System Specification - Cooperative Research Platform

Gergo Ferenc Igneczi[1*], Dávid Józsa[1†] and Mátyás Mesics[1†]

[1]Vehicle Industry Research Center, University of Győr, Egyetem sq., Győr, 9024, Győr-Moson-Sopron, Hungary.
[2]ZalaZONE Innovation Park, University of Győr, Dr. Michelberger Pál str., Zalaegerszeg, 8900, Zala, Hungary.

*Corresponding author(s). E-mail(s): gergo.igneczi@ga.sze.hu;
Contributing authors: jozsa.david@ga.sze.hu; mesics.matyas@ga.sze.hu;
†These authors contributed equally to this work.

Abstract

System developed, owned and maintained by University of Győr to accomplish various automated driving function tasks.

Keywords: none

1 Goal of the document

This document summarizes the details of how the Cooperative Research Platform (CRP) is built up. It consists of the multiple layers:

- System Functionality,
- Architecture,
- Scenario Coverage and demonstration basis

2 Supplements

2.1 Corresponding terminology

- E2E: end-to-end, usually referred to as standalone operation of a function, without the need of e.g., pre-recorded data, and this function can be used by a non-technician user
- function: practical manifestation of technical implementation
- architecture: collection of components that are arranged into a pre-defined structure,
- ground architecture: white-paper definition of system components, without dependencies like Autoware,
- function architecture: real structure of the system components, that are directly usable in the vehicle.

2.2 Coordinate Frames

2.3 Nomenclature

- α_f - front road-wheel angle
- δ - position offset to the centerline of the lane
- ψ - yawrate of the vehicle (rotational speed around ζ axis)
- v_ξ - longitudinal speed of the vehicle, in the vehicle frame (ξ, η, ζ)
- a_ξ - longitudinal acceleration of the vehicle, in the vehicle frame (ξ, η, ζ)
- L_w - wheel base
- θ - orientation of the vehicle in the global (x, y, z) coordinate frame

2.4 Testing Concept

Implement function code in function layer, then integrate to application layer. At this stage, record raw data (mcaps) together with vehicle and controller integration layers. The resulting measurement file can be used for open-loop tests, that is satisfactory except for vehicle control components.

3 Function Specification

This Section describes the high-level specifications of the covered functionality. Autoware architecture:
https://app.diagrams.net/?lightbox=1#Uhttps%3A%2F%
2Fautowarefoundation.github.io%2Fautoware-documentation%
2Fmain%2Fdesign%2Fautoware-architecture%2Fnode-diagram%
2Foverall-node-diagram-autoware-universe.drawio.svg

3.1 Intelligent Speed Adjustment

- Step 1 functionality: longitudinal speed control adjusted to static information, such as curve and local regulations (speed limit).
- Step 2 functionality: step 1 + speed adjustment on dynamic information, such as moving objects (e.g., followed vehicle).

For both: speed range is $0.0 \leq v_x \leq 150.0kph$, which therefore includes automatic start/stop functionality. Function is illustrated in Figure 1.

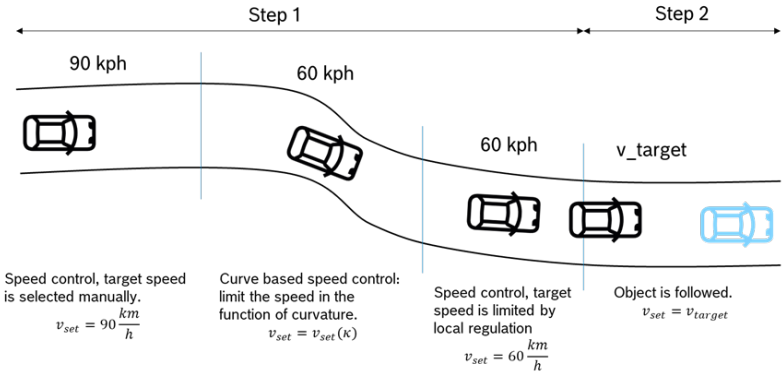

Fig. 1: Function illustration, both step 1 and step 2 functionality.

3.2 Longitudinal Emergency Function

Functionality: vehicle or delegated sensors provide information about static / dynamic objects. The function decides proper strategy to stop the vehicle (and where to stop it). Then, this strategy is accomplished by applying proper braking force. Function use cases are shown in Figure 2. Operation range: $0.0 \leq v_x \leq 150.0kph$.

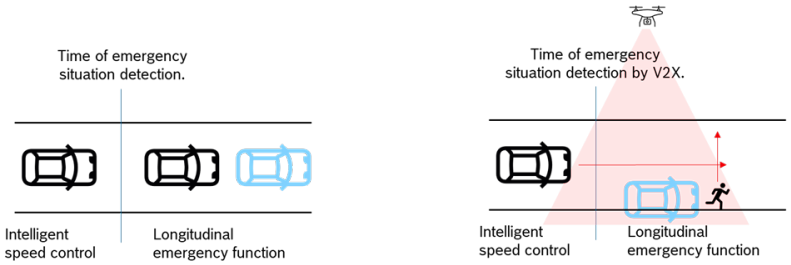

Fig. 2: Longitudinal emergency use cases from vehicle sensors and infrastructure sensors.

3.3 Lane Follow

- Step 1 functionality: vehicle is running in a lane, which is bounded by lane edges (markers or only the edge of the drivable surface) and the vehicle follows the centerline of the lane (or externally defined local trajectory). Operation range: $a_{y,max} = 5m/s^2$, $0.0kph \leq v_x \leq 150.0kph$, $\sigma^2_{e_y} \leq 0.1m$.

- Step 2 functionality: Drivable corridor is shifted due to e.g., temporarily shifted road works, which is bounded by 3D obstacles like cones, walls...etc. Vehicle (with lower dynamics) can still navigate through this drivable corridors. Operation range: $a_{y,max} = 3m/s^2$, $0.0kph \leq v_x \leq 110.0kph$, $\sigma^2_{e_y)} \leq 0.1m$.

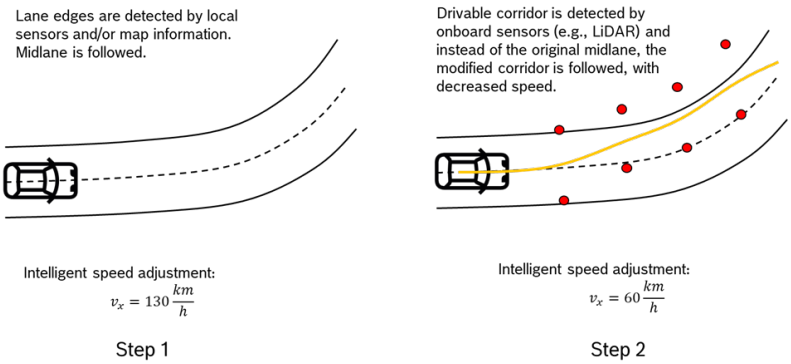

Fig. 3: Lane follow functionality steps and covered operation.

4 Architecture Specification

4.1 Comprehensive notes

The architecture is defined based on the E2E function specifications. There are two main concepts that must be considered at all time:

- fulfill E2E function requirements with the least architecture components,
- re-use Autoware components where possible, but keeping its number (or number/-function) as low as possible.

Therefore, the work model shown in Figure 4 is strongly recommended. This concept may hinder the efficient/reliable planning of tasks regarding architecture definition, but ensures that the above two concepts are respected.

4.2 General Architecture

4.2.1 High-level Black-box architecture

4.2.2 Gray-box architecture

4.2.3 Functional (ROS2) architecture

4.3 Function Architecture Specification

4.3.1 Intelligent Speed Adjustment

Step 1 architecture is shown in Figure 8. Note: even if we only control the vehicle

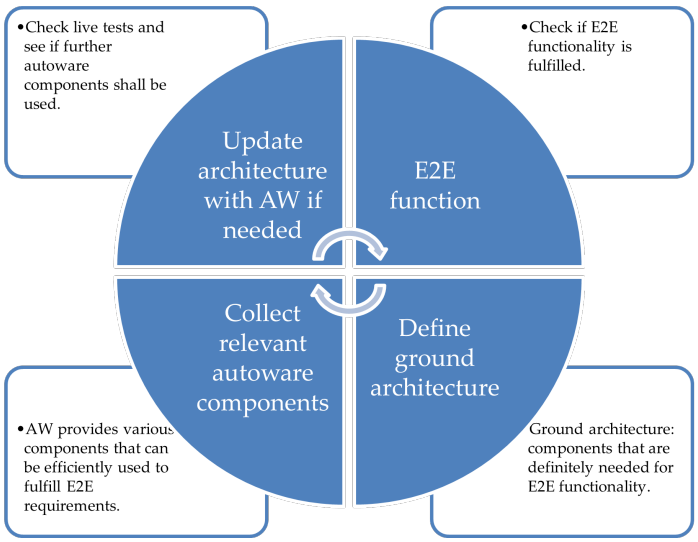

Fig. 4: Working cycle - proposal

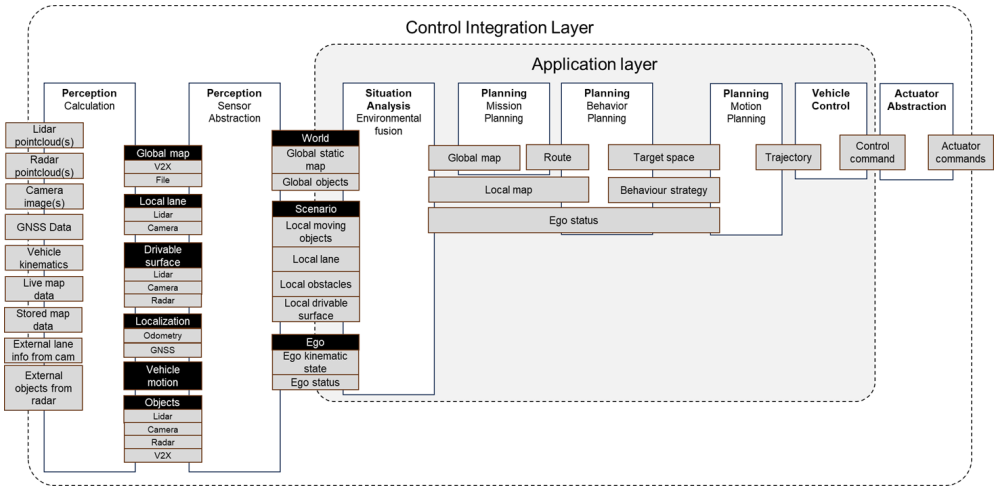

Fig. 5: Black box architecture

longitudinally, the lateral path shall be filled with dummy values. Idea: add a straight line with no offset. Later, it must be solved that the vehicle is longitudinally controlled by the system, but laterally by the driver.
Step 2 architecture is shown in Figure 9.

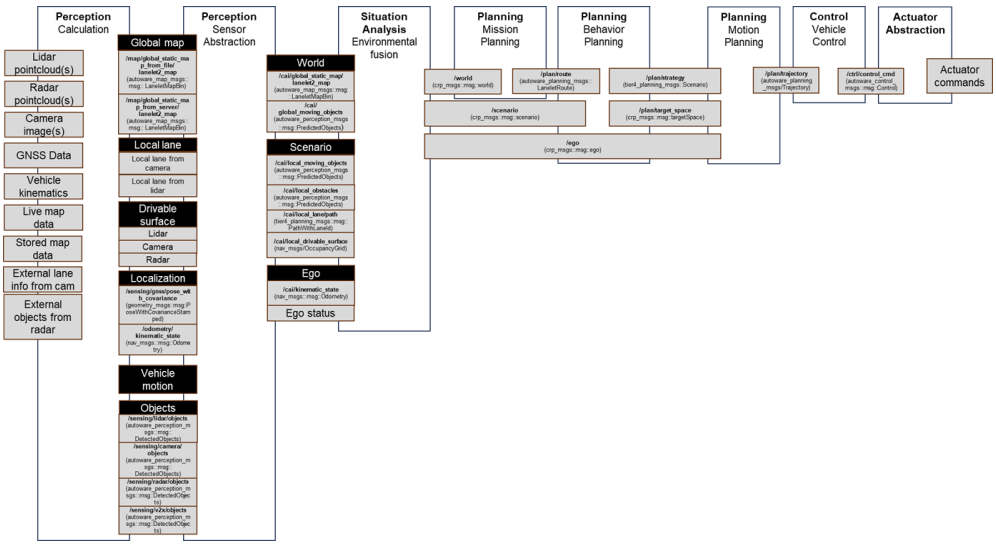

Fig. 6: Gray box architecture.

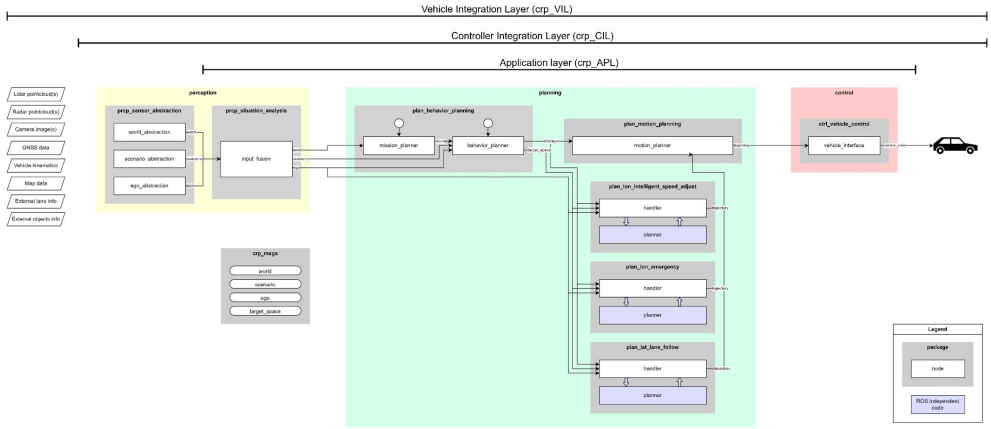

Fig. 7: Functional architecture

4.3.2 Longitudinal emergency function

Based on distributed sensor data calculate the trigger of the emergency scenario. Corresponding architecture is shown in Figure 10.

4.3.3 Lane Follow

Lane follow functional architecture is shown in Figure 11. Lane follow architecture consists of new components of path planner, that takes geometry information of the road, and plans a smooth (drivable) path, which is then handed over to the lateral

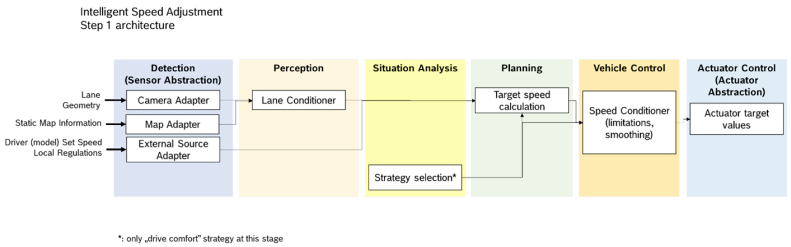

Fig. 8: Architecture components of step 1 functionality of ISA

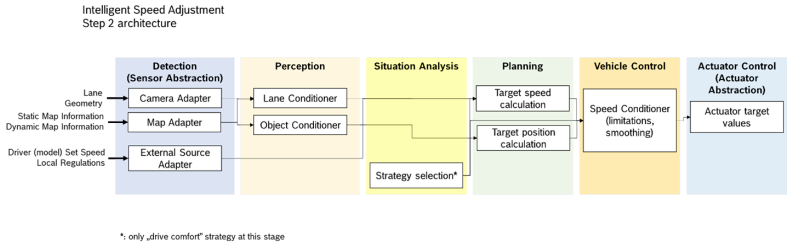

Fig. 9: Architecture components of step 2 functionality of ISA

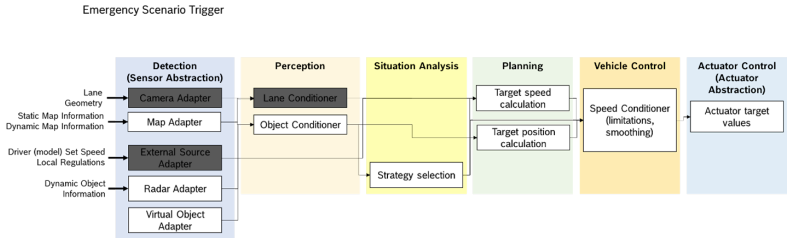

Fig. 10: Architecture components of step 1 longitudinal emergency function.

controller component. This controller controls only the lateral movement of the vehicle, producing output to the actuator control (i.e., steering angle). Note: route input comes from mission planner, which is currently not part of the architecture. During integration process, it may be extended. Step 2 architecture is shown in Figure 12.

5 Message Definitions

5.1 CRP messages

5.1.1 crp_msgs/msg/scenario

This interface holds information of four main types:

- local moving objects: highest layer which is associated with other objects that move around the ego vehicle, such as other vehicles, pedestrians, animals...etc.

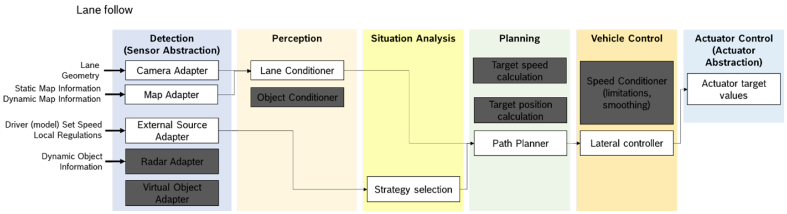

Fig. 11: Architecture components of step 1 functionality of LF (lane follower) without behaviour layer.

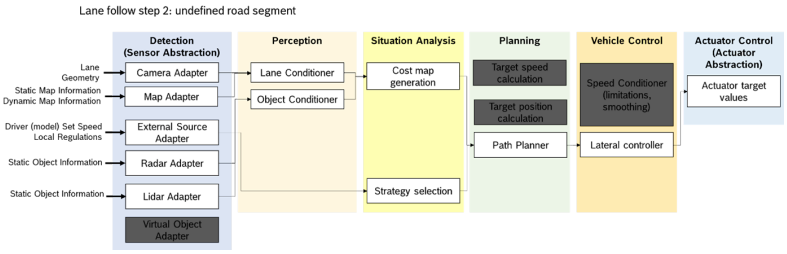

Fig. 12: Architecture components of step 2 functionality of LF (lane follower) without behaviour layer.

- local obstacles: static items that are located around the ego vehicle.
- local lanes: the lanes that are mainly marked by painted markers and form the static driving corridors.
- local drivable surface: the most indefinite representation of the local environment, in the form of a generic occupancy grid.

This interface type (in contrast to globally defined 'world' interface) must contain data with as high accuracy as possible. Message definition:

```
std_msgs/Header header

autoware_perception_msgs/PredictedObject[] local_moving_objects
autoware_perception_msgs/PredictedObject[] local_obstacles
tier4_planning_msgs/PathWithLaneId[] lanes
// traffic rules information to be added
nav_msgs/OccupancyGrid free_space
std_msgs/Float32 maximum_speed
```

Note: the traffic rules collect all types of information that are coming from the static rules and can impact the selected bahviour. These are like:

- stop lines (stop signs and lines)
- speed limitation
- traffic light information (semi-static)...etc.

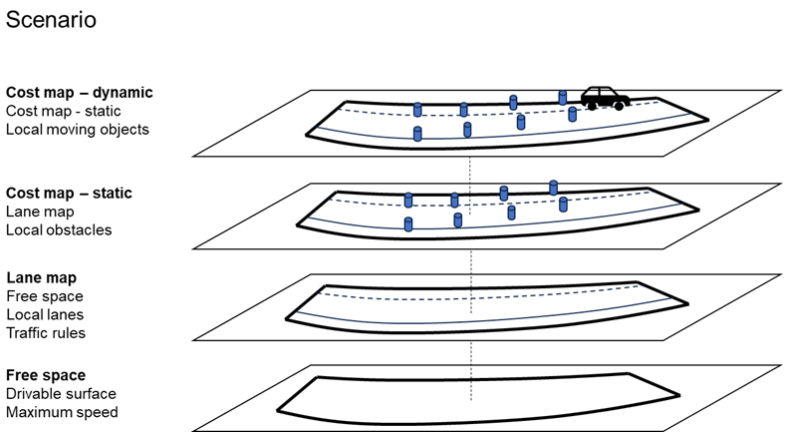

Fig. 13: Illustration of the scenario layers.

5.1.2 crp_msgs/msg/world

5.1.3 crp_msgs/msg/ego

This interface contains every relevant information about the ego vehicle:

- pose: position, orientations and uncertainty
- velocity: linear and angular
- acceleration: linear and angular
- wheel angle

Message definition:

```
std_msgs/Header header

geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
geometry_msgs/AccelWithCovariance accel
float32 wheel_angle
```

5.2 Tier4 messages in use

5.2.1 Path

This is a tier4 autoware message extension, with the following definition:

```
std_msgs/Header header

tier4_planning_msgs/PathPoint[] points
nav_msgs/OccupancyGrid drivable_area
```

5.2.2 Path point

```
uint8 REFERENCE=0
uint8 FIXED=1
geometry_msgs/Pose pose
geometry_msgs/Twist twist
uint8 type
```

5.2.3 Trajectory

```
std_msgs/Header header
tier4_planning_msgs/TrajectoryPoint[] points
```

5.2.4 Trajectory point

```
geometry_msgs/Pose pose
geometry_msgs/Twist twist
geometry_msgs/Accel accel
```

6 Coding rules

6.1 Naming convention

- Use camel case everywhere if not stated otherwise - example: *latLaneFollow*
- Classes and structs should start with upper case letter - example: *MotionHandler*
- Methods should be camel case (starting with small case) - example: *scenarioCallback*
- Method arguments have no prefix/suffix, they must be given with camel case - example: *plannerInput*
- Constant variables should be all capital letters. Instead of camel case the words should be separated with '_' - example: *MAX_SPEED*
- In classes all member variables should start with the 'm_' prefix - example: *m_vxEgo*
- The runtime calibratable parameters should start with the 'p_' prefix - example: *p_mainThreshold*
- In special cases (e.g., subscribers/publishers) extra name tags can be added with lower case, separated by underscore - example: *m_sub_strategy_*
- Pointers should comply with the previous rules but should have a '_' suffix - example: *m_sub_strategy_*
- Namespaces should be fully lower case - example: *crp*
- Maximum namespace depth should be 3
- File and folder names should be camel case where it is not restricted (e.g. ROS2 naming rules)

6.2 Cplusplus

6.2.1 General

- Header and source files should be separated ('include' folder for headers and 'src' folder for source files)
- Header files should only contain declarations, the definitions should be in a source file that includes the header
- If applicable, include further header files in the main header file, not in the source file (source file (i.e., cpp) should only include its own declaration header)
- Functional code should be included in a separate cmake file (with .cmake extension), which is then included in the main CMakeList.txt (this way, cmake file with functional sources is ROS agnostic)

6.2.2 Header files

Every header file should...

- have the '.hpp' extension,
- have header guards (#ifndef, #define, #endif) and it should be all capitals,
- contain max. one class.

6.2.3 Source files

Every source file should...

- have the ".cpp" extension.

6.3 ROS2

6.3.1 Packages

Non-driver package names should start with and abbreviation of the following categories:

- prcp (perception)
- plan (planning)
- ctrl (control)

All dependencies must be set in the package.xml and in the CMakeList.txt.

6.4 Python

6.4.1 General

- Python files should have the '.py' extension.
- In classes private variables should start with the '__' prefix - example: *__m_vxEgo*
- In classes protected variables should start with the '_' prefix - example: *_m_vxEgo*
- Methods and classes should have 2 empty lines after the definition

7 Package documentations

7.1 pacmod_extender

7.1.1 Purpose

The purpose of this package is to extend the default pacmod capabilities on the Lexus vehicle by decoding CAN messages or calculating new data from the inputs. The package is designed to work seamlessly with the already existing pacmod3 system.

7.1.2 Usage

The package can be used by running the executable node. This way it uses the default namespace for subscriptions and publishers:

```
ros2 run pacmod_extender pacmod_extender_node
```

The other way is to use the launcher. This launcher is tailored for the Lexus vehicle by giving the executable the necessary namespace to match the other components:

```
ros2 launch pacmod_extender pacmod_extender.launch.py
```

7.1.3 IO

Input topics are given in Table 1, outputs are given in Table 2.

Table 1: Pacmod extender inputs.

Data	Message name	Message Type
Raw can data	pacmod/can_tx	can_msgs/msg /Frame
Vehicle test	vehicle_status	geometry_msgs/msg/TwistStamped

Table 2: Pacmod extender output.

Data	Message name	Message Type
Linear acceleration	pacmod/linear_accel_rpt	pacmod3_msgs/msg/LinearAccelRpt
Calculated yaw rate	pacmod/yaw_rate_calc_rpt	Pacmod3_msgs/msg/YawRateRpt

7.1.4 Inner workings

The main functionality is the decoding of previously not used CAN messages. The decodings are defined in the PacmodDefinitions class. Every message has a decode method that requires the CAN message as parameter. The message IDs are stored as

constants in the class. The PacmodExtender class is the main class that is executed as a node. It subscribes to the inputs, uses the PacmodDefinitions class to decode the CAN messages and outputs the new messages. The output rate of every message depends on the input frequencies. Every output value is in SI units. Decodings:

- Linear acceleration
- longitudinal, lateral, vertical acceleration in m/s

Calculations:

- Yaw rate $\psi = v_\xi tan(\frac{\alpha_f}{L_w})$, where ψ is the yaw rate, α_f is the front road-wheel angle and L_w is the wheelbase.