# open robinos specification

## architecture, mechanisms, interfaces, and testing

1.1.0, Released

## Legal disclaimer

# Table of Contents

# 1. Objective

With functions such as Lane Assist, Automated Emergency Braking, and Adaptive Cruise Control, the current generation of automobiles has taken the first steps towards automated driving. The automotive industry is heading towards highly and even completely automated vehicles. This is no simple task. Among other things, we need a scalable systems architecture agreed on in the industry that covers all driver assistance functions and integrates them into one fully automated system.

This open interface specification describes a functional architecture to address this problem. It gives an initial overview on how to structure functionally necessary parts, and can be seen as a starting point for collaboration work.

## 1.1. Structure of the document

Excluding the section you are currently reading, this document begins in chapter 2 with an outlining of the motivation that leads towards the desire for, and creation of, an open functional architecture for automated driving systems.

chapter 3 introduces some basic assumptions about the architecture, including coordinate systems, units of length, time, uncertainty, and other conventions.

The following chapter 4 gives a birds-eye overview of the proposed architecture.

chapter 5 up to chapter 12 then provide a deeper view into specific sections of the architecture, software modules, and interfaces. Each of these sections:

- Describes the mechanisms with which the software modules making up the architecture section interact, which module provides information to other modules, and what might happen if information does not arrive (or not in a timely fashion) or is wrong, outdated, or otherwise corrupted.
- Describes the interfaces used for interaction, providing operational data and metadata, callback mechanisms, etc. These are given in tabular and abstracted form because this document describes the reference architecture. Concrete implementations (such as EB robinos for ADTF) may deviate somewhat from the reference.
- Provides some comments on testing and verifying the correct operation of each section of the architecture and the modules it consists of.
- Provides some comments on functional and operational safety concerning the section of the architecture and its modules. This last paragraph of each section is, by necessity, sparse on specific requirements imposed by ISO 26262 or similar standards. These requirements can ultimately only be derived from a concrete function specification and broken down into software modules in a top-down fashion, while the open robinos specification moves the other way, providing building blocks bottom-up. Where ASIL requirements or similar are touched, they are only to be interpreted as specifications for elements out of

context. On the other hand, the paragraph is intended to be quite specific in some places for solutions of the problem of operational safety, which ISO 26262 does not address, for example providing suggestions for certain algorithmic redundancies which can help detect or even eliminate known weak spots of "gold-standard" algorithms employed in the industry today.

Finally, chapter 13 describes the license for this document.

## 1.2. What this document is (or aims to be), and what it is not

This document aims to provide a description of software building blocks for automated driving, and the mechanisms with which they interact together with some ideas on how to use these building blocks to build actual automated-driving applications. This combination of building blocks, mechanisms, and reference applications make up a reference architecture for automated driving functions. In this case, automated driving does not only mean highly automated (SAE level 3) vehicles, but rather anything from the most simple emergency brake and lane departure warning up to fully automated driving, and even providing links into the cloud.

The idea for this architecture, the need for it in the marketplace, and most of the mechanisms and building blocks that make up its parts, are born from more than a combined century of experience developing perception, sensor data fusion, control, and automation systems from robotics research to level-4 automated driving production systems.

Since today's in-vehicle electronic architectures are so incredibly diverse, the document specifically tackles the topic of functional software architecture, and does not enter the domain of mapping this software architecture to hardware, control units, or networks of these elements (except in some examples). The interface descriptions presented in this document are not data type specifications on programming language level (i.e. C struct). This means that all used types (int, vector, etc.) are not further determined or documented.

This document is also not complete and probably will not be for quite some time. There is no fully automated driving system on the market yet that could simply be documented. Therefore, while many of the elements are tried and tested in functional development, some are also prototypes that are known from experience to have worked before, but have not been implemented yet as a complete system. Some are also not possible to specify in an in-depth manner since they tie in too closely with vehicle specifics. It is our goal to work towards providing a specification that is complete (as far as possible), accompanied with a full reference and production implementation for all of the components and mechanisms.

In addition, while the document is currently created and published by Elektrobit Automotive GmbH, we do not think it is reasonable to expect (or even possible) for one corporation to single-handedly provide a full specification of an automated driving software solution that would fit all applications. Therefore, this document is a call for discussion, and for partners to discuss with, just as much as it is a specification. If you or the company you belong to is interested in joining the discussion, please do not hesitate to join@open-robinos.com.

# 2. Motivation for an open architecture: Reducing complexity

Today's driver assistance systems are – due to factors such as costs, scalability for various vehicle market segments and the space in the vehicle itself – usually divided into few "packages", which only carry out their assigned task. In this arrangement, it is often customary to bundle functionality close to the relevant sensors, which can save lots of space (package optimization). At the same time, the individual functions are independent of one another, which simplifies both selling separate units to the customer (combinatorial optimization) and employing various suppliers (vendor optimization). These factors give the automotive manufacturer an economic advantage. In addition, the individual functions can be tested independently of one another, which significantly simplifies this process, as there is little effort involved in integrating the functions with each other. In Landau notation, used in computer science to describe the complexity of algorithms, the complexity for the integration of n functions with this approach is `O(n)`.

This principle works exceptionally well as long as the individual systems can carry out their functions more or less independently and only serve to assist the driver. If the goal is to have an autonomous vehicle, these systems need to connect. A simplified example, using the two functions adaptive cruise control and emergency brake assist, shows how quickly the level of complexity rises when various functions have to communicate with one another (see figure 2.1).

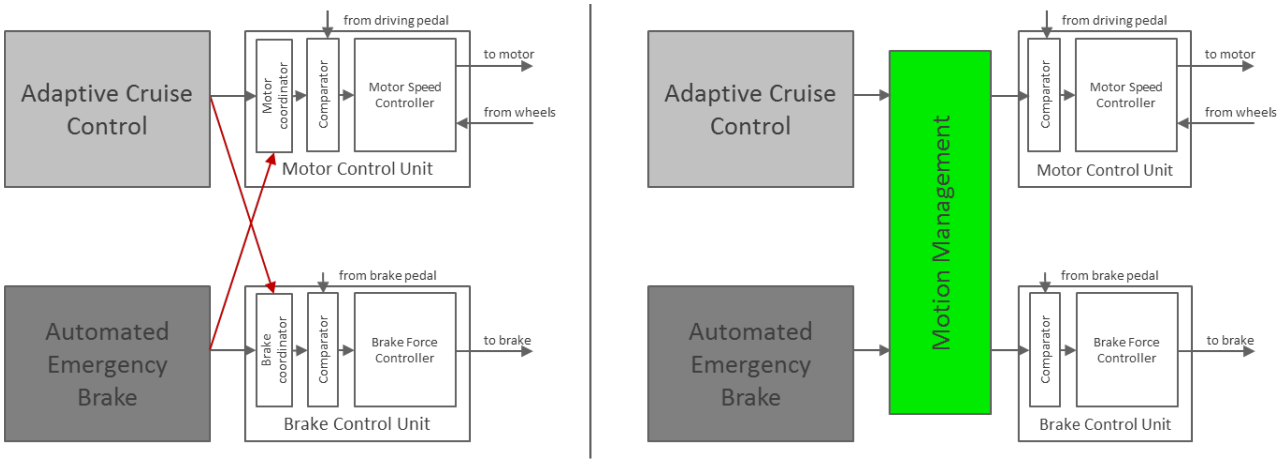


Figure 2.1. Coordination between braking and engine management module (source: Elektrobit)

In this case, the control units need a coordinating module to ensure that the functions do not conflict with one another. This is not collaboration yet, but a very rudimentary form of coordination that only takes care of possible conflicts. If this system structure enables the functions to be available in various combinations (combinatorial optimization), then the time and effort needed to test and certify all possible combinations grows. The complexity for k systems is expressed by `O(2k-1)`. The use of modules made by various suppliers, as is usually the case, often complicates the process ever further.

To manage this complexity, the current trend in the automobile industry is to develop central processing units for one particular activity (e.g. driver assistance). The sensor data is sent to one centralized processing unit and then, as required, made available to the functions. The advantage: the car manufacturer achieves a separation of hardware and software. This means a growth in the possibilities for vendor optimization, but it also means a decrease in scalability because the central processing unit has to be integrated into the vehicle regardless of the chosen functional configuration. Any scalability must now be handled within the ECU itself, e.g. not populating certain parts of the circuit board. Complexity also increases because, in addition to the integration of the functions with each other, there also has to be an integration into the central processing unit, e.g. as software components running in an AUTOSAR context.

## 2.1. Dealing with complexity

Software development has come up with various mechanisms to deal with highly complex architectures. Three of them are especially promising with respect to their use for automotive applications.

One of these is abstraction. When a software component has to respond to requests from too many partners, it can be helpful to insert a coordinator module in between. In a vehicle this could, for example, be a “motion management” that controls the access to the motor and brake control unit (see figure 2.1). This module receives acceleration or braking orders from the driver assistance systems, but decides which orders are to be carried out and how (for example through brakes applied to the wheels or by using the engine brake) according to its own internal logic.

This new module reduces complexity enormously: the coordinator modules for engine and brakes are no longer necessary because motion management now carries out their tasks. Above all, the functions and the actuators only have to use one communications channel, that being the one to the motion management. Now the complexity derived from the number of functions no longer develops exponentially ($O(2k-1)$), but linearly ($O(k)$). That means that large numbers of systems, such as are needed for autonomous driving, can work easily together.

Object oriented system design has given us the concept of polymorphy: software components that carry out similar functions, even if their internal structures are significantly different, should communicate with the outside world via the same interface. These so-called polymorphisms can also be of help in automotive software architectures because the communications paths offer a great potential for the reduction of complexity. The developer of a software component that has to send various “partners” similar instructions would not have to write a new code for every kind of instruction – a standardized interface can be used for communication with various other components. This reduces the complexity from $O(k)$ to $O(1)$.

Standardization takes us another step further in complexity reduction. When calling for tenders, the OEM can reference standards, and thereby increasing the level of understanding between the seller and the buyer of a software module. Functions and modules of various manufacturers can simply be exchanged. That reduces development costs and risk. The suppliers and component developers profit, too: with a standardized interface, a software module could be sold to several car manufacturers. Test and tool suppliers can concentrate on the

contents of the value creation and not so much on individual interface adaptations. In automotive software, AUTOSAR and ADASIS are among the most important standards for software systems, just as the upcoming SENSORIS standard for cloud-based map enrichment, a development that is just beginning.

# 3. Basic specifications

All data elements have units according to the SI system, unless otherwise specified.

The DIN ISO 8855 specification provides the basis for all Cartesian coordinate system elements.

## 3.1. Common data types

For simplification, this section specifies a set of data structures that capture measurements, i.e. scalar values or vectors together with specifications of uncertainty derived when obtaining them. These data types will be used by many other specifications later on.

### 3.1.1. Uncertain measurements in 1D (Meas1D)

Defines a scalar with standard deviation and a validity flag – further used as standard data type for 1D measurements.

Table 3.1. Measurements in 1D

| Name | Data type | Unit | Description |
|---|---|---|---|
| Measurement 1D | float | m | |
| Standard Deviation 1D | float | m | |
| Is Valid | bool | True, False | Shows that the linked measurement is valid |

### 3.1.2. Uncertain measurements in 2D (Meas2D)

Defines a 2-vector for measurements in the [x,y] plane with standard deviation ellipsis and a validity flag – further used as standard data type for [x,y] measurements.

The variances/covariances are specified as ellipsis parameters rather than the customary 2x2 covariance matrices to eliminate the over-determination inherent in that representation.

Table 3.2. Measurements in 2D

| Name | Data type | Unit | Description |
|---|---|---|---|
| Measurement 2D [x,y] | float[2] | m, m | Given as vector for values in respect to x, y |

| Name | Data type | Unit | Description |
|---|---|---|---|
| Standard Deviation 2D [major, minor] | float[3] | m, m, deg | Standard deviation given as ellipsis parameters for ground plane(x-y-plane) as major, minor axis length[m] and orientation [deg]. The orientation refers to 0 along the x-axis and increases counter-clockwise. |
| Is Valid | bool | True, False | Shows that the linked vector is valid |

### 3.1.3. Uncertain measurements in 3D (Meas3D)

Defines a 3-vector for measurements in the [x,y,z] plane. The uncertainty is given as standard deviation ellipsis for the (co)variance of x and y; z is specified as a scalar / standard deviation uncorrelated to x/y for simplicity. This structure is further used as standard data type for [x,y,z] measurements.

Table 3.3. Measurements in 3D

| Name | Data type | Unit | Description |
|---|---|---|---|
| Measurement 3D [x,y,z] | float[3] | m, m, m | Given as vector for values in respect to x, y, and z |
| Standard Deviation 3D [major, minor, angle, vertical] | float[4] | m, m, deg, m | Standard deviation given as ellipse parameters for ground plane(x-y-plane) as major, minor axis length[m] and orientation [deg]. The orientation refers to 0 along the x-axis and increases counter-clockwise. Further the standard deviation along the height axis is given as single value[m]. |
| Is Valid | bool | True, False | Shows that the linked vector is valid |

## 3.2. Time

Absolute time is defined in microseconds, according to international Coordinated Universal Time (UTC), starting at January 01, 1970, as a 64-bit signed integer (as customary on modern operating systems). Communication to the outside of the vehicle (vehicle-to-infrastructure, as well as GPS²), uses absolute time.

A vehicle moving at 200km/h moves 55.6m/s or ~6cm/ms. Counting time in milliseconds is insufficient for high-dynamic situations as the base of fusion and control systems as it introduces an error in the range of several centimeters. Therefore, a second relative time counter is introduced.

In most regular in-vehicle communications, only the relative timestamp is used, In most communications with the outside world, only the absolute timestamp is used. In some specific cases, both may be used in conjunction.

Table 3.4. Time

| Name | Data type | Unit | Description |
|---|---|---|---|
| Absolute Time | int64 | µs | Time according to international Coordinated Universal Time (UTC) since 01.01.1970 |
| Relative Time | int64 | µs | Relative based on a common in-vehicle clock |

## 3.3. Coordinate systems

The following chapters introduce coordinate systems that are used later on (see figure 3.1). The coordinate systems are defined from global to local, and comprised of:

- Geographic coordinate system (GCS)
- Local coordinate system (LCS) and local area
- Construction coordinate system
- Vehicle coordinate system
- Sensor coordinate system
- Anchor coordinate system

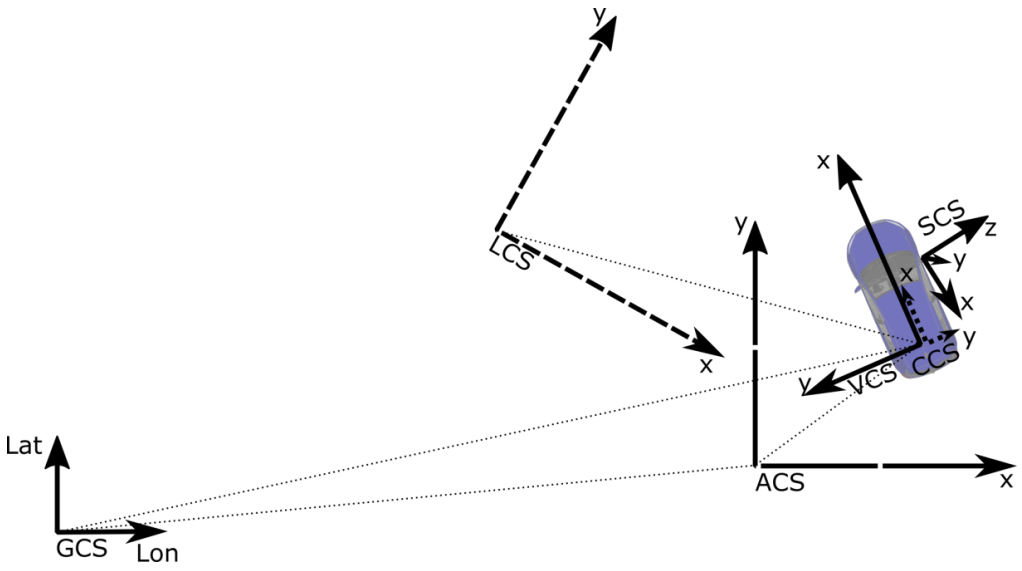


Figure 3.1. Coordinate systems overview with GCS (Geographic-), LCS (Local-), CCS (Construction-), VCS (Vehicle-), SCS (Sensor-) and ACS (Anchor coordinate system)

### 3.3.1. Geographic coordinate system (GCS)

Geographic coordinates describe a position on the Earth's surface. The open robinos specification refers to two standards, the Universal Transverse Mercator (UTM) as well as the World Geodetic System 1984 (WGS84).

Table 3.5. WGS84

| Name | Data type | Unit | Description |
|---|---|---|---|
| Position [lat, lon, alt] | float [3] | deg, deg, m | Given as vector for values in respect to latitude[deg], longitude[deg] and altitude[m] |

Table 3.6. UTM

| Name | Data type | Unit | Description |
|---|---|---|---|
| Position [x, y, z] | float[3] | m, m, m | Given as vector for values in respect to x, y, z and the related UTM zone ID |
| Zone ID | int | | The Earth's surface is split into 6-degree wide longitudinal bands. Each band is based on a cylindrical approximation of the surface. These stripes start from 180-degree western longitude with UTM ID 1 and go towards 180-degree eastern longitude with UTM ID 60. e.g. Germany is mainly located within zone 32. |

### 3.3.2. Local coordinate system (LCS) and local area

A local coordinate system (LCS) is an arbitrarily set Cartesian coordinate system. It sets the origin for the spatial relative described motion of the vehicle and other parts of the environmental model. It can be described in the geographic coordinate system (GCS) in order to link the cars movement globally. The relation between GCS and LCS can change over time, if a reset of the LCS is necessary.

A local area is defined as a square area around the local coordinate system origin and covers an area of interest for the car. The size of the area is defined by giving a maximal distance to the center. This distance is used to build the surface of the area.

Table 3.7. Local coordinate system

| Name | Data type | Unit | Description |
|---|---|---|---|
| Origin in GCS | float[3] | WGS or UTM | Given as a vector and states the relation of LCS to the global coordinate system. See section 3.3.1 |

| Name | Data type | Unit | Description |
|---|---|---|---|
| Yaw Angle | float | deg | z axis is aligned in bot coordinates (GCS and LCS), therefore only a yaw angle is defined |
| Local area - maximal distance | float | m | Sets a square area aligned with the [x,y] axis |

### 3.3.3. Construction coordinate system (CCS)

This coordinate system is the base to describe all parts of a car. It varies among brands and platforms and is most likely not linked to a specific part of the vehicle. It was set at the beginning of construction (of a much older version) of the car and is needed for calibrating and as common origin for every sensor. Furthermore the sensor as well as the vehicle coordinate system can be described in this system in order to respect production as well as fusion aspects.

Table 3.8. Construction coordinate system

| Name | Data type | Unit | Description |
|---|---|---|---|
| Relation to other coordinate systems [x,y,z,r,p,y] | float[6] | m, m, m, rad, rad, rad | Given as vector, describing the translation and rotation to other non-geographical coordinate systems |

### 3.3.4. Vehicle coordinate system (VCS)

figure 3.2 shows the vehicle with its coordinate system. This coordinate system is used to describe the environment of a car in respect to itself. The origin is located on the road surface close to the middle of the rear axis. In automotive applications, the x-axis points in the primary direction of motion and the z-axis points to the sky. The yaw, pitch and roll axis describe the orientation of the chassis. All detected objects, lanes, etc. from the sensors are transformed to vehicle coordinates to be comparable.

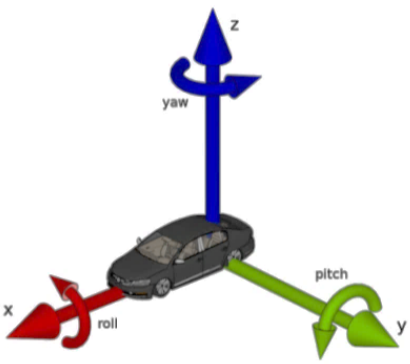


Figure 3.2. Vehicle coordinate system

Table 3.9. Vehicle coordinate system

| Name | Data type | Unit | Description |
|---|---|---|---|
| Relation to other coordinate systems [x,y,z,r,p,y] | float[6] | m, m, m, rad, rad, rad | Given as vector, describing the translation and rotation to other non-geographical coordinate systems |

## 3.3.5. Sensor coordinate system

Each sensor uses a specific coordinate system. Depending on the sensor, the delivered data can be 3D point clouds, 2D images, depth images, etc.

As the system assumes that detected objects are relative to the sensor, extrinsic parameters (position and orientation according to the vehicle they are mounted on) of the sensor shall be known. This allows transformation to vehicle or other coordinates.

Table 3.10. Sensor coordinate system

| Name | Data type | Unit | Description |
|---|---|---|---|
| Relation to other coordinate systems [x,y,z,r,p,y] | float[6] | m, m, m, rad, rad, rad | Given as vector, describing the translation and rotation to other non-geographical coordinate systems |

## 3.3.6. Anchor coordinate system (ACS)

The anchor coordinate system is a discrete Cartesian coordinate system to the origin of a grid. It is arbitrarily set in either a georeferenced or locally referenced coordinate system.

Table 3.11. Anchor coordinate system

| Name | Data type | Unit | Description |
|---|---|---|---|
| Origin in GCS | float | | States the relation of ACS to the global coordinate system. See section 3.3.1 |
| Relation to other coordinate systems [x,y,z,r,p,y] | float[6] | m, m, m, rad, rad, rad | Given as vector, describing the translation and rotation to other non-geographical coordinate systems |
| Discrete grid coordinates [x,y,z] | int, int, int | | Coordinates of a grid cell by counting the cell beginning at the anchor, z axis only for 3D grids with voxels |

# 4. Architecture overview

The top-level architecture is depicted in figure 4.1. The architecture follows the path of data processing from the set of sensors on the left to the set of actuators on the right. There are five big blocks surrounded by abstraction layers in between: Sensor Data Fusion, Situative Behavior Arbitration, Motion Management, HMI Management, and Safety Management contain functional algorithms for automated driving. The Vehicle Abstraction for Sensors as well as for Actuators and the Functions Specific Views ensure an easy adaptation to specific OEM E/E design decisions as well as to the addressed set of behaviors.

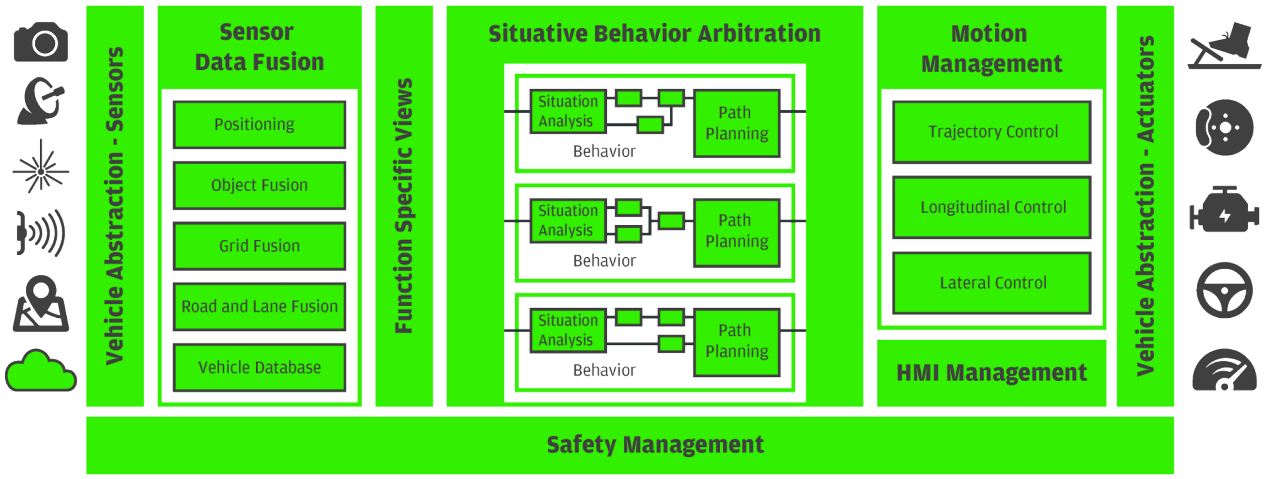


Figure 4.1. Architecture Overview

figure 4.1 shows a functional architecture. This architecture, and this document, only describes what modules must be present, their function, and the data they use for communication.

More details, for instance about the communication protocols or the run time environment to be used, are out of scope for this document. These additional details strictly depend on the specific system and context in which open robinos is implemented. For example, some legacy systems are still using a classic AUTOSAR, whereas new systems may take advantage of the new Adaptive AUTOSAR features. In the prototyping phase, everything is running on runtime environments such as ADTF. Some other systems use an all-on-one-device approach, other prefer to distribute the computation across several ECUs, as shown in figure 4.2.

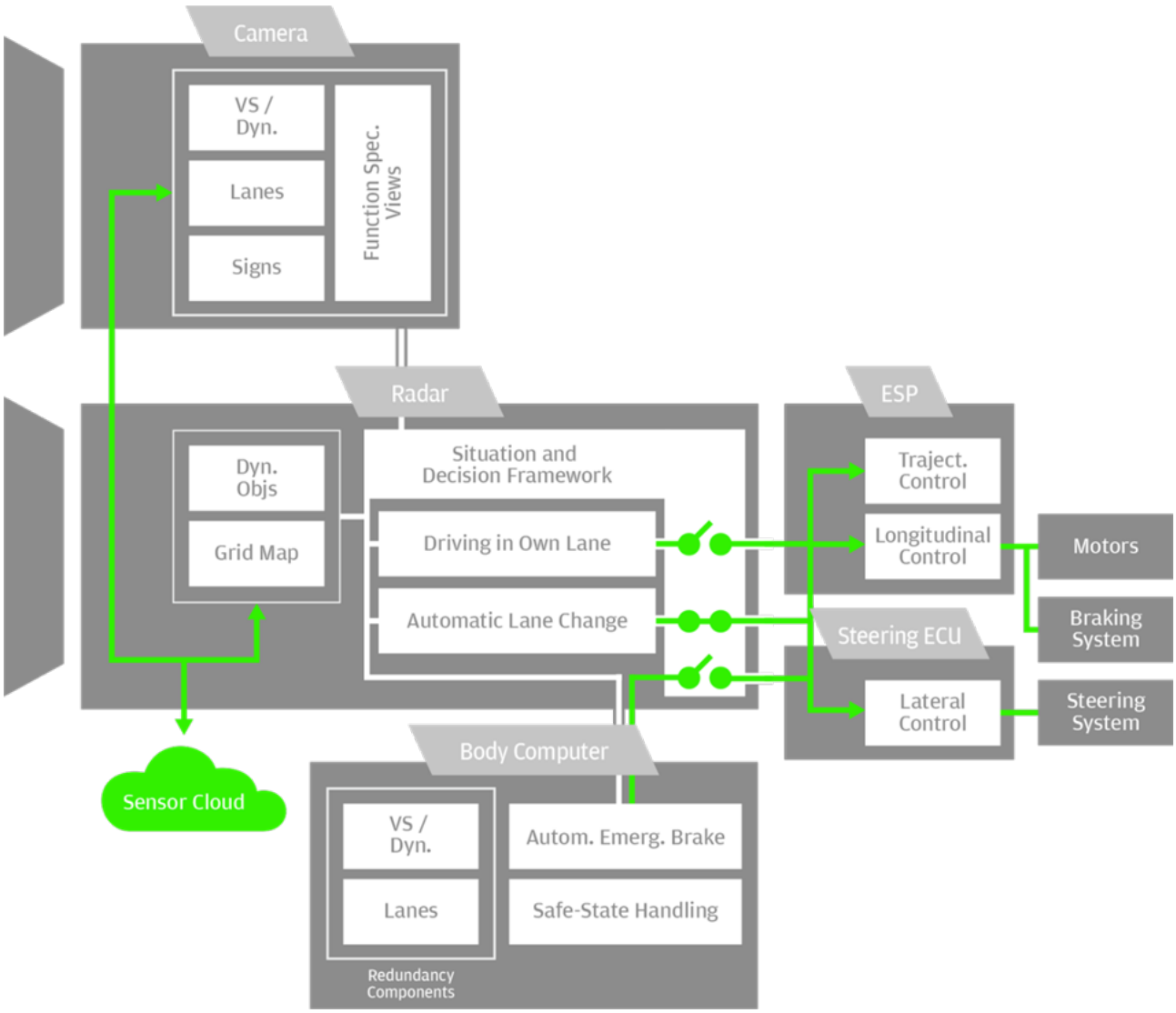


Figure 4.2. Computation across several ECUs

# 5. Vehicle abstraction - Sensors

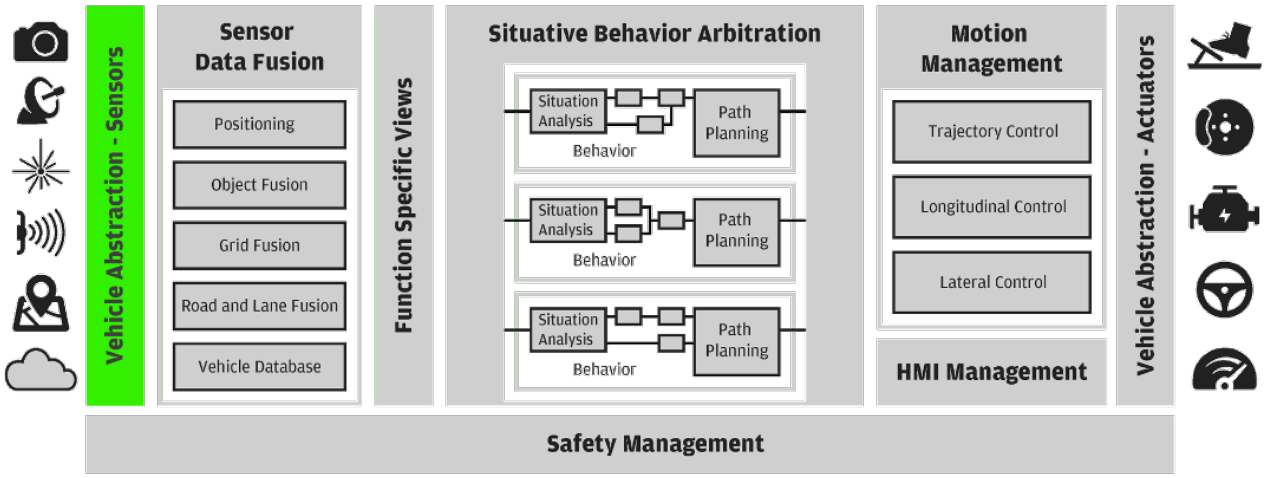


Figure 5.1. Vehicle Abstraction – Sensors as part of the Architecture overview

The Vehicle Abstraction – Sensors block on the left side addresses the task of transforming individual sensor interfaces into the open robinos specification. On one hand, these are simple transformations, such as converting acceleration from the unit [g] to [m/s²]. On the other hand, these are abstractions on sensor data based on a type clustering.

The initial clustering separates sensors into three groups:

- Interoceptive sensors: These sensors are mounted on or in the car and focus on the vehicle itself. They measure the dynamic states of the car and other internal data. Typical examples of these sensors are gyros, accelerometers, steering angle sensors, as well as sensors for wiper activity, turn indicators, and so on.
- Exteroceptive sensors: These sensors are mounted on or in the car, focus on the vehicle´s surroundings, and measure the car´s environment. Typical examples of these sensors are radars, lasers, ultrasonic sensors, cameras, etc.
- Meta sensors: These elements can be seen as a sensor, but are usually a source of data derived from measurements of other sensors. Typical examples of these sensors are cloud data, navigation maps, Car2X, etc.

This chapter captures a first selection of the necessary set of specifications and we invite the whole automotive industry to extend this as well as the other chapters. Please join by contacting us via join@open-robinos.com.

## 5.1. Standard sensor information

This information usually applies to many of the sensors.

Table 5.1. Standard sensor information

| Name | Data type | Unit | Description |
|---|---|---|---|
| Mounting position [x, y, z, roll, pitch, yaw] | float[6] | m, m, m, rad, rad, rad | Given as vector for translation and rotation of the SCS in respect to the VCS, see section 3.3 |
| Sensor ID | int | | Describes the sensor and ensure a conclusion on the actual source |

# 5.2. Interfaces for interoceptive sensors

The following specifications apply to sensors that are mounted in the car, focusing on state of the car itself.

## 5.2.1. Dynamic state vector

The following data is used to derive the dynamic state vector.

Table 5.2. Dynamic state vector

| Name | Data type | Unit | Description |
|---|---|---|---|
| Vehicle State | | | Refer to section 6.1 – some sensors directly supply data about some of the elements described in this chapter |
| Wheel Ticks | int[n] | | Given as vector for n wheels |
| Steering Wheel Angle | float | deg | Given as single value. Not necessarily linear related to actual steering angle |
| Steering Wheel Rate | float | deg/s | Given as single value |
| Magnetic Flux | float[3] | T | Given as vector of x, y, and z |
| Barometric Pressure | float | kPa | Given as single value |

## 5.2.2. Car component states

The following data is used to record the states of the various car components.

Table 5.3. Car component states

| Name | Data type | Unit | Description |
|---|---|---|---|
| Motor State | enum | | STATE_UNDEFINED, STATE_STANDBY, STATE_NOTREADY, STATE_READY |

| Name | Data type | Unit | Description |
|---|---|---|---|
| Gear Direction | enum | | GEAR_DIRECTION_FORWARD, GEAR_-DIRECTION_BACKWARD, GEAR_DIREC-TION_NEUTRAL, GEAR_DIRECTION_UN-KNOWN |
| Steering Wheel Angle | float | rad | |
| Wheel Angle | float[n] | rad | Given as vector for n wheels |

# 5.3. Interfaces for exteroceptive sensors

The following section describes two basic possibilities to receive sensor data: a scan based approach and an object list approach. The open robinos specification expects intelligent sensors and the given specifications can be seen as a starting point and frame to describe the extensive capabilities of exteroceptive sensors among the industry. Please join by contacting us via join@open-robinos.com.

## 5.3.1. Scan-based exteroceptive sensors

The following abstraction assumes that each sensor is intended to describe measurements in spatial relation to its mounting position. Therefore, the scan point itself is an abstract data representing this spatial relation. Furthermore, every scan point carries further attributes that vary among the type of sensor.

Table 5.4. Scan-based exteroceptive sensors

| Name | Data type | Unit | Description |
|---|---|---|---|
| Scan Point [x,y,z] or [dist, angle] | float[3] or float[2] | m, m, m, or m, rad | Given as vector for measurements in the sensor coordinate system. Depending on the sensor technology, this will be a set of [x,y] or [x,y,z] coordinates (Lidar, Camera) or [dist, angle] tuples for an ultrasonic sensor |
| Scan Point Variance [x,y,z] or [dist, angle] | float[3] or float[2] | $m^2$, $m^2$, $m^2$, or $m^2$, $rad^2$ | Given as vector for measurements in the sensor coordinate system |
| Scan Point Attributes | | | Classifies a Scan Point as dirt or ground as simple example |

## 5.3.2. Object based exteroceptive sensors

The objects are described within the section for the environment model. For specifications, see section 6.2.

# 5.4. Interfaces for meta-sensors

The open robinos concept uses the existing specifications for meta-sensors as shown in the following table.

Table 5.5. Interfaces for meta-sensors

| Meta-sensor | Linked specification | Description |
|---|---|---|
| Map | NDS | http://www.nds-association.org/ Navigation Data Standard |
| electronic horizon | ADASIS Forum | http://adasis.org/ Advanced map-enhanced driver assistance systems |

In addition to the input from the meta-sensors, there are specifications for enriching such databases. For more information, refer to chapter 11.

## 5.4.1. Electronic Horizon

The ADASIS Forum is currently specifying the version 3 of its standard, covering a detailing level of roads, lanes, borders, and intersections (http://adasis.org/). The ADASIS standard itself is a semi open standard. Non-members of the forum get access to it with a time delay.

Modern navigation systems provide to ADAS applications, typically on-board of other control units (ECU), predictive and optimized data of the route in front of the vehicle as well as attributes along its route. This representation is called the ADAS electronic horizon (eHo).

From an architectural point of view, the main entities of a map-based ADAS application are:

- ADASIS Electronic Horizon Provider (EHP), which generates ADAS Horizon and messages which will be sent to ADAS applications.
- Bus (CAN for ADASISv2 and Ethernet for ADASISv3), transport layer. Vehicle's bus system to broad eHo data on car.
- ADASIS Electronic Horizon Reconstructor (EHR), which will receive, parses all incoming ADASIS messages, reassemble them into a complete horizon representation which will be provided outside.

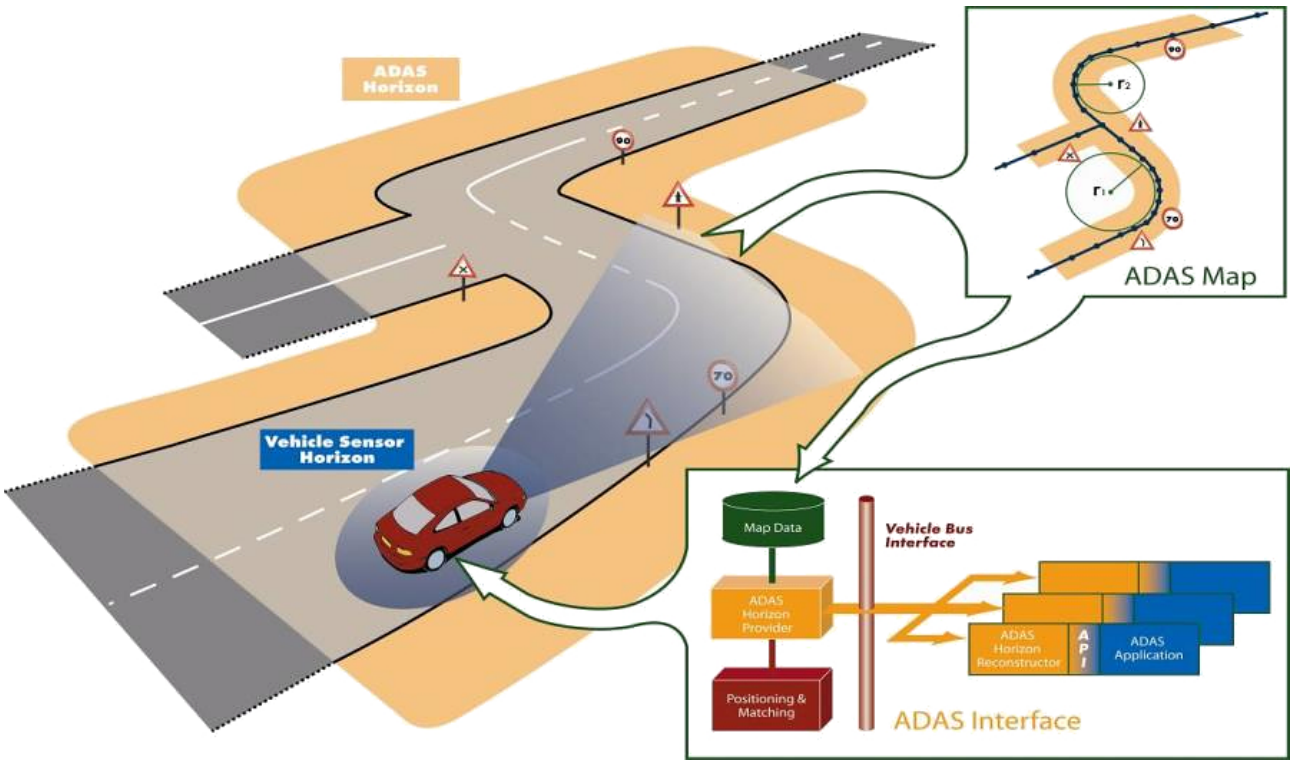


Figure 5.2. Electronic horizon - overview

Both the provider and the reconstructor are out of scope in this document. Here we describe the data that comes out of the reconstructor and that can be used as input for the road and lane fusion model. Some data requires a search functionality that explores the ADASIS tree, but this shall be provided by the electronic horizon.

Most of the definitions and constants used here come from the ADASISv3 specification. We refer the reader to the ADASISv3 specification for more details.

Table 5.6. Position

| Name | Data type | Unit | Description |
|---|---|---|---|
| Path ID | int | | ID number of the path where the vehicle is located |
| Offset | int | | Offset from the beginning of the path |
| Accuracy | float | | |
| Lateral offset | int | | |
| Position Status | enum | | It could be “off road”, “off map” or “on map” |
| Speed | float | | Vehicle Speed |
| Relative Heading | float | | |
| Probability | float | | Probability of the correctness of positioning |
| Lane ID | int | | Lane ID where the vehicle is located |

| Name | Data type | Unit | Description |
|---|---|---|---|
| Preferred Path | int | | |
| Position Age | int64 | µs | |
| Timestamp | int64 | µs | |

Table 5.7. Stub

| Name | Data type | Unit | Description |
|---|---|---|---|
| Path ID | int | | ID of the path containing the stub |
| Offset | int | | Offset where the stub is located |
| Subpath ID | int | | ID of the new path |
| Instance ID | int | | |
| Angle | float | | |
| Is Complex Intersection | bool | | |
| Right Of Way | enum | | One of the values “Must Yield” right of way, “Has Right Of Way” or “Unknown” |

Table 5.8. Profile

| Name | Data type | Unit | Description |
|---|---|---|---|
| Instance ID | int | | |
| Path ID | int | | ID of the path the profile refers to |
| Confidence | float | | |
| Lane number | int | | |
| Offset | int | | Start offset |
| End Offset | int | | End Offset |
| Profile Type | enum | | One of the profile types defined by ADASISv3 |
| Value | blob | | Value of the profile from “start offset” to “end offset”. The actual data type depends on the profile type |

This record is used to describe one control point of the profile for a single attribute on a single path.

Table 5.9. Global data

| Name | Data type | Unit | Description |
|---|---|---|---|
| Country Code | enum | | |
| Region Code | enum | | The region code semantic depends on the country code |

| Name | Data type | Unit | Description |
|---|---|---|---|
| Map Provider | | | |
| Driving Side | enum | | Left/Right |
| Unit System | enum | | SI/Imperial |
| Version Protocol | | | |
| Version Hardware | | | |
| Version Map | | | |

Table 5.10. Curve

| Name | Data type | Unit | Description |
|---|---|---|---|
| Curve Type | enum | | Interpolation method between points: Poly-line/Bezier |
| Interpolation Type | enum | | Spot/Step/Linear |
| Number of Points | int | | |
| Points | WGS84[] | | See section 3.3.1 |

Table 5.11. Lane connectivity

| Name | Data type | Unit | Description |
|---|---|---|---|
| Initial Path ID | int | | |
| New Path ID | int | | |
| Initial Lane Number | int | | |
| New Lane Number | int | | |

Table 5.12. Lane descriptor

| Name | Data type | Unit | Description |
|---|---|---|---|
| Lane Direction | enum | | None/Along driving direction/Against driving direction/Both |
| Transition | enum | | None/Opening/Closing/Merging/Splitting |
| Lane Type | enum | | See ADASIS specification |
| Center Object Type | enum | | Centerline/Lane Marking/Guard Rail/Fence/Curb |
| Left Object Type | enum | | Centerline/Lane Marking/Guard Rail/Fence/Curb |
| Right Object Type | enum | | Centerline/Lane Marking/Guard Rail/Fence/Curb |
| Center Marking | enum | | See ADASIS Specification |

| Name | Data type | Unit | Description |
|---|---|---|---|
| Left Marking | enum | | See ADASIS Specification |
| Right Marking | enum | | See ADASIS Specification |
| Center Geometry | curve | | |
| Left Geometry | curve | | |
| Right Geometry | curve | | |
| Lane Number | int | | |
| Current Model | Lane Connectivity | | It can be empty |
| Next Model | Lane Connectivity | | It can be empty |

# 6. Sensor data fusion

The Sensor Data Fusion block, depicted in figure 6.1, addresses the task of computing an environmental model as well as a model of the car´s states and capabilities. Among the industry, this block has been given much focus over the last years compared to other blocks of the entire architecture. Nevertheless, there is nothing considered to be “the environmental model” and no open standard for a collection of interfaces as there is for maps or the electronic horizon protocols, for example (NDS, ADASIS). This chapter captures a first selection of the necessary set of specifications and we invite the whole automotive industry to extend this as well as the other chapters. Please join by contacting us via join@open-robinos.com.

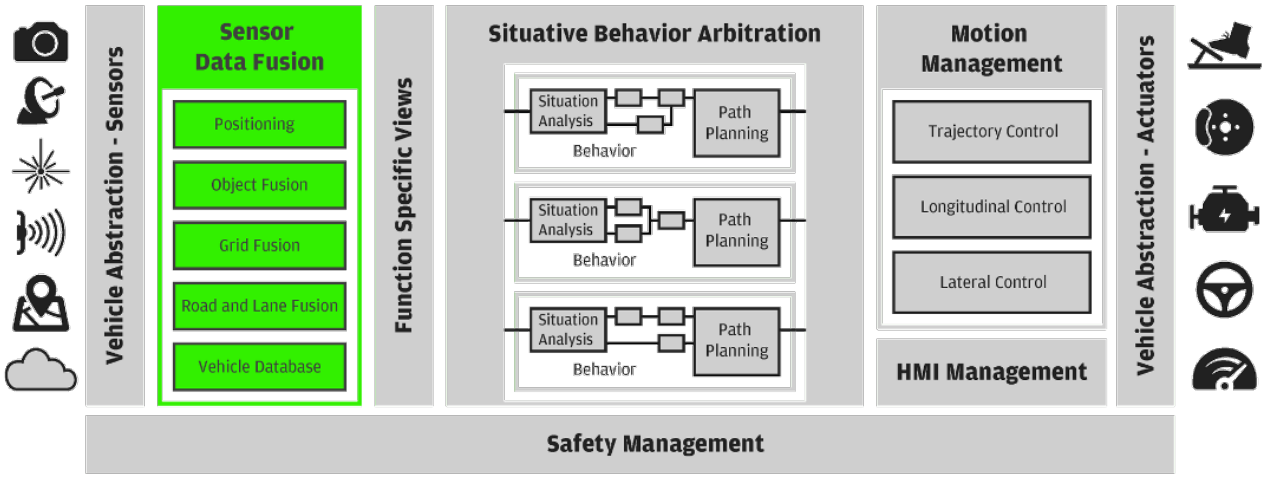


Figure 6.1. Sensor Data Fusion as part of the Architecture overview

The list below documents the SW components that Sensor Data Fusion is composed of and the environment model aspects they are responsible for:

- Positioning
  - Vehicle position
  - Vehicle dynamics
- Objects
  - Objects in the environment
  - Convoy tracks
- Grid fusion
  - Occupancy grid
- Road and lane fusion
  - Roads and lanes
- Vehicle database

- Vehicle state (excluding position and dynamics)

# 6.1. Positioning

Vehicle Dynamics and Position describes the dynamic states of the car as well as its position in global and local coordinate systems. The dynamic states of parts of the vehicle may differ, since the chassis, the body, the wheels, etc. have their own dynamics. The following specification states the vehicle dynamic only and focuses on a description of the chassis only.

The block HD Positioning in the architecture shown in figure 6.1 is responsible for deriving the (single source of truth) representation of vehicle dynamics and position. Depending on the configuration, the HD Positioning component can deliver a vehicle position in global and/or local coordinate system. The relation between local and global coordinate systems is not fixed over time and might change during runtime.

Table 6.1. Local coordinate system

| Name | Data type | Unit | Description |
|---|---|---|---|
| Timestamp | int64, int64 | µs, µs | See section 3.2 for timing definitions |
| Position | Meas3D | m | Given as vector for values in respect to x, y, and z with uncertainty – see section 3.1.3 |
| Velocity | Meas3D | m/s | Given as vector for values in respect to x, y, and z with uncertainty – see section 3.1.3 |
| Acceleration | Meas3D | m/s² | Given as vector for values in respect to x, y, and z with uncertainty – see section 3.1.3 |
| Orientation [roll, pitch, yaw] | float[3] | deg, deg, deg | Given as vector for values in respect of roll, pitch, and yaw |
| Standard Deviation Orientation [roll, pitch, yaw] | float[3] | deg, deg, deg | Standard deviations given as vector for values in respect of roll, pitch, and yaw |
| Angular Velocity [roll, pitch, yaw] | float[3] | deg/s, deg/s, deg/s | Given as vector for angular velocity values in respect of roll, pitch, and yaw |
| Standard Deviation Angular Velocity [roll, pitch, yaw] | float[3] | deg/s, deg/s, deg/s | Standard deviations given as vector for angular velocity values in respect of roll, pitch, and yaw |

Table 6.2. Global coordinate system

| Name | Data type | Unit | Description |
|---|---|---|---|
| Timestamp | int64, int64 | µs, µs | See section 3.2 for timing definitions |

| Name | Data type | Unit | Description |
|---|---|---|---|
| Position [lat, lon, alt] | Meas3D | deg, deg, m | Given as vector for values in respect to latitude[deg], longitude[deg], and altitude[m] with uncertainty – see section 3.1.3 |
| Velocity [e, n, u] | Meas3D | m/s | Given as vector for values in respect to east, north, and up with uncertainty – see section 3.1.3 |
| Acceleration [e, n, u] | Meas3D | m/s² | Given as vector for values in respect to east, north, and up with uncertainty – see section 3.1.3 |
| Orientation [roll, pitch, yaw] | float[3] | deg, deg, deg | Given as vector for values in respect of roll, pitch, and yaw |
| Standard Deviation Orientation [roll, pitch, yaw] | float[3] | deg, deg, deg | Standard deviations given as vector for values in respect of roll, pitch, and yaw |
| Angular Velocity [roll, pitch, yaw] | float[3] | deg/s, deg/s, deg/s | Given as vector for angular velocity values in respect of roll, pitch, and yaw |
| Standard Deviation Angular Velocity [roll, pitch, yaw] | float[3] | deg/s, deg/s, deg/s | Standard deviations given as vector for angular velocity values in respect of roll, pitch, and yaw |

## 6.2. Object fusion

The term object describes all items of the environment that can be thought of as independent from each other, moving according to certain rules (or standing still), and which can be represented in a simplified way as a box with dynamic states and further attributes. Objects are given as a list of such and focus on modeling the dynamic environment. The figure 6.2 gives an impression of dynamic objects represented in an object list based on a radar sensor. Examples of objects are vehicles, pedestrians, cyclists, or road signs. Things that are not well suited for description as objects are roadside boundaries or curbs, because they tend to be curved at their beginning and tend not to be observable, so to describe them with boxes would lose significant data.

The Object Fusion block in the architecture shown in figure 6.1 is responsible for deriving the (single source of truth) representation of objects.

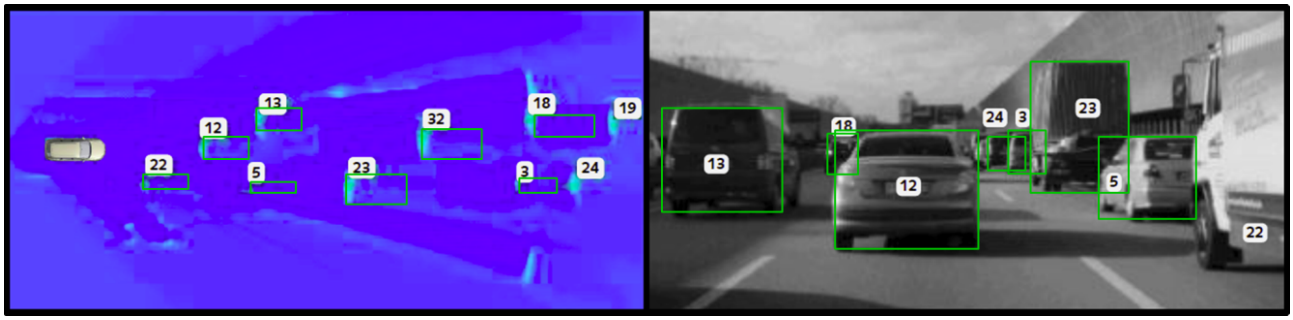


Figure 6.2. Dynamic objects based on a radar data sensor function (source: M. Reichel, Situationsanalyse für fortschrittliche Fahrerassistenzsysteme, Dissertation, Braunschweig, 2013)

Since at any given point there are probably many objects around the vehicle, the object specification is given in two parts: objects, and an object list.

## 6.2.1. Object IDs

The identifying element of an object is its (numeric) ID. Object fusion must do what it can to make this ID remain constant for as long as possible. This means also not re-using the object ID if not absolutely necessary, and leaving as much time as possible between two objects with the same ID occurring in the data stream. An object's ID changing or an object leaving a sensor and a different object with the same ID re-entering some cycles later cause many errors in development.

That being said, in reality it is not always possible to prevent ID re-use. Therefore, a Boolean field is specified which says that this is a "new object", which must be set in the very first cycle that an object appears in and re-set in all others. If a function that uses an object list sees the "new object" flag set to TRUE for an object for whose ID it still stores any data, it must clear that data immediately.

## 6.2.2. Object Classification, and Motion Models

An object is assigned a certain class, e.g. vehicle, pedestrian, or static object. This is important since objects with a certain class tend to move in a certain way (for example, a pedestrian can jump to the side, while a vehicle cannot). Therefore, an object's class contains important information about its dynamics. Classification can be obtained directly by certain sensors (e.g. cameras, LIDARs), or derived implicitly by observing the object's motion for a while, although the former is by far preferable.

The motion model directly implies certain properties about the object to be meaningful or meaningless. For example, a vehicle will have a certain longitudinal velocity, yaw angle, and yaw rate, but usually no lateral velocity; to describe a pedestrian's yaw rate is certainly possible but poses little constraint on the pedestrian's range of motion, but the pedestrian may have lateral velocity instead.

## 6.2.3. Object reference point

As convention, the output point of an object (its x/y location) is always the center of the object's rear end, because in typical ACC or AEB functions, it provides the distance to the point that the ego vehicle is most likely to collide with (0 in figure 6.3).

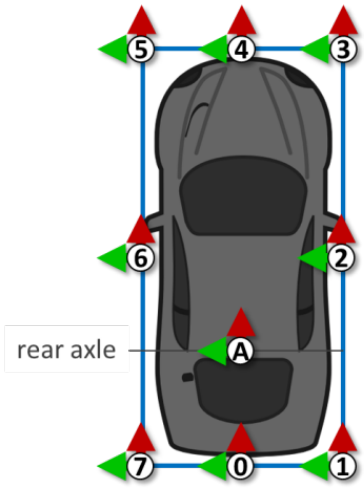


Figure 6.3. Coordinate points on an Ego vehicle

**NOTE**

**Tracking an object's motion**

Internally, an object fusion component will most likely not use that point to track the object's motion, but rather the point labeled "A", which is the center of the object's rear axle. For most vehicles, this is the point that it will turn about when driving a curve. It is probably not directly observable so it must be conjectured or derived from the object's actual motion; however a conjectured turning point is still better than assuming the vehicle turns about 0, which it certainly does not.

When observing an object, it may happen that point 0 is not directly visible (for example, when the object arrives at the neighboring lane from behind, so that we see it from the front). Therefore, a sensor providing an object observation should provide the point that it is most likely to observe as the object's output position, and specify this, as a number of 0 to 7.

For simplification, and since most observed objects tend to stand or move on the ground, the z (up) coordinate is neglected in these considerations.

## 6.2.4. Object and Object List Format

Table 6.3. Object format

| Name | Data type | Unit | Description |
|---|---|---|---|
| ID | int | | Object ID, see section 6.2.1 |
| New object | enum | | Describes if the object appears for the first time. Values: True, false or unknown. See section 6.2.1 |
| Reference point | enum | | Position, velocity and acceleration refer to this point at the object, see section 6.2.3 |
| Position | Meas3D | m | Relative to the vehicle coordinate system, section 3.3.3 and with uncertainty – see section 3.1.3 |
| Yaw angle | Meas1D | rad | Given in the vehicle coordinate system, section 3.3.3 with uncertainty (see section 3.1.1) |
| Velocity | Meas3D | m/s | Relative to the ego, given in the vehicle coordinate system, (see chapter section 6.2.3) and with uncertainty (see chapter section 3.1.3) |
| Yaw rate | Meas1D | rad/s | Given in absolute values over ground with uncertainty (see section 3.1.1) |
| Acceleration | Meas3D | $m/s^2$ | Relative to the ego, given in the vehicle coordinate system, (see chapter section 6.2.3) and with uncertainty (see chapter section 3.1.3) |
| Length | Meas1D | m | Object length with uncertainty (see chapter section 3.1.3) |
| Width | Meas1D | m | Object width with uncertainty (see chapter section 3.1.3) |
| Height | Meas1D | m | Object height with uncertainty (see chapter section 3.1.3) |
| Class | enum | | Type of the object, e.g. unknown, passenger car, truck, (motor)cycle, pedestrian or static object |
| Is movable | enum | | Describes if the object has been seen moving even if it is static now. Values: True, false or unknown |

| Name | Data type | Unit | Description |
|---|---|---|---|
| Last seen by sensor | Time | | Describes at what point in time the sensor has seen the object last. If the sensor has never seen this object, the corresponding element of the vector is set to 0. |
| Existence probability | float | | Probability of the existence of this object |

Table 6.4. Object list format

| Name | Data type | Unit | Description |
|---|---|---|---|
| Timestamp | Time | | See section 3.2 for timing definitions |
| Object count | int | | Number of objects |
| Objects | Object[] | | Object description: See above |
| Sensor ID | enum | | ID of the object list source, e.g. front left ultrasonic sensor or object fusion |
| Sensor type | enum | | Measurement principle of the object list source, e.g. ultrasonic measurement or object fusion |

### 6.2.5. Convoy Tracks

Convoy tracks represent the path on which at least two cars moved or will move along as depicted in figure 6.4. It models the common behavior of drivers to follow each other or to keep within the lane boundaries. This representation is suited for scenarios with dense and well-structured traffic as a perfect complement of a camera-based lane marking detection.

Since the block Object Fusion in the architecture shown in figure 6.1 is responsible for deriving the representation of dynamic objects, it is also responsible for the convoy tracks.

The given specification assumes a polyline model wherein the centerline points are tracked without using a geometrical model. Further models will follow with increasing versions of this specification.

Figure 6.4. Merging Convoy Tracks (source: M. Reichel, Situationsanalyse für fortschrittliche Fahrerassistenzsysteme, Dissertation, Braunschweig, 2013)

Table 6.5. Convoy tracks

| Name | Data type | Unit | Description |
|---|---|---|---|
| Timestamp | int64, int64 | µs, µs | See section 3.2 for timing definitions |
| ID | int | | Convoy track ID |
| Age | int | ms | Time since creation of the track |
| Number of linked objects | int | | Number of linked objects a corresponding object list (see section 6.2) |
| Linked Objects | int[] | | Given as vector and stating all objects that build the convoy track |
| Number of center polyline points | int | | Number of points for the following centerline |
| Centerline points [x,y] | Meas2D | m | Given as vector for the x,y coordinates of the center polyline relative to the vehicle coordinate system, (see section 3.3.4) and with uncertainty (see section 3.1.2) |
| Width as left and right offset | Meas1D[2] | m, m | Given as vector for left and right offset for each point of the center polyline and with uncertainty (see chapter section 3.1.1) |

# 6.3. Grid fusion

## 6.3.1. Occupancy Grid

The occupancy grid is the most common usage of a grid representation of the environment. It describes the occupancy within a discretized 2D surface model. The grid itself is the base to contribute to further parts of the environmental model (freespace, objects and road model) as well as a possible starting point for direct functional computations as path planning for example. As centralized computing architectures become standard and bandwidth problems between ECUs become less important, the direct usage of grids can become reality for driving behaviors.

The block Grid Fusion in the architecture shown in figure 6.1 is responsible for Grids and therefore also for the occupancy grid.

Figure 6.5. Example of two occupancy grid layers and a path-planning taking into account both layers

In figure 6.5, the bottom occupancy layer represents a V2X given infrastructure, and the top layer fuses extrinsic sensor data input and has a higher resolution. The colors range from red (occupied) over blue (unknown) to green (free).

Table 6.6. Occupancy grid

| Name | Data type | Unit | Description |
|---|---|---|---|
| Timestamp | int64, int64 | µs, µs | See section 3.2 for timing definitions |
| ID | int | | Occupancy grid ID to distinguish between different occupancy grids |
| Mutex | | | Locking functionality to the grid |
| Grid block dimension [x,y] | int[2] | | The number of grid blocks in x and y dimension |
| Grid block count [x,y] | int [2] | | The number of cells in a grid block in x and y dimension |
| Grid resolution [x,y] | float[2] | m, m | The resolution of a grid cell in x and y dimension |
| Anchor coordinate system | ACS | | See section 3.3.6 for the coordinate system definitions |
| Grid occupancy layer [occupied, free] | float | | The occupancy probability. For Dempster-Shafer based evidence computation, occupied and free is modeled separately for each grid cell. For a Bayesian evidence computation only one occupancy float is needed, ranging from free (0) to occupied (1) |

# 6.4. Road and Lane fusion

The block Road and Lane Fusion in the architecture shown in figure 6.1 is responsible for inferring a model of the environmental road network from sensor measurements as well as from a-priori knowledge of digital maps and for providing this model via the Road and Lane fusion interface.

The model shall cover all road network related environment aspects that are of interest for the driving functions, i.e. lane topology (logical structure) and 2D/3D geometry, lane accurate position of the ego car in the road network, traffic regulations, road and lane boundaries, road surface material etc.

Since most of these environment aspects are more of meta-level rather than physical nature, the way of inference may depend on the particular driving function. From this follows the necessity for multi-hypothesis support in the representation of some of the environment aspects.

The function specific view blocks shown in figure 6.1 are responsible for selecting the data signals and hypotheses that are relevant to the respective driving functions and converting these data signals onto an interface adapted to the driving function’s needs.

Since there are many environment aspects covered by the model, they are grouped into subgroups of the same level of abstraction. So far, there are:

- The top-level (lane agnostic) medium range road network topology that only represents intersection-free segments of the network, the intersections in the network, the drivable paths through the network, the ego car position and its most probable path through the network. Only a single hypothesis is given, presumably taken from a digital map.
- The top-level road network geometry, i.e. road centerline geometries and positions.
- The low-level near range road network topology that represents the lanes of the different building blocks of the top-level road topology and their relations, the lane accurate ego car position and its most probable path through the lane network. Several hypotheses may be given.
- The low-level road network geometry, i.e. lane boundary and centerline geometries and positions.
- Convoy tracks and the drivable area used for inferring the model fit here relating to their level of abstraction. As their perception is done by Object Fusion and Grid Fusion respectively, the data representation is described in section 6.2 and section 6.3.1. Only a single hypothesis is given.
- The physical lane boundary indicators, i.e. lane markings, guard rails, gantries et cetera including their geometry and position.

The digital map data contains aspects of at least several levels of abstraction and its representation is predefined by its provider (i.e. ADASIS).

One possibility to implement the road and lane fusion interface is to enrich the digital map data type / interface with the road and lane fusion interface signals.

### 6.4.1. Top-level road network topology

In the data representation, the road network consists of two types of parts:

- A roadway segment is a linear segment of a road that consists of a not necessarily constant number of parallel non-intersecting lanes. It is either completely drivable in both directions or completely drivable in one direction. No lane of another segment may merge into or split off the interior of the segment. The segments are directed, i.e. one of their endpoints is referred to as front, the other endpoint as tail. The direction from front to tail is called segment direction.
- At the end of a segment there are either other segments directly connected in a lane intersection free manner or the part of the road network containing the intersecting lanes is represented as roadway intersection.

Besides naming the roadway segments and intersections, the data representation holds roadway segment connectors, which encode the immediate directed drivable connections between the roadway segment ends and specify if and which roadway intersection is traversed.

The ego position and most probable path are encoded by an ordered list of ego path segments. They specify the current and most probably following roadway segments the ego is driving on, including driving direction.

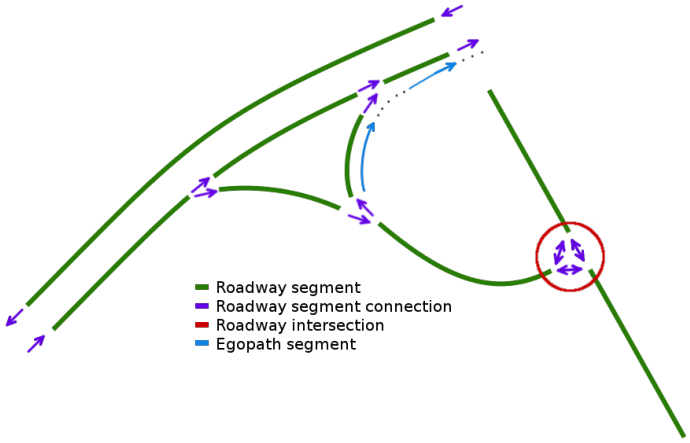


Figure 6.6. Visualization of the top-level road network topology

figure 6.6 shows a highway consisting of separate roadways, a country road, and a highway entry/exit for one of the highways driving directions.

One highway roadway is described by a single roadway segment, the other roadway by a roadway segment before the exit, one in between exit and entry, and one after the entry.

The only spot in the topology where driving corridors are intersecting each other is the junction between country road and highway entry and exit lanes. This is described by referencing the junction in the segment connections.

The ego position is just before entering the acceleration lane; therefore, there are two ego path segments.

The following signals implement the data representation:

Table 6.7. TopLevelRoadNetworkTopology signals

| Name | Data type | Unit | Description |
|---|---|---|---|
| roadwaySegments | Vector of RoadwaySegment | | |
| roadwayIntersections | Vector of RoadwayIntersection | | |
| roadwaySegmentConnections | Vector of RoadwaySegmentConnection | | |
| mostProbableEgoPathSegments | Vector of EgoPathSegment | | The vector is ordered according to the temporal order of the ego's prospective segment traverse, i.e. the first vector element specifies the current ego position and direction. |

Table 6.8. RoadwaySegment signals

| Name | Data type | Unit | Description |
|---|---|---|---|
| id | Unsigned int | | |
| centerlineID | Unsigned int | | Reference to an element of top-level road network geometry, see section 6.4.2. |
| roadwayPartIDs | Vector of unsigned int | | Reference to elements of low-level road network topology, see section 6.4.3.<br><br>The roadway part IDs are ordered according to the part positions inside the segment in segment direction. |

Table 6.9. RoadwayIntersection signals

| Name | Data type | Unit | Description |
|---|---|---|---|
| id | Unsigned int | | |
| roadwayPartIDs | Vector of unsigned int | | Reference to elements of low-level road network topology, see section 6.4.3. |

Table 6.10. RoadwaySegmentConnection signals

| Name | Data type | Unit | Description |
|---|---|---|---|
| connector | LinearSegmentConnector | | See section 6.4.6. |
| traversedIntersectionID | Unsigned int | | If no intersection is traversed, an invalid ID must be used (i.e. 0). |

Table 6.11. EgoPathSegment signals

| Name | Data type | Unit | Description |
|---|---|---|---|
| roadwaySegmentID | Unsigned int | | |
| drivingDirection | Enum Direction | | Either IN_SEGMENT_DIRECTION or AGAINST_SEGMENT_DIRECTION. This specifies whether the ego is traversing the segment from front to tail or from tail to front. |

## 6.4.2. Top-level road network geometry

Regarding the top-level road networks geometry, only the roadway segment centerline is of particular interest. The detailed geometry (i.e. width of the road) is available in the low-level road network geometry.

A 2D polygonal chain in local coordinates represents a roadway segment centerline.

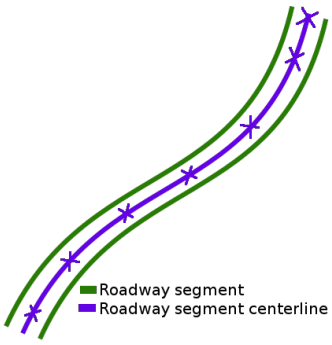


Figure 6.7. Visualization of the top-level road network geometry

figure 6.7 shows a roadway segment from the top-level road network topology that is enriched by geometric information in the form of a polygonal chain specifying the location of the segments centerline.

The following signals implement the geometry representation:

Table 6.12. TopLevelRoadNetworkGeometry signals

| Name | Data type | Unit | Description |
|---|---|---|---|
| roadwaySegmentCenterlines | Vector of LinearSegmentGeometry2D | | See section 6.4.6. |

## 6.4.3. Low-level road network topology

The low-level road network topology data structure [1] offers support for multi-hypothesis representation of how the building blocks of the top-level road network are composed of lanes and how these lanes are interconnected.

In order to represent the hypotheses, the roadway segments and roadway intersections are partitioned into roadway parts – each roadway part will have one or multiple roadway part hypotheses. Roadway segments are partitioned in parts only in a longitudinal direction, i.e. a hypothesis always describes the whole width of a longitudinal interval of the roadway. In addition, the partitioning is done such that:

- Each roadway part has a fixed number of lanes in all hypotheses over its length
- Roadway parts are as long as possible

Each roadway part hypothesis consists of an arbitrary number of lane part hypotheses. Hypotheses of the same roadway part may share lane part hypotheses. Each lane part hypothesis has a right and left boundary part hypothesis. They may also be shared between lane hypotheses of the same roadway part.

[1] The idea was taken from Dierkes, Frank, et al. "Towards a Multi-hypothesis Road Representation for Automated Driving." Intelligent Transportation Systems (ITSC), 2015 IEEE 18th International Conference on. IEEE, 2015. and adapted.

Lane part hypotheses and boundary part hypotheses are directed, the same wording as that of roadway segments is used. Lane part hypotheses inherit their direction from their roadway segment, boundary part hypotheses inherit their direction from the lane part hypotheses they are referenced by.

Compatible hypotheses of neighboring roadway parts are connected via roadway part hypothesis connections. They incorporate information about how their lanes are connected in the form of roadway part hypothesis lane connections. Compatible means here, that it is plausible that both hypotheses are valid (and not only either the first or the second) – and only in this case a connection information is meaning and useful.

The current lane accurate position and driving direction of the ego is described by a list of ego lane matchings. Each matching describes one hypothesis consisting of lane accurate position and driving direction of the ego and hypothesis probability.

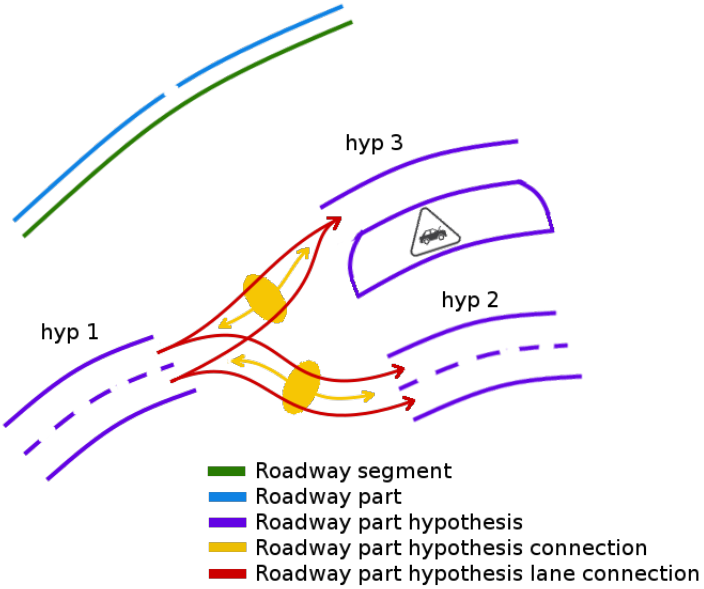


Figure 6.8. Visualization of the low-level road network topology

figure 6.8 shows a roadway segment that is partitioned into two roadway parts. The left roadway part has a single hypothesis (hyp 1), consisting of two lanes. The right roadway part has two hypotheses (hyp 2 and hyp 3). Both of these are compatible with hyp 1.

Hyp 2 consists of two lanes. Each lane in hyp 2 is connected with the respective lane in hyp 1. Hyp 3 may be based on an observed convoy track merge: Both lanes of hyp 1 are merging into the single lane of hyp 3. A broken down vehicle could be blocking the missing lane.

Table 6.13. LowLevelRoadNetworkTopology signals

| Name | Data type | Unit | Description |
|---|---|---|---|
| roadwayParts | Vector of RoadwayPart | | |

| Name | Data type | Unit | Description |
|---|---|---|---|
| roadwayPartHypothesis-Connections | Vector of RoadwayPartHypothesisConnection | | |

Table 6.14. RoadwayPart signals

| Name | Data type | Unit | Description |
|---|---|---|---|
| id | Unsigned int | | |
| hypotheses | Vector of RoadwayPartHypothesis | | |
| lanePartHypotheses | Vector of lanePartHypothesis | | The lane part hypotheses are shared between roadway part hypotheses of the same roadway part and are thus stored in this struct. |
| boundaryPartHypotheses | Vector of BoundaryPartHypothesis | | The boundary part hypotheses are shared between lane part hypotheses of the same roadway part and are thus stored in this struct. |

Table 6.15. RoadwayPartHypothesis signals

| Name | Data type | Unit | Description |
|---|---|---|---|
| id | Unsigned int | | |
| laneHypothesisIDs | Vector of unsigned int | | If the roadway part is a segment, lane hypotheses are ordered by their topological sequence in the segment from right to left. |
| probableEgoLaneMatchings | Vector of EgoLaneMatching | | |

Table 6.16. LanePartHypothesis signals

| Name | Data type | Unit | Description |
|---|---|---|---|
| id | Unsigned int | | |
| leftBoundaryHypothesisId | Unsigned int | | |
| rightBoundaryHypothesisId | Unsigned int | | |
| presentInDigitalMapData | bool | | True, if a lane contained in the digital map matches this lane. |

| Name | Data type | Unit | Description |
|---|---|---|---|
| centerlineID | Unsigned int | | Reference to an element of low-level road geometry, see section 6.4.4. |
| matchedConvoyTrackID | Unsigned int | | ID of a convoy track matching this lane, see section 6.2.5.<br><br>If no convoy track matches the lane, an invalid ID must be used (i.e. 0). |

Table 6.17. BoundaryPartHypothesis signals

| Name | Data type | Unit | Description |
|---|---|---|---|
| id | Unsigned int | | |
| lineID | Unsigned int | | Reference to an element of low-level road geometry, see section 6.4.4. |
| matchedPhysicalLaneBoundaries | Vector of unsigned int | | Reference to elements of physical lane boundary indicators, see section 6.4.5. |

Table 6.18. EgoLaneMatching signals

| Name | Data type | Unit | Description |
|---|---|---|---|
| laneHypothesisId | Unsigned int | | |
| drivingDirection | Enum Direction | | Either IN_SEGMENT_DIRECTION or AGAINST_SEGMENT_DIRECTION. |
| probability | Float | | |

Table 6.19. RoadwayPartHypothesisConnection signals

| Name | Data type | Unit | Description |
|---|---|---|---|
| FirstHypothesisID | Unsigned int | | |
| SecondHypothesisID | Unsigned int | | |
| laneConnections | Vector of LinearSegmentConnector | | See section 6.4.6. |

## 6.4.4. Low-level road network geometry

The low-level road network geometry describes the geometry of lane segment hypotheses.

A lane part boundary hypothesis line describes the geometry of a boundary part hypothesis and a lane part hypothesis centerline describes the geometry of the centerline of a lane part hypothesis. Both are represented by 3D polygonal chains in local coordinates.

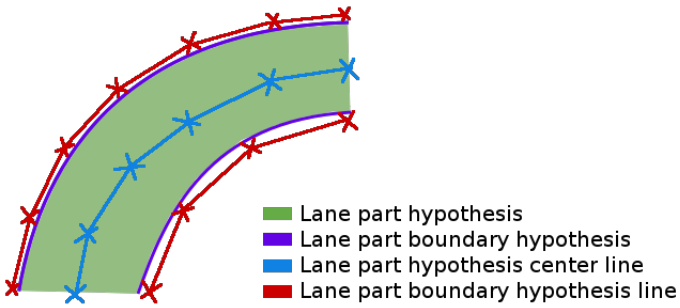


Figure 6.9. Visualization of the low-level road network geometry

figure 6.9 shows a hypothesis of a lane part and the corresponding boundary hypotheses from the low-level road network topology being enriched by geometric information in the form of polygonal chains. For each boundary, one polyline specifies its location and for each lane, one specifies its centerline's location.

The data representation is implemented by the following signals:

Table 6.20. LowLevelRoadNetworkGeometry signals

| Name | Data type | Unit | Description |
|---|---|---|---|
| lanePartHypothesisCenterlines | Vector of LinearSegmentGeometry3D | | See section 6.4.6. |
| lanePartBoundaryHypothesisLines | Vector of LinearSegmentGeometry3D | | See section 6.4.6. |

## 6.4.5. Physical lane boundary indicators

One of the physical entities that can be used to infer knowledge about the road network – and especially about the lane boundaries – is linear static elements in the environment that are or are parallel to lane boundaries. For simplicity, they are referred to as physical lane boundaries and are modeled as follows:

- The geometry of a lane boundary is approximated by a simple polygonal chain and thus is represented by the sequence of vertices of the polygonal chain
- Each linear element has a type and optionally type dependent attributes (i.e. color for the type lane marking)
- The endings of two linear elements can be connected to each other by a lane boundary connector.
- Physical lane boundaries are directed, the same wording as that of roadway segments is used. They inherit their direction from the boundary part hypotheses they are referenced by.

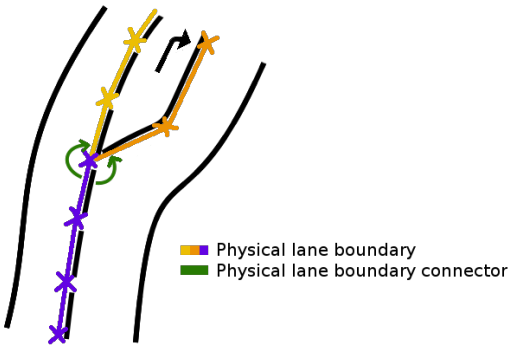


Figure 6.10. Visualization of the physical lane boundary indicators

figure 6.10 shows the position of lane markings described by polygonal chains. A lane marking splitting into two lane markings is described by two connections between the corresponding endpoints of their three polygonal chains.

The data representation is implemented by the following signals:

Table 6.21. PhysicalLaneBoundaries signals

| Name | Data type | Unit | Description |
|---|---|---|---|
| physicalLaneBoundaries | Vector of PhysicalLaneBoundary | | |
| laneBoundaryConnectors | Vector of LinearSegmentConnector | | See section 6.4.6. |

Table 6.22. PhysicalLaneBoundary signals

| Name | Data type | Unit | Description |
|---|---|---|---|
| id | Unsigned int | | |
| Type | Enum LaneBoundaryType | | e.g. LANE_MARKING, FENCE, KERB. |
| markingType | Enum LaneMarkingType | | Only used for boundaries of type LANE_MARKING, e.g. SOLID, DASHED. |
| markingColor | Enum LaneMarkingColor | | Only used for boundaries of type LANE_MARKING, e.g. YELLOW, WHITE. |
| polygonalChainNodes | Vector of Meas3D | [m] | See section 3.1.3. Positions are given in local coordinates. |

## 6.4.6. Auxiliary types

In several levels of abstraction the following auxiliary types are used:

1. A linear segment connector is a directed link between two segment ends
2. A segment end is either the front or the tail of some specified segment in the environment model (e.g. of a physical lane boundary)
3. A linear segment geometry holds spatial information of a segment in the environment model. This information is given by a polygonal chain of nodes given in either 2D (ground plane) or 3D.

Linear segment connector
: A directed link between two segment ends

Segment end
: Either the front or the tail of a specified segment in the environment model e.g. of a physical lane boundary

Linear segment geometry
: Spatial information of a segment in the environment model, that is given by a polygonal chain of 2D nodes or 3D nodes

Table 6.23. LinearSegmentConnector signals

| Name | Data type | Unit | Description |
|---|---|---|---|
| start | SegmentEnd | | segment end where the connection is starting. |
| end | SegmentEnd | | segment end where the connection is ending. |

Table 6.24. SegmentEnd signals

| Name | Data type | Unit | Description |
|---|---|---|---|
| segmentId | Unsigned int | | It should be apparent from the context to which type of entity the ID refers to. |
| positionInSegment | Enum positionInSegment | | Either FRONT or TAIL. |

Table 6.25. LinearSegmentGeometry2D signals

| Name | Data type | Unit | Description |
|---|---|---|---|
| id | Unsigned int | | |
| polygonalChainNodes | Vector of Meas2D | [m] | See section 3.1.2. Positions are given in local coordinates. They are ordered by their succession in the chain from front to tail point. |

Table 6.26. LinearSegmentGeometry3D signals

| Name | Data type | Unit | Description |
|---|---|---|---|
| id | Unsigned int | | |
| polygonalChainNodes | Vector of Meas3D | [m] | See section 3.1.3. Positions are given in local coordinates. They are ordered by their succession in the chain from front to tail point. |

## 6.5. Vehicle database

Next to the perception of the vehicle´s environment, the car has also a computational representation of itself. This covers static knowledge as data of mounting positions of sensors, the E/E system as well as dynamically changing knowledge about physical properties and own capabilities, for example the current mass, fuel range, blind sensors or actuator status.

# 7. Function specific views

The Function Specific Views block (figure 7.1) extracts information of the environment model in order to give the subset of information to modules that are downstream of the sensor data fusion. It also solves ambiguities that the sensor data fusion block explicitly models. For example, the existence of white and yellow paintings leads to several lane interpretations. A function specific view uses the knowledge of the aim of the function to solve such ambiguities. A piloted function will try to keep within the overlap of both – a lane departure warning might take into account the other boundaries.

The function specific view is not responsible for deriving further abstraction levels (for example of collaboration predictions, behavior classifications or others) – it rather shrinks the information to the required subset of the following function.

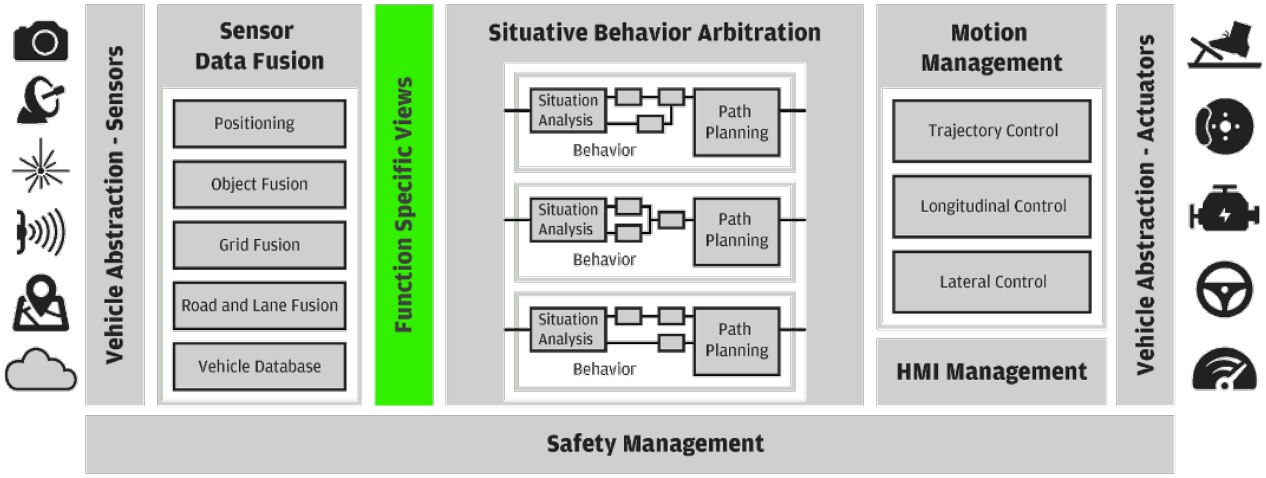


Figure 7.1. Function Specific Views as part of the architecture overview

## 7.1. Freespace

The Freespace describes an area in which the car can probably move without collision. It is a 2D representation without any height information of a surface that is not necessarily flat but can be driven by the car. The necessity to derive such representation is that an inverse object list representation is not a freespace as shown in figure 7.2.

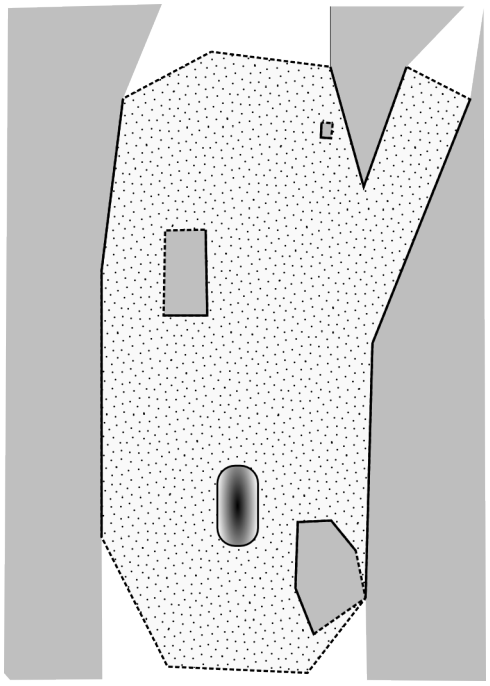

Figure 7.2. Example of a Freespace

The car is in the bottom center of the picture, the free space is represented by the dotted area. Gray areas represent physical objects, like other cars, guard rails, trees, etc. Obstacles can be found within the free space polygon. White areas are free but are not part of the freespace, because, for instance, out of sensor reach. Continuous lines denote the transition from free space to not free space: a space is not free if the sensors identify an object (static or dynamic) in it. Dashed lines mark the transition from free space to unknown: the state of the space after the line is unknown because the sensors cannot sense it (either it is too far away or the sensor is blocked by other objects)

The area of a freespace is described using a closed contour polyline as envelope and further polygons to model not free inclusions within the freespace envelope.

All polygons are meant to be closed polygons, that is, the last point in the list is connected to the first one.

Table 7.1. Freespace

| Name | Data type | Unit | Description |
|---|---|---|---|
| Timestamp | int64, int64 | µs, µs | See section 3.2 for timing definitions |
| ID | int | | Freespace ID |
| Number of envelope contour points | int | | Number of envelope contour points |
| Envelope contour points [x,y] | Meas2D | m | [x,y] coordinates of the envelope contour given in VSC (section 3.3.4), the contour points describe a polygon by connecting them in ascended sequence and are |

| Name | Data type | Unit | Description |
|---|---|---|---|
| | | | described as uncertain values (see section 3.1.2) |
| Segment transition to not-free | bool[] | | Tells if the segment in the free space was created because of a transition from free to not free (TRUE) or from free to unknown (FALSE) |
| Number of inclusions | int | | Number of inclusions within the envelope contour |
| Number of inclusion contour points | int | | Number of inclusion contour points of each inclusion |
| Inclusion contour points [x,y] | float[][] | m | Given as vector of vectors for the x,y coordinates of the inclusions also described as polygons by connecting the points in ascended sequence, modeled as uncertain values (see see section 3.1.3) |
| Segment transition to not-free | bool[] | | Tells if the segment in the free space was created because of a transition from free to not free (TRUE) or from free to unknown (FALSE) |
| Probability threshold | float | | States the threshold of occupancy probability for free for extracting the freespace out of a grid |

# 8. Situative behavior arbitration

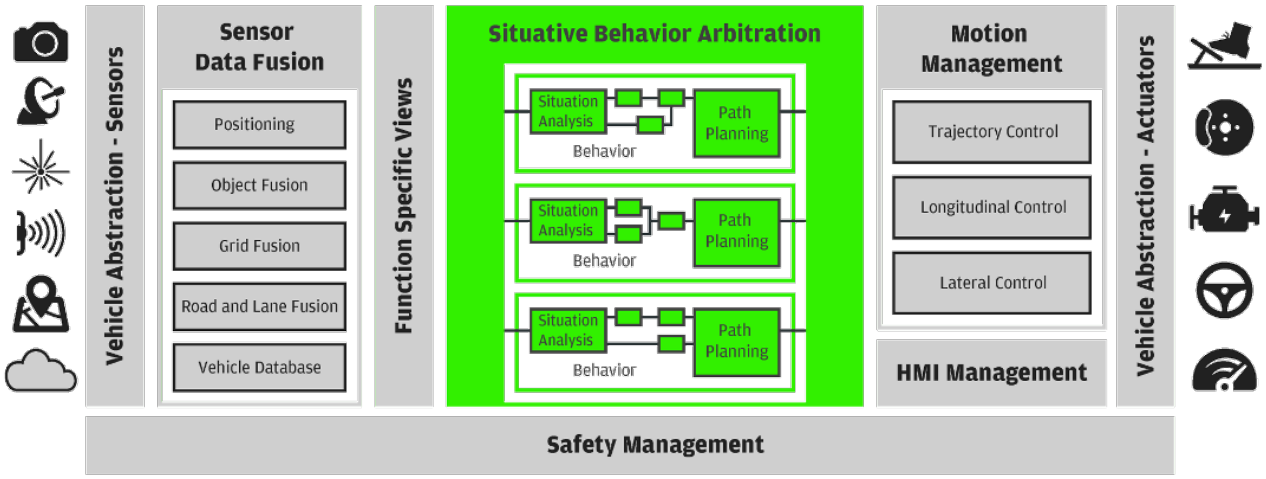


Figure 8.1. Situative Behavior Arbitration as part of the architecture overview

Every DAS system revolves around the idea of functions, since they are what the end customer experiences. Functions usually accept some sensory input, perform some situation analysis, make decisions about how to operate, and plan some vehicle behavior (longitudinal or lateral motion, or trajectory-based motion as a combination of both) as the result, as well as some HMI output.

For example, an automated emergency brake function will continuously check the differential velocity and distance to the leading vehicle, and use this data to determine a time-to-collision (TTC) value. If the TTC drops below a first threshold, the function will trigger an (HMI) alert to the driver to ask for evasive action; if the TTC drops below a second threshold, the function will communicate with the motion control components to perform an emergency braking action. If the function (or entire system) makes a mistake in this process, it might call for an emergency braking action that is not necessary, maybe because there is no leading vehicle at all. Since this causes a hazard for any vehicles that might follow, this function would have to fulfill high functional safety requirements; but since it is still an emergency braking assistant, the driver is still the fallback. So in case that an error is detected (e.g. the function receives no environment information anymore), it can simply switch off and trigger a warning to the driver that it is no longer working.

An example of how to construct this simple function in the open robinos architecture can be found in below (figure 8.2, “Example for three coordinated behaviors realizing a level-2 automated system”).

It gets more and more interesting the more functions are active at the same time, and the higher their safety requirements are. To guarantee coherent behavior and safety, all the functions moving the vehicle must be co-ordinated, ideally in a centralized location, although distribution is also possible. The idea of "situative behavior arbitration" is the suggested mechanism within open robinos to perform this co-ordination. It is built around the following core concepts:

- Function-behavior decomposition: It should be possible to distinguish between customer functions (as a user-experience concept), and elementary behaviors of which they consist (as a technical, software-block

concept). Therefore, a customer function such as level-2 lane-keeping adaptive cruise control can be split into the elementary behaviors of "keep distance from leading vehicle" and "center in own lane". The arbitration system is responsible for coordinating these two elementary behaviors into one single function. This is not mandatory, however; modularity commands that it is quite possible to create a behavior that does full trajectory planning, integrating these two behaviors into one.

- Distribution of interpretation: The architecture should not have to know in advance the details of every function's limits and way of operation. Situation analysis is therefore a part of the function itself.
- Centralization of decision: The decision which behavior(s) have control over (parts of) the vehicle's motion, is not made by the behaviors themselves but by a central concept called the situative behavior arbitration block. In a multi-behavior system, behaviors should communicate with this block and not with the control components directly.

The architectural concept used to realize this concept is the situative behavior arbitration block, which can be thought of as a "plugin container" for the behavior (or function) modules. The interface of a behavior used to express its desire, or necessity, to operate is very simple and is defined by four scalars:

- Applicability: The ability of the function to operate, expressed in percent. E.g. a lane change assist that sees clear lane markings, road side boundaries and convoy tracks, and an empty left lane, might set this to 100%; if it only sees convoy tracks and lane markings are unclear (e.g. in a construction site), it might set this to 30%; if the left lane is occupied, it will always set this to 0%.
- Desire: The desire of the function to operate, expressed in percent. E.g. an adaptive cruise control function set to 130km/h on an empty highway might set this to 100%; if it finds itself behind a truck going 80km/h it might set this to 50%.
- Risk: A scalar, expressed in percent, giving an assessment of the risk involved when performing the behavior. A lane changing assistant that sees a perfectly clear left lane might set this to 20% (since visibility from the ego vehicle will always be obstructed), one that sees a slowly-approaching vehicle to the rear in the left lane might set this to 50%, one that sees a vehicle arriving with high difference velocity might set it to 90%.
- Comfort: A scalar, expressed in percent, giving an assessment of the comfort to the driver that performing a certain motion will entail; expected high lateral or longitudinal acceleration or deceleration will result in a low comfort level, gentle motions in a high one.

Table 8.1. Situative behavior

| Name | Data type | Unit | Description |
|---|---|---|---|
| Timestamp | int64, int64 | µs, µs | See section 3.2 for timing definitions |
| ID | int | | Behavior ID |
| Applicability | int | % | Applicability of the behavior |
| Desire | int | % | Behavior's desire to operate |
| Risk | int | % | Predicted risk, should behavior be executed |

| Name | Data type | Unit | Description |
|---|---|---|---|
| Comfort | int | % | Comfort to the driver, should behavior be executed |

figure 8.2 shows an example of three coordinated behaviors realizing a level-2 automated system which is able to center the vehicle in its lane, follow traffic, (semi-) automatically change lanes and perform an emergency braking maneuver if required.

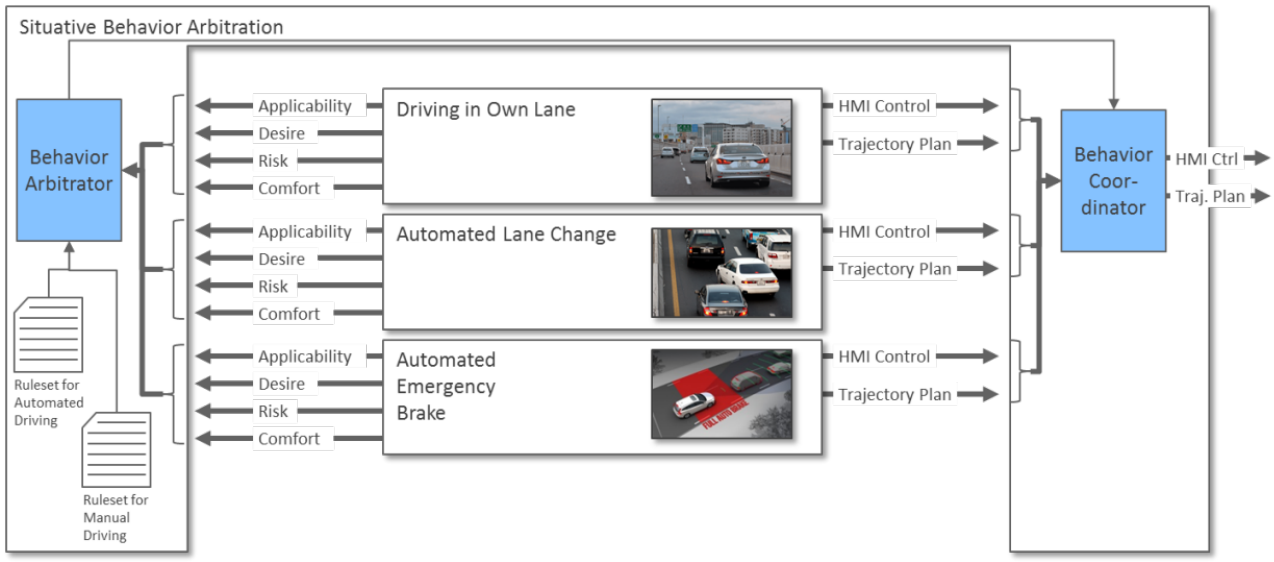


Figure 8.2. Example for three coordinated behaviors realizing a level-2 automated system

figure 8.3 shows the basic construction of a behavior module, and the few interfaces it needs to fulfill to be integrated into the situative behavior arbitration module. Note that applicability and desire are computed by the situation interpretation module (ante facto), while behavior assessment to provide risk and comfort ideas needs to have some idea of the vehicle motion, so can only be computed after motion planning has been performed.

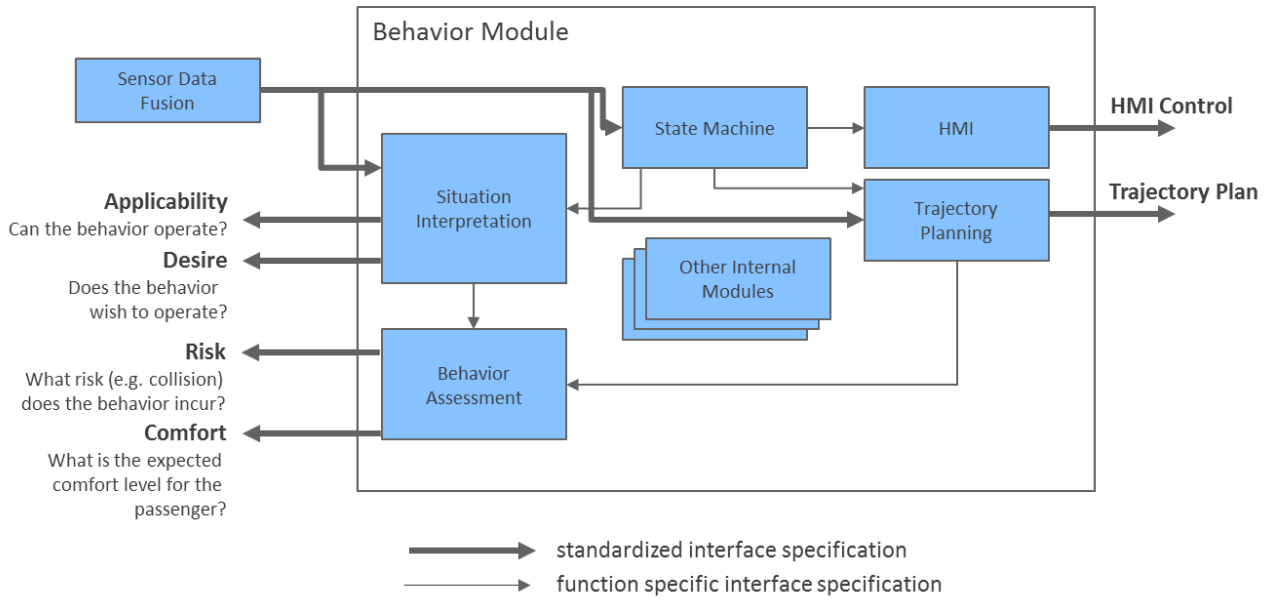


Figure 8.3. Basic construction of a behavior module

# 9. Motion management

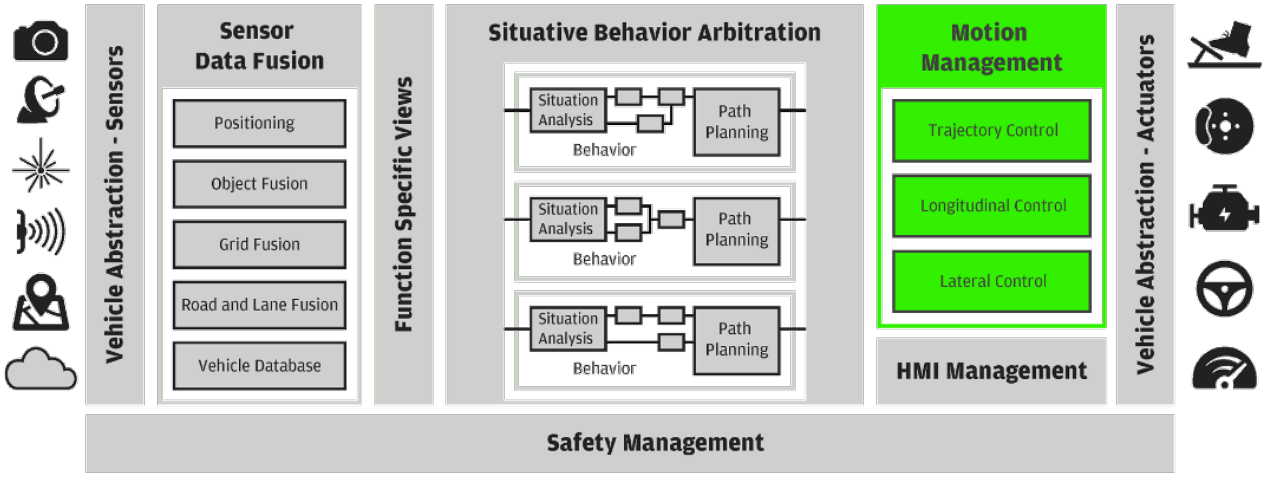


Figure 9.1. Motion Management as part of the architecture overview

The Motion Management block in the architecture does the abstraction of the increasing amount of actuators on the output and the increasing amount of behaviors on the other. It accepts motion commands, and controls the vehicle accordingly. Furthermore, it selects the actuator set that introduces the necessary changes in the vehicle´s dynamics.

As an example for such abstraction, the following of a curved path will be described: As input the block accepts for example a trajectory. Such trajectory describes the desired future behavior in terms of when to be where in which pose – but it does not need to specify whether the change in yaw rate over time is introduced by steering the front wheels, steering the back wheels, torque vectoring, single wheel braking, …. or a combination of these things. As a result, the behavior planning and the Safety Management of fallback layers become less complex.

The Motion Management described in the following sections accept trajectory input (used for level 2 automation and above), longitudinal only input (used for ACC, AEB, and similar), and lateral only input (used for LKA and similar).

## 9.1. Path

The interface should be able to realize both complex back-and-forth paths for parking applications and simple linear paths for highway driving. For this purpose, a path is made of separate segments that are driven in forward or reverse gear (see figure 9.2). A path without directional changes contains only one segment.

Path segments are specified as polygons consisting of a list of control points (x, y) with a vehicle orientation and a suggested velocity.[1]

[1]This concept is rather over-determined and may have to be revised.

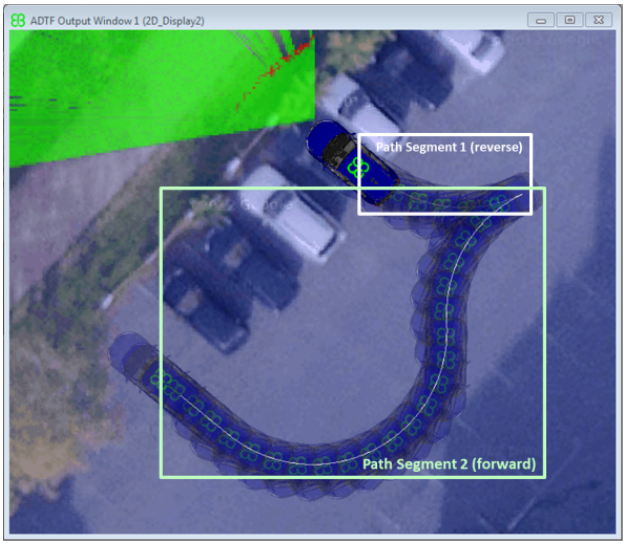


Figure 9.2. Two segment path, one with backward direction and one with forward direction (source: Elektrobit)

**NOTE**

**Paths, not trajectories**

This section talks about paths, not trajectories. Full trajectory planning is often not possible within a real traffic situation since motion velocity needs to be adapted to that of other traffic participants. The specification tries to handle this by introducing "suggested velocities", but these are advisory. The geometrical construct is more important than the temporal. Therefore, we use the word path even though some connection to timing is introduced.

Table 9.1. Path control point

| Name | Data type | Unit | Description |
|---|---|---|---|
| X | float | m | X position of the control point, relative to anchor coordinate system |
| Y | float | m | Y position of the control point, relative to anchor coordinate system |
| Yaw | float | rad | Yaw of the vehicle when control point is reached, relative to anchor coordinate system |
| V | float | m/s | Proposed velocity of the vehicle when control point is reached |

Table 9.2. Path section

| Name | Data type | Unit | Description |
|---|---|---|---|
| Path Direction | enum | | Forward, backward, stopped, unknown |
| Number of Control Points | int | | |
| Control Points | Control Points[] | | See above |

Table 9.3. Path description

| Name | Data type | Unit | Description |
|---|---|---|---|
| TimeStamp | int64, int64 | µs, µs | See section 3.2 |
| Number of Path Sections | int | | |
| Path Sections | Path sections[] | | See above |

# 9.2. Longitudinal control

Longitudinal control is directly realized via acceleration value, as it is directly observable. The longitudinal controller is responsible for mapping this acceleration to the vehicle's possibilities (motor throttle / stall moment, recuperative braking, disc brakes, ...).

The same interface is used for the concept of brake jerks. Since this means a very high deceleration in comparison to the ones preceding and following, together with the promise that the next controller value will ease up on that deceleration again, and the longitudinal controller must not smooth out this spike, there are high functional safety aspects associated with this concept. A brake jerk must therefore be marked by a separate Boolean value. A real implementation might also use this flag to trigger a predefined brake jerk instead of the acceleration value contained in the data.

Table 9.4. Longitudinal control input

| Name | Data type | Unit | Description |
|---|---|---|---|
| Timestamp | int64, int64 | µs, µs | See section 3.2 |
| Acceleration | float | m/s² | Positive: acceleration, negative: deceleration |
| Brake jerk | bool | | Distinction between emergency (fast) braking and comfort braking (see text above within this chapter) |

# 9.3. Lateral only control

The input for lateral control is vehicle yaw rate, as it is directly observable. The lateral controller is responsible for mapping this yaw rate to the vehicle's possibilities (front / rear steering, one-sided braking, ...).

Table 9.5. Lateral control input

| Name | Data type | Unit | Description |
|---|---|---|---|
| Timestamp | int64, int64 | µs, µs | See section 3.2 |
| Yaw Rate | float | rad/s | Positive: to the left, negative: to the right |

# 10. HMI management

As the Motion Management is the abstraction layer to the increasing amount of actuators – the HMI Management does such abstraction for the increasing amount of displaying options. Dashboard, head-up display with augmented reality features, central monitor or new curved displays among the interior design need to be managed in what, when, where and how information is transported to the driver / passengers.

While deriving an architecture for automated driving it became clear that there is the necessity for such functional part. This section will be expanded in upcoming versions of this specification. We would like to invite the community to contribute. Please join by contacting us via join@open-robinos.com.

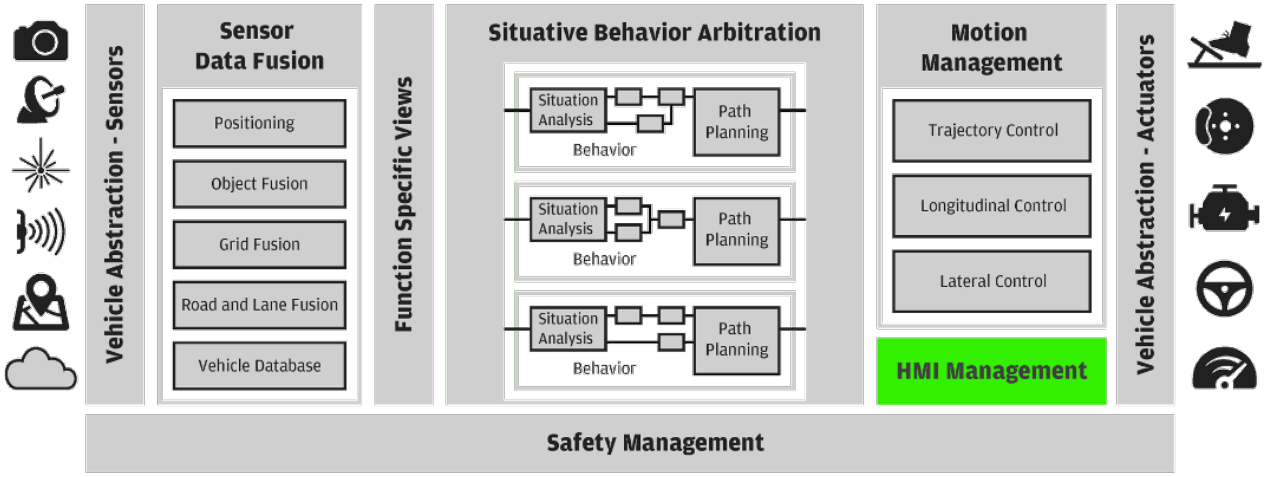


Figure 10.1. HMI Management as part of the architecture overview

# 11. Vehicle abstraction - Actuators

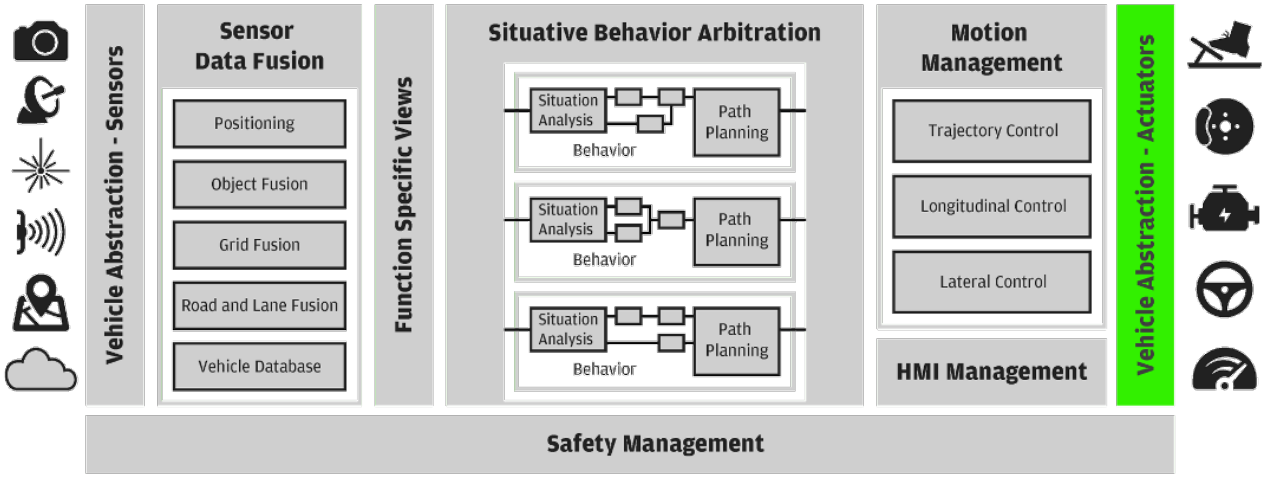


Figure 11.1. Vehicle Abstraction – Actuators as part of the architecture overview

The Vehicle Abstraction – Actuators block addresses the task of transforming the open robinos specification into individual actuator interfaces. On the one hand, these are easy transformations like transforming an angle from [rad] to [degree]. On the other hand, these are abstractions to different types of actuators. The following actuator types can be defined:

- Vehicle dynamic influencing actuators: These actuators are capable of changing the vehicle´s dynamics.
- Communication and other actuators: These elements communicate with the outside world of a vehicle. On the one hand, these are indicators. On the other hand, these are communication elements to transport information to the cloud or V2X. Furthermore, this group includes helping actuators such as belt tensioners or wipers.

The specification within this section will increase with the participation of more actuator supplier. Everyone can participate at writing the open robinos specification. Please join by contacting us via join@open-robinos.com.

## 11.1. Vehicle dynamic influencing actuators

This list shows a specification for drive by wire vehicles as well as a simple abstraction for electric vehicles.

Table 11.1. A specification for drive by wire vehicles

| Name | Data type | Unit | Description |
|---|---|---|---|
| Timestamp | int64, int64 | µs, µs | See section 3.2 for timing definitions |
| Ignition | bool | | Start ignition |

| Name | Data type | Unit | Description |
|---|---|---|---|
| Accelerator pedal position | float | % | Requested position of the accelerator pedal [0 ... 100] used for drive by wire equipped cars |
| Brake pedal position | float | % | Requested position of the brake pedal [0 ... 100] used for drive by wire equipped cars |
| Steering wheel position | float | % | Requested position of the steering wheel [-100 … 100] used for drive by wire equipped cars |
| Gear request | int | | Requested gear (P, N, D, R) for drive by wire equipped cars |
| Hand Brake Apply | int | | Activate the hand brake |
| Hand Brake Release | int | | Release the hand brake |
| Yaw Rate | float | rad | Target yaw rate |
| Velocity | float | m/s | Target longitudinal velocity |
| Acceleration | float | m/s² | Target longitudinal acceleration |

# 11.2. Communication and other actuators

The following specification describes the interface to actuators as beams or wipers.

Table 11.2. The interface to actuators

| Name | Data type | Unit | Description |
|---|---|---|---|
| Timestamp | int64, int64 | µs, µs | See section 3.2 for timing definitions |
| Turn Signal Left | int | | Activate turn signal left |
| Turn Signal Right | int | | Activate turn signal right |
| Hazard Lights | int | | Activate hazard lights |
| Parking Lights | int | | Activate parking lights |
| Lower Beam | int | | Activate lower beam |
| Full Beam | int | | Activate full beam |
| Wiper Interval | int | | Wipers are activated |
| Wiper Step One | int | | Wipers are activated, period is slow |
| Wiper Step Two | int | | Wipers are activated, period is fast |
| Horn | int | | Activate the horn |

| Name | Data type | Unit | Description |
|---|---|---|---|
| Front Windscreen Washer System | int | | Activate the front windscreen wash system |
| Back Windscreen Washer System | int | | Activate the back windscreen wash system |

In addition to the actuators within the car, there are communication actuators to enable the car to communicate with its environment and the cloud via standardized communication protocols (as WAVE for example). One important specification that is currently in the progress of being founded is SENSORIS (hosted by ERTICO http://ertico.com/). The open robinos specification does not want to specify this communication channel twice; therefore, we would like to refer to SENSORIS.

# 12. Safety Management

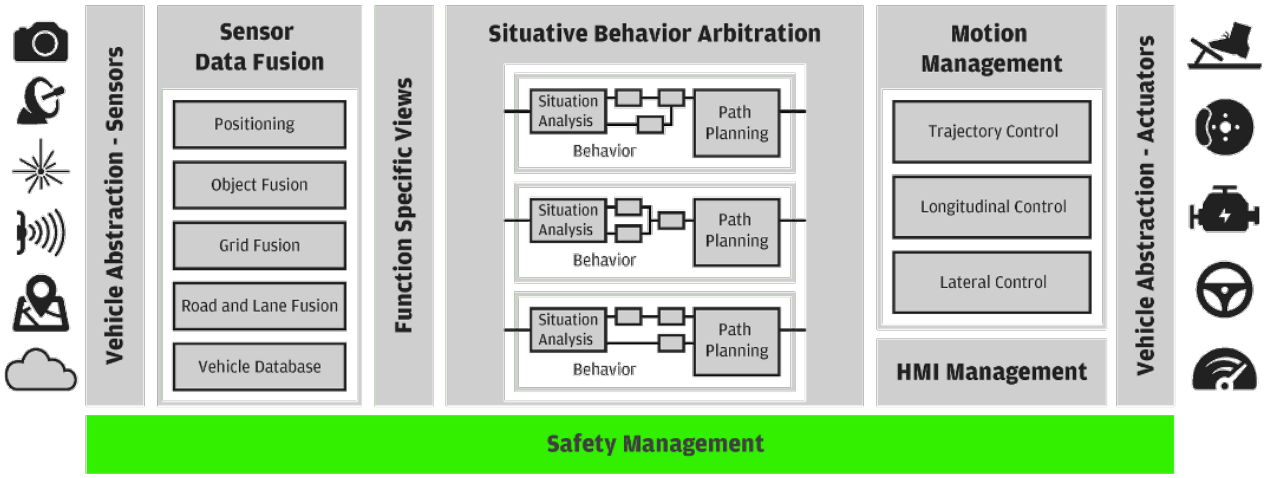


Figure 12.1. Safety Management as part of the architecture overview

The Safety Management block covers two basic tasks:

- Perform plausibility checks on data transferred between the modules in the functional architecture
- Collect all plausibility check results and derive suitable strategies in terms of safety and error handling

## 12.1. Plausibility checking

A standard plausibility check is the evaluation of data thresholds and timing conditions. Next to such easy implementations, there exist also further interpretation steps in order to evaluate greater and more complex parts of the architecture. As an example, the result of a path planning based on a grid-based sensor data fusion can be checked against a second approach of sensor data fusion in order to ensure that the first computation chain is working correctly.

The result of such safety and error finding driven evaluation is a list of system states:

Table 12.1. System states

| Name | Data type | Unit | Description |
|---|---|---|---|
| Timestamp | int64, int64 | µs, µs | See section 3.2 for timing definitions |
| Check Result | bool[] | | List of results for every checked state - OK or not OK |

## 12.2. Deriving safety and error strategies

The list of check results is used to decide whether a specific behavior can be activate due to is activation prerequisites or – if the behavior is already active – what fail operational mechanism shall be activated and what information shall be stored for later error handlings.

# 13. License

The open robinos specification is provided to facilitate a common understanding of the architecture, interfaces and testing requirements of automated driving vehicles. It intends to enable and accelerate the development of such systems across the industry.

# 14. List of authors


| | |
|---|---|
| Manuel Frei | Elektrobit Automotive GmbH |
| Dr. Björn Giesler | Elektrobit Automotive GmbH |
| Dr. Matthias Haringer | Elektrobit Automotive GmbH |
| Florian Janßen | Elektrobit Automotive GmbH |
| Pierre Marie Kengne Nzegne | Elektrobit Automotive GmbH |
| Dr. Sebastian Ohl | Elektrobit Automotive GmbH |
| Dr. Michael Reichel | Elektrobit Automotive GmbH |
| Felix Tennert | Elektrobit Automotive GmbH |


If you or your company is interested in co-creating the open robinos specification, please join by contacting us via join@open-robinos.com.

# Glossary