

TeamMate System Architecture including open API for 2nd Cycle

Project Number:	690705
Classification	
Deliverable No.:	D.5.1
Work Package(s):	WP5
Document Version:	8.1
Issue Date:	31/10/2018
Document Timescale:	
Start of the Document:	11/07/2018
Final version due:	31/10/2018
Compiled by:	Mohamed Cherif RAHAL
Authors:	Mohamed Cherif RAHAL (VED) Steve Pechberti (VED) Stefan Suck (OFF) Mark Eilers (HMT) Elisa Landini (REL) Fabio Tango (CRF) Adam Knapp (BIT) Lynda Halit (VED) Daniel Twumasi (HMT) Alain Giralt (CAF)
Technical Approval:	Fabio Tango (CRF)
Issue Authorisation:	Andreas Lüdtke, (OFF)

© All rights reserved by AutoMate consortium

This document is supplied by the specific AutoMate work package quoted above on the express condition that it is treated as confidential to those specifically mentioned on the distribution list. No use may be made thereof other than expressly authorised by the AutoMate Project Board.



DISTRIBUTION LIST		
Copy type ¹	Company and Location	Recipient
T	AutoMate Consortium	all AutoMate Partners

¹ Copy types: E=Email, C=Controlled copy (paper), D=electronic copy on Disk or other medium, T=Team site (Sharepoint)

Content table

List of Figures	5
List of Tables	6
List of Abbreviations	7
Executive Summary	8
1 Introduction	9
2 Global overview of the TeamMate system	10
2.1 The TeamMate concept of cooperation	10
2.2 The enablers	14
3 TeamMate functional Architecture	20
3.1 Information exchange between enablers	23
3.2 Dataflow diagram and Enablers interconnection	23
4 Data flow, data structures	26
4.1 Standards	26
4.1.1 Time synchronization	27
4.1.2 Coordinate systems	27
4.2 Data Structures	27
4.2.1 Pose and motion of the TeamMate vehicle	27
4.2.2 Pose, Motion, and dimension of detected objects	28
4.2.3 Semantic annotation of detected objects	29
4.2.4 Prediction of the spatial and temporal evolution of detected objects	30
4.2.5 Digital Maps	30
4.2.6 Safety corridors	30
4.3 Data interface	32
4.3.1 Environment model	32
4.3.2 Static environment model	32
4.3.3 Dynamic environment model	33
4.3.4 Evolution of the traffic scene	33
4.3.5 Safety corridor	34
4.3.6 Planned Trajectory	35
4.3.7 Driver's state	36
4.3.8 Online Learning from the Driver	39
4.3.9 Component Communication Framework	40
5 The TemMate API for the second cycle	42
5.1 Commuication of the TeamMate car with its envirennement: the V2X related standards	52



5.1.1	ETSI TC ITS V2X Reference Architecture	53
5.1.2	Decentralized environmental notification message	54
5.2	Third party HMI SDK specification	56
5.2.1	DQuid SDK definition	56
6	Conclusion	61
7	References.....	62
Appendix 1.....		63
Example of JAVA code generation		63
Example of C++ code generation		64

List of Figures

Figure 1: State machine that shows how the cooperation is implemented ..	13
Figure 2: State diagram of AutoMate Human Machine Cooperation.....	13
Figure 3 : Sketch of the intended overall TeamMate System architecture ...	20
Figure 4 : Sketch of the model-based architecture of decision modules.....	21
Figure 5 :Sketch of manoeuvre planning architecture	22
Figure 6 : Sketch of TeamMate-HMI architecture	23
Figure 7: Current AutoMate system architecture	24
Figure 8: Exemplary realization of a network of components.	41
Figure 9: Example of communication within the component communication framework.	42
Figure 10. ETSI TC ITS reference architecture	52
Figure 11. General structure of DENM [2]	54
Figure 12: DQuid SDK high-level architecture.....	57
Figure 13: Write the DQuidObject's properties	58
Figure 14: Subscribe/update the DQuidObject's properties.....	58
Figure 15: Example of the structure of the properties in JSON	60

List of Tables

Table 1: different types of alternate controls.	11
Table 2: How the enablers in all WPs contribute to cooperation.....	16
Table 3: The EgoState data structure.	27
Table 4: The Object data structure.....	28
Table 5: The ObjectAnnotation data structure.....	29
Table 6: The ObjectTrack data structure	30
Table 7: The 2DPoint data structure.....	31
Table 8: The Polyline data structure.....	31
Table 9: The ObjectPolyline data structure.....	31
Table 10: The SafetyCorridor data structure.....	31
Table 11: Content of the message defining the static environment model. .	32
Table 12: Content of the message defining the dynamic environment model.	33
Table 13: Content of the message defining the semantic prediction.....	33
Table 14: Content of the message defining the probabilistic prediction	34
Table 15: Content of the message defining the probabilistic prediction	35
Table 16: Content of the message defining the planned trajectory.	35
Table 17: Content of the Drowsiness message.....	36
Table 18: Content of the Visual Attention Fast message.....	37
Table 19: Content of the Visual Attention slow message	38
Table 20: Content of the Cognitive Distraction message	38
Table 21: Content of the driver's-state Raw-data message.....	39
Table 22: Online learning from the driver.....	39



List of Abbreviations

A2H : Automation to Human.

AM : Automation Mode.

API : Application Programming Interface.

CAM : Cooperative Awareness Message.

CSM : Control Sharing Mode.

DENM : Decentralized Environmental Notification Message.

DIR : Driver intention recognition

H2A : Human to Automation.

HMI : Human Machine Interface.

MM : Manual Mode.

SDK : Software Development Kit.

SMM : Safe Manoeuvre Mode.

TCP : Transmission Control Protocol.

TOR : take-over request.

UDP : User Datagram Protocol.

V2X : Véhicule to all communication.

Executive Summary

This deliverable D5.1 presents the global AutoMate System architecture and the adapted architectures for each demonstrator, making sure that the architecture works on all demonstrators of the AutoMate project, considering their different initial starting point. An explanation of dataflow, within the SW/HW construct and a clarification of stateflow concerning the teammate car, is also provided, together with protocols for communication.

Another important part is the definition of interfaces between the modules, as well as the common data formats standards and communication protocols. Therefore, the TeamMate Application Programming Interface (API) is defined in terms of principles, standards, interfaces, and data structure that enables the communication of information between components in the TeamMate ecosystem, based on exchanging messages in a publisher-subscriber messaging patterns. The goal is to have common data formats for interfacing the modules. We provided a first definition of a set of such data structures. However, it is worth to note that the definition is non-exhaustive and may be subject to change if the need arises during integration and advances in the development of enablers for the TeamMate demonstrators.

Finally in this document, the software modules are specified and dedicated to their corresponding enablers. Moreover, the teammate cooperation modes are explained, including the concept of Automation to Human (A2H), as well as Human to Automation (H2A) communication.



1 Introduction

This document presents the global AutoMate System architecture. The purpose of the Automate project is to build a complete software/hardware concept for the teammate car. Therefore, each module has to be developed and programmed and finally all modules have to be put together. Due to the number and complexity of these modules, the composition can become a complex task and needs to be well coordinated. Also one needs to make sure, that the architecture works on all demonstrators of the Automate project considering their different initial architectures. For this reason WP5 deals only with the issues mentioned above and this document presents the global teammate architecture, as well as the adapted architectures for each demonstrator. Another important part is the definition of interfaces between the modules, as well as common data formats standards and communication protocols.

Therefore, the TeamMate Application Programming Interface (API) is defined in terms of principles, standards, interfaces, and data structure that enables the communication of information between components in the TeamMate ecosystem. In the following, we will present a common design principle for the communication between components in the TeamMate ecosystem, based on exchanging messages in a publisher-subscriber messaging patterns. Messages will be defined in terms of data structures with fixed semantics. We provide a first definition of a set of such data structures. We note that the definition is non-exhaustive and may be subject to change if the need arises during integration and advances in the development of enablers for the TeamMate demonstrators.

Thus, in this document the above mentioned will be elaborated in more detail. In section 2 the software modules are specified and dedicated to their corresponding enablers and the teammate cooperation modes are specified. Also the concept of Automation to human (A2H), as well as human to automation (H2A) communication is explained. Section 3 contains the Teammate global teammate architecture concept, an explanation of dataflow within the software construct and a clarification of stateflow concerning the teammate car. Also protocols for communication are defined. Subsequently in section 4 the API is defined in order to have common data formats for interfacing the modules.

2



3 Global overview of the TeamMate system

3.1 The TeamMate concept of cooperation

As pointed out by C. Sentouh and colleagues², the human-machine cooperation is a challenging problem since the introduction of automated systems in the various fields of human activity, especially in the aviation and automotive fields. According to Piaget (1977)³, we can assume the following definition of cooperation: "Cooperate in action is to operate in common, that is to say, adjust with new operations, the operations performed by each partner, it's coordinate the operations of each partner in a single operating system in which the acts themselves of collaboration constitute the integral operations". This leads us to the following questions, as pointed out by Hoc and colleagues⁴:

- When to intervene to assist the driver?
- How to do it and at what degree?
- What the effect will this intervention on the driver?
- Finally, whom assign responsibility for the driving?

Sheridan⁵ gave the definition of "Sharing control" where the human operator and machine work together, simultaneously, to make or perform a task. There can be also other possible definitions for alternate control, illustrated in the following table, where one of the two agents is responsible of a function, and either the human operator or the machine performs the function from time to time (a change of active agent):

² Chouki Sentouh, Jean-Christophe Popieul, Serge Debernard, Serge Boverie. Human-Machine Interaction in Automated Vehicle: The ABV Project.

³ J. Piaget. "Etudes sociologiques" (3e ´ed). Geneve: Droz, 1977.

⁴ J.M. Hoc. Towards a cognitive approach to human-machine cooperation in dynamic situations. International Journal of Human-Computer Studies, Volume 54, Issue 4, April 2001, pp.509-540.

⁵ T.B. Sheridan. Telerobotics, automation, and human supervisory control. Cambridge, MA: MIT Press, 1992.




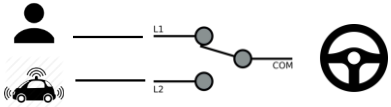
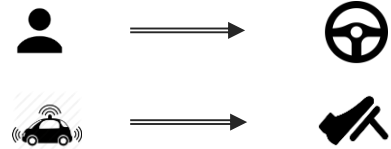
Shared Control	Working in the same task at the same time	
Traded Control	Working in the same task at different time	
Cooperative Control	Working in different tasks at the same time	

Table 1: different types of alternate controls.

Besides of this approach, there are also many experiments carried out on the full automation of driving, including presentations of the Google Driverless Car in 2010⁶ and VisLab Driverless Car in 2013⁷. These experiments aim to demonstrate the full automation, where the driver is completely out of the driving task. However, as highlighted by many works⁸, there can be the paradox for which, when the autonomous system reaches its limits, it requires to the human to take back the vehicle control (exactly the same human agent regarded as “a problem” up to that time and completely out-of-the-loop).

The top-level objective of AutoMate is to develop, evaluate and demonstrate the “TeamMate Car” concept as a major enabler of highly automated vehicles. This concept consists of considering the driver and the automation as members of one team that understand and support each other in pursuing cooperatively the goal of driving safely, efficiently and comfortably from A to B. As a consequence, in order to show how the enablers contribute to the implementation of this concept, it is important to briefly explain why the cooperation is needed, and how the human and the automation can support

⁶ J. Markoff. Google Cars Drive Themselves, in Traffic. The New York Times, 2010.

⁷ PROUD Car Test 2013. <http://vislab.it/vislab-events-2/>.

⁸ L. Bainbridge. Ironies of automation. Automatica, vol. 19, no. 6, pp. 775-779, 1983.



each other to create a safe, efficient and comfortable driving experience. Therefore, the future generations of driver assistance systems (ADAS) and autonomous functions (ADFs) must be developed to ensure a smooth action of the controller continuously, while keeping the driver in-the-loop without generating negative interference.

The AutoMate approach is based on the mutual complementarity between the driver and the automation: this support is achieved through the cooperation between the team members.

The cooperation is bidirectional: while the Automation to Human Cooperation (A2H) is used to complement the human limits, the Human to Automation Cooperation (H2A) is implemented to allow the driver to support the automation to overcome its limits.

According to AutoMate concept, the cooperation is made of two types of support: in perception and in action.

The complementarity between the driver and the automation is the conceptual solution to compensate the reciprocal limitations, while the cooperation is how the complementarity is implemented. In this context, AutoMate project has integrated the problem of interaction with the driver, in the design process of the system, by considering the task sharing and degree of freedom, authority, level of automation and Human- Machine Interface (HMI).

Figure 1 shows how both the A2H and the H2A cooperation between Manual mode and Automatic mode can be implemented in perception (state A and B) and in action (state C and D).

Figure 2 shows the detailed implementation of this cooperation including a Control Sharing mode.

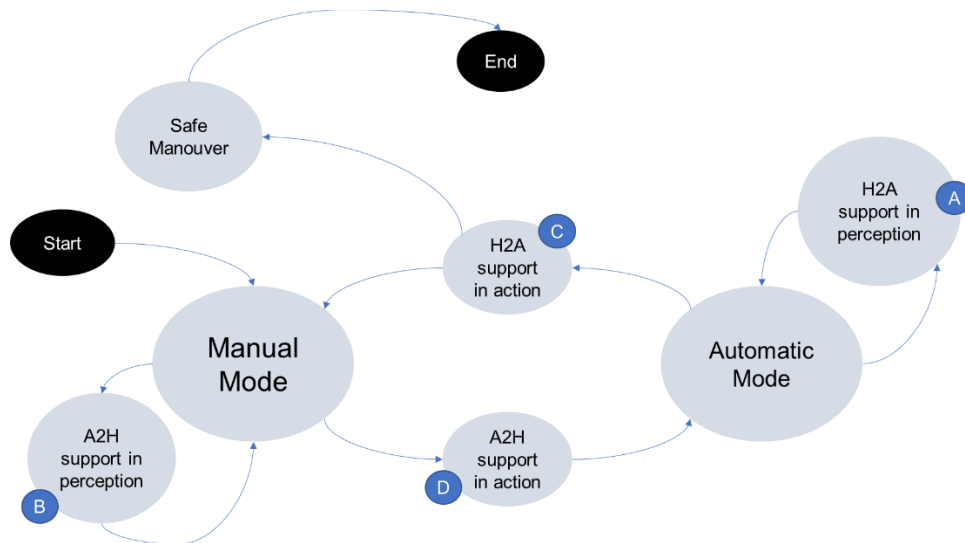


Figure 1: State machine that shows how the cooperation is implemented

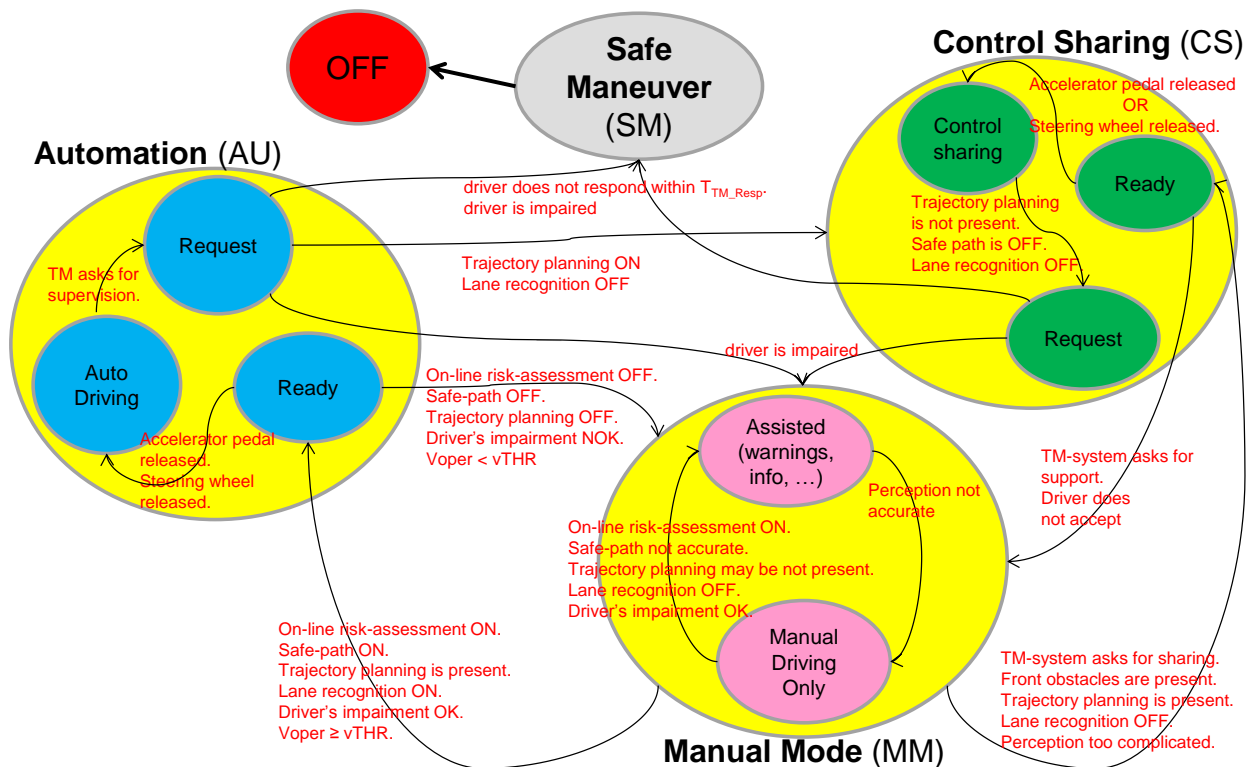


Figure 2: State diagram of AutoMate Human Machine Cooperation.

With reference to the figure, there are four main states:

1. Manual Mode (MM)



2. Control Sharing (CS)
3. Automation (AU)
4. Safe Manoeuvre (SM)

In MM, the driver is fully responsible of both lateral and longitudinal driving tasks, even if the machine-agent can provide assistance in terms of warning and information. On the other way around, in the AM, the system is fully responsible both lateral and longitudinal driving tasks (full automation); however, the machine-agent can require the intervention of the human-agent (take-over request, TOR in short), using appropriate strategies to take him/her into the control-loop again, depending on the cognitive status. In the CSM, the two types of control are separated: typically, the longitudinal task is under the system responsibility, while the lateral task under the driver control. Eventually, the last mode is an emergency shutdown (SMM), in which a minimum risk manoeuvre is foreseen, by stopping it in an automatic manner in case the driver is not responding to a TOR in order to preserve the safety of the vehicle and its passengers.

In each of these four modes, different sub-modes have been defined, whose transitions from one to another are represented by arcs, characterised by different parameters, related to the vehicle (e.g. speed), to the external situation (e.g. end secured path), to the driver status (e.g. drowsiness) and finally to a controller state (e.g. system OK). The different transitions between the block are reported in red in the figure on those arcs.

Every time the system starts in the MM block. When the driver wants more support from the automation, s/he can make this request and, if the conditions of transition are satisfied, the system can enter the AM block. If not, but there are the conditions of a shared control, then the system goes into the CSM block. Of course, the situation is highly dynamic, depending on the external environment, as well as on the status of the driver and of the system.

3.2 The enablers

The common global TeamMate architecture can be considered as a framework that allows to understand how the enablers, integrated into the different demonstrators, have a crucial role in the concrete implementation of the concept and in the achievement of the TeamMate features.

This chapter describes the role of the enablers as a means to implement the concept of cooperation.



The enablers have different roles, which can be divided into three categories:

- **Enablers of automation**, i.e. Automated Driving Functions (ADFs) to allow an effective automated driving - Enabler 4 (Adaptive driving Manoeuvre Planning, Execution and Learning) and Enabler 5 (Online Risk Assessment);
- **Enablers of adaptive automation**, i.e. systems and technologies designed to allow a tailored and adaptive driving experience – Enabler 1 (Sensor and communication platform), Enabler 2 (Probabilistic Driver Modelling and Learning), and Enabler 3 (Probabilistic Vehicle and Situation Modelling);
- **Enablers of cooperation**, e.g. the systems that allow the cooperation between the technological and the human agents – Enabler 6 (TeamMate HMI)

Moreover, all enablers support the cooperation in one or both aforementioned directions, i.e. A2H and H2A.

Table 2 shows the role and relevance of each enabler in the cooperation. The role of the enablers developed in the different WP, and how they contribute to the cooperation when integrated in the system architecture, is reported in the table. These roles, also reported in D2.4, D3.5 and D4.4, have been merged into a single table to understand the relationship between the enablers into the TeamMate architecture.

Table 2: How the enablers in all WPs contribute to cooperation

WP	ID	Enabler	Enabler Owner	Aim of the enabler	Direction of support	
					Automation to Human	Human to Automation
WP2	Enabler 1: Sensor and communication platform					
	E1.1	Driver monitoring system with driver state model for distraction and drowsiness	CAF	Sensors and models to detect driver’s visual distraction and drowsiness detection and classification	Enabler E1.1 is needed to implement a support in perception to complement the perception of the driver about the his/her state	
	E1.2	V2X communication	BIT	To Allow the communication between the vehicle and everything.	Enabler E1.2 is needed to implement a support in perception to complement the perception of the driver about the environment	
	Enabler 2: Probabilistic Driver Modelling and Learning					
	E2.1	Driver intention recognition	OFF	To Classify the current driver state, describe the interdependencies between the driver’s state, type, behaviour and environment and predict the driver intention	Enabler E2.1 is needed to implement a support in perception to complement the perception of the driver about his/her state	
WP3	Enabler 3: Probabilistic Vehicle and Situation Modelling					
	E3.1	Situation and vehicle model	DLR OFF	To estimate the dynamic vehicle and	Enabler E3.1 is needed to	



				object state and position	implement a support in perception to complement the perception of the driver about the situation and the vehicle	
E3.2	Driving task Model	DLR		To define the driver's tasks to understand the expected behaviour (Paper Enabler)	Enabler E3.2 is needed to implement a support in action along with E6.1 (Interaction Strategy) to provide the driver with an effective means to interact with the automation in case of need.	
Enabler 4: Adaptive driving Manoeuvre Planning, Execution and Learning						
E4.1	Planning and execution of safe manoeuvre	ULM VED		To plan the possible manoeuvres and select the most effective, efficient and comfortable.	Enabler E4.1 is needed to implement a support in action to complement the ability of the driver to intervene in case of risk	



	E4.2	Learning of intention from the driver	OFF HMT	To learns the driver's intention to predict the expected behaviour	Enabler E4.2 is needed to implement a support in perception to complement the ability of the driver to assess the risk in case of risky behaviour	
Enabler 5: Online Risk Assessment						
	E5.1	Online risk assessment	OFF DLR HMT	To define a safety zone where the vehicle is not likely to collide either with obstacles or other vehicles (according to the prediction of their future position).	Enabler E5.1 is needed to implement a support in perception to complement the ability of the driver to assess the risk	
Enabler 6: TeamMate HMI						
WP4	E6.1	Interaction Modality	ULM	To define the best way to allow the driver to provide feedback to the HMI		In perception and in action (negotiation-based HMI) to allow the driver to answer the request of support of the automation
	E6.2	TeamMate multimodal HMI	REL	To show information on different device and through different sebsorial channels, in	In perception and in action (warning-based HMI) either to inform	In perception and in action (negotiation-based HMI)



				order to either inform the driver, explain manoeuvres and situations, allow him to interact with the vehicle	the driver about a potential risk or to take the control of the vehicle	to ask the driver either for support in perception or in action
	E6.3	AR	HMT	To show information on the windshield to improve their comprehensibility (for simulators only).	In perception (warning-based HMI) to inform the driver about a potential risk	In perception and in action (negotiation-based HMI) to ask the driver either for support in perception or in action

4 TeamMate functional Architecture

The intended Overall TeamMate functional-Architecture is presented in the DOW as shown in Figure 3.

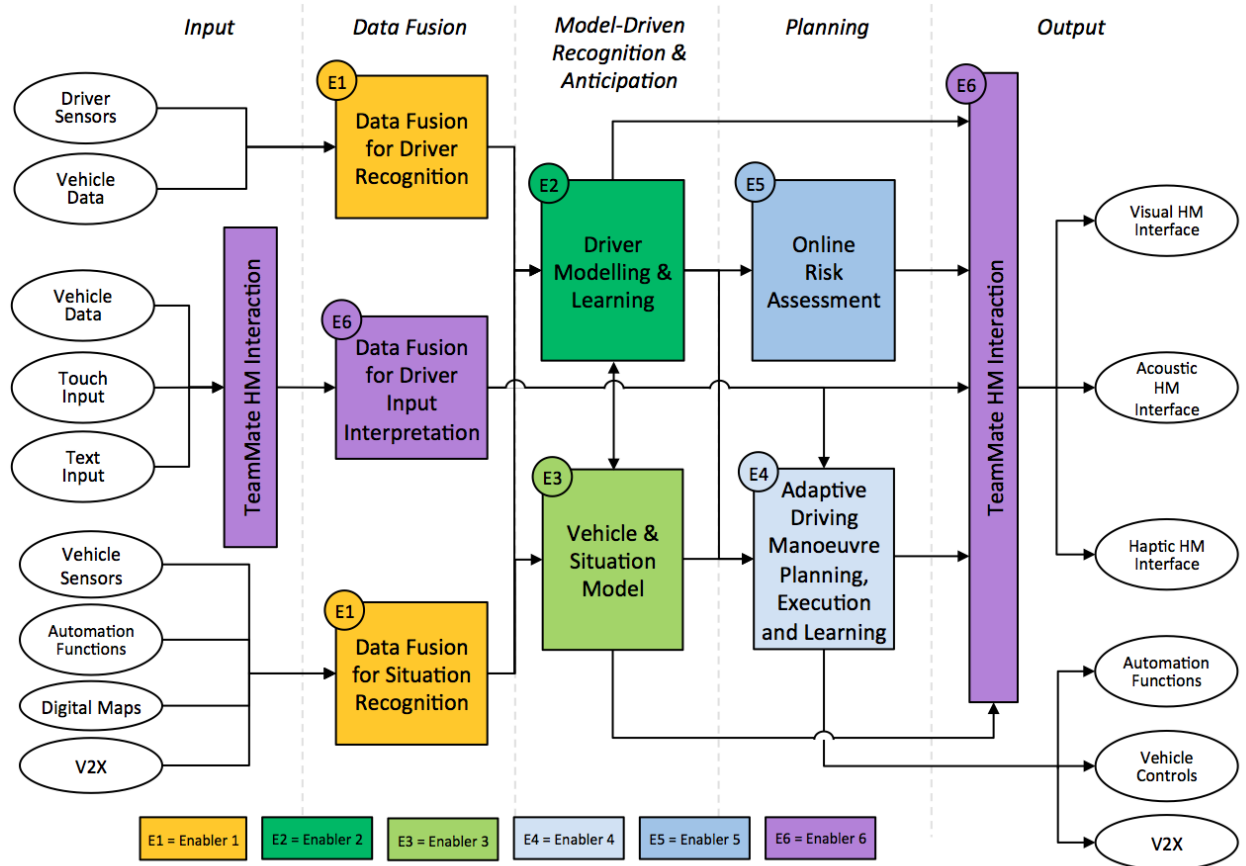


Figure 3 : Sketch of the intended overall TeamMate System architecture

The model-based architecture (illustrated in Figure 4) takes the fused data to interpret and assess the vehicle, traffic situation and driver.

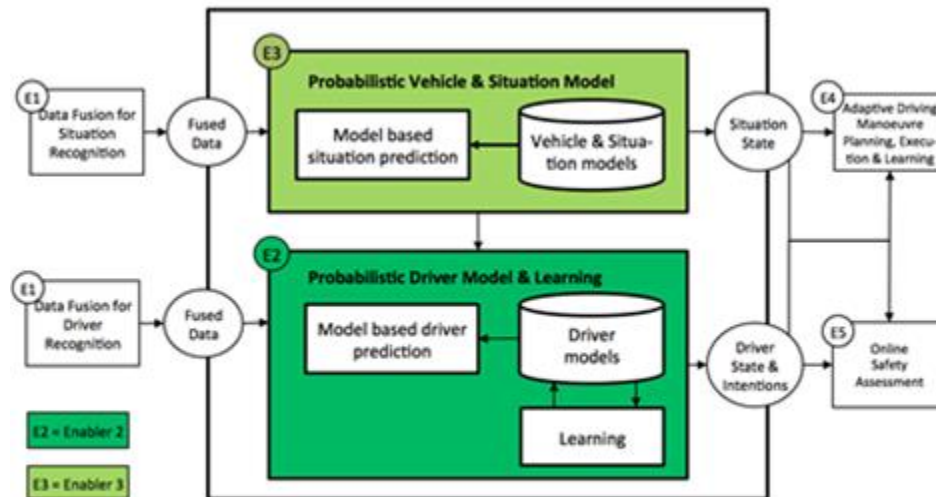


Figure 4 : Sketch of the model-based architecture of decision modules

The vehicle & situation modelling components classifies complex traffic situations and anticipates a sequence of likely spatial and temporal evolutions of the world by using vehicle and situation models. Thereby, it will interact with the driver model to incorporate possible interventions of the human driver. The driver model & learning component will infer driver states and intentions, which includes likely temporal evolutions of states and intentions. As behaviours and preferences vary across drivers, a learning component is included. It uses the inputs the driver makes or the behaviour shown by the driver to update the driver model. This ensures that the driver state prediction is well adapted to the individual driver.

The adaptive driving manoeuvres planning, execution and learning component takes the input, state and intentions of the driver and the current situation state as input.

Based on these data, potential strategic manoeuvres are identified and planned up to a concrete action sequence on an operational level. The resulting plans include a suitable task distribution plan between driver and automation. The manoeuvre planning component includes a learning algorithm that takes the behaviour of the driver and driver's responses to proposed manoeuvre plans as input. Based on this data the component learns and improves manoeuvre plans and recognizes driver's preferences for certain manoeuvre plans. The online risk assessment component defines a safety corridor based on the current state of the driver (e.g. distraction) and the state of the situation (including an anticipated sequence of likely spatial and temporal evolutions). The safety corridors are continuously monitored and updated in order to respond to a critical development of the driver or situation state.

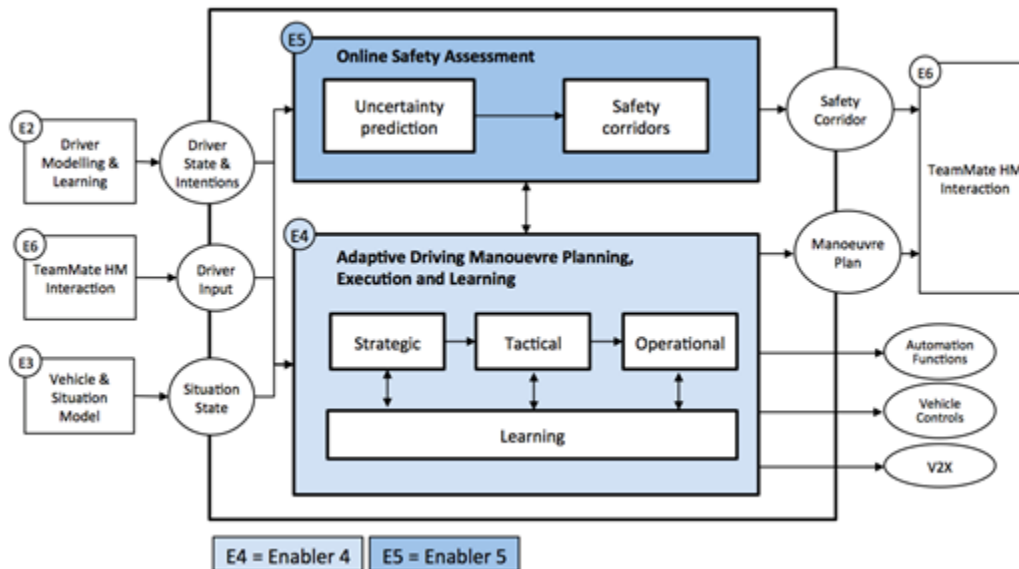


Figure 5 :Sketch of manoeuvre planning architecture

Based on the online risk assessment, the manoeuvre planning component selects suitable manoeuvres to be proposed to the driver. Depending on the planned task sharing the manoeuvre planning component executes the tasks that are assigned to the automation.

The TeamMate HMI provides intuitive bidirectional communication mechanisms between driver and automation. The driver can communicate with the automation via different modalities, e.g., touch input. The TeamMate HMI fuses the input from these modalities and interprets it based on stored personalized, multi-modal communication preferences ("concurrent abbreviations"). The interpreted driver input is then passed on to Enabler 2 and 4. Vice versa, the automation uses the HMI to communicate with the driver via different modalities. For example, the HMI takes the manoeuvre plans proposed by Enabler 4 and suggests these to the driver. It chooses a communication strategy that again relies on stored personalized, multi-modal communication preferences. The HMI does not only suggest this support to the driver, but also selects and communicates information of the current situation and driver state, which enables the driver to understand the rationale behind the suggested support.

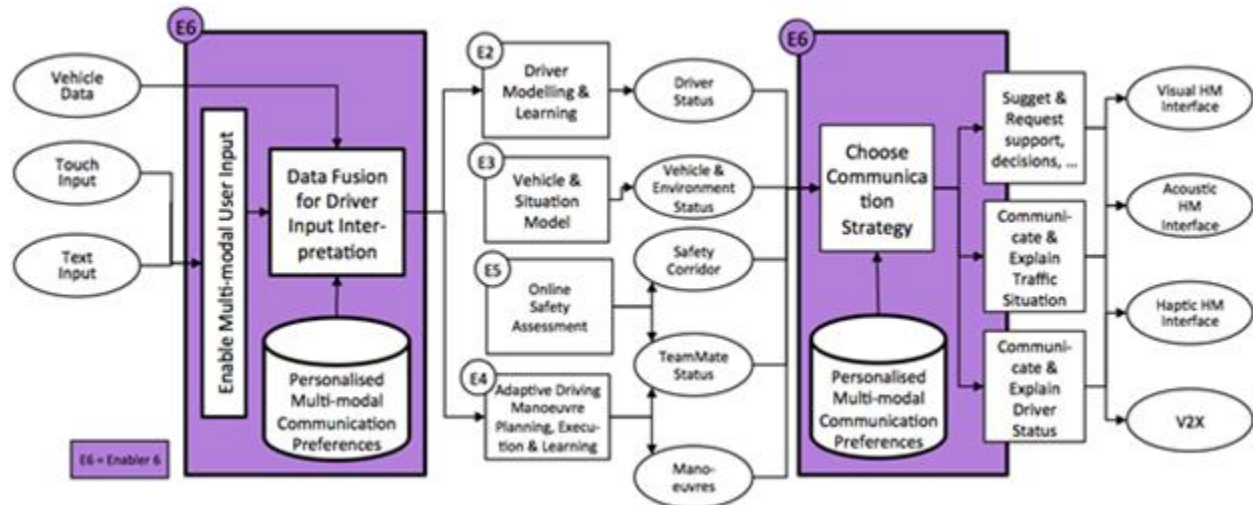


Figure 6 : Sketch of TeamMate-HMI architecture

4.1 Information exchange between enablers

The TeamMate ecosystem will be realized as a distributed application, where the TeamMate components communicate by exchanging messages in a publisher-subscriber messaging pattern. More specifically, the TeamMate ecosystem will be realized as a client-server model. Each component may act as a server that provides services to other components. Simultaneously, each component may act as a client by requesting services from other components. Within the TeamMate ecosystem, services are defined in terms of the communication of messages that encapsulate data structures with fixed semantics. As communication protocol, we plan the use of the Transmission Control Protocol (TCP), or if not feasible, the User Datagram Protocol (UDP). Components may act in a time-triggered or event-triggered mode. In a time-triggered mode, a component uses an internal schedule and timer to request necessary information from other components. The information can be used to proactively prepare the information that will be requested by other components. In an event-triggered mode, a component only requests information from other components to provide a service requested by another component.

A template for the development of components, providing the necessary functionality for communication between components will be prepared for the TeamMate Extension SDK.

4.2 Dataflow diagram and Enablers interconnection

Within AutoMate all the data provided by the sensors, communication, the map layer and vehicle data are put onto a TeamMate communication bus making them available to all other components. From a functional point of view, the sensor platform includes both the sensors themselves but also the data fusion components. The data delivered by all types of sensors will be fused to provide coherent, reliable information about the driver, environment and vehicle. Furthermore, a communication platform will be developed, that connects the components to the vehicle and allows all components to exchange data with each other and to control the vehicle itself.

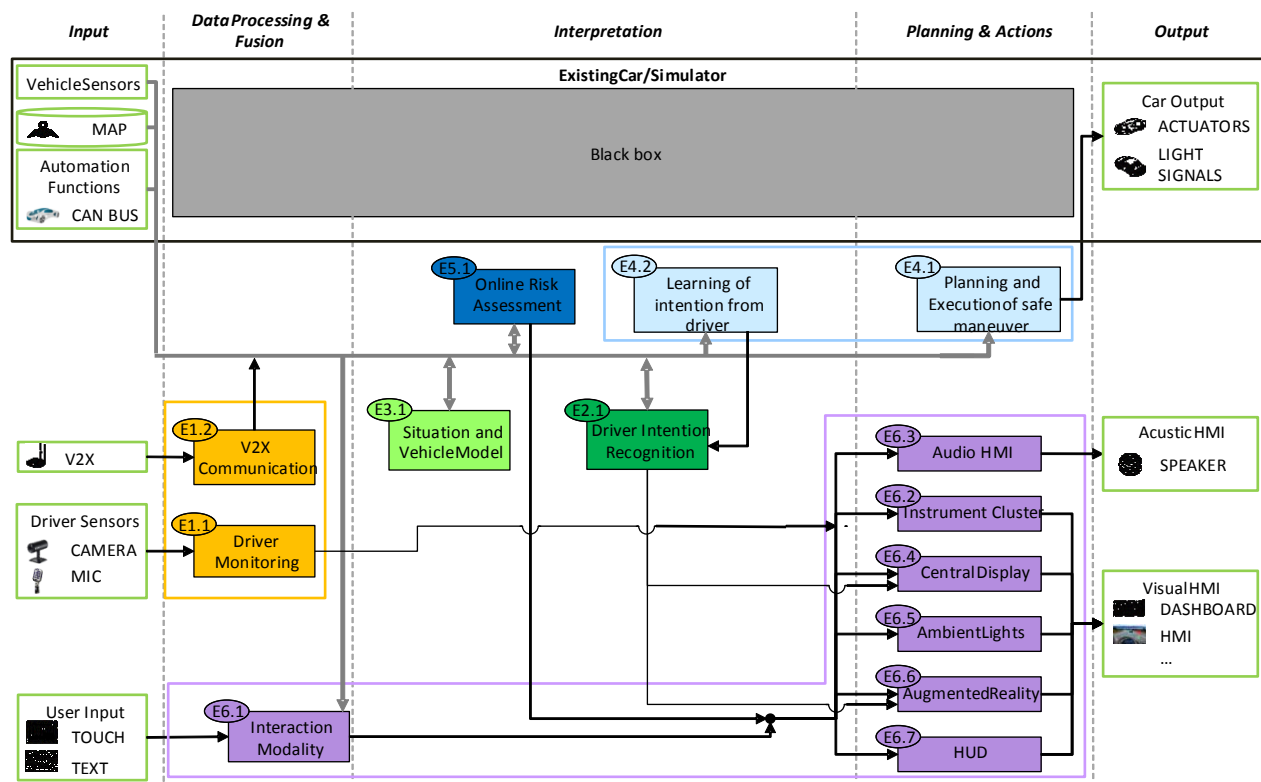


Figure 7: Current AutoMate system architecture

The current AutoMate system architecture is shown in Figure 7: Current AutoMate system architecture. It illustrates the relations of the automate enablers among each other and together with a given platform (vehicle or simulator) during the second cycle. The depicted layout is kept close to the current implementation while staying general enough to be applicable for all demonstrators.

The AutoMate enablers support different functional steps: "data processing & fusion", "interpretation", and "planning & actions" which are represented by the corresponding sections. Each enabler is represented by a software **component dependent on their concerns**. A **message bus oriented** data



exchange between the components is implied to support a communication via one or more channels.

The existing vehicle or simulator may already have modules which also perform one or more of the aforementioned functional steps, but the TeamMate system does not need to know how these internal modules of the platform work or interact. Thus, wide parts of the existing platform are considered as a black box.

However, the simulator or the vehicle has to provide access to certain input and output interfaces in order to enable the AutoMate system to receive necessary data and to deliver processing results or to execute actions. Input data from automation functions, maps, and vehicle sensor are at least expected to be provided by the existing vehicle or simulator. Further inputs introduced by the Teammate architecture are V2X data, driver sensor data, and user input via touch or text interfaces. The TeamMate system can deliver its output via acoustic and visual human-machine interfaces to the driver. Further the existing platform is required to provide certain output interfaces, for example to car actuators and light signals.

Inside the architecture there are several data flows that shall now be described from the perspective of the enablers.

The **Driver State Monitoring (E1.1)** receives its data directly from sensors related to the driver like a camera. The module infers the driver state in terms of drowsiness and attentiveness and provides it as output data, which can be consumed by other enablers. Currently it is used by *HMI enablers (E6.x)* like the Instrument Cluster to trigger messages for the driver, e.g., to suggest a transition to automatic mode.

The **V2X Communication (E1.2)** is directly connected to a V2X data receiver. The received data is interpreted and made available for other enabler via the aforementioned message bus. Currently the information is used to inform the driver via one of the *HMI enablers (E6.x)* about certain conditions on the route which might require a mode transition to manual or shared control.

The **Situation and Vehicle Model (E3.1)** receives its input data, e.g., map, ego vehicle, other vehicle data via the message bus. The output data, an interpretation of the traffic situation and a spatial and temporal prediction of traffic participants within sensor range is then also made available via the bus. Currently it is intended that this data is consumed by the *Online Risk Assessment* and the *Driver Intention Recognition*.

The **Driver Intention Recognition (E2.1)** consumes data from the *Situation and Vehicle Model* and from the car or simulator, which is received via the message bus. The recognized intention probabilities, for example for a lane change, are then returned to the bus and to the *HMI enablers* where the



intention of the driver can be visualized in order to inform the driver that the TeamMate car is aware of his intention.

The **Learning of intention from driver (E4.2)** is closely related to the *Driver Intention Recognition* (DIR). It receives data from the Situation and Vehicle Model and from the car or simulator via the message bus. Additionally it has also access to the driver model storage of the DIR. The enabler updates the parameters of the DIR model based on observed evidence. After an update the new model parameters are stored and the DIR is informed to reload its model to operate with the new parameters.

The **Online Risk Assessment (E5.1)** consumes data from the Situation and Vehicle Model and from the car or simulator. All data is again received via the message bus. The generated output, safety corridors that quantify the safety of the current and near-future traffic situation, is fed to the message bus. Currently this output is consumed by the *Planning and Execution of safe maneuver* in order to plan the trajectory and by *HMI enablers*, which can combine this information with the output of the DIR to warn the driver if a predicted intention would lead to an unsafe maneuver.

The **Planning and Execution of safe maneuver (E4.1)** receives its input data from the car or simulator as well as the interpreted data from the *Online Risk Assessment* via the message bus. It attempts to plan a trajectory which is then provided to the output interface of the simulator or car so that the vehicle can follow the planned path.

The **HMI enablers (6.x)** receive data from the car or the simulator in order to present the current vehicle status. They also consume data from the *VX2 Communication* and the *Driver State Monitoring* and the output of the *Driver Intention Recognition* and the *Online Risk Assessment*. Additionally they may process data from the user input interfaces, e.g., text or touch. The output of the HMI enablers is then directly provided to the driver via the corresponding visual or acoustic interface.

5 Data flow, data structures

5.1 Standards

To unify the interpretation of information, we will define the following standards for time synchronization via timestamps and the use of coordinate systems.

5.1.1 Time synchronization

All time stamps are expected either in standard Universal Time Coordinated (UTC) or a local time stamp, in milliseconds (ms). Each component providing information containing time stamps must define whether UTC or local time is used.

5.1.2 Coordinate systems

All coordinates are either expected in a global coordinate system based on UTM coordinates or in a local coordinate system based on the car reference system as defined in the norm ISO 8855 "Road vehicles - Vehicle dynamics and road-holding ability - Vocabulary" using the location of the TeamMate vehicle as origin. Each component providing information containing coordinates must define whether a global or local reference frame is used.

5.2 Data Structures

Information requested or provided by components is intended to make use of the following data structures. Libraries implementing these data structures will be prepared for the TeamMate Extension SDK.

5.2.1 Pose and motion of the TeamMate vehicle

Information about the position, orientation, and motion of the TeamMate vehicle is stored in a data structure *EgoState*. For the *EgoState*, we define the coordinates to follow the global UTM reference frame. The members of the *EgoState* are defined in Table 3.

Table 3: The EgoState data structure.

Data	Type	Unit	Description
TimeStandard	uint_16	1	Indicator whether the time is provided as UTC date and time (0) or local (1)
Timestamp	uint_64	ms	Timestamp
CoordinateStandard	uint_16	1	Indicator whether coordinates are based on a global UTM reference frame (0) or local frame (1). For the EgoState, only the global UTM reference frame is accepted.
PositionX	uint_16	m	X-position of the centre of the bounding box
PositionY	float_64	m	Y-position of the centre of the bounding box

Heading	float_64	rad	Heading in respect to the x-axis
VelocityX	float_64	m/s	Velocity in longitudinal direction
VelocityY	float_64	m/s	Velocity in lateral direction
AccelerationX	float_64	m/s ²	Acceleration in longitudinal direction
AccelerationY	float_64	m/s ²	Acceleration in lateral direction
YawRate	float_64	rad/s	Radial velocity
PoseMotionCovMat	float_64[8][8]	1	Covariance matrix for pose and motion

5.2.2 Pose, Motion, and dimension of detected objects

Information about a single object detected by the sensors of the TeamMate vehicle is intended to be stored in a data structure *Object*, as defined Table 4. The dimension of an objects is assumed to be represented as a two-dimensional bounding box with width and length. All coordinates are defined either in a global or local reference frame.

Table 4: The Object data structure.

Data	Type	Unit	Description
ID	uint_32	1	Unique object identifier
TimeStandard	uint_16	1	Indicator whether the time is provided as UTC date and time (0) or local (1)
Timestamp	uint_64	ms	Timestamp
CoordinateStandard	uint_16	1	Indicator whether coordinates are based on a global UTM reference frame (0) or local frame (1). For the EgoState, only the global UTM reference frame is accepted.
PositionX	uint_16	m	X-position of the centre of the bounding box
PositionY	float_64	m	Y-position of the centre of the bounding box
Heading	float_64	rad	Heading in respect to the x-axis
VelocityX	float_64	m/s	Velocity in longitudinal direction
VelocityY	float_64	m/s	Velocity in lateral direction

AccelerationX	float_64	m/s ²	Acceleration in longitudinal direction
AccelerationY	float_64	m/s ²	Acceleration in lateral direction
YawRate	float_64	rad/s	Radial velocity
PoseMotionCovMat	float_64[8][8]	1	Covariance matrix for pose and motion
Length	float_64	m	Length of the bounding box in longitudinal direction
Width	float_64	m	Width of the bounding box in lateral direction
LengthWidthCovMat	float_64[2][2]	1	Covariance matrix for length and width
Dynamic	uint_16	1	Indicator whether the object represents a static (0) or dynamic (1) object
ExistenceProbability	float_64	1	Confidence that the detected object is existing.

5.2.3 Semantic annotation of detected objects

Information about detected objects may be annotated by additional information defined in the data structure *ObjectAnnotation* (Table 5).

Table 5: The ObjectAnnotation data structure.

Data	Type	Unit	Description
ID	uint_32	1	Unique object identifier
TimeStandard	uint_16	1	Indicator whether the time is provided as UTC date and time (0) or local (1)
Timestamp	uint_64	ms	Timestamp
SemanticClass	uint_32	1	The semantic class of the object.
SemanticClassProbability	float_64	1	The probability that the object belongs to the stated semantic class.
AllowedManeuver	uint_32[8]	1	Vector of manoeuvres the object is allowed to perform: Start (0), FollowLane (1), ChangeLane (3), TurnLeft (4), TurnRight (5), SlowDown (6), Stop (7).

AllowedManeuverProbability	float_64	1	Probability for each manoeuvre.
----------------------------	----------	---	---------------------------------

5.2.4 Prediction of the spatial and temporal evolution of detected objects

Information about the spatial and temporal evolution of a detected object is summarized in the data structure *ObjectTrack*, as defined in Table 6.

Table 6: The ObjectTrack data structure

Data	Type	Unit	Description
ID	uint_32	1	Unique object identifier
TimeStandard	uint_16	1	Indicator whether the time is provided as UTC date and time (0) or local (1)
Timestamp	uint_64	ms	Timestamp
Size	uint_32	1	The number of entries in the ObjList
ObjList	Object[Size]	1	A vector of Objects, specifying the evolution of the object's state at specific points in time (as indicated by the Object.Timestamp).

5.2.5 Digital Maps

Many components of the TeamMate vehicle, incl. driver-, vehicle-, and situation-models, as well as online risk assessment will require high-accurate topographic information about the geometry and semantic of the current and future environment, incl. knowledge of roads, lanes, lane types, curbs, road and lane markings, intersections, traffic lights and signals, speed limits etc. Usually, such information is already available in autonomous vehicles in terms of a digital map.

The digital map will be summarized in a data structure *DigitalRoadMap*. The DigitalRoadMap is currently derived from harmonizing the different approaches currently used in each demonstrator.

5.2.6 Safety corridors

Information about the area of collision-free travel estimated over a specific temporal interval is summarized in the data structure *SafetyCorridor*, as defined in Table 10, recursively using the *2DPoint* (Table 7), *Polyline* (Table 8), and *ObjectPolyline* (Table 9) in the process.

Table 7: The 2DPoint data structure.

Date	Type	Unit	Description
X	float_64	m	X position
Y	float_64	m	Y position

Table 8: The Polyline data structure.

Data	Type	Unit	Description
Size	uint_16	1	Number of points in the PointsArr
PointsArr	2DPoint[Size]	1	Vector of 2D Points

Table 9: The ObjectPolyline data structure.

Data	Type	Unit	Description
Size	uint_16	1	Number of polylines in the PolylineList
Type	uint_16	1	Indicator whether the PolylineList is associated with the lane boundaries (0) or an object.
ID	uint_32	1	Unique identifier referring to the object ID if the PolylineList is associated with an object.
PolylineList	Polyline[Size]	1	Vector of polylines

Table 10: The SafetyCorridor data structure.

Date	Type	Unit	Description
TimeStandard	uint_16	1	Indicator whether the time is provided as UTC date and time (0) or local (1)
TimestampStart	uint_64	ms	Timestamp indicating the beginning of the temporal interval t .
TimestampEnd	uint_64	ms	Timestamp indicating the end of the temporal interval $t + \Delta t$.
Size	uint_16	1	Number of ObjectPolylines of ObjectPolylineList

ObjectPolylineList	ObjectPolyline[Size]	1	Vector of ObjectPolylines defining the safety corridor over the temporal interval $[t, t + \Delta t]$
--------------------	----------------------	---	---

5.3 Data interface

The data interface is defined by the set and structure of messages that components provide to a requesting client.

5.3.1 Environment model

The environment model is assumed to be provided as a service by the sensor and communication platform of a TeamMate vehicle. It consists of a static environment model and a dynamic environment model.

5.3.2 Static environment model

The static model contains all measured and validated information about the static scene, as defined in Table 11.

Table 11: Content of the message defining the static environment model.

Data	Type	Unit	Description
TimeStandard	uint_16	1	Indicator whether the time is provided as UTC date and time (0) or local (1)
Timestamp	uint_64	ms	Timestamp
Size	uint_16	1	Number of objects in the StaticObjectList
StaticObjectList	Object[Size]	1	Vector of objects defining the state of all static objects detected by the TeamMate vehicle
DigitalRoadMap	DigitalRoadMap	1	The digital road map

5.3.3 Dynamic environment model

The dynamic model contains two different classes of information. First, it contains the dynamically detected, classified and predicted objects. Secondly, it contains the ego vehicle description.

Table 12: Content of the message defining the dynamic environment model.

Data	Type	Unit	Description
TimeStandard	uint_16	1	Indicator whether the time is provided as UTC date and time (0) or local (1)
Timestamp	uint_64	ms	Timestamp
Size	uint_16	1	Number of objects in the DynamicObjectList
DynamicObjectList	Object[Size]	1	Vector of objects defining the state of all dynamic objects detected by the TeamMate vehicle.
EgoState	EgoState	1	EgoState defining the state of the TeamMate vehicle.

5.3.4 Evolution of the traffic scene

The evolution of the traffic scene is a service expected to be provided by components/enablers implementing driver-, vehicle-, and situation models. It consists of a semantic annotation of sensor data and a probabilistic prediction of the spatial and temporal evolution of objects detected by the TeamMate vehicle.

5.3.4.1 Semantic annotation

The semantic annotation augments the dynamic objects with additional semantic information. The semantic prediction will be provided by the following table.

Table 13: Content of the message defining the semantic prediction

Date	Type	Unit	Description
TimeStandard	uint_16	1	Indicator whether the time is provided as UTC date and time (0) or local (1)

Timestamp	uint_64	ms	Timestamp
Size	uint_16	1	Number of objects in the ObjectList and the ObjectAnnotationList
ObjectList	Object[Size]	1	Vector of objects defining the state of all objects for which a semantic annotation is available.
ObjectAnnotationList	ObjectAnnotation[Size]	1	EgoState defining the state of the TeamMate vehicle.

5.3.4.2 Probabilistic prediction

The probabilistic prediction defines the estimated evolution of the traffic scene.

Table 14: Content of the message defining the probabilistic prediction

Data	Type	Unit	Description
TimeStandard	uint_16	1	Indicator whether the time is provided as UTC date and time (0) or local (1)
Timestamp	uint_64	ms	Timestamp
Size	uint_16	1	Number of elements in the ObjectTrackList
ObjectTrackList	ObjectTrack[Size]	1	Vector of ObjectTracks defining the temporal and spatial evolution of the state of objects detected by the TeamMate vehicle

5.3.5 Safety corridor

The safety corridor defines a region in which the TeamMate vehicle can travel safely, i.e., with an upper-bound on the probability of a collision with the road

boundaries or other traffic participants. Safety corridors are provided by components/enablers for online risk assessment.

Table 15: Content of the message defining the probabilistic prediction

Data	Type	Unit	Description
TimeStandard	uint_16	1	Indicator whether the time is provided as UTC date and time (0) or local (1)
Timestamp	uint_64	ms	Timestamp
CoordinateStandard	uint_16	1	Indicator whether coordinates are based on a global UTM reference frame (0) or local frame (1).
Size	uint_16	1	Number of safety corridors in the SafetyCorridorList
SafetyCorridorList	SafetyCorridor[Size]	1	Vector of SafetyCorridors over adjacent temporal intervals, defining a joint safety corridor over a prediction horizon $[t, t + \text{Size}\Delta t]$

5.3.6 Planned Trajectory

The planned trajectory defines the planned trajectory of the TeamMate vehicle and is provided by components/enablers for trajectory planning and execution.

Table 16: Content of the message defining the planned trajectory.

Data	Type	Unit	Description
TimeStandard	uint_16	1	Indicator whether the time is provided as UTC date and time (0) or local (1)



Timestamp		uint_64	ms	Timestamp, specifying the creation date of the trajectory
CoordinateStandard		uint_16	1	Indicator whether coordinates are based on a global UTM reference frame (0) or local frame (1).
Size		uint_16	1	Number of points and associated timestamps in the Trajectory and TimestampList
Trajectory		2DPoint[Size]	1	Vector of 2DPoints defining the planned trajectory
TimestampList		uint_64[Size]	ms	Vector of timestamp corresponding to each 2DPoint in the Trajectory

5.3.7 Driver's state

5.3.7.1 Drowsiness

This message regroups fields associated with driver's drowsiness.

The "Drowsiness State" and "Drowsiness Level" fields are both about the level of alertness/sleepiness of the driver; simply "Drowsiness Level" is a continuous variable with values between 0 and 1, whereas "Drowsiness State" is a discrete integer variable identifying 4 state of drowsiness. The confidence variable is about the reliability of both drowsiness field.

Frame rate: ~1 Hz.

Table 17: Content of the Drowsiness message

Data	Type	Unit	Range	Description
------	------	------	-------	-------------

timestamp	UInt64	ms	0...	Current UTC time
Drowsiness State	Int32	1	0..4	0 = unknown, 1=alert, 2=slightly drowsy, 3=drowsy, 4=sleepy,
Drowsiness Level	Float	1	0..1	0 -->Fully alert; 1-->Maximum Sleepiness
Confidence Level	Float	1	0..1	0 = no confidence, 1 = full confidence
Microsleep Event	Int32	1	0, 1	0 = No micro sleep event, 1 = Micro sleep event

5.3.7.2 Visual attention fast

This interface is directly related to driver's eyes and head gaze, and is thus published at a video-rate frequency.

Frame rate: ~30 Hz.

Table 18: Content of the Visual Attention Fast message

Data	Type	Unit	Range	Description
timestamp	UInt64	ms	0...	Current UTC time
Observed areas	Int32	1	0..7	0 = UNKNOWN // Unknown area 1 = ON-Road // looking ahead at the road 2 = OFF-Road 3 = LR // Left rear view Mirror area 4 = RR // Right Rear view 5 = CR // Central Rear view mirror 6 = IC // Instrument cluster 7 = CD // Central Display
Confidence Level	Float	1	0..1	0 = no confidence, 1 = full confidence
Look Time	UInt32	ms	0...	Time the driver has been continuously looking at the current instrument.

5.3.7.3 Visual attention slow

This message regroups fields associated with driver's visual attention.

The level of visual attention is a continuous variable with values between 0 and 1, whereas state of visual attention is a discrete integer variable identifying 3 state of attention. The confidence variable rates both fields.

Frame rate: ~2 Hz.

Table 19: Content of the Visual Attention slow message

Data	Type	Unit	Range	Description
timestamp	Uint64	ms	0...	Current UTC time
Visual Time Sharing	Float	1	0..1	Ratio of attention time ON-ROAD/ (OFF-ROAD+ON-ROAD)
Confidence Level	Float	1	0..1	0=no confidence, 1=full confidence
Attention State	Int32	1	0..5	0 = unknown, 1=Attentive, 2=Mid attention, 3=Distracted
Attention Level	Float	1	0..1	0 -->Fully attentive; 1-->Distracted
Confidence Level	Float	1	0..1	0 = no confidence, 1 = full confidence

5.3.7.4 Cognitive distraction

This message regroups fields associated with driver's cognitive distraction. The Level of cognitive distraction and state of cognitive distraction fields are both about the level of Cognitive Distraction function. The level of cognitive distraction is a continuous variable with values between 0 and 1, whereas state of cognitive distraction is a discrete integer variable identifying 3 levels of Cognitive Distraction. The confidence variable rates both fields. Frame rate: ~1 Hz.

Table 20: Content of the Cognitive Distraction message

Data	Type	Unit	Range	Description
timestamp	Uint64	ms	0...	Current UTC time
Cognitive Distraction State	Int32	1	0..5	0 = unknown, 1=Not distracted, 2= mid distraction, 3= fully distracted
Cognitive Distraction Level	Float	1	0..1	0-->Fully alert; 1-->Maximum
Confidence Level	Float	1	0..1	0 = no confidence, 1 = full confidence

5.3.7.5 Driver's state raw-data

This message contains the data provided by the face tracker. Frame rate: ~30 Hz.

Table 21: Content of the driver's-state Raw-data message

Data	Type	Unit	Range	Description
Timestamp	UInt64	ms	0...	Current UTC time
headPos	Float3	m		3D Position of head in meters
headPosQ	Float	1	0..1	Head position quality: (0.0 = no head tracking, 0.1 = Face Detection, 0.2 = Face Refinder, 0.2..1.0 = Head Tracking).
headYaw	Float	°		Head heading angle in degrees
headPitch	Float	°		Head pitch angle in degrees
headRoll	Float	°		Head roll angle in degrees
headRotQ	Float	1	0..1	Head rotation quality
gazeSrc	Float3	m		3D Position of origin of gaze vector, in meters. This is the consensus of the values for both eyes.
gazeDir	Float3	1		Unit vector giving the gaze direction in 3D (average of both eyes)
gazeQ	Float	1	0 .. 1	Quality of gaze origin & direction
leftEyeOpen	Float	mm	0...	The distance between the eyelids of the left eye
leftEyeOpenQ	Float	1	0..1	Quality of left eye opening
rightEyeOpen	Float	mm	0...	The distance between the eyelids of the right eye
rightEyeOpenQ	Float	1	0..1	Quality of right eye opening

5.3.8 Online Learning from the Driver

The Online Learning from the Driver loads the initial model of the Driver Intention Recognition and updates the model parameters during driving based on the sensory input received. The required input for learning is the same as for the Driver Intention Recognition meaning the initial model, environment data from the static and the dynamic environment model (track-objects, vehicle data). If an updated version of the Driver Intention Recognition model is ready it is signaled via the output message. The Intention Recognition can then load the updated model parameters.

Table 22: Online learning from the driver

Date	Type	Unit	Description
updateReady	UInt_16	1	Flag to signalize that a model update is ready
modelLocation	String	1	Path to updated model



5.3.9 Component Communication Framework

Within AutoMate, the different demonstrators make use of pre-existing middleware solutions that provide their own dedicated SDKs for the integration of enablers and communication. Where applicable, enabler providers will make use of such pre-existing SDKs to expedite the integration process.

For a more *general* solution, HMT is currently developing a framework for communication as a part of the TeamMate SDK, called the *component framework*. The component framework is implemented in C++ and allows (external) developers to embed their enablers into C++ components with pre-implemented functionality for communication.

The primary building block within the component framework is that of a *component*, which acts as a template that implements all necessary functionalities for communication in which an enabler can be embedded. Based on a client-server model, components may act as servers by providing services to other components (in terms of messages), or as clients, by requesting these services from other components.

Components can have multiple inputs and (optionally) a single output. Each input can be connected to the output of a single other component, while each output may be connected to multiple inputs. A component provides a primary execute method that can be used to read all current inputs and produce a new output. Components are executed periodically, with each component being allowed to work at its own frequency. Components are organized in component sets that will be executed as a single unit.

By now, we prepared three types of components to facilitate different use cases:

- Reader components have a set of input but no output. Such components can be used, e.g., by enablers that write data from the network into a database.
- Writer components have a single output but no input. Such components can be used, e.g., by enablers that read data from a database to serve it to other components.
- Lastly, reader/writer components combine the abilities of the reader and writer components by providing a set of inputs and a single output.

In addition to the above, the component communication framework also supports standalone inputs and outputs that can be integrated into existing codebases to enable communication between components and different middlewares.

Figure 8 shows an exemplary utilization of the component framework, consisting of five components, organized in two separate component sets. The first component set consists of two writer components, that generate messages, e.g., by retrieving data from a database, and periodically send their outputs over the network. The second component set consists of three components, a reader-writer and two reader components. The reader/writer component reads from its inputs at its own frequency and produces output to be consumed by the two reader components, writing the received data into a database and displaying it via a graphical user interface.

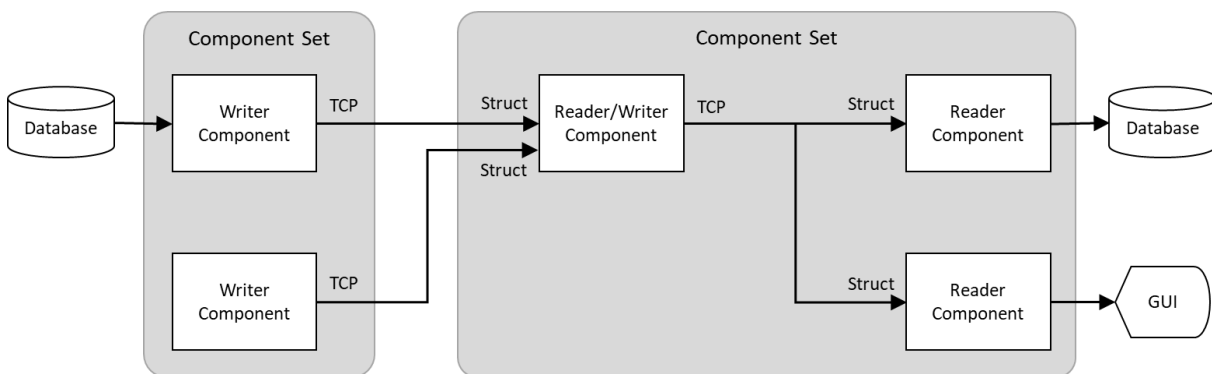


Figure 8: Exemplary realization of a network of components.

The component framework supports both TCP/IP and UDP communication, implemented using the *Boost ASIO* framework. Messages can be serialized using *Boost Serialization* or Google's *protobuf* to enable communication with software written in languages other than C++.

Figure 9 shows an exemplary communication pattern for a reader/writer component receiving messages (M) from two connected server components at its inputs. As messages may arrive asynchronously based on the sending components internal frequencies, they are put into a message queue. As soon as the components execution period starts, it can read from its inputs. When this happens, the component receives the most current message available (displayed in red) on each input, while all previous messages will be discarded (displayed in black). The enabler embedded within the component can then process the messages to produce its own output (O) and distribute it to its own clients.

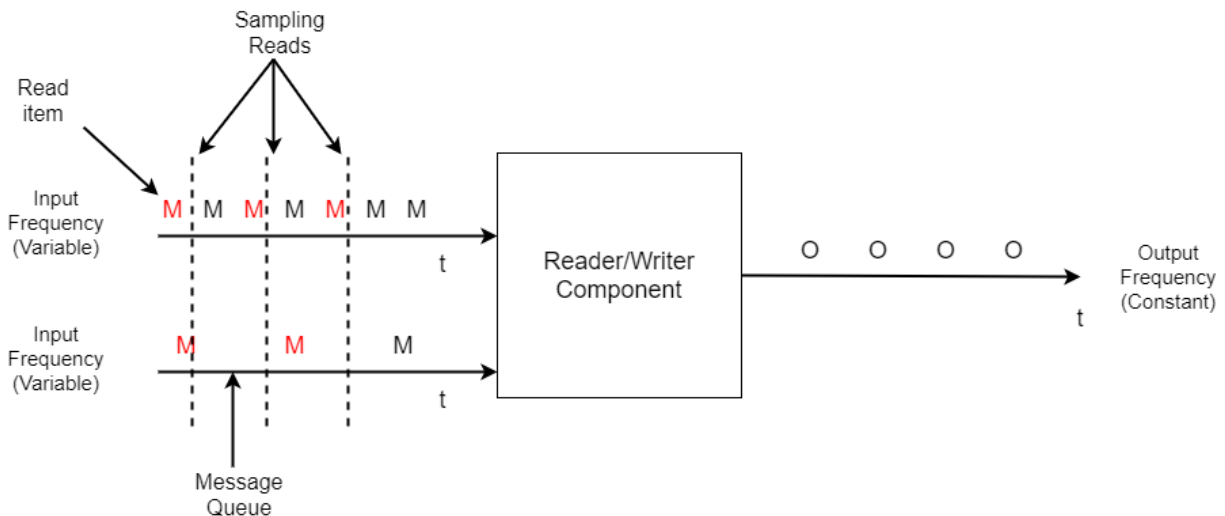


Figure 9: Example of communication within the component communication framework.

General remarks: These descriptions will be translated into protobuf messages in order to enrich the 2nd cycle API and provide a global API including all the exchanged information between the TeamMate system and its host car and between all the enablers constituting the architecture of our system, since we are in subscriber/publisher functioning mode there is no need to define the message chart diagram, since subscribers have to pay attention to the timestamp of the data exchanged in order to process it.

6 The TemMate API for the second cycle

In this section VED propose a definition of the general API of the TeamMate system. This API is extracted from the main inputs of the TemMate architecture shown in the section 4. We describe the exchanged information between the system and the sensors in order to define the TeamMate system functional and ready to be integrated in a specific car.

Several solutions could be used to fulfill these requirements but to be able to provide to third party a way to interoperate with other enablers or sensors all messages are defined using protobuf protocol⁹. In this way, the enablers could be defined in any language (C/C++, JAVA ...), in that way we have a convenient way to integrate TemMate system messages in their implementations.

⁹ <https://github.com/protocolbuffers/protobuf>
⁹<https://developers.google.com/protocol-buffers/>

The communication between the TeamMate system and the host vehicle/simulator can follow the description given in the section 5.3.9. If the host vehicle owner has a specific middleware it is on its own responsibility to adapt the right wrappers in order to make the system or a part of the TeamMate system communicate with the different sensors and low level organs of the vehicle.

To define the requested messages for the TeamMATE system, we rely on discussions between AutoMATE partners in charge on enabler definitions specified during different workshops of integration (Ulm University (ULM), Paris1 (VED), Versailles1 (VED), Paris2 (VED), and Versailles2 (VED) which took place during the demonstrators definition.

The first defined messages are about the information requested by the enablers to operate computations, their main topics are the environment definition in terms of static and dynamic informations and the ego-vehicle state.

The Ego-vehicle data message

The first requested message deals with the ego-vehicle state. Several informations have been requested as defined in the following table.

Name	Type	Units	Comments
timestamp	Long (64 bits)	Ms	Expressed since 01/01/1970-00:00:00
rear_axle_center_x	Float (32 bits)	M	X-coordinate of the Rear Axle Center Expressed in the UTM referential
rear_axle_center_y	Float (32 bits)	M	Y-coordinate of the Rear Axle Center Expressed in the UTM referential
rear_axle_center_x_std	Float (32 bits)	M	Standard deviation of the X-coordinate of the Rear Axle Center Expressed in the UTM referential
rear_axle_center_y_std	Float (32 bits)	M	Standard deviation of the Y-coordinate of the Rear Axle Center Expressed in the UTM referential
vehicle_width	Float (32 bits)	M	The width of the Ego-Vehicle
vehicle_height	Float (32 bits)	M	The height of the Ego-Vehicle
vehicle_heading	Float (32 bits)	degree	The heading of ego-vehicle defined from North, clockwise
vehicle_heading_std	Float (32 bits)	degree	Standard deviation of the heading of ego-vehicle, clockwise
vehicle_speed_x	Float (32 bits)	m.s ⁻¹	The longitudinal component of ego-vehicle velocity

vehicle_speed_y	Float (32 bits)	m.s-1	The lateral component of ego-vehicle velocity
vehicle_speed_norm	Float (32 bits)	m.s-1	The euclydian norm of ego-vehicle velocity
vehicle_yaw_rate	Float (32 bits)	degree.s-1	The yaw rate of ego-vehicle
steering_angle	Float (32 bits)	degree	The steering angle of ego-vehicle front wheels

The protobuf translation of the previous table is he following:

```
package eu.automate.openapi.messages;

message EgoVehicleMessage {

    required int64 timestamp = 10; //
    expressed in milliseconds since 01/01/1970-00:00:00

    required float rear_axle_center_x = 20; //
    expressed in meters
    required float rear_axle_center_y = 21; //
    expressed in meters

    required float rear_axle_center_x_std = 30; // standard
    deviation, expressed in meters
    required float rear_axle_center_y_std = 31; // standard
    deviation, expressed in meters

    required float vehicle_width = 40; //
    expressed in meters
    required float vehicle_height = 41; //
    expressed in meters

    required float vehicle_heading = 50; //
    expressed in degree, from North, clockwise
    required float vehicle_heading_std = 51; // standard
    deviation, expressed in degree

    required float vehicle_speed_x = 60; //
    expressed in meters per seconds
    required float vehicle_speed_y = 61; //
    expressed in meters per seconds
    required float vehicle_speed_norm = 62; //
    euclydian norm, expressed in meters per seconds

    required float vehicle_yaw_rate = 70; //
    expressed in degrees per seconds

    required float steering_angle = 80; //
    expressed in degrees

    // extensions 100 to 199;
}
```

In addition to the ego-vehicle state, some enablers need also information about the surrounding environment. Two kinds of information define the surrounding environment of the ego-vehicle, such that static information which typically refers to the definition of the road network near the vehicle and

dynamic information corresponding to the different surrounding obstacles and temporary modifications on the static layer (the road).

Static message is built according to information provided by a third party map system, where we only preserve information about the current road element and the ones defined in the current defined journey. All of this correspond to an elongated road with or without intersections.

Dynamic information come from the preception layer of the TeamMate vehicle, it mainly deals with surrounding obstacles.

The Map message

Name	Type	Units	Comments
id	Long (64 bits)	-	A unique ID for this map definition
nbLanes	Integer (32 bits)	-	The maximum number of lanes
right_lane_info	LaneInfo (see below)	-	Road information about the left lane
middle_lane_info	LaneInfo (see below)	-	Road information about the middle lane
left_lane_info	LaneInfo (see below)	-	Road information about the right lane

The map is defined as a succession of waypoints. For each waypoints in the map, we provide several information related to the driving lane.

Name	Type	Units	Comments
availability	Boolean	-	A unique ID for this map definition
center_x	Float (32 bits)	m	The maximum number of lanes
center_y	Float (32 bits)	m	Road information about the left lane
half_width	Float (32 bits)	m	Road information about the middle lane
mandatory_speed_limit	Float (32 bits)	m.s-1	Road information about the right lane
right_marking	Integer (32 bits)	[0 - 1]	0 if dashed marking, 1 if continuous marking, -1 otherwise
left_marking	Integer (32 bits)	[0 - 1]	0 if dashed marking, 1 if continuous marking, -1 otherwise
road_heading	Float (32 bits)	degree	expressed from North, clockwise

The protobuf message translation merging these two tables is the following:



```
package eu.automate.openapi.messages;

message MapMessage {

    required int32      id                = 1;          // An unique ID

    required int32      nbLanes           = 2;          // The number of
available lanes

    message LaneInfo {

        required bool   availability      = 1;          // True if exists,
False otherwise

        required float  center_x         = 10;         // The center of
lane x-coordinate, expressed in meters (UTM referential)

        required float  center_y         = 11;         // The center of
lane y-coordinate, expressed in meters (UTM referential)

        required float  half_width       = 20;         // The half width
of lane, expressed in meters

        required int32  mandatory_speed_limit = 30;     // The mandatory
speed limit at this position expressed in kilometers per hours

        required int32  right_marking     = 40;         // = 0 if dashed
marking, = 1 if continuous marking

        required int32  left_marking      = 41;         // = 0 if dashed
marking, = 1 if continuous marking

        optional float  road_heading      = 50;         // expressed in
degree, from North, clockwise

    }

    required LaneInfo   right_lane_info    = 10;
    required LaneInfo   middle_lane_info   = 20;
    required LaneInfo   left_lane_info     = 21;

    // extensions 100 to 199;

}
```

The objects message

The last mandated message for the environment is about obstacles as below.

Name	Type	Units	Comments
id	Long (64 bits)	-	A unique ID for this list of obstacles
timestamp	Long (64 bits)	ms	Expressed since 01/01/1970-00:00:00
nbObjects	Integer (32 bits)	-	The number of detected obstacles
object_info	ObjectInfo[]	-	The characteristics of each detected objet

And an ObjectInfo is defined as:

Name	Type	Units	Comments
id	Long (64 bits)	int	A unique id per obstacle
relative_position_x	Float (32 bits)	m	The x-coordinate center of bounding box Expressed in the ego-vehicle referential
relative_position_y	Float (32 bits)	m	The y-coordinate center of bounding box Expressed in the ego-vehicle referential
relative_uncertainty_p_ [xx, xy yx, yy]	Float * 4 (32 bits)	m	The uncertainty matrix of relative position of center of bounding box
relative_velocity_x	Float (32 bits)	m.s-1	The longitudinal component of the object velocity Expressed in the ego-vehicle referential
relative_velocity_y	Float (32 bits)	m.s-1	The lateral component of the object velocity Expressed in the ego-vehicle referential
relative_uncertainty_v_ [xx, xy yx, yy]	Float * 4 (32 bits)	m.s-1	The uncertainty matrix of relative velocity of center of bounding box
absolute_position_x	Float (32 bits)	m	The x-coordinate center of bounding box Expressed in the UTM referential
absolute_position_y	Float (32 bits)	m	The y-coordinate center of bounding box Expressed in the UTM referential
absolute_uncertainty_p_ [xx, xy yx, yy]	Float * 4 (32 bits)	m	The uncertainty matrix of absolute position of center of bounding box
absolute_velocity_x	Float (32 bits)	m.s-1	The longitudinal component of the object velocity Expressed in the UTM referential
absolute_velocity_y	Float (32 bits)	m.s-1	The lateral component of the object velocity Expressed in the UTM referential
absolute_uncertainty_v_ [xx, xy yx, yy]	Float * 4 (32 bits)	m.s-1	The uncertainty matrix of absolute velocity of center of bounding box
yaw	Float (32 bits)	degree	the orientation of the bounding box Expressed in the ego-vehicle referential
yaw_std	Float (32 bits)	degree	The standard deviation of orientation
width	Float (32 bits)	m	The visible width of the object
height	Float (32 bits)	m	The visible height of the object
width_std	Float (32 bits)	m	The standard deviation of width
height_std	Float	m	The standard deviation of height

	(32 bits)		
label	Integer (32 bits)	- (cf. IBEO)	The classification result, same as IBEO
label_score	Float (32 bits)	[0, 1]	The score of classification
existence_probability	Float (32 bits)	[0, 1]	The existence probability

The following protobuf message of the obstacles is the following:

```
package eu.automate.openapi.messages;

message ObjectMessage {
    required int32 id = 1; // An unique ID

    required int64 timestamp = 10; // expressed in
    milliseconds since 01/01/1970-00:00:00

    required int32 nbObjects = 20; // the number of objects
    in this message

    message ObjectInfo {
        required int32 id = 1; // An unique ID

        required float relative_position_x = 10; // The x-coordinate
        center of bounding box, expressed in meters (Vehicle referential)
        required float relative_position_y = 11; // The y-coordinate
        center of bounding box, expressed in meters (Vehicle referential)

        required float relative_uncertainty_p_xx = 20; // The (0,0)-coordinate
        of uncertainty matrix of relative position of center of bounding box
        required float relative_uncertainty_p_xy = 21; // The (0,1)-coordinate
        of uncertainty matrix of relative position of center of bounding box
        required float relative_uncertainty_p_yx = 22; // The (1,0)-coordinate
        of uncertainty matrix of relative position of center of bounding box
        required float relative_uncertainty_p_yy = 23; // The (1,1)-coordinate
        of uncertainty matrix of relative position of center of bounding box

        required float relative_velocity_x = 30; // The x-coordinate
        center of bounding box, expressed in meters (Vehicle referential)
        required float relative_velocity_y = 31; // The y-coordinate
        center of bounding box, expressed in meters (Vehicle referential)

        required float relative_uncertainty_v_xx = 40; // The (0,0)-coordinate
        of uncertainty matrix of relative velocity of object
        required float relative_uncertainty_v_xy = 41; // The (0,1)-coordinate
        of uncertainty matrix of relative velocity of object
        required float relative_uncertainty_v_yx = 42; // The (1,0)-coordinate
        of uncertainty matrix of relative velocity of object
        required float relative_uncertainty_v_yy = 43; // The (1,1)-coordinate
        of uncertainty matrix of relative velocity of object

        required float absolute_position_x = 110; // The x-
        coordinate center of bounding box, expressed in meters (UTM referential)
        required float absolute_position_y = 111; // The y-
        coordinate center of bounding box, expressed in meters (UTM referential)

        required float absolute_uncertainty_p_xx = 120; // The (0,0)-
        coordinate of uncertainty matrix of absolute position of center of bounding box
```



```

        required float absolute_uncertainty_p_xy = 121;      // The (0,0)-
coordinate of uncertainty matrix of absolute position of center of bounding box
        required float absolute_uncertainty_p_yx = 122;      // The (0,0)-
coordinate of uncertainty matrix of absolute position of center of bounding box
        required float absolute_uncertainty_p_yy = 123;      // The (0,0)-
coordinate of uncertainty matrix of absolute position of center of bounding box

        required float absolute_velocity_x      = 130;      // The x-
coordinate center of bounding box, expressed in meters (UTM referential)
        required float absolute_velocity_y      = 131;      // The y-
coordinate center of bounding box, expressed in meters (UTM referential)

        required float absolute_uncertainty_v_xx = 140;      // The (0,0)-
coordinate of uncertainty matrix of absolute velocity of object
        required float absolute_uncertainty_v_xy = 141;      // The (0,0)-
coordinate of uncertainty matrix of absolute velocity of object
        required float absolute_uncertainty_v_yx = 142;      // The (0,0)-
coordinate of uncertainty matrix of absolute velocity of object
        required float absolute_uncertainty_v_yy = 143;      // The (0,0)-
coordinate of uncertainty matrix of absolute velocity of object

        required float yaw                      = 200;      // The orientation
of the bounding box expressed in degree (Vehicle referential)
        required float yaw_std                  = 201;      // The standard
deviation of orientation

        required float width                    = 210;      // The width of the
object, expressed in meters
        required float height                   = 211;      // The height of
the object, expressed in meters

        required float width_std                = 220;      // The standard
deviation of width
        required float height_std               = 221;      // The standard
deviation of height

        optional int32 label                    = 230;      // The
classification result, same as IBEO
        optional float label_score              = 231;      // The score of
classification, in [0, 1]

        optional float existence_probability    = 240;      // The existence
probability, in [0, 1]
    }

    repeated ObjectInfo object_info              = 100;
}

```

The next defined messages are the standardized output of enablers requested to be able to interoperate all together and provide the interface requested in order to switch from one specific implementation to another.

The trajectory message

Trajectory messages are generated by the MotionPlanning enabler, it provide a list of waypoint, these waypoints are used by the controller which translate the waypoints and the curvature into control variables (acceleration, braking and steering).

Name	Type	Units	Comments
N	Long (64 bits)	-	A unique ID for this generated trajectory
global_timestamp	Long (64 bits)	ms	The expected time to reach the 1st way point Expressed since 01/01/1970-00:00:00
waypoint_info	WayPointInfo[]	-	The characteristics of each waypoint

And a waypoint is defined as below:

Name	Type	Units	Comments
time_to_reach	Long (64 bits)	-	The expected time the ego-vehicle reach the position
relative_position_x	Float (32 bits)	m	The x-coordinate way point, Expressed in the vehicle referential
relative_position_y	Float (32 bits)	m	The y-coordinate way point, Expressed in the vehicle referential
absolute_position_x	Float (32 bits)	m	The x-coordinate way point, Expressed in the UTM referential
absolute_position_y	Float (32 bits)	m	The y-coordinate way point, Expressed in the UTM referential

```
package eu.automate.openapi.messages;

message TrajectoryMessage {
    required int32 N = 1; // number of way points
    required int64 global_timestamp = 10; // expected time to reach the 1st way point
    message WayPointInfo {
        required int64 time_to_reach = 1; // An unique ID
        required float relative_position_x = 10; // The x-coordinate way point, expressed in meters (Vehicle referential)
        required float relative_position_y = 11; // The y-coordinate way point, expressed in meters (Vehicle referential)
        required float absolute_position_x = 110; // The x-coordinate way point, expressed in meters (UTM referential)
        required float absolute_position_y = 111; // The y-coordinate way point, expressed in meters (UTM referential)
    }
}
```

```
repeated WayPointInfo waypoint_info = 100;
}
```

For more details on the other messages and enablers present in the API please download the full version on the repository of the project in the folder:

/Workpackage Documents/WP5-TeamMate Architecture System Integration and Implementation/Deliverables/D5.1/New/OpenAPI

See the appendix for more details about the username and password for the connection.

6.1 Communication of the TeamMate car with its environment: the V2X related standards

This subsection introduces the architecture and the standards of communication of the TeamMate car with the surrounding environment. Indeed we dealt in this section with the ETSI TC ITS reference architecture using [1]. The reference architecture gives an overview about the applied V2X communication in the project. Then the details of the standards and their application in the project are discussed using [2][3] and [4].

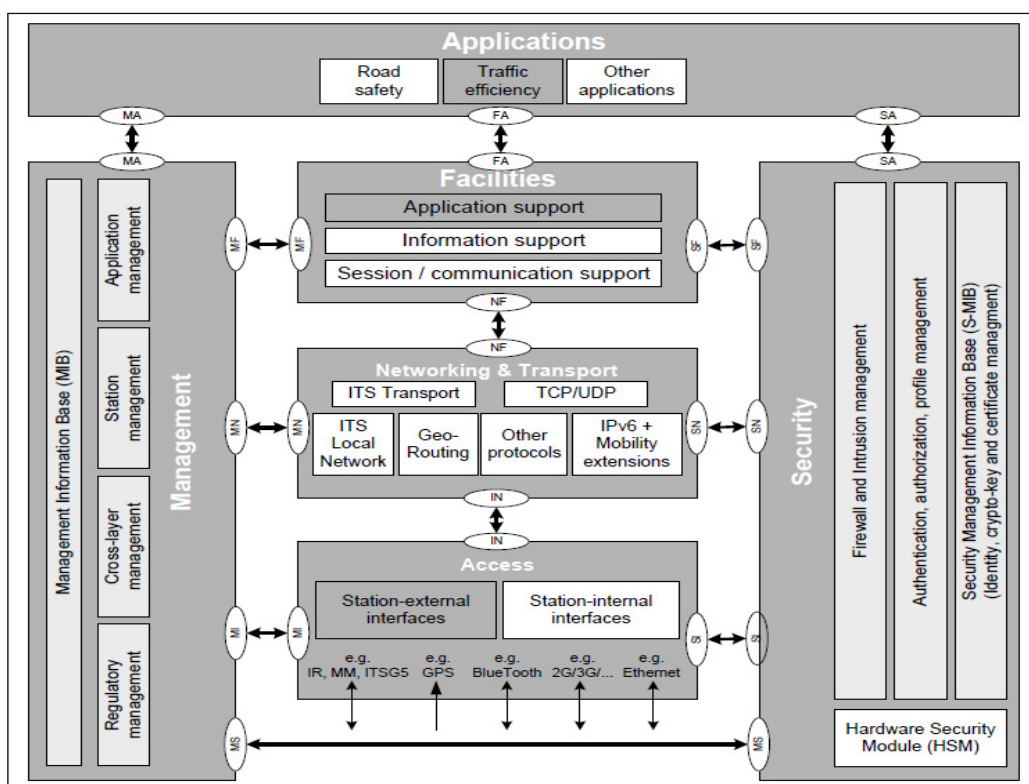


Figure 10. ETSI TC ITS reference architecture



6.1.1 ETSI TC ITS V2X Reference Architecture

ETSI TC ITS has defined a reference architecture as shown in Figure 10, which is similar to the U.S. architecture. The reference architecture has layered structure, where each layer has its own dedicated tasks following ISO Open System Interconnection (OSI) model. The reference architecture is based on a slightly modified IEEE 802.11p at the access layer. It enables new networking functionalities based on geographical addressing at the network layer, and new facilities layer on top that allows a set of rich messages, which support different types of applications. Compared to the U.S. ITS architecture, the ETSI TC ITS architecture includes more features at the network layer to support further communication scenarios, such as multi-hop forwarding. The functionalities of the facilities layer are very similar in both architectures as most of them have been initially defined by the U.S. standard, and then adopted and slightly modified by the EU standard (ETSI).

V2X is intended to enable critical safety applications first, where vehicles and road infrastructure cooperate by exchanging real-time information to be used for the prediction and the avoidance of accidents and, thus, to improve road safety. This type of application requires fast communication. Once the technology is deployed, it will also allow new traffic efficiency applications, as well as useful infotainment and added-value applications. The technology will support also the autonomous driving application, as it is important for an autonomous vehicle to communicate with other self-driving vehicles around it to negotiate the sharing of the road resources.

These applications require well-defined messages that could provide all of the required information in an efficient and reliable manner. ETSI TC ITS has been working on defining key messages at the facilities layer, such as cooperative awareness messages (CAM) and decentralized environmental notification messages (DENM). The CAM is intended to be sent by each vehicle at least once every second and at most 10 times per second, based on the vehicle dynamics. Each CAM message includes a list of information about the location and status of the vehicle. The CAM exchange enables each vehicle to build a local map about all vehicles in the surrounding. While CAM is a proactive message, the DENM is a reactive message and triggered by an event, e.g., when planned road maintenance is going on, a DENM generation mechanism is triggered by the road operator to initiate the related DENM informing all vehicles within the relevant geographical area about the roadwork. As mentioned some messages like DENM need to be disseminated within a limited geographical area and to support, that there was a need to enable new dissemination algorithms at the transport and network layers. The GeoNetworking functionalities at the network layer of the ETSI TC ITS have been introduced to support geographical-based routing mechanisms where a packet is forwarded based on geographical addressing schemes that use geospatial coordinates.

6.1.2 Decentralized environmental notification message

Decentralized environmental notification message supports road safety applications by informing all vehicles within an area of relevance about road hazard [2]. As mentioned, DENM is a reactive message, and it is typically triggered by an event. The DENM generation and broadcast can be initiated by the road operator directly (e.g., planned road maintenance starts in an area) or by a vehicle that notice such event using its sensors (e.g., icing on the road). In the project, road works warning message will be used in the Martha scenario as use case specific DENM.

The following figure depicts the general structure of decentralized environmental notification message.

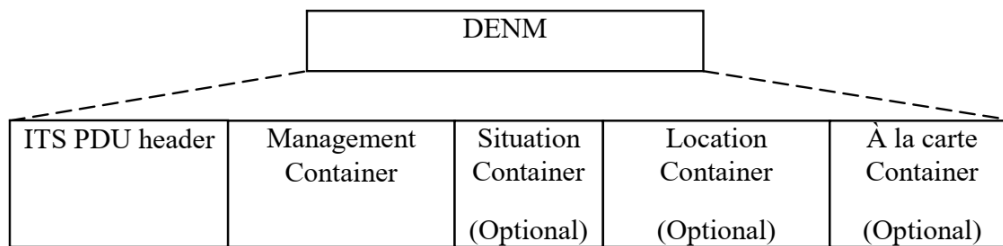


Figure 11. General structure of DENM [2]

The message is structured with a header and data containers. Each container is extensible to support potential extension of the DENM content for future ITS application needs.

The ITS PDU header is a common header for all ITS message types. It includes among others the protocol version, message ID and station ID. It also contains the information about how should threat the ITS station the message. For DENM the GeoBroadcast functionality of the GeoNetworking protocol is used. It supports multi-hop packet forwarding in order to route the DENM from the source to the defined geographical destination area.

The management container container is mandatory and shall be present in all transmitted DENM. This container contains information for the DENM protocol operation using several parameters:

- *actionID* is composed of the station ID of the detecting ITS-S and a sequence number. The concept of the this parameter is introduced as the event identifier. An *actionID* enables a receiving ITS-S to distinguish an event detected by different ITS stations, or different event detected by the same ITS-S.
- *referenceTime* is the parameter that enables the distinction of different DENM updates about one event.
- *termination* allows the receiving ITS-S to derive the DENM type. If present in DENM, it includes two values i.e., cancellation DENM or negation DENM.
- *validityDuration* parameter indicates the end of a DENM validity. It may be used to indicate an estimated or preset duration of the event persistence, in case such duration is known in advance. This parameter



may not be present in a DENM, in case the detecting ITS-S is not able to provide the event duration information. In this case, a default value is set by ITS-S for internal protocol operation.

- *transmissionInterval* is present in DENM when facilities layer forwarding is activated. It indicates the time interval of DENM transmission at the originating ITS-S.

The situation container contains event type information and an indicator of the event detection performance. Each event type is identified with an integer type event code. This list of event codes is extensible. The event type is composed of two data elements, namely the cause code and sub-cause code. The cause code is 3 and the sub-cause code is 3 or 4 in case of road works warning.

The location container includes information that describes the location referencing information at the event position. The location referencing information for DENM is a list of traces. Each trace is composed of a list of waypoints that construct a path approaching to the event position. This location referencing information enables receivers to estimate its relevance to the event, by comparing its own itinerary path to each trace contained in the received DENM. In addition, the location container may also include information that represents the detection history of a plain event (e.g., an extreme weather condition event), if the same event was detected by a moving vehicle along its travel path in the past.

The à la carte container contains additional information that is not provided by other containers. This container provides the possibility for ITS-S application to include application specific data in a DENM. All information included in the à la carte container is optional. They shall be present when the data is provided by the ITS-S application.

For example, *roadWorks* container may be added for the roadwork use case as specified in ETSI TS 101 539-1 [3]. It includes information of the roadwork zone and specific access conditions:

- CauseCode: 3 in case of roadwork.
- SubCauseCode: 3 or 4 in case of roadwork.
- RoadworkSegmentDescriptor: this parameter contains the geographic position of the start of roadwork areas and the stop of roadwork area.
- RoadworkClosedLanes: it describes the total number of lanes, identification of closed lanes according to vehicles types.
- SpeedLimits: it presents the regulatory speed limits per remaining open lane and according to vehicle types.
- AuthorizedVehicleTypes: this optional parameter contains the list of authorized vehicle types.
- RecommendedItinerary: this optional parameter contains the list of waypoints to re-access a road, which is closed on a given segment.

Besides the latter two parameters, all the others are mandatory in the *roadWorks* container.

The ASN.1 unaligned PER encoding rules are used for DENM encoding and decoding, in order to optimize the message size.

6.2 Third party HMI SDK specification

This section is dedicated to the description of the third party SDK developed in the field of the project and the description of the used protocols and standards of the V2X communication.

Indeed, In Automate, we will develop SDKs to allow third parties re-using the data extracted from the TeamMate car (about the vehicle, the environment and the driver) and develop their own applications based on this data.

Thanks to the installation of an IoT embedded device of DQUID¹⁰, the real-time information about the vehicle, the environment and the driver (available on the CAN bus) will be extracted, elaborated and made available. The SDKs developed in the Automate project will be used to seamlessly create new mobile applications that exploit this data.

An example of a potential application is the re-use of driving data by an insurance company. In fact, it could use the real-time information to identify the driving behaviour and then associate the risk of the driver (to optimize the driver's profile). These systems are already available¹¹ and could be further improved by the introduction of the Automate SDK.

This approach is in line with the EU strategy about the re-use of available data to strengthen the EU economy and creation of an innovation ecosystem.

6.2.1 DQuid SDK definition

A data exchange protocol (from now on called "DQuid protocol") has been defined to share data between the DQuid SDK - integrated into the mobile application - and the DQuid Stack - integrated into the application running on the vehicle (as shown in Figure 12).

¹⁰ www.dquid.com

¹¹ <http://www.gruppoac.it/insurance-gestione-crash-driving-behaviour-profilazione-driver/>

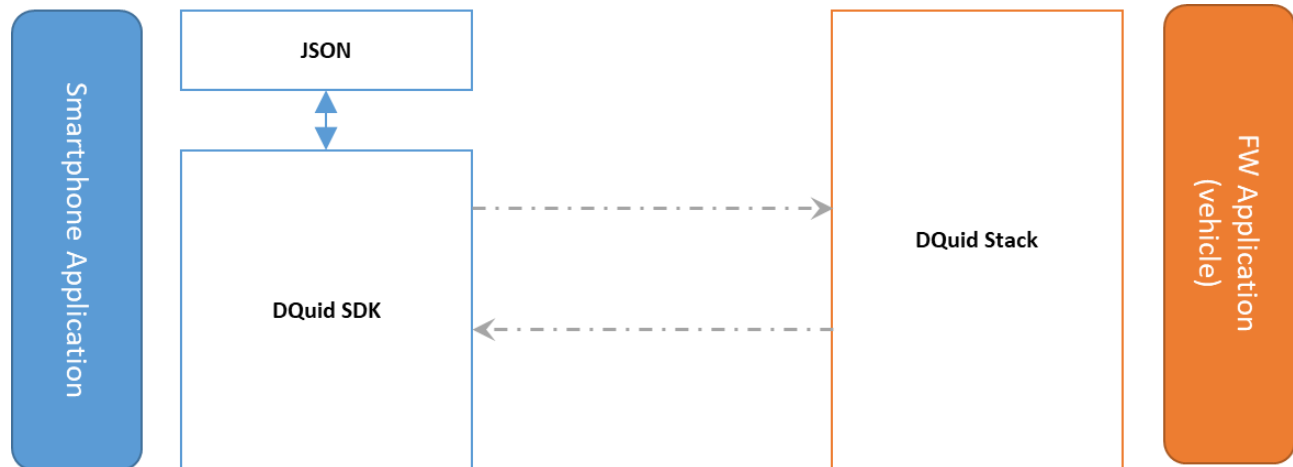


Figure 12: DQuid SDK high-level architecture

The mobile application (Android / iOS) links to the DQuid SDK framework that is in charge of establishing a communication with the embedded device on a specific link (e.g. BLE) and exchange data over this link.

The DQuid SDK provides the following features:

- Discovery of the devices embedding the DQuid Stack
- Connection/Disconnection to/from a specific embedded device (one connection at a time)
- Connection/Disconnection notification
- Definition of object's properties (DQSignal) with attributes (size, type, readable/writable). These properties must also be specified in the embedded firmware application integrating the DQuid Stack.
- Properties' read/write
- Properties' Subscribe/Unsubscribe
- Properties' update notification (in case of property subscription).

The DQuid Stack provides the following features:

- Definition of object's properties (DQSignal) with attributes (size, type, readable/writable). These properties must also be specified in the mobile application integrating the DQuid SDK.
- Properties' Subscribe/Unsubscribe
- Notify the embedded application when a request of a write operation for a property is received by the DQuid SDK.
- Property's value update (DQSignal)

The embedded device installed on the vehicle integrating the DQuid Stack is represented in the DQuid SDK as a DQuidObject with one or more properties (DQSignal).

The mobile application can

- read/write the DQuidObject's properties (Figure 13)
- subscribe to all DQuidObject's properties in order to receive notifications when properties data are updated by the embedded device (Figure 14).

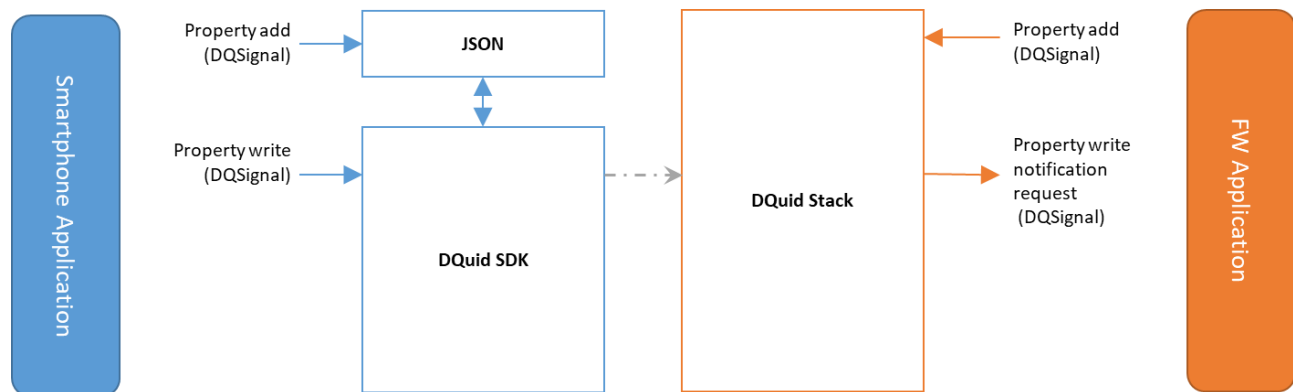


Figure 13: Write the DQuidObject's properties

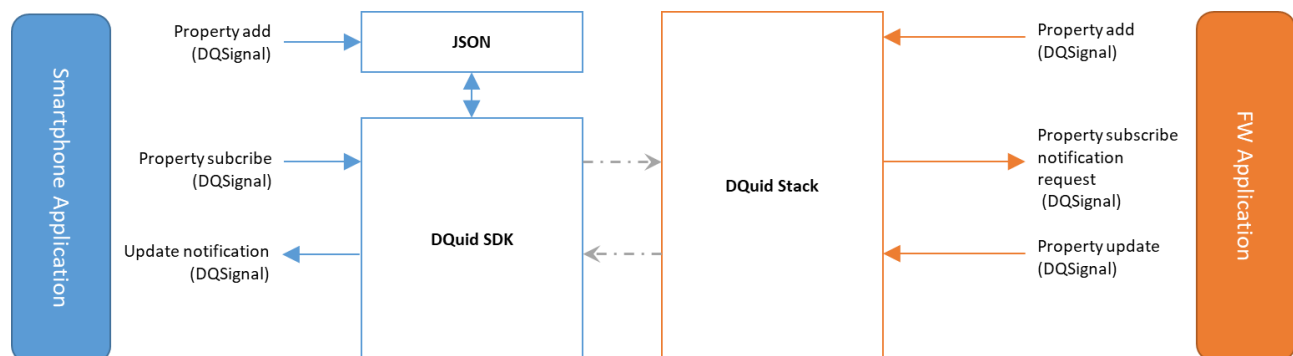


Figure 14: Subscribe/update the DQuidObject's properties

DQuidObject's properties are defined by the mobile application developer toward the DQuid SDK, and the properties are stored in JSON format (a sample is shown in Figure 15).

In order to implement the communication with the vehicle, the DQuid SDK provides a module for the CAN signal properties management.

Every CAN signal in a CAN message is translated into a DQSignal property. DQSignals properties of type CAN have additional attributes to store specific CAN signal data needed to extract signal's information from the CAN message:

- Start Bit
- Length
- CAN message ID



- CAN channel ID

DQuid SDK provides a method to parse the CAN db (Vector format .dbc) and automatically populate the properties' JSON file with all CAN signals and messages stored into the DBC.

```
{
  "objectName": "OBJECT_NAME",
  "serialNumber": "SERIAL_NUMBER",
  "unitType": 1,
  "visibility": "private",
  "objectProperties": [
    {
      "propertyName": "readProperty",
      "channel": {
        "type": 255,
        "id": 1
      },
      "propertyReadConf": {
        "payloadLen": 2,
        "propertyVisibility": "public",
        "identifier": 0,
        "dataType": 3
      },
      "propertyUm": ""
    }, {
      "propertyName": "writeProperty",
      "channel": {
        "type": 255,
        "id": 1
      },
      "propertyReadConf": {
        "payloadLen": 2,
        "propertyVisibility": "public",
        "identifier": 1,
        "dataType": 3
      },
      "propertyUm": ""
    }, {
      "propertyName": "readWriteProperty",
      "channel": {
        "type": 255,
        "id": 1
      },
      "propertyReadConf": {
        "payloadLen": 2,
        "propertyVisibility": "public",
        "identifier": 2,
        "dataType": 3
      },
      "propertyUm": ""
    }
  ]
}
```



Figure 15: Example of the structure of the properties in JSON

As regards the DQuid SDK, it provides the following features linked to the CAN Module:

- .dbc database parsing and automatic creation of the DQSignal properties with all attributes (Start Bit, Length, CAN message ID, CAN channel ID).
- Subscription/Unsubscription of CAN signals (properties' names are in the form «CANMessage.CANSignal»). For each CAN signal, it allows to define:
 - the transmit rate of each property from the DQuid Stack to the DQuid SDK
 - the way the signal is elaborated between consecutive BLE transmissions (LAS, AVERAGE, MIN, MAX). This elaboration is allowed only for 32bits unsigned signals.
- Subscription/Unsubscription of CAN messages (properties' names are in the form «CANMessage»). For each CAN message, it allows to define the transmit rate of each property from the DQuid Stack to the DQuid SDK.
- CAN signal/message update notification for all subscribed signals. The DQuid SDK applies the proper offset and scale factor to provide the CAN signal physical value to the mobile application.
- CAN message write
 - Optionally, it allows to define the CAN message transmit rate (on the CAN network)
 - It allows to update the payload of the CAN message transmitted over the CAN network.

On the other hand, for the DQuid Stack, it provides the following features linked to the CAN Module:

- CAN signals subscribe/unsubscribe notification
- CAN messages subscribe/unsubscribe notification
- It automatically updates the DQuid SDK with the subscribed CAN signal/message data according to the periodicity specified during subscription.
- It automatically elaborated the CAN signal value according to the configuration specified by the DQuid SDK during subscription.
- CAN signal/messages update: the DQuid Stack provides a method the embedded application can invoke when a CAN message is received from the CAN network. The Stack will then manage the update of all subscribed properties (CAN signals) embedded in the CAN message.

CAN message transmission on the CAN network: the transmission rate is defined by the configuration received by the DQuid SDK



7 Conclusion

In this deliverable D5.1, we presented the global AutoMATE System architecture, showing that it works on all demonstrators of the Automate project considering their different initial architectures. In addition, we refined the functional blocks of the TeamMate Car, including their interconnections and the data flow between them. In this way, we have also formally specified the system architecture, including the technical details/enablers of the three demonstrator platforms we are using: driving simulators provided by ULM, REL and VED; prototype vehicles provided by ULM, CRF and VED. An explanation of dataflow within the software construct and a clarification of stateflow concerning the teammate car is given as well.

Another important part is the definition of interfaces between the modules, as well as common data formats standards and communication protocols. Therefore, the TeamMate Application Programming Interface (API) has been defined in terms of principles, standards, interfaces and data structure that enable the communication of information between components in the TeamMate ecosystem. Thus, we presented a common design principle for the communication between components in the TeamMate ecosystem, based on exchanging messages in a publisher-subscriber messaging patterns. Messages are defined in terms of data structures with fixed semantics. We provided a first definition of a set of such data structures. It is worth to note here that the definition is non-exhaustive and may be subject to change if the need arises during integration and advances in the development of enablers for the TeamMate demonstrators.

Finally, in this document, we have also described the TeamMate Extension Standard Development Kit (SDK) for third party applications and hardware (smartphones, tablets, etc.) – at least in a first version – that are based on the TeamMate Car framework and the requirements from WP1. The SDK, together with the API, allow third parties to build new components or to replace or adjust TeamMate components. Therefore, the SDK provide the necessary libraries and APIs (e.g. for communication bus access), compilers and runtime environments (e.g. for testing).

This document constitutes the basis for the description of the baseline cars (deliverable D5.2) and of the first implementation of the TeamMate cars in the three demonstrators with a specific focus on the deployment (deliverable D5.3), as well as the refinement of the TeamMate System Architecture (including API) in the 3rd project cycle (deliverable D5.4) which will include: (1) the enablers architecture, (2) the final API for the teammate system.



8 References

- [1] E. B. Hamida, H. Noura and W. Znaidi, "Security of Cooperative Intelligent Transport Systems: Standards, Threats Analysis and Cryptographic Countermeasures", Electronics 2015, 4, pp. 380-423
- [2] ETSI EN 302 637-3 V1.2.1 – ITS Basic Set of Applications; Part 3: Specifications of Decentralized Environmental Notification Basic Service (2014-09)
- [3] ETSI TS 101 539-1 V1.1.1 – ITS V2X Applications; Part 1: Road Hazard Signalling (RHS) application requirements specification (2013-08)
- [4] Campolo, C., A. Molinaro, and R. Scopigno, "Vehicular ad hoc Networks. Standards, Solutions, and Research", Springer, 2015, ISBN: 978-3-319-15496-1

Appendix 1

This appendix aims to show examples of fragments of code generated for the API of the TeamMate architecture.

The resulting codes and the API are regularly updated and uploaded on the project repository.

<https://vprojects.offis.de/predict/ajaxplorer>

Username : reviewer

Password : Te4mM@te

under the following folder

/Workpackage Documents/WP5-TeamMate Architecture System Integration and
Implementation/Deliverables/D5.1/New/OpenAPI

Example of JAVA code generation

```
package eu.automate.openapi.messages;

public final class MapMessageOuterClass {
    private MapMessageOuterClass() {}
    public static void registerAllExtensions(
        com.google.protobuf.ExtensionRegistry registry) {
    }
    public interface MapMessageOrBuilder extends
        // @@protoc_insertion_point(interface_extends:eu.automate.openapi.messages.MapMessage)
        com.google.protobuf.MessageOrBuilder {

        /**
         * <code>required int32 id = 1;</code>
         *
         * <pre>
         * An unique ID
         * </pre>
         */
        boolean hasId();
        /**
         * <code>required int32 id = 1;</code>
         *
         * <pre>
         * An unique ID
         * </pre>
         */
        int getId();

        /**
         * <code>required int32 nbLanes = 2;</code>
         *
         * <pre>
         * The number of available lanes
         * </pre>
         */
        boolean hasNbLanes();
    }
}
```



```
/**
 * <code>required int32 nbLanes = 2;</code>
 *
 * <pre>
 * The number of available lanes
 * </pre>
 */
int getNbLanes();

/**
 * <code>required .eu.automate.openapi.messages.MapMessage.LaneInfo right_lane_info = 10;</code>
 */
boolean hasRightLaneInfo();
/**
 * <code>required .eu.automate.openapi.messages.MapMessage.LaneInfo right_lane_info = 10;</code>
 */
eu.automate.openapi.messages.MapMessageOuterClass.MapMessage.LaneInfo getRightLaneInfo();
/**
 * <code>required .eu.automate.openapi.messages.MapMessage.LaneInfo right_lane_info = 10;</code>
 */
eu.automate.openapi.messages.MapMessageOuterClass.MapMessage.LaneInfoOrBuilder
getRightLaneInfoOrBuilder();

/**
 * <code>required .eu.automate.openapi.messages.MapMessage.LaneInfo middle_lane_info = 20;</code>
 */
boolean hasMiddleLaneInfo();
/**
 * <code>required .eu.automate.openapi.messages.MapMessage.LaneInfo middle_lane_info = 20;</code>
 */
eu.automate.openapi.messages.MapMessageOuterClass.MapMessage.LaneInfo getMiddleLaneInfo();
/**
 * <code>required .eu.automate.openapi.messages.MapMessage.LaneInfo middle_lane_info = 20;</code>
 */
```

Example of C++ code generation

```
// MapMessage.h
// Generated by the protocol buffer compiler. DO NOT EDIT!
// source: MapMessage.proto

#ifndef PROTOBUF_MapMessage_2eproto__INCLUDED
#define PROTOBUF_MapMessage_2eproto__INCLUDED

#include <string>

#include <google/protobuf/stubs/common.h>

#if GOOGLE_PROTOBUF_VERSION < 2006000
#error This file was generated by a newer version of protoc which is
#error incompatible with your Protocol Buffer headers. Please update
#error your headers.
#endif
#if 2006001 < GOOGLE_PROTOBUF_MIN_PROTOC_VERSION
#error This file was generated by an older version of protoc which is
#error incompatible with your Protocol Buffer headers. Please
#error regenerate this file with a newer version of protoc.
#endif

#include <google/protobuf/generated_message_util.h>
#include <google/protobuf/message.h>
```




```
#include <google/protobuf/repeated_field.h>
#include <google/protobuf/extension_set.h>
#include <google/protobuf/unknown_field_set.h>
// @@protoc_insertion_point(includes)

namespace eu {
namespace automate {
namespace openapi {
namespace messages {

// Internal implementation detail -- do not call these.
void protobuf_AddDesc_MapMessage_2eproto();
void protobuf_AssignDesc_MapMessage_2eproto();
void protobuf_ShutdownFile_MapMessage_2eproto();

class MapMessage;
class MapMessage_LaneInfo;

// =====

class MapMessage_LaneInfo : public ::google::protobuf::Message {
public:
  MapMessage_LaneInfo();
  virtual ~MapMessage_LaneInfo();

  MapMessage_LaneInfo(const MapMessage_LaneInfo& from);

  inline MapMessage_LaneInfo& operator=(const MapMessage_LaneInfo& from) {
    CopyFrom(from);
    return *this;
  }

  inline const ::google::protobuf::UnknownFieldSet& unknown_fields() const {
    return _unknown_fields_;
  }

  inline ::google::protobuf::UnknownFieldSet* mutable_unknown_fields() {
    return &_unknown_fields_;
  }

  static const ::google::protobuf::Descriptor* descriptor();
  static const MapMessage_LaneInfo& default_instance();

  void Swap(MapMessage_LaneInfo* other);

  // implements Message -----
  MapMessage_LaneInfo* New() const;
  void CopyFrom(const ::google::protobuf::Message& from);
  void MergeFrom(const ::google::protobuf::Message& from);
  void CopyFrom(const MapMessage_LaneInfo& from);
  void MergeFrom(const MapMessage_LaneInfo& from);
  void Clear();
  bool IsInitialized() const;

  int ByteSize() const;
  bool MergePartialFromCodedStream(
    ::google::protobuf::io::CodedInputStream* input);
  void SerializeWithCachedSizes(
    ::google::protobuf::io::CodedOutputStream* output) const;
  ::google::protobuf::uint8* SerializeWithCachedSizesToArray(::google::protobuf::uint8* output) const;
  int GetCachedSize() const { return _cached_size_; }
private:
  void SharedCtor();
  void SharedDtor();
  void SetCachedSize(int size) const;
public:
  ::google::protobuf::Metadata GetMetadata() const;
```



```
// nested types -----

// accessors -----

// required bool availability = 1;
inline bool has_availability() const;
inline void clear_availability();
static const int kAvailabilityFieldNumber = 1;
inline bool availability() const;
inline void set_availability(bool value);

// required float center_x = 10;
inline bool has_center_x() const;
inline void clear_center_x();
static const int kCenterXFieldNumber = 10;
inline float center_x() const;
inline void set_center_x(float value);

// required float center_y = 11;
inline bool has_center_y() const;
inline void clear_center_y();
static const int kCenterYFieldNumber = 11;
inline float center_y() const;
inline void set_center_y(float value);

// required float half_width = 20;
inline bool has_half_width() const;
inline void clear_half_width();
static const int kHalfWidthFieldNumber = 20;
inline float half_width() const;
inline void set_half_width(float value);

// required int32 mandatory_speed_limit = 30;
inline bool has_mandatory_speed_limit() const;
inline void clear_mandatory_speed_limit();
static const int kMandatorySpeedLimitFieldNumber = 30;
inline ::google::protobuf::int32 mandatory_speed_limit() const;
inline void set_mandatory_speed_limit(::google::protobuf::int32 value);

// required int32 right_marking = 40;
inline bool has_right_marking() const;
inline void clear_right_marking();
static const int kRightMarkingFieldNumber = 40;
inline ::google::protobuf::int32 right_marking() const;
inline void set_right_marking(::google::protobuf::int32 value);

// required int32 left_marking = 41;
inline bool has_left_marking() const;
inline void clear_left_marking();
static const int kLeftMarkingFieldNumber = 41;
inline ::google::protobuf::int32 left_marking() const;
inline void set_left_marking(::google::protobuf::int32 value);

// optional float road_heading = 50;
inline bool has_road_heading() const;
inline void clear_road_heading();
static const int kRoadHeadingFieldNumber = 50;
inline float road_heading() const;
inline void set_road_heading(float value);
```