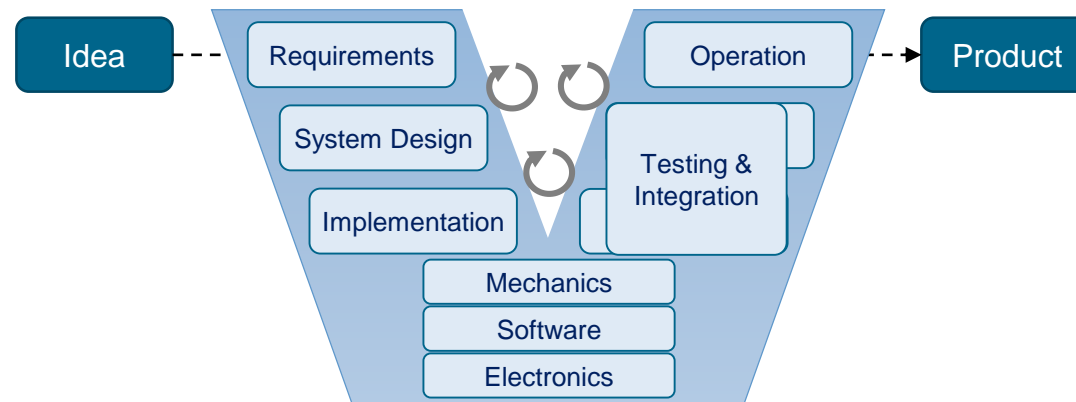
TASTE the Workshop | November 21, 2024

Transformation Hub “Automotive Software Engineering” (TASTE)

Dr. Ingo Stierand
German Aerospace Center (DLR)
Institute of Systems Engineering for Future Mobility
Oldenburg, Germany



- Systems engineering is an interdisciplinary approach focused on designing, integrating, and managing complex systems throughout their lifecycle.

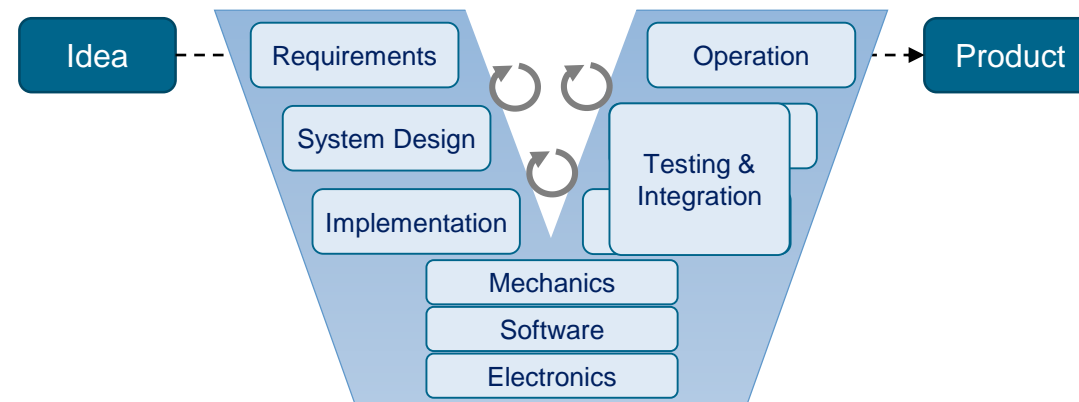


- Objective: To ensure that all system components work together seamlessly to achieve a common goal.
- Key Aspects:
 - Holistic view: Encompasses hardware, software, people, and processes.
 - Emphasis on lifecycle: From concept development to decommissioning.

The Systems Engineering Process

Stages in the Process:

- **Requirements Analysis:** Identifying the needs and constraints.
- **System Design:** Conceptualizing and architecting the system.
- **Implementation:** Building or coding the system components.
- **Integration and Testing:** Ensuring components work together as intended.
- **Operation and Maintenance:** Managing the system's functionality over time.



Iterative Nature: Feedback loops at each stage for continuous improvement.

Systems Engineering Challenges



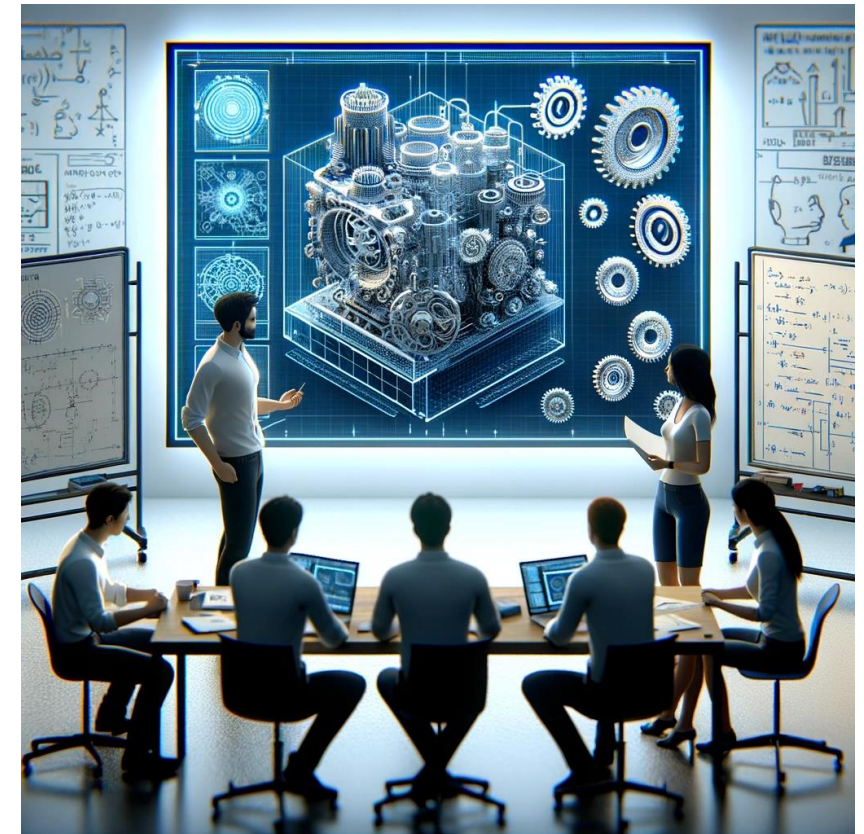
Systems engineering encounters several significant challenges, among which are the following:

- **Complexity Management:** Modern systems are highly complex, integrating numerous components and subsystems. Managing this complexity, and predicting how changes in one part will affect the whole system, is a fundamental challenge.
- **Risk Management:** Identifying, analyzing, and mitigating (technical, financial, operational) risks in large-scale projects is crucial. Engineers must develop strategies to manage these risks, often in presence of uncertainty and limited information.
- **Interdisciplinary Collaboration:** Systems engineering requires collaboration across various disciplines, including mechanical, electrical, software, and human factors. Ensuring effective communication and understanding among experts from these diverse fields is crucial.
- **Rapid Technological Changes:** The pace of technological advancement is rapid, often outstripping the lifecycle of existing systems. Keeping up with these changes, integrating new technologies into existing systems is a critical challenge.
- **Requirements Management:** Defining and managing the requirements of a complex system is an intricate task. Requirements often change due to various factors like market demands, regulatory changes, or technological advancements. Adapting to these changes without significantly disrupting the project's progress is a key challenge.
- **Regulatory Compliance and Safety:** Ensuring compliance with industry standards and safety regulations is crucial. This becomes even more challenging where different countries may have varying regulations.
- **Data Management and Security:** Managing the security and integrity of data is paramount. Systems engineers must ensure data is protected against unauthorized access and cyber threats.

Why Model-Based (Systems) Engineering

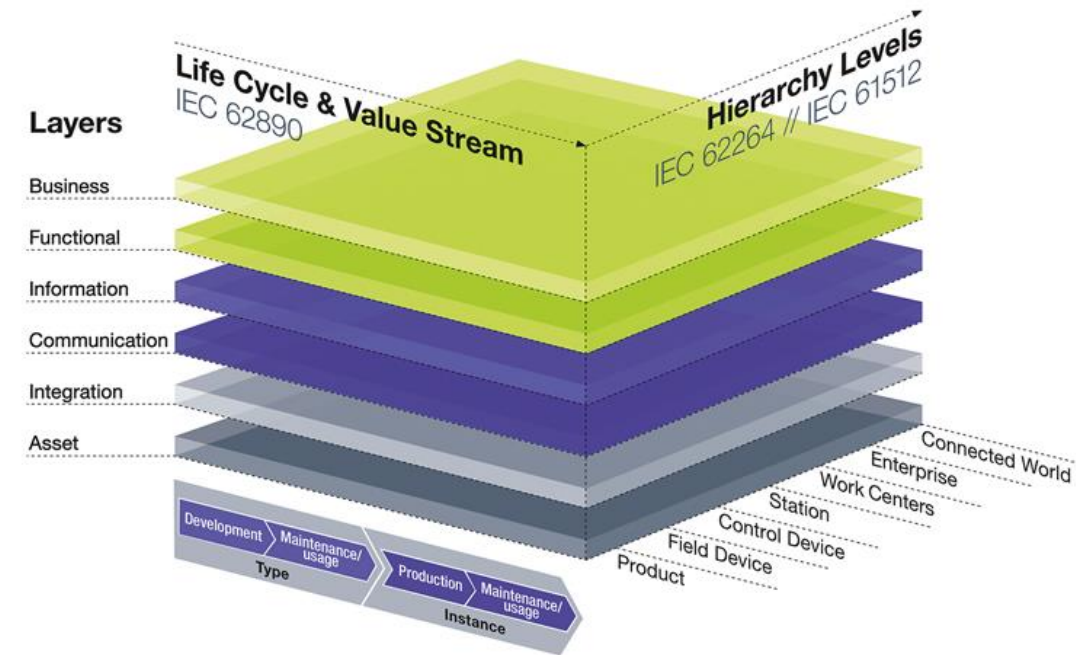
Model-Based Systems Engineering (MBSE) uses models as an integral part of the technical baseline that includes the requirements, analysis, design, implementation, and verification.

- **Complexity Management:** Comprehensive models that represent all aspects of a system help in managing complexity by providing a clear and consistent representation.
- **Collaboration:** Models as a common language provide a shared framework that everyone can understand and contribute to.
- **Adaptability:** Models can be updated more easily than traditional documents, which enables engineers to more rapidly adapt to evolving requirements.
- **Requirements Management:** MBSE enables linking system requirements directly to elements within the model. This ensures traceability and helps to see the impact of changes.
- **Risk Management:** By simulating and analyzing models, MBSE allows engineers to identify potential issues and risks early in the design process.
- **Regulatory Compliance and Safety:** Thoroughness and accuracy of MBSE models assist in ensuring that systems meet regulations and safety requirements.
- **Data Management:** MBSE models can integrate data from various sources and maintain data consistency throughout the system's lifecycle.



Modeling Frameworks

- Each application domain comes with particular demands, regulations, and constraints.
- Modeling frameworks help structuring the development process:
 - Layers (perspectives, viewpoints)
 - Life cycle phases
 - Hierarchy levels
 - ...
- Many modeling languages exist to “fill in”:
 - Business Process Model and Notation (BPMN)
 - Structured Goal Notation (GSN)
 - UML diagrams
 - ...

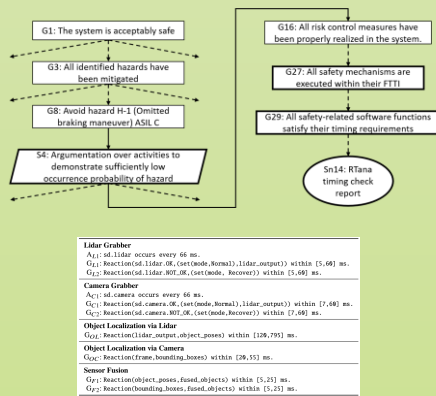


Reference Architecture Model Industry 4.0 – RAMI 4.0 (Source: DKE)

- Focus on (safety-critical) cyber-physical systems
- Provision of semantically consistent, continuous, traceable MBSE along viewpoints and abstraction levels

Requirements Viewpoint

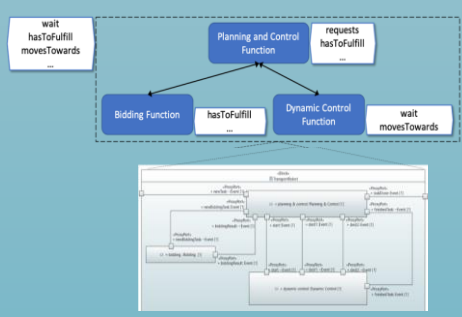
- Operational context
- Goals
- Top-level requirements



Lidar Grabber
A ₁ : lid.lidar occurs every 66 ms.
G ₁ : Reaction(s ₁ .lidar_ok, (set(mode_Normal), lidar_output)) within (5,66) ms.
G ₂ : Reaction(s ₁ .lidar_not_ok, (set(mode_Recover))) within (5,66) ms.
Camera Grabber
A ₁ : c ₁ .camera occurs every 66 ms.
G ₁ : Reaction(s ₁ .camera_ok, (set(mode_Normal), lidar_output)) within (7,66) ms.
G ₂ : Reaction(s ₁ .camera_not_ok, (set(mode_Recover))) within (7,66) ms.
Object Localisation via Lidar
G ₁ : Reaction(lidar_camera_object_poses) within (126,793) ms.
Object Localisation via Camera
G ₁ : Reaction(frame_bounding_boxes) within (28,35) ms.
Scene Profile
G ₁ : Reaction(object_poses_found_objects) within (5,23) ms.
G ₂ : Reaction(bounding_boxes_found_objects) within (5,23) ms.

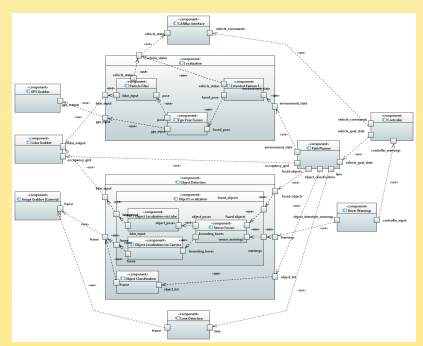
Functional Viewpoint

- System behavior, and
- Dependencies
- Decomposition of functions



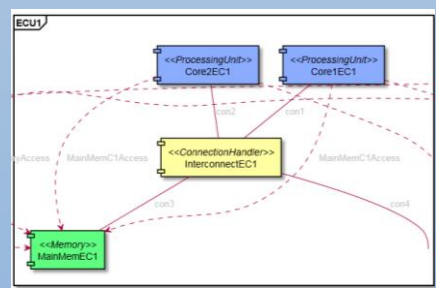
Logical Viewpoint

- Decomposition into sub-systems and components
- Assigning functions to logical units



Technical Viewpoint

- Hardware/Software Design
- Scheduling/middleware
- Optimization of resources
- Communication

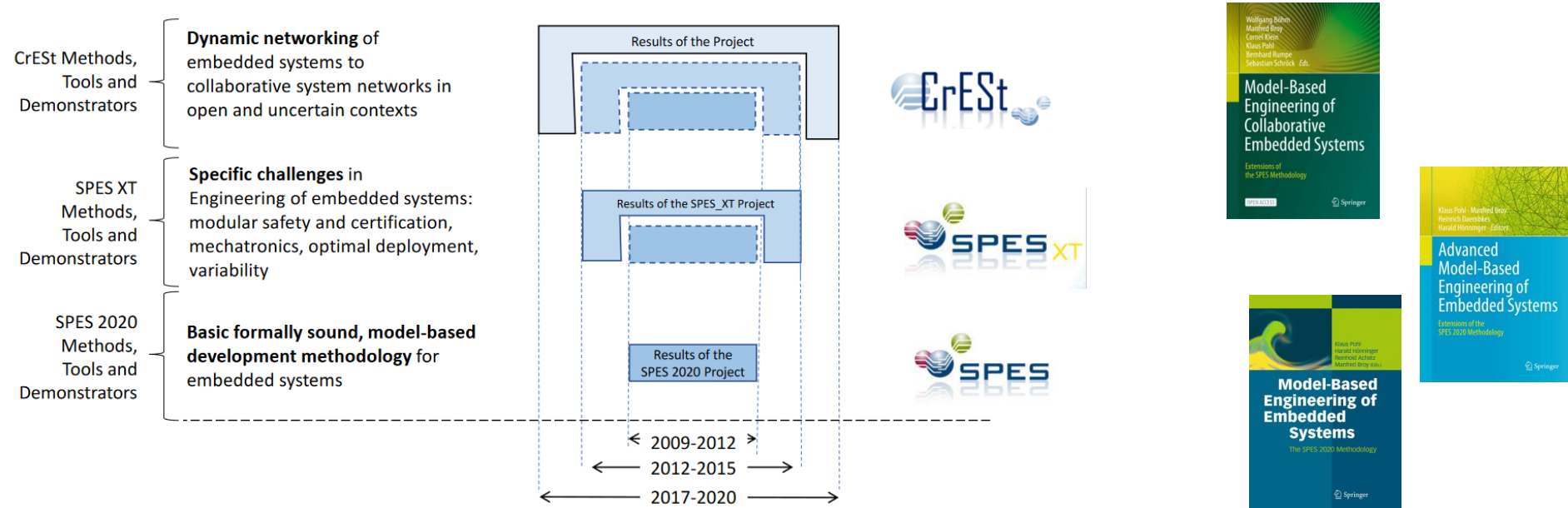


Abstraction levels

History: From SPES 2020 to CrESt

Vision

The development of software intense systems (CPS) can be accomplished through a set of **integrated modeling techniques**, their use and integration in the development is fully understood.



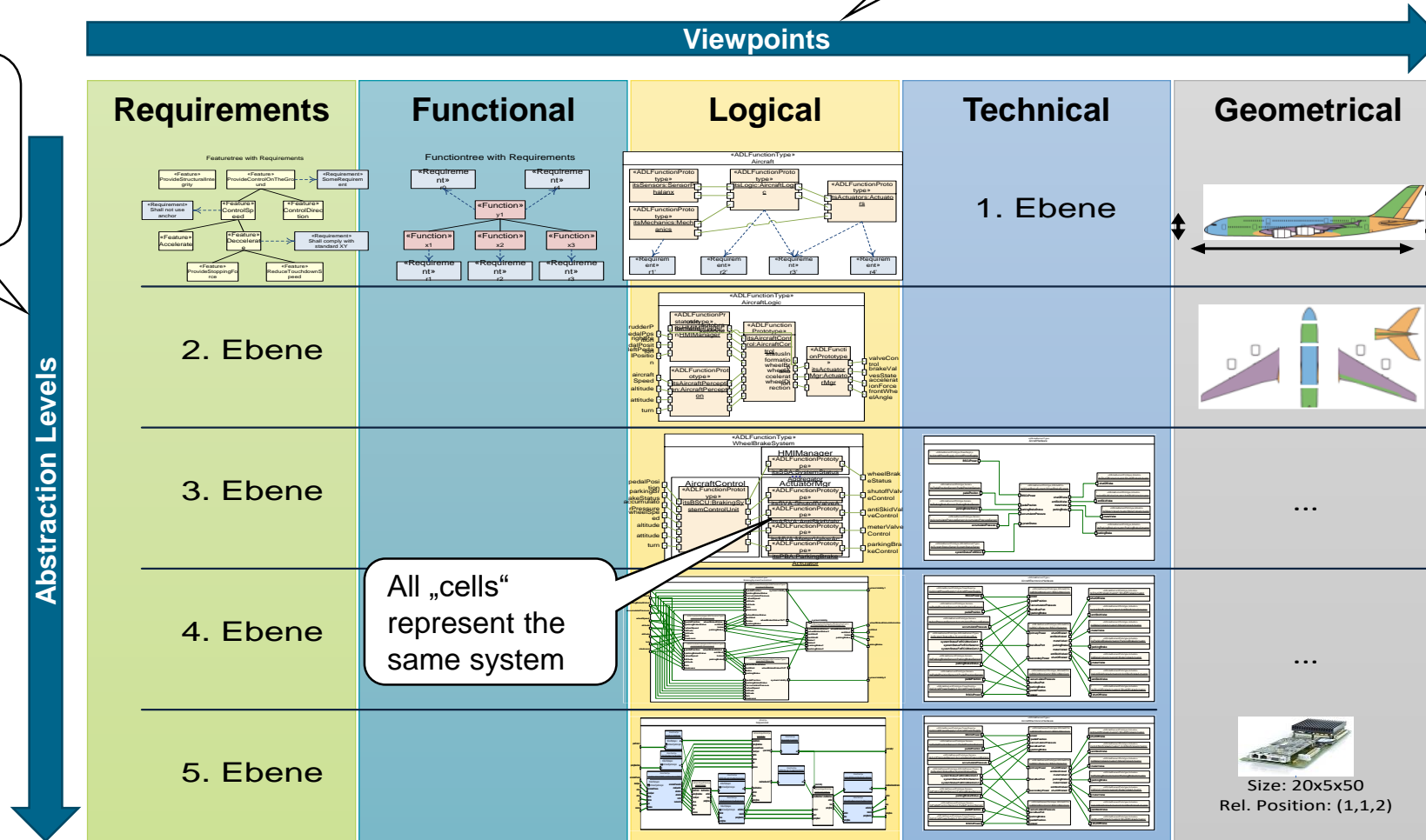
- **Compositional Semantic Framework**
 - Contract-based Design in the SPES Modelling Framework
- A "Meta Theory" of Contract-Based Design
 - What contracts are – exactly – good for
- Example
 - Applying contract-based design in safety-critical automotive system development

Compositional Semantic Framework

Abstraction Levels and Viewpoints

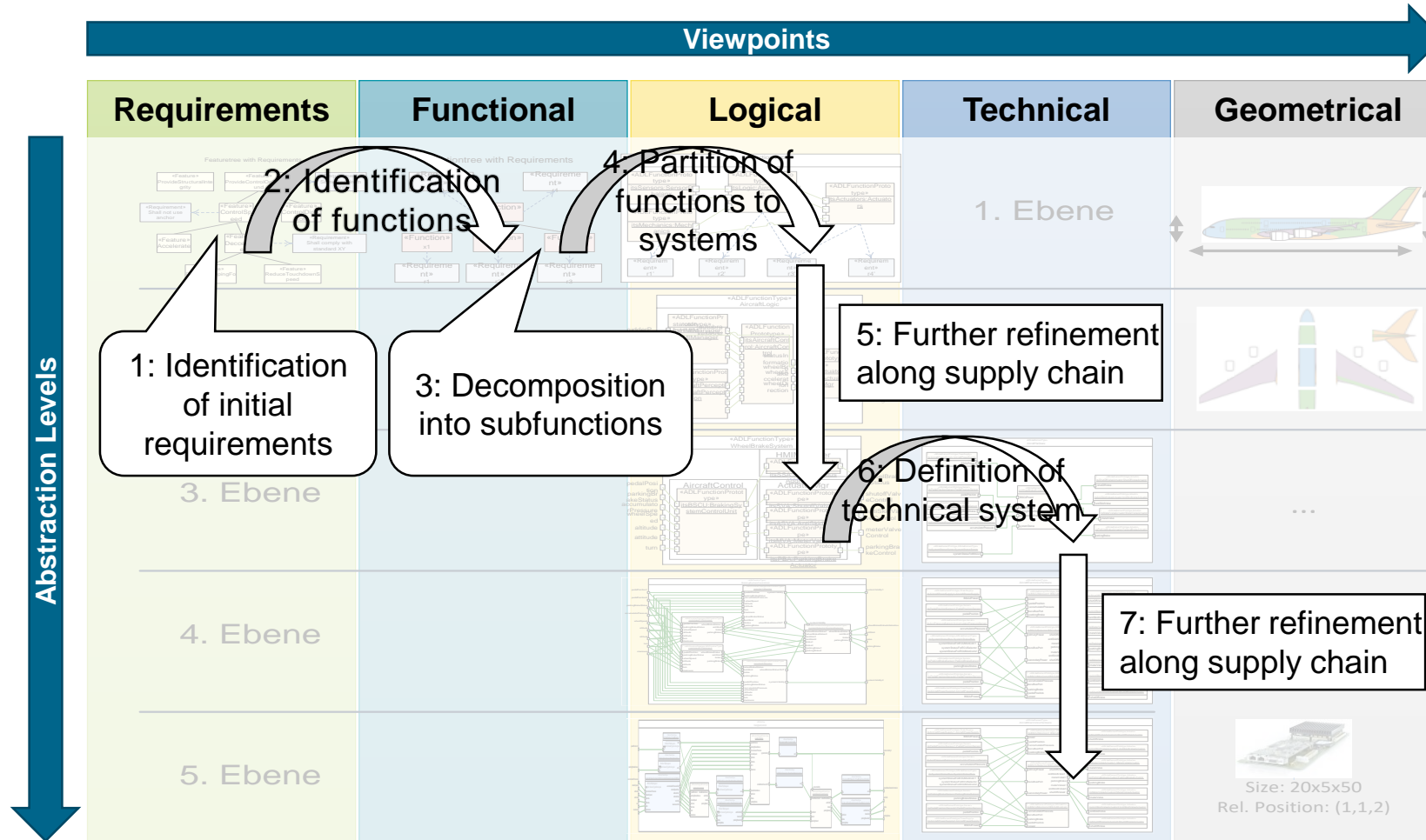
Viewpoints represent design under different concerns

Abstraction levels represent design with different granularity



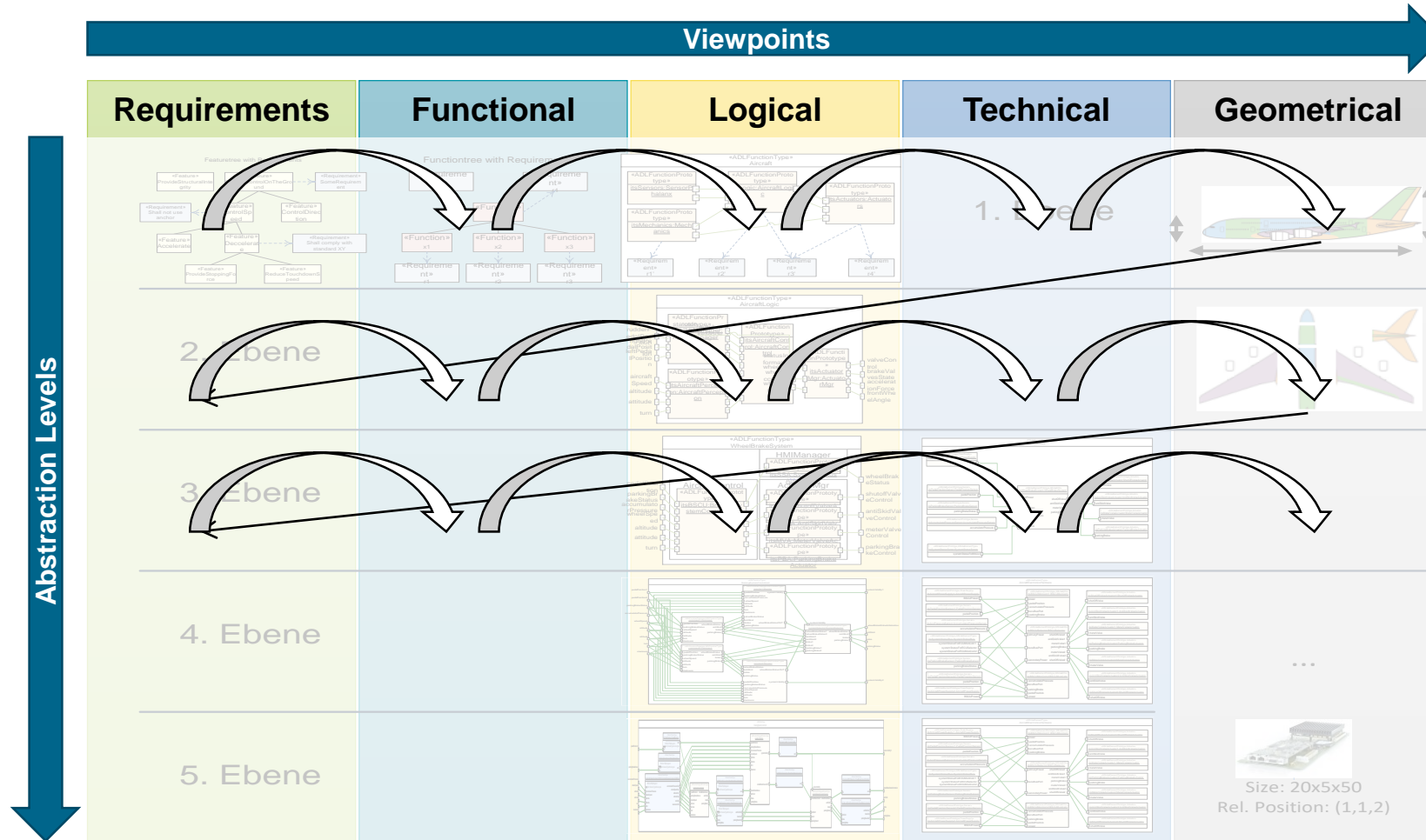
Compositional Semantic Framework

Example Design Steps



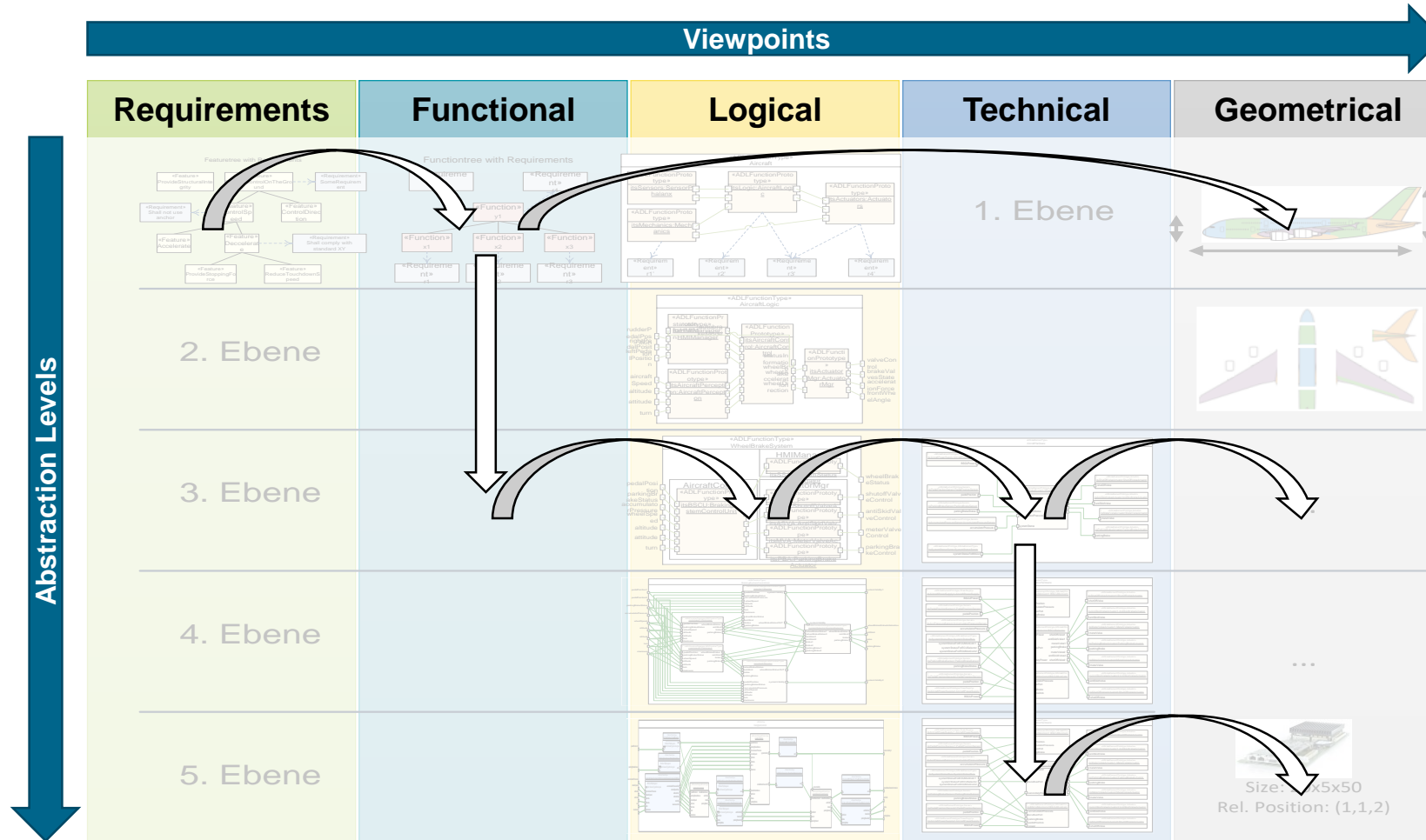
Compositional Semantic Framework

Example Design Steps



Compositional Semantic Framework

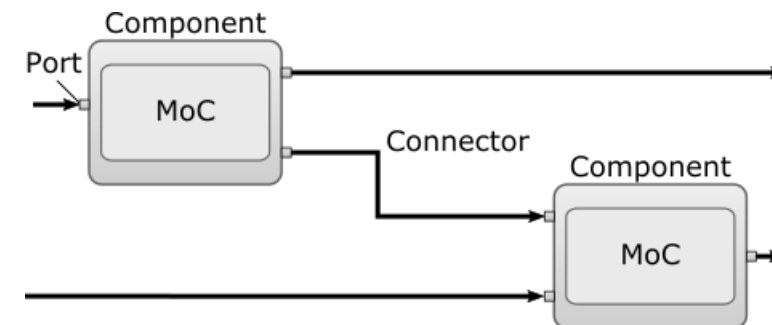
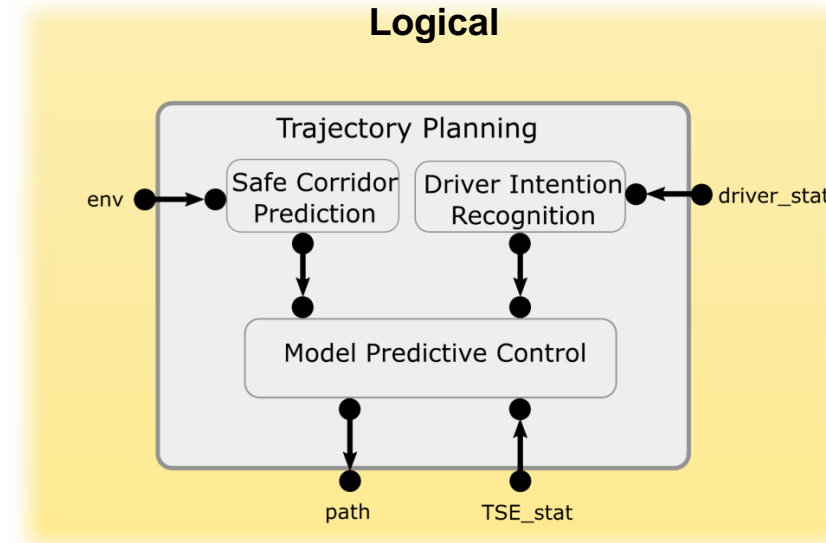
Example Design Steps



Compositional Semantic Framework

Components and Hierarchy

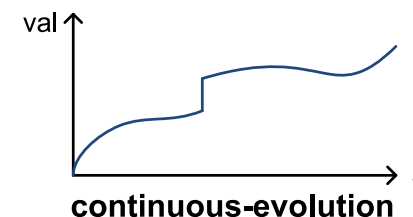
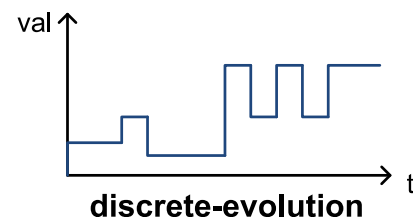
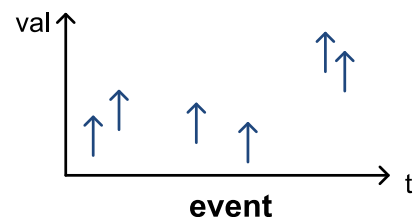
- Components
 - Basic design entity to structure models
 - Well defined interfaces
 - Can be reused in different design contexts
- Typed Ports
 - Define syntactical interface to adjacent components / environment
- Hierarchy and Composition
 - Allows deeply nested component hierarchies
 - Supports top-down and bottom-up design
- Connectors
 - Component interaction via port connections
 - Simple and complex connectors



Compositional Semantic Framework

Component Behavior (Semantic Domain)

- Behavior is visible at component ports
 - Common dense time domain: $\mathbb{T} = \mathbb{R}^{\geq 0}$ (for example)
 - Port types and value domains: D_p for port p
 - Behavior in term of signals: $s_p: \mathbb{T} \rightarrow V_p \cup \{\perp\}$
 - \perp means “absent value”
- Allows for specification of different “types” of behavior:
 - Discrete event** (absent values except for discrete set of time points)
 - Discrete evolution** (changes only at discrete time points)
 - Continuous evolution**

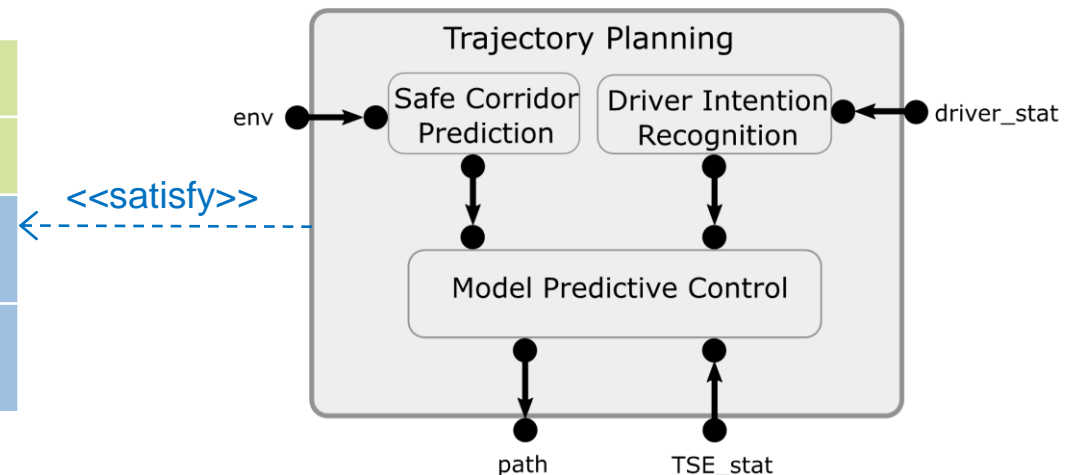


Compositional Semantic Framework

Contracts – Assume/Guarantee Reasoning

- **Assumptions (green):**
 - Specify necessary conditions of environment and surrounding components for component to work properly
- **Guarantees (blue):**
 - Specify required behavior that must be guaranteed by implementation if used in context compliant to assumptions
- Split in assumptions and guarantees allows **compositional** reasoning schemes
 - Implementations can be developed independently
 - Reduces verification complexity
 - Detect integration issues early in design process before developing implementations

Assumption A:	driver_stat occurs every 1s.
	TSE_stat occurs after every path.
Guarantee G:	Reaction(TSE_stat.reached,path) within [150,1500] ms.
	Reaction(TSE_stat.paa,path) within [150,200] ms.



Compositional Semantic Framework

Summary



- Contract-Based Design is a key enabler for rigorous formal system design
 - Contracts essentially are a particular way to specify behavior
 - CBD supports systems engineering processes
- Contract-Based Design is not a System Design Methodology
 - It enfold its benefits only in conjunction with other design paradigms
 - Such as the SPES Modeling Framework
- Contract-Based Design has a well-defined theoretical foundation
 - A “meta theory” of contract-based design
 - And many potential instances of it (like A/G contracts)

Agenda

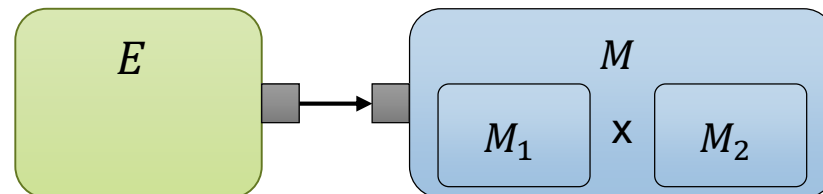


- Compositional Semantic Framework
 - Contract-based Design in the SPES Modelling Framework
- **A "Meta Theory" of Contract-Based Design**
 - What contracts are – exactly – good for
- Example
 - Applying contract-based design in safety-critical automotive system development

Contract-Based Design

Components

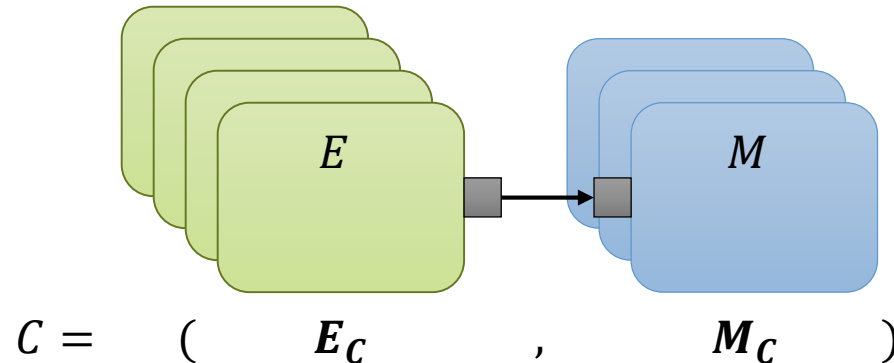
- Contract-Based Design implies existence of components
 - A *component* is the basic design entity.
 - Components can be *composed*: $M_1 \times M_2$.
 - Not all components can be composed
 - Components are *composable* if $M_1 \times M_2$ is well-defined.
 - An *environment* of M is a component E such that $E \times M$ is composable.



Contract-Based Design

Contracts

- A contract $C = (E_C, M_C)$ specifies two sets of components
 - Each pair $E \models^E C$ and $M \models^M C$ must be composable
 - $E \in E_C$ is an *environment* of C : $E \models^E C \Leftrightarrow E \in E_C$
 - $M \in M_C$ is an *implementation* of C : $M \models^M C \Leftrightarrow M \in M_C$



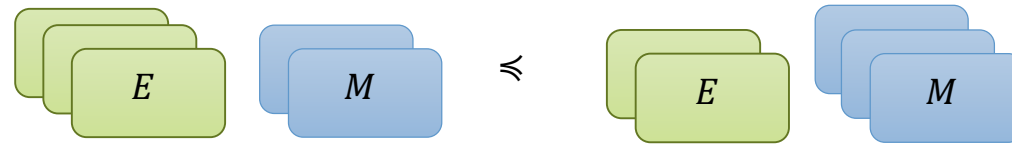
- We say, C is *consistent* iff it has at least one implementation: $M_C \neq \emptyset$
- We say, C is *compatible* iff it has at least one environment: $E_C \neq \emptyset$

Contract-Based Design

Refinement, Conjunction, Composition

- Refinement:

Contract C' refines C , $C' \preceq C$, iff $E_{C'} \supseteq E_C$ and $M_{C'} \subseteq M_C$

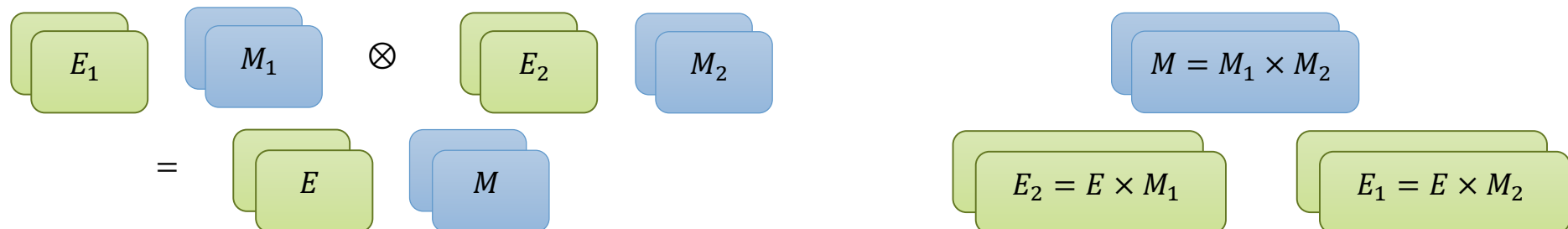


- Conjunction:

For all C_1, C_2 exists $\text{infimum}\{C_1, C_2\}$

- Composition:

$$C_1 \otimes C_2 = \min \left\{ C \mid \begin{array}{l} \forall M_1 \models^M C_1 \\ \forall M_2 \models^M C_2 \\ \forall E \models^E C \end{array} \Rightarrow \begin{array}{l} M_1 \times M_2 \models^M C \\ E \times M_1 \models^E C_2 \\ E \times M_2 \models^E C_1 \end{array} \right\}$$



Contract-Based Design

Important Properties



▪ Refinement

- Let be $C' \preceq C$
- Any implementation of C' is an implementation of C :
- Any environment of C is an environment of C' :

$$M \models^M C' \Rightarrow M \models^M C$$

$$E \models^E C \Rightarrow E \models^E C'$$

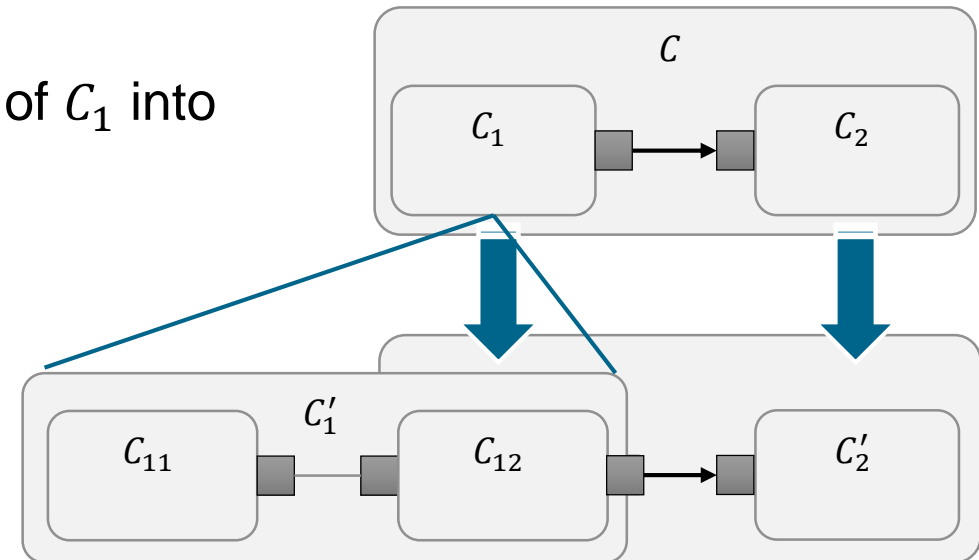
▪ Independent Implementability

- Let be $C'_1 \preceq C_1$ and $C'_2 \preceq C_2$
- Composition of C'_1 and C'_2 refines composition of C_1 and C_2 : $C'_1 \otimes C'_2 \preceq C_1 \otimes C_2$

Contract-Based Design

Important Properties Applied: Virtual Integration Test

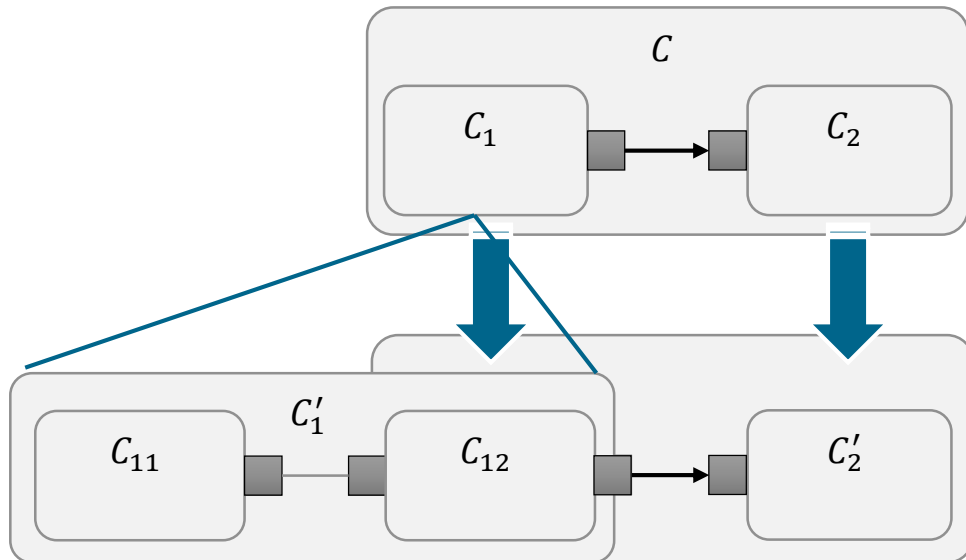
- Typical design steps “decomposition” and “refinement”:
 1. Component C is split into components C_1 and C_2 , respectively.
 - E.g. a complex driving function into planning and execution.
 2. Later on, components C_1 and C_2 are further refined.
 - Resulting in C'_1 and C'_2 , respectively.
 3. Component C'_1 results from further decomposition of C_1 into C_{11} and C_{12} .
 - C'_2 may be a re-used / library component.



Contract-Based Design

Important Properties Applied: Virtual Integration Test

- What are the proof obligation to ensure that C is satisfied?



$$C \cong C_1 \otimes C_2 \quad \text{VIT}$$

$$\left. \begin{array}{l} C_1 \cong C'_1 \\ C_2 \cong C'_2 \end{array} \right\} \Rightarrow C_1 \otimes C_2 \cong C'_1 \otimes C'_2$$

(Independent Implementability)

Refinement

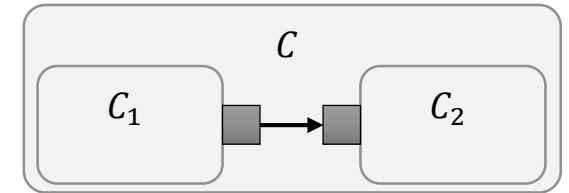
$$C'_1 \cong C_{11} \otimes C_{12} \quad \text{VIT}$$

$$C \cong (C_{11} \otimes C_{12}) \otimes C'_2$$

Contract-Based Design

Virtual Integration Test Conditions

- Virtual Integration Test: C is given as well as C_1 and C_2 .
 - Calculating $C_1 \otimes C_2$ and then checking $C_1 \otimes C_2 \preceq C$ is not necessary.



- Recall:

- $C' \preceq C$ iff $E_{C'} \supseteq E_C$ and $M_{C'} \subseteq M_C$

- $C_1 \otimes C_2 = \min \left\{ C \mid \begin{array}{l} \forall M_1 \models^M C_1 \\ \forall M_2 \models^M C_2 \\ \forall E \models^E C \end{array} \Rightarrow \begin{array}{l} M_1 \times M_2 \models^M C \\ E \times M_1 \models^E C_2 \\ E \times M_2 \models^E C_1 \end{array} \right\}$

- For any contract C with

$$\begin{array}{l} \forall M_1 \models^M C_1 \\ \forall M_2 \models^M C_2 \\ \forall E \models^E C \end{array} \Rightarrow \begin{array}{l} M_1 \times M_2 \models^M C \\ E \times M_1 \models^E C_2 \\ E \times M_2 \models^E C_1 \end{array} \quad \text{VIT Conditions}$$

holds that $C_1 \otimes C_2 \preceq C$.

Contract-Based Design

Summary



- Modern contract-based design is based on a meta-theory
 - Systems are constituted by components;
 - Contracts specify valid environments and implementations;
 - Contract-Based Design formally defines crucial properties for the development process
 - Refinement, composition, satisfaction, virtual integration testing, ...
- There are many different contract theories
 - We will see more about A/G-contracts

Agenda

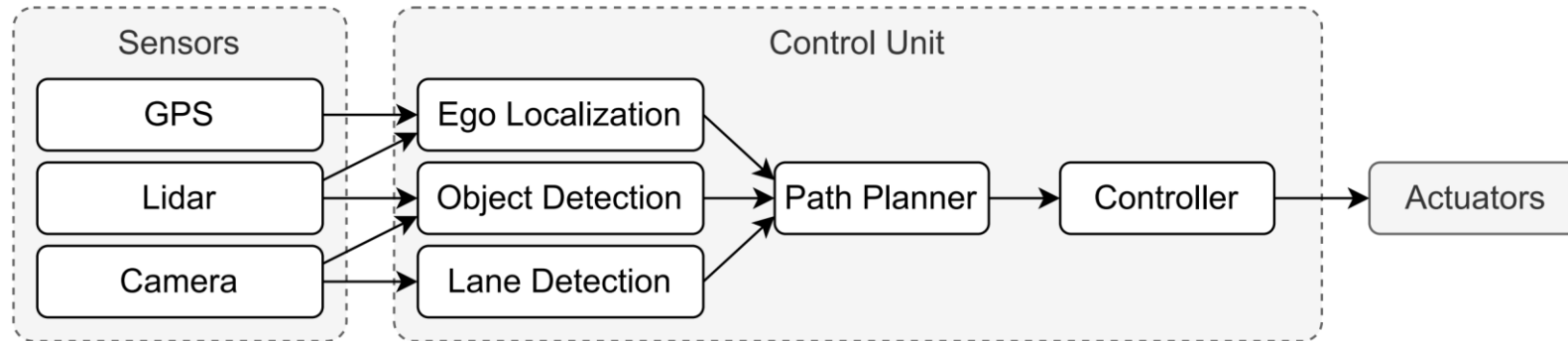


- **Compositional Semantic Framework**
 - Contract-based Design in the SPES Modelling Framework
- A "Meta Theory" of Contract-Based Design
 - What contracts are – exactly – good for
- **Example**
 - Applying contract-based design in safety-critical automotive system development

Application Example

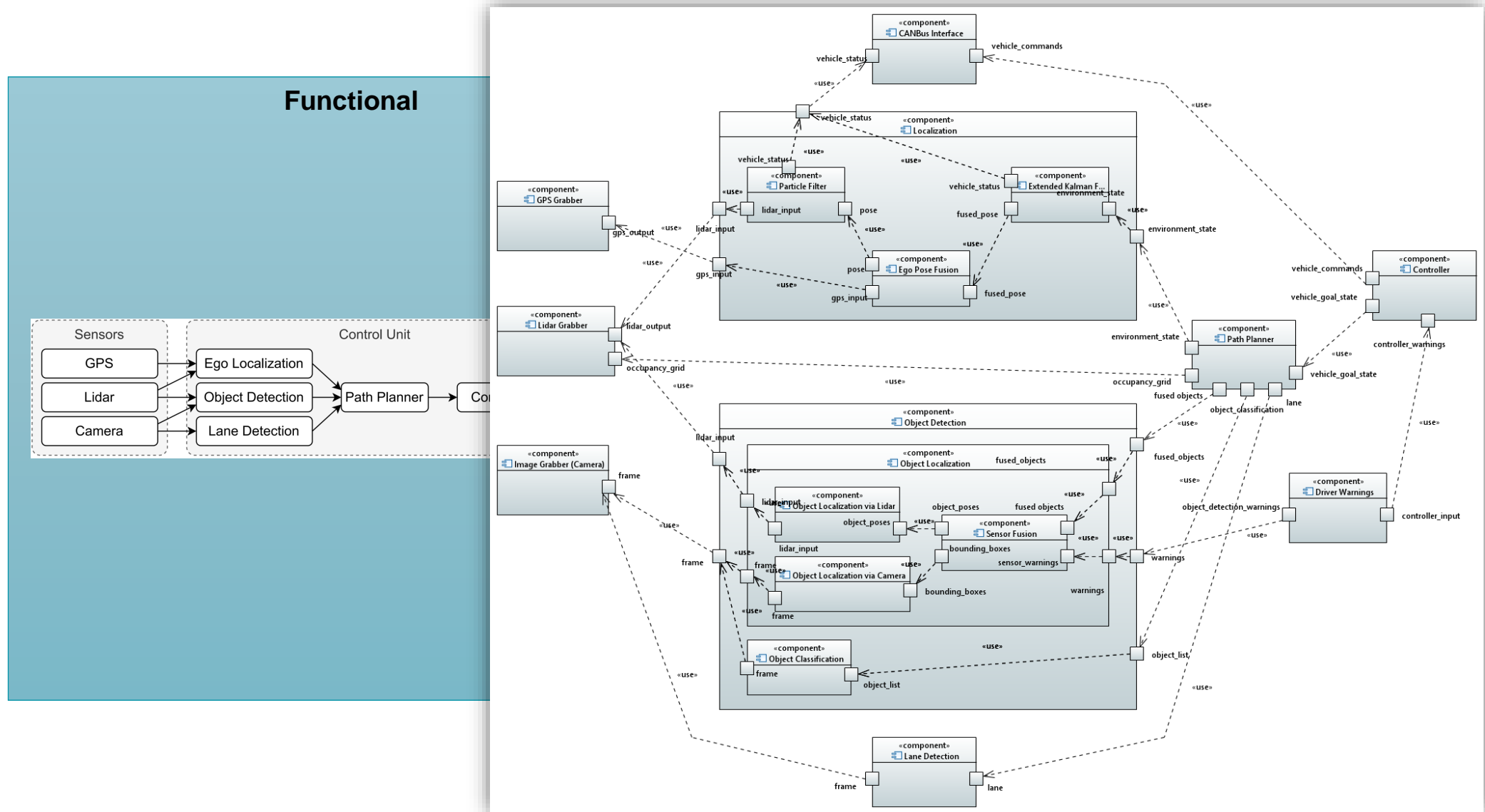
Overview

- (Part of) a highly automated vehicle that shall perform two main functions:
 - i. following a predefined route,
 - ii. avoiding collisions with traffic participants / obstacles.

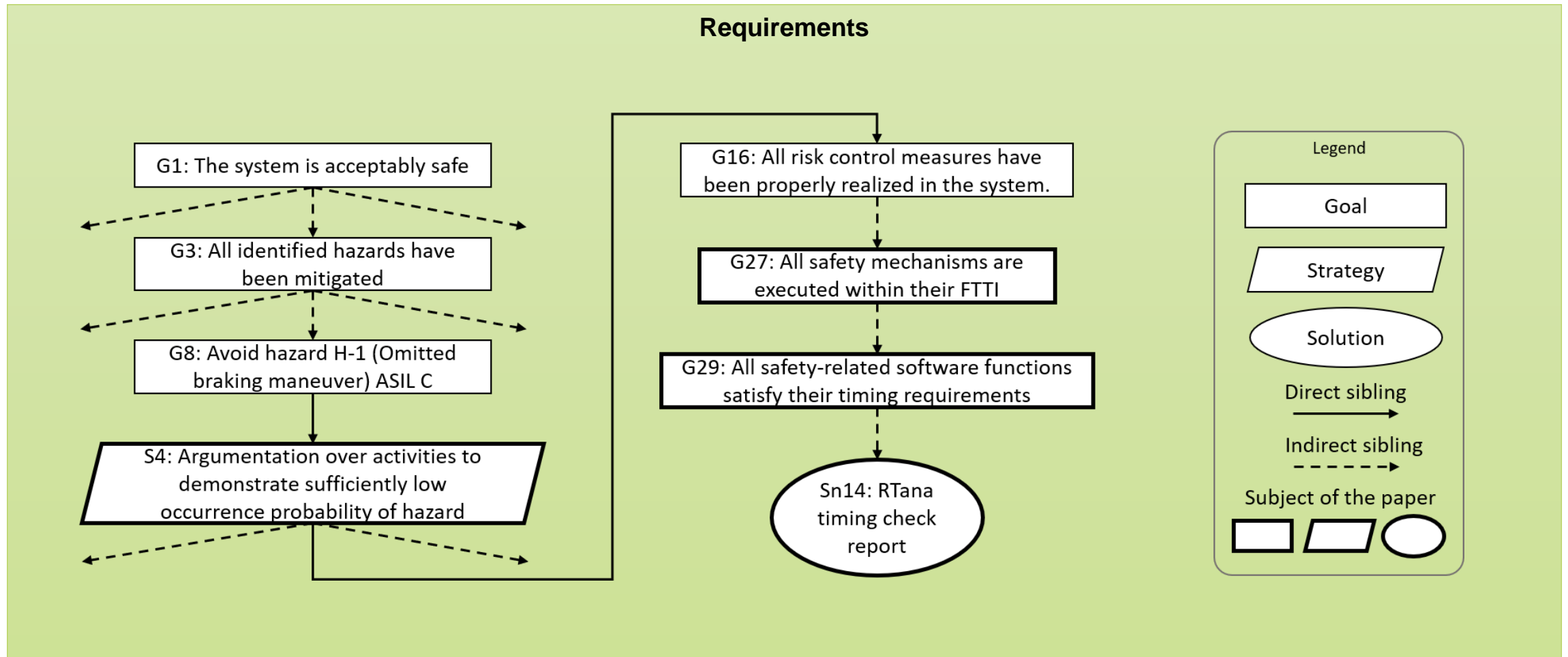


- Sensors, Control Unit, Actuators:
 - Ego Localization (localize the ego vehicle),
 - Object Detection (detect, localize, and classify traffic participants and obstacles),
 - Lane Detection (detect lane boundaries),
 - Path Planner (plan maneuvers),
 - Controller (and calculate actuator commands).

Where are we in the SPES Framework?



Safety Case (Excerpt)



Hazard Analysis and Risk Assessment

Hazards – Safety Goals – Safety Concept

- HARA: Identification of hazards and corresponding risks
 - Here: **H-1: Omitted braking maneuver**
(a necessary braking maneuver of the ego vehicle is not performed (in time))

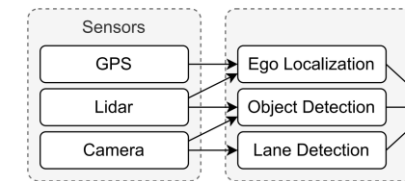
- Fault Analysis (FTA, FMEA) [...]

- Two failure modes are considered:

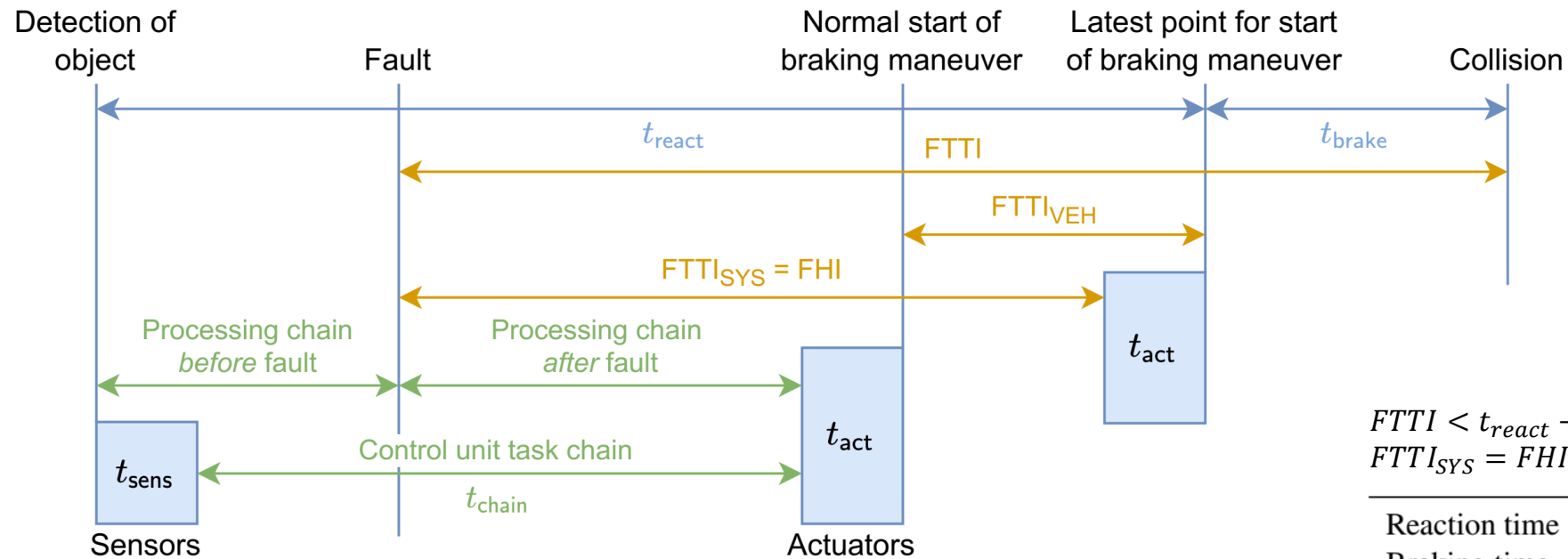
1. One of *Lidar* or *Camera* fails permanently.
2. One or both of *Lidar* and *Camera* fail for a limited amount of time.

- Safety Requirements:

ID	Description
SG-1	The system shall prevent omitting required braking maneuvers.
SR-1-1a	The system shall use lidar and camera for object detection.
SR-1-1b	The system shall ensure that objects are detected if lidar or camera fails.
SR-1-1.1	The system shall identify sensor failures.
SR-1-1.1.1	The system shall identify when the lidar sensor has failed.
SR-1-1.1.2	The system shall identify when the camera has failed.
SR-1-1.2	The system shall mitigate sensor failures.
SR-1-1.2.1	The system shall detect objects using lidar.
SR-1-1.2.2	The system shall detect objects using camera.
SR-1-1.2.3	The system shall fuse the objects detected by lidar and camera.
SR-1-1.3	The system shall use only information from working sensors.
SR-1-2	...



Fault-Tolerant Time Interval (FTTI)



$$FTTI < t_{react} + t_{brake} \approx 4.57s$$

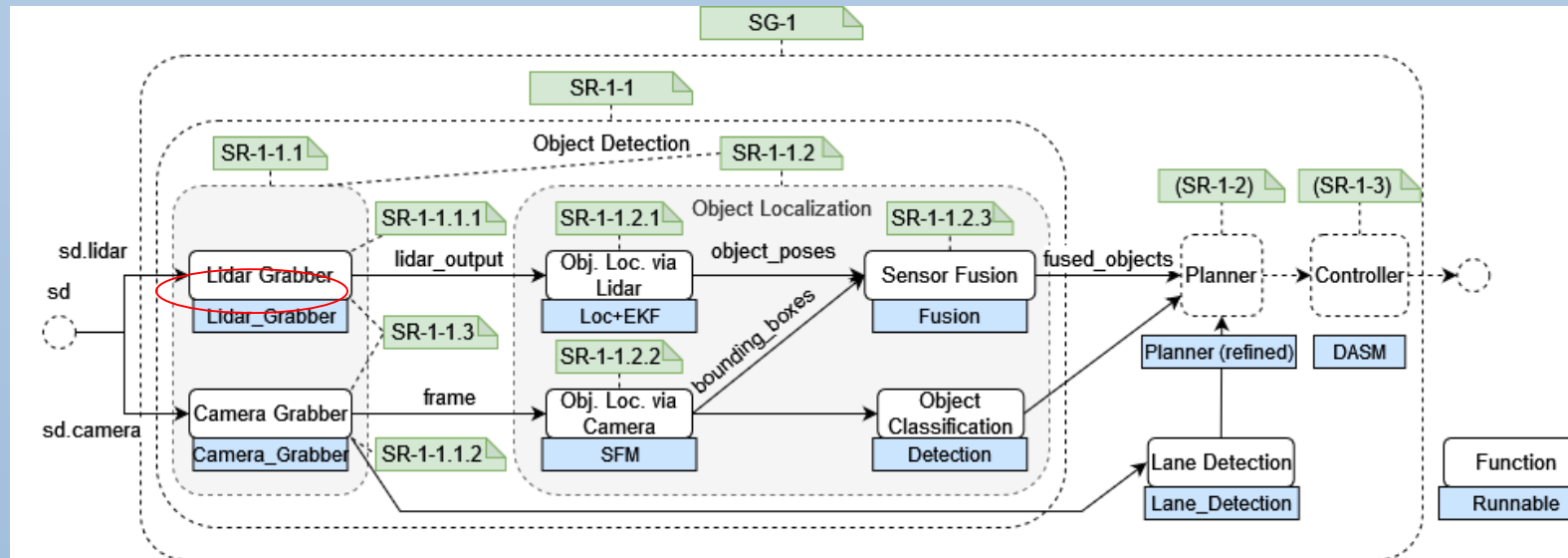
$$FTTI_{SYS} = FHI < t_{react} + t_{act} + t_{sense} \approx 2.43s$$

Reaction time	t_{react}	2.57 s
Braking time	t_{brake}	2.0 s
Sensor processing time	t_{sense}	50 ms
Actuator time	t_{act}	100 ms
Control unit WCRT	t_{chain}	< 800 ms

Remark: We consider a *safety mechanism implemented with emergency operation* [ISO26262:2018]

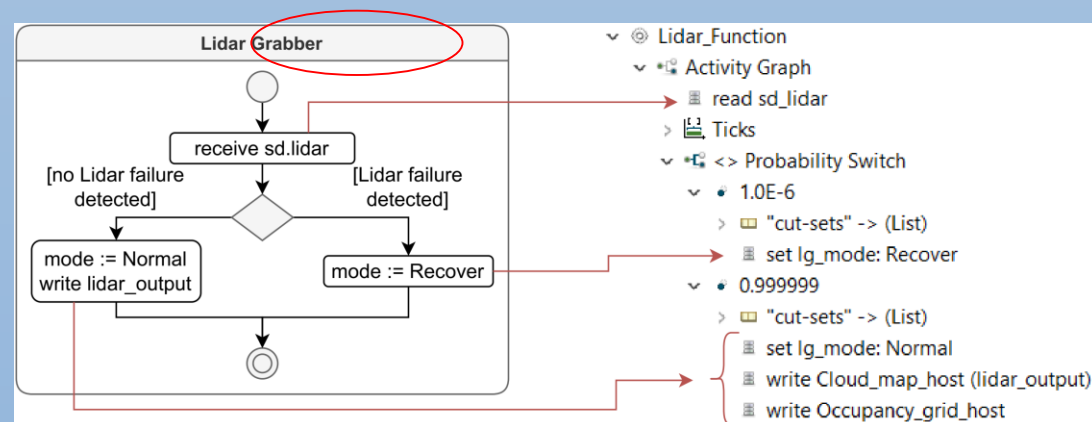
Mapping Requirements to Technical Elements

Technical Viewpoint



ID	Description
SG-1	The system shall prevent omitting required braking maneuvers.
SR-1-1a	The system shall use lidar and camera for object detection.
SR-1-1b	The system shall ensure that objects are detected if lidar or camera fails.
SR-1-1.1	The system shall identify sensor failures.
SR-1-1.1.1	The system shall identify when the lidar sensor has failed.
SR-1-1.1.2	The system shall identify when the camera has failed.
SR-1-1.2	The system shall mitigate sensor failures.
SR-1-1.2.1	The system shall detect objects using lidar.
SR-1-1.2.2	The system shall detect objects using camera.
SR-1-1.2.3	The system shall fuse the objects detected by lidar and camera.
SR-1-1.3	The system shall use only information from working sensors.
SR-1-2	...

Implementation in APP4MC:



Mapping Requirements to Technical Elements

Timing Contracts

A: sd occurs every 66 ms.
 G1: Reaction(sd.OK,fused_objects) within [100,1006] ms in mode Normal.
 G2: Reaction(sd.NOT_OK,set(mode,Recover)) within [5,66] ms in mode Normal.
 G3: Reaction(sd.OK,fused_objects) within [100,940] ms in mode Recover.

Υ ? (VIT)

Lidar Grabber

A_{L1} : sd.lidar occurs every 66 ms.
 G_{L1} : Reaction(sd.lidar.OK,(set(mode,Normal),lidar_output)) within [5,60] ms.
 G_{L2} : Reaction(sd.lidar.NOT_OK,(set(mode,Recover)) within [5,60] ms.

Camera Grabber

A_{C1} : sd.camera occurs every 66 ms.
 G_{C1} : Reaction(sd.camera.OK,(set(mode,Normal),lidar_output)) within [7,60] ms.
 G_{C2} : Reaction(sd.camera.NOT_OK,(set(mode,Recover)) within [7,60] ms.

Object Localization via Lidar

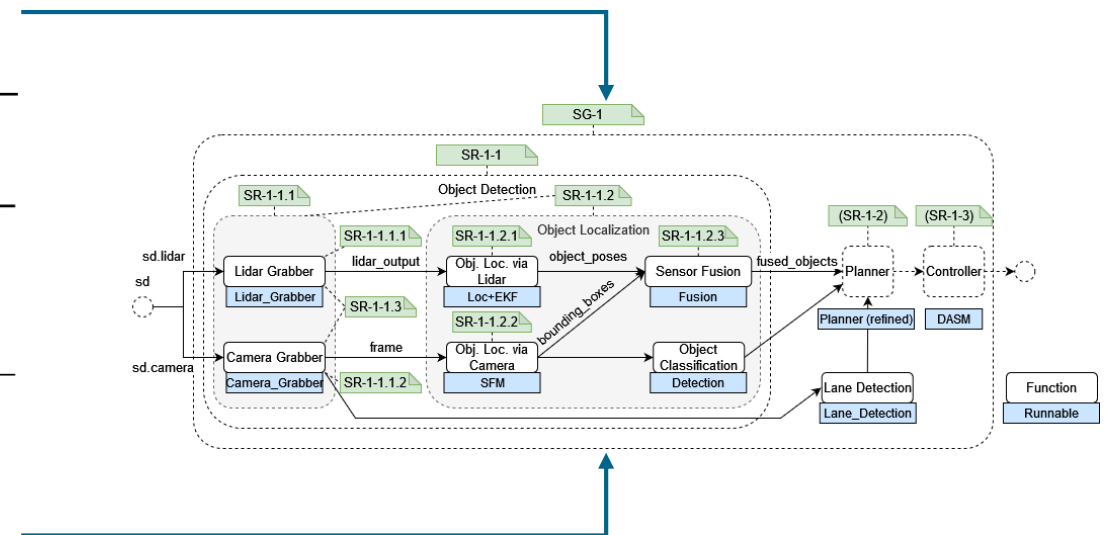
G_{OL} : Reaction(lidar_output,object_poses) within [120,795] ms.

Object Localization via Camera

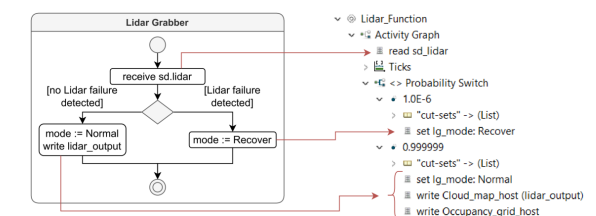
G_{OC} : Reaction(frame,bounding_boxes) within [20,55] ms.

Sensor Fusion

G_{F1} : Reaction(object_poses,fused_objects) within [5,25] ms.
 G_{F2} : Reaction(bounding_boxes,fused_objects) within [5,25] ms.



\Rightarrow ? (satisfaction)



Mapping Requirements to Technical Elements

Virtual Integration Testing

Exercise

A: sd occurs every 66 ms.

G1: Reaction(sd.OK,fused_objects) within [100,1006] ms in mode Normal.

G2: Reaction(sd.NOT_OK,set(mode,Recover)) within [5,66] ms in mode Normal.

G3: Reaction(sd.OK,fused_objects) within [100,940] ms in mode Recover.

Lidar Grabber

A_{L1}: sd.lidar occurs every 66 ms.

G_{L1}: Reaction(sd.lidar.OK,(set(mode,Normal),lidar_output)) within [5,60] ms.

G_{L2}: Reaction(sd.lidar.NOT_OK,(set(mode,Recover))) within [5,60] ms.

Camera Grabber

A_{C1}: sd.camera occurs every 66 ms.

G_{C1}: Reaction(sd.camera.OK,(set(mode,Normal),lidar_output)) within [7,60] ms.

G_{C2}: Reaction(sd.camera.NOT_OK,(set(mode,Recover))) within [7,60] ms.

Object Localization via Lidar

G_{OL}: Reaction(lidar_output,object_poses) within [120,795] ms.

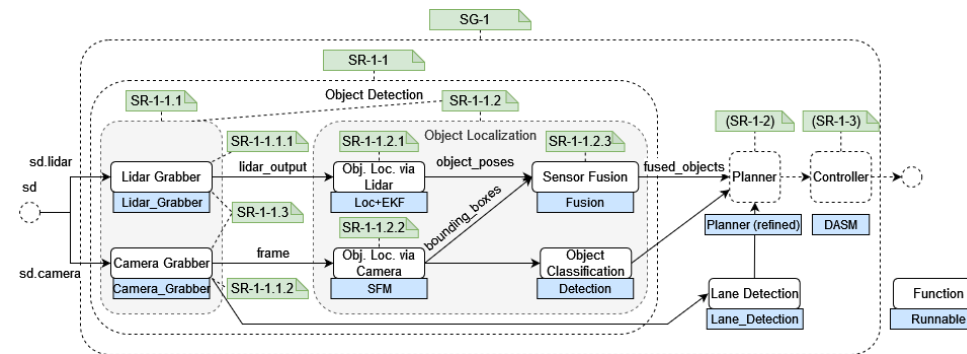
Object Localization via Camera

G_{OC}: Reaction(frame,bounding_boxes) within [20,55] ms.

Sensor Fusion

G_{F1}: Reaction(object_poses,fused_objects) within [5,25] ms.

G_{F2}: Reaction(bounding_boxes,fused_objects) within [5,25] ms.



■ Prerequisites:

- Signal “sd” contains both “sd.lidar” and “sd.camera”
- Signal “sd” is OK means sensors are OK:
 - `sd.OK := sd.lidar.OK or sd.camera.OK`
- Mode of “object detection” is the shared with sub components “camera grabber” and “lidar grabber”:
 - `Object Detection.mode = Lidar Grabber.mode = Camera Grabber.mode`

Mapping Requirements to Technical Elements

Virtual Integration Testing

Exercise

A: sd occurs every 66 ms.

G1: Reaction(sd.OK,fused_objects) within [100,1006] ms in mode Normal.

G2: Reaction(sd.NOT_OK,set(mode,Recover)) within [5,66] ms in mode Normal.

G3: Reaction(sd.OK,fused_objects) within [100,940] ms in mode Recover.

Lidar Grabber

A_{L1}: sd.lidar occurs every 66 ms.

G_{L1}: Reaction(sd.lidar.OK,(set(mode,Normal),lidar_output)) within [5,60] ms.

G_{L2}: Reaction(sd.lidar.NOT_OK,(set(mode,Recover))) within [5,60] ms.

Camera Grabber

A_{C1}: sd.camera occurs every 66 ms.

G_{C1}: Reaction(sd.camera.OK,(set(mode,Normal),lidar_output)) within [7,60] ms.

G_{C2}: Reaction(sd.camera.NOT_OK,(set(mode,Recover))) within [7,60] ms.

Object Localization via Lidar

G_{OL}: Reaction(lidar_output,object_poses) within [120,795] ms.

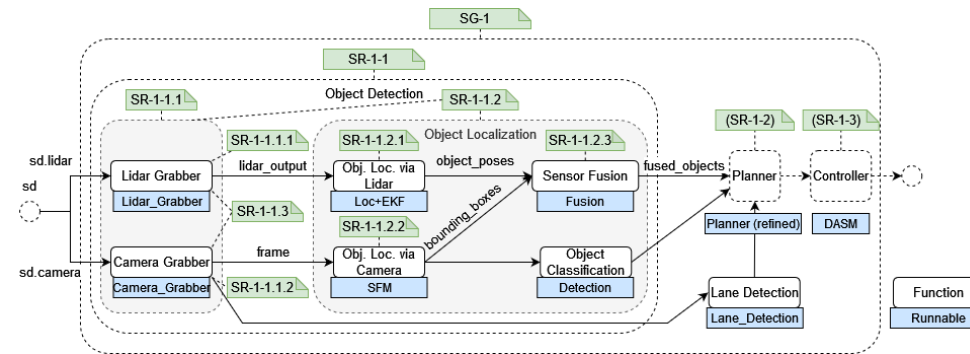
Object Localization via Camera

G_{OC}: Reaction(frame,bounding_boxes) within [20,55] ms.

Sensor Fusion

G_{F1}: Reaction(object_poses,fused_objects) within [5,25] ms.

G_{F2}: Reaction(bounding_boxes,fused_objects) within [5,25] ms.



Assumptions:

Object Detection:

- A: sd occurs every 66ms

Lidar Grabber:

- A_{L1}: sd.lidar occurs every 66ms

Camera Grabber:

- A_{C1}: sd.camera occurs every 66ms

- Check ($A \subseteq A_{L1} \wedge A \subseteq A_{C1}$): ✓

Mapping Requirements to Technical Elements

Virtual Integration Testing

Exercise

A: sd occurs every 66 ms.

G1: Reaction(sd.OK,fused_objects) within [100,1006] ms in mode Normal.

G2: Reaction(sd.NOT_OK,set(mode,Recover)) within [5,66] ms in mode Normal.

G3: Reaction(sd.OK,fused_objects) within [100,940] ms in mode Recover.

Lidar Grabber

A_{L1}: sd.lidar occurs every 66 ms.

G_{L1}: Reaction(sd.lidar.OK,(set(mode,Normal),lidar_output)) within [5,60] ms.

G_{L2}: Reaction(sd.lidar.NOT_OK,(set(mode,Recover)) within [5,60] ms.

Camera Grabber

A_{C1}: sd.camera occurs every 66 ms.

G_{C1}: Reaction(sd.camera.OK,(set(mode,Normal),lidar_output)) within [7,60] ms.

G_{C2}: Reaction(sd.camera.NOT_OK,(set(mode,Recover)) within [7,60] ms.

Object Localization via Lidar

G_{OL}: Reaction(lidar_output,object_poses) within [120,795] ms.

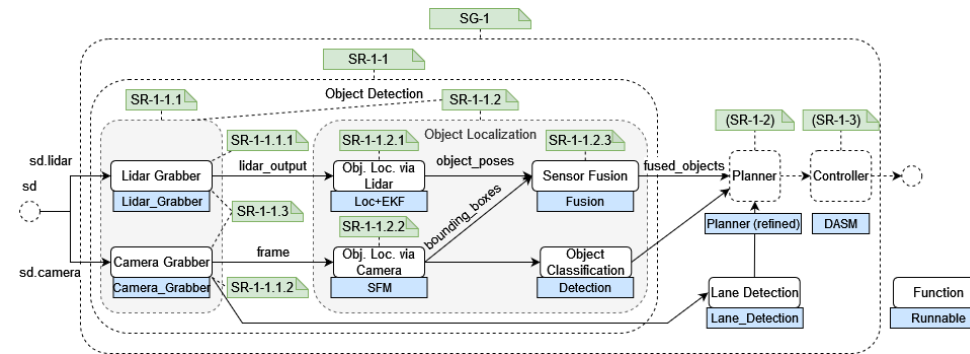
Object Localization via Camera

G_{OC}: Reaction(frame,bounding_boxes) within [20,55] ms.

Sensor Fusion

G_{F1}: Reaction(object_poses,fused_objects) within [5,25] ms.

G_{F2}: Reaction(bounding_boxes,fused_objects) within [5,25] ms.



Guarantees:

- Object Detection:
 - G1: Reaction(sd,fused_objects) within [100,1006]ms in mode Normal.
- Lidar Grabber:
 - G_{L1}: Reaction(sd.lidar.OK,lidar_output) within [5,60]ms.
- Object Localization via Lidar:
 - G_{OL}: Reaction(lidar_output,object_poses) within [120,795]ms.
- Sensor Fusion:
 - G_{F1}: Reaction(object_poses,fused_objects) within [5,25]ms.
- Check: $5 + 120 + 5 = 130, 60 + 795 + 25 = 875, [130,875] \subseteq [100,1006]$ ✓

Mapping Requirements to Technical Elements

Virtual Integration Testing

Exercise

A: sd occurs every 66 ms.

G1: Reaction(sd.OK,fused_objects) within [100,1006] ms in mode Normal.

G2: Reaction(sd.NOT_OK,set(mode,Recover)) within [5,66] ms in mode Normal.

G3: Reaction(sd.OK,fused_objects) within [100,940] ms in mode Recover.

Lidar Grabber

A_{L1}: sd.lidar occurs every 66 ms.

G_{L1}: Reaction(sd.lidar.OK,(set(mode,Normal),lidar_output)) within [5,60] ms.

G_{L2}: Reaction(sd.lidar.NOT_OK,(set(mode,Recover))) within [5,60] ms.

Camera Grabber

A_{C1}: sd.camera occurs every 66 ms.

G_{C1}: Reaction(sd.camera.OK,(set(mode,Normal),lidar_output)) within [7,60] ms.

G_{C2}: Reaction(sd.camera.NOT_OK,(set(mode,Recover))) within [7,60] ms.

Object Localization via Lidar

G_{OL}: Reaction(lidar_output,object_poses) within [120,795] ms.

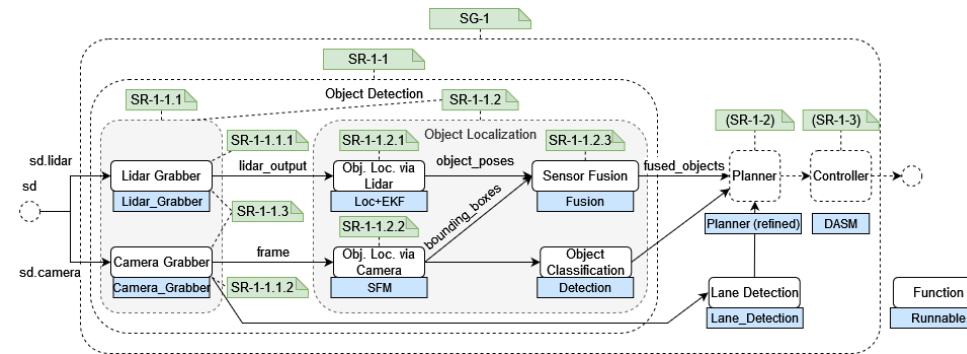
Object Localization via Camera

G_{OC}: Reaction(frame,bounding_boxes) within [20,55] ms.

Sensor Fusion

G_{F1}: Reaction(object_poses,fused_objects) within [5,25] ms.

G_{F2}: Reaction(bounding_boxes,fused_objects) within [5,25] ms.



Guarantees:

Object Detection:

- G1: Reaction(sd,fused_objects) within [100,1006]ms in mode Normal.

Camera Grabber:

- G_{C1}: Reaction(sd.camera.OK,frame) within [7,60]ms.

Object Localization via Camera:

- G_{OC}: Reaction(frame,object_poses) within [20,55]ms.

Sensor Fusion:

- G_{F1}: Reaction(object_poses,fused_objects) within [5,25]ms.

- Check: $7 + 20 + 5 = 32, 60 + 55 + 25 = 140, [32,140] \subseteq [100,1006]$ ⚡

Mapping Requirements to Technical Elements

Virtual Integration Testing

Exercise

A: sd occurs every 66 ms.

G1: Reaction(sd.OK,fused_objects) within [100,1006] ms in mode Normal.

G2: Reaction(sd.NOT_OK,set(mode,Recover)) within [5,66] ms in mode Normal.

G3: Reaction(sd.OK,fused_objects) within [100,940] ms in mode Recover.

Lidar Grabber

A_{L1}: sd.lidar occurs every 66 ms.

G_{L1}: Reaction(sd.lidar.OK,(set(mode,Normal),lidar_output)) within [5,60] ms.

G_{L2}: Reaction(sd.lidar.NOT_OK,(set(mode,Recover)) within [5,60] ms.

Camera Grabber

A_{C1}: sd.camera occurs every 66 ms.

G_{C1}: Reaction(sd.camera.OK,(set(mode,Normal),lidar_output)) within [7,60] ms.

G_{C2}: Reaction(sd.camera.NOT_OK,(set(mode,Recover)) within [7,60] ms.

Object Localization via Lidar

G_{OL}: Reaction(lidar_output,object_poses) within [120,795] ms.

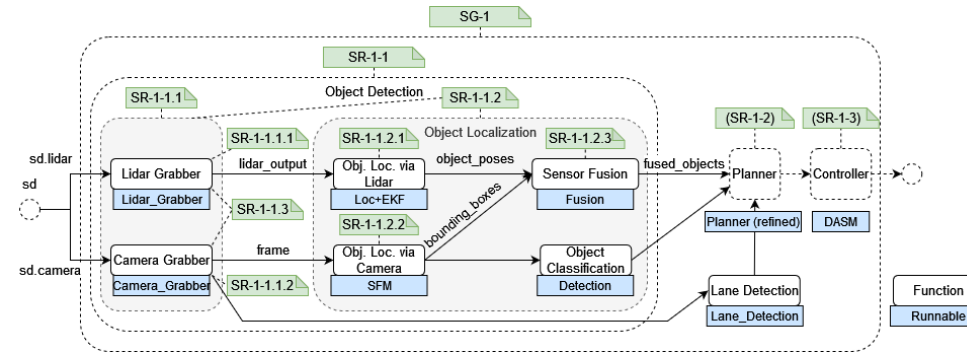
Object Localization via Camera

G_{OC}: Reaction(frame,bounding_boxes) within [20,55] ms.

Sensor Fusion

G_{F1}: Reaction(object_poses,fused_objects) within [5,25] ms.

G_{F2}: Reaction(bounding_boxes,fused_objects) within [5,25] ms.



Guarantees:

Object Detection:

- G1: Reaction(sd.NOK,set(mode,Recover)) within [5,66]ms in mode Normal.

Lidar Grabber:

- GL1: Reaction(sd.lidar.OK,set(mode,Normal)) within [5,60]ms.
- GL2: Reaction(sd.lidar.NOK,set(mode,Recover)) within [5,60]ms in mode Normal.

Camera Grabber:

- GC1: Reaction(sd.camera.OK,set(mode,Normal)) within [7,60]ms.
- GC2: Reaction(sd.camera.NOK,set(mode,Recover)) within [7,60]ms in mode Normal.

Check: ⚡

Satisfaction Checking

Lidar Grabber

A_{L1} : sd.lidar occurs every 66 ms.

G_{L1} : Reaction(sd.lidar.OK, (set(mode,Normal),lidar_output)) within [5,60] ms.

G_{L2} : Reaction(sd.lidar.NOT_OK, (set(mode, Recover)) within [5,60] ms.

Camera Grabber

A_{C1} : sd.camera occurs every 66 ms.

G_{C1} : Reaction(sd.camera.OK, (set(mode,Normal),lidar_output)) within [7,60] ms.

G_{C2} : Reaction(sd.camera.NOT_OK, (set(mode,Recover)) within [7,60] ms.

Object Localization via Lidar

G_{OL} : Reaction(lidar_output,object_poses) within [120,795] ms.

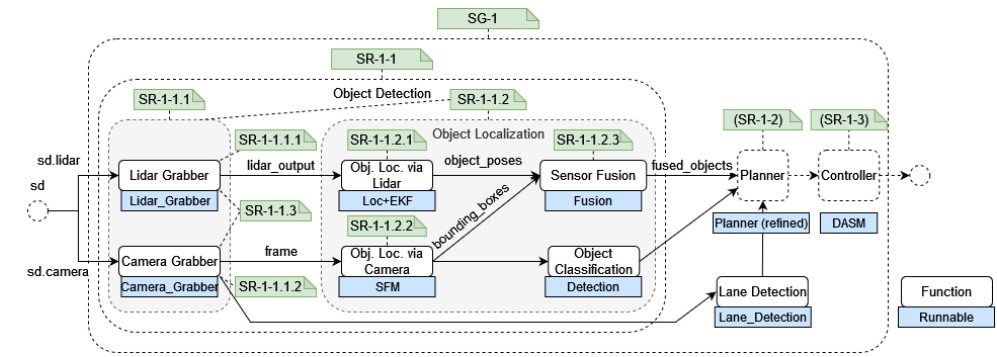
Object Localization via Camera

G_{OC} : Reaction(frame,bounding_boxes) within [20,55] ms.

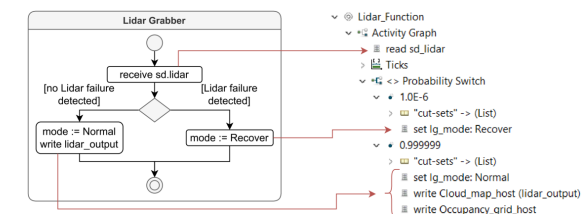
Sensor Fusion

G_{F1} : Reaction(object_poses,fused_objects) within [5,25] ms.

G_{F2} : Reaction(bounding_boxes,fused_objects) within [5,25] ms.



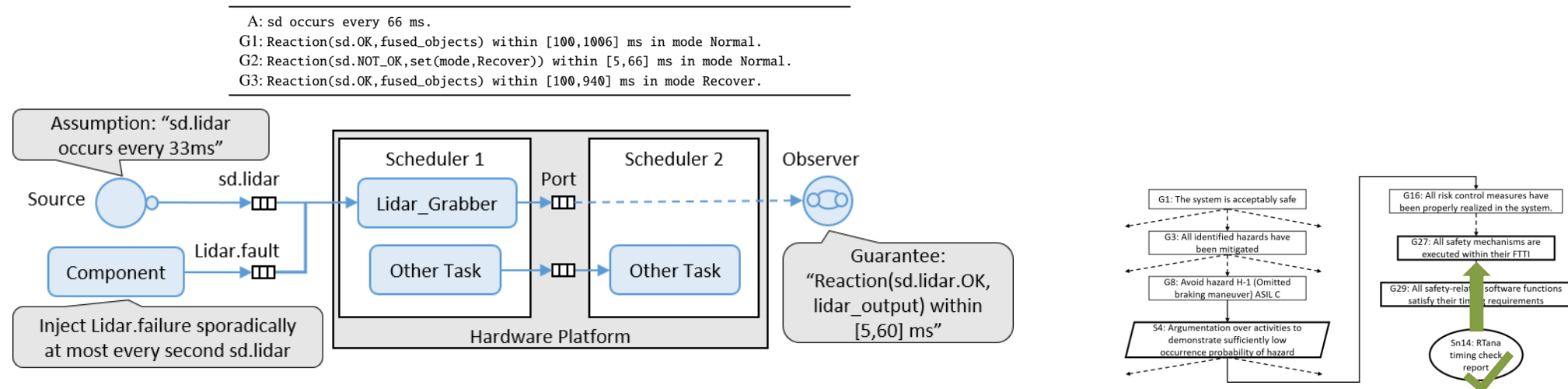
≡ ?



Satisfaction Checking

Approach and Structure of the RTana_{2sim} Model

- APP4MC model is translated into RTana_{2sim} model
- Assumptions are translated into event sources
- Guarantees are translated into observer automata
- Additional components provide for fault injection



Satisfaction Checking Example

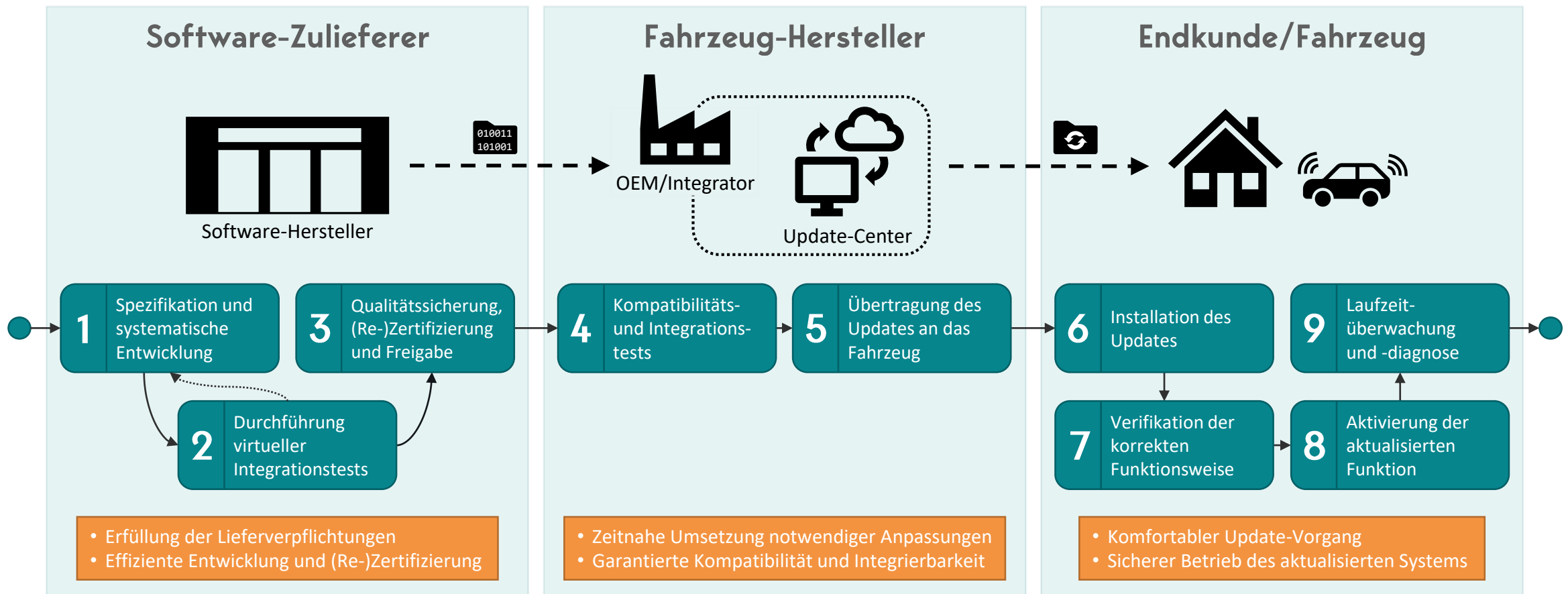
The screenshot displays the `rtana2sim` software interface. The main window is divided into several sections:

- Input:** A code editor showing a system model with constants, signals, and sources. The code includes copyright information for DLR and defines various system parameters and components.
- Log:** A window showing the execution log, including messages like "Start RTana parser...", "Started in Analysis Mode", and "Analysis: started".
- Transitions:** A list of system transitions, including `dummy`, `OS_Overhead_trigg`, `Lidar_Grabber_trigg`, and others.
- Sources/Components:** A list of system components, including `periodic_100ms`, `periodic_33ms`, `periodic_5ms`, and others.
- State Transition Diagram:** A large diagram showing the state space of the system. The diagram is a grid of states, with transitions between them. The states are color-coded (green, yellow, orange, red) to represent different levels of satisfaction or error. The diagram shows a complex, interconnected state space.

At the bottom right, there is a "Show Error Trace" button.

Entwicklung von sicheren Over-the-Air-Updates

Beteiligte Akteure und relevante Prozessschritte

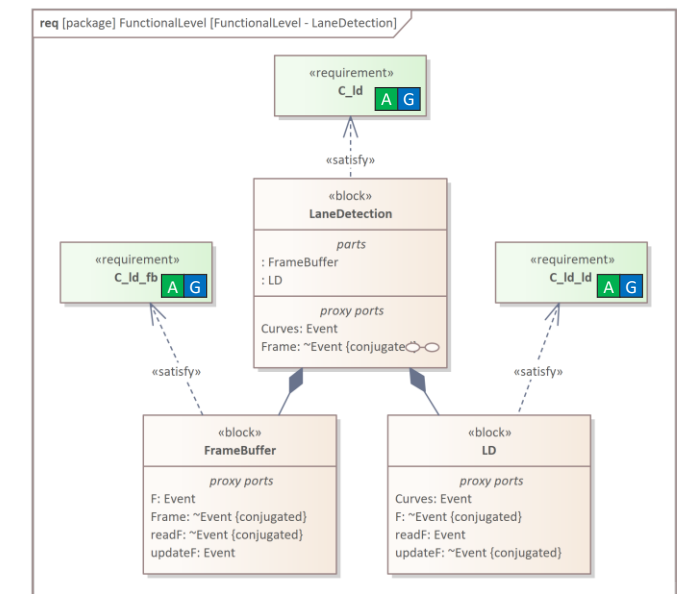
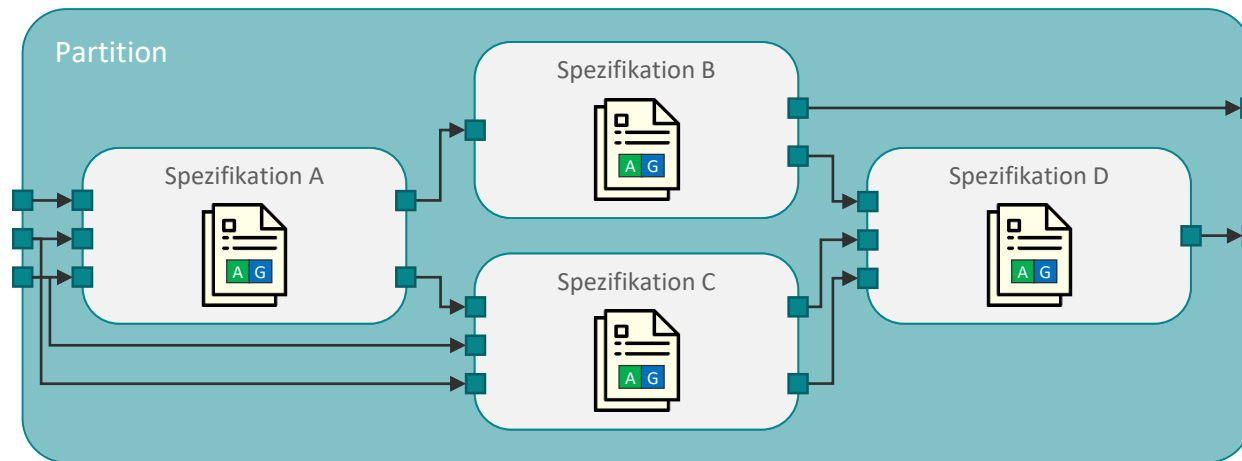


Prozessschritte beim Software-Zulieferer

Entwicklung neuer Funktionen und Updates

1. Systematische Entwicklung neuer Funktionen und Updates

- Spezifikation des gewünschten Verhaltens von Software-Komponenten
 - z.B. unter Verwendung von Assume/Guarantee-Contracts (u.a. für Zeit- und Speicherbudgets)
 - Ausnutzung formal definierter Kompositions- und Verfeinerungsoperationen
- Prozessbegleitende Verfeinerung der Spezifikationen



Bildquelle des Symbols für Spezifikationen: Modifizierte Version eines Icons von *smalllikeart* auf Flaticon.
URL: https://www.flaticon.com/de/kostenloses-icon/dokument_888034

Conclusion



- Model-based systems engineering helps in solving many challenges in engineering processes
- The SPES modeling framework aims at supporting engineering of (safety-critical) cyber-physical systems
- Contract-based design provides formal design and engineering support:
 - Correctness of key design steps becomes verifiable
 - Enables tool support in verification tasks
- Industrial example demonstrates applicability