POLITECNICO DI TORINO

Master's Degree in Computer engineering

Politecnico
di Torino

Master's Degree Thesis

A methodology for the design of an automotive network architecture

Supervisors

Prof. Fulvio RISSO

Ing. Paolo PICCIAFOCO

Candidate

Luca VALENTINI

April 2022

Summary

More than 100 years have passed since the first car was put on the market, and the concept of the car has changed drastically over time. Initially, the car was conceived as a vehicle whose task was limited to transporting people from point A to point B. Now it continues to have its transport function but is also seen as a platform for technology, where software is as important as hardware. The introduction of new technologies and artificial intelligence has drastically revolutionized the automotive sector, forcing it to employ a different approach for the construction of the vehicle architecture, adopting a service-oriented methodology with the aim of optimizing computational performance. This thesis, carried out in collaboration with Italdesign Giugiaro S.p.A., is aimed at investigating and developing models for the design of a service-oriented network architecture using the PREEvision software. The PREEvision software uses a graphical notation for the design and requires the definition of requirements, use-cases, logic diagrams, software diagrams and hardware (network) diagrams for the generation of the communication model.

The project introduces the problem and complexity of modern vehicular architectures and shows a methodology for its design. By means of the PREEvision software and using the CAN network only, a simplified part of the network architecture concerning the adaptive cruise control functionality is created. Through the creation of various diagrams in the various layers provided by PREEvision, the network architecture is created, starting from a logical schema, up to the definition of signals, frames and the automatic generation of ARXML files for future ECU development. Afterwards, the architecture is interrogated through metrics, i.e. functions applied to the created communication model, to assess the efficiency of the architecture in the design phase.

Acknowledgements

Ringrazio la mia famiglia che mi ha permesso di intraprendere questo percorso di studi.

Ringrazio Giulia per essere stata fondamentale al fine di raggiungere questo grande traguardo. Grazie di esserci sempre.

Ringrazio i miei nonni Lorenzo, Maria, Giuseppe e Anna per l'amore che mi hanno saputo donare e per l'appoggio che non mi hanno mai fatto mancare.
A Nonno Lore, che nonostante il destino ci abbia diviso, continua a proteggermi e a fare parte dei miei pensieri.

Ringrazio tutti i miei amici perché il loro sostegno e le grandi aspettative che hanno sempre avuto nei miei confronti mi hanno spronato ad andare avanti e migliorarmi.

Ringrazio Paolo e Domenico per avermi seguito durante il percorso di tesi. Grazie alla loro disponibilità e alla loro conoscenza sono riuscito ad ambientarmi in un mondo che non era il mio.

Ringrazio il professore Fulvio Risso per la disponibilità e la passione che mi ha trasmesso in questi due anni di magistrale.

Table of Contents

List of Tables

List of Figures

Acronyms

AI
Artificial Intelligence

ECU
Electronic Control Unit

E/E
Electrical/Electronic

ACC
Adaptive Cruise Control

ARXML
AUTOSAR Xtensible Markup Language

CAN
Controller Area Network

CAN DBC
CAN database

SOA
Service Oriented Architecture

ADAS
Advanced driver assistance system

SW-C
Software Component

VFB

Virtual Function Bus

RTE

Runtime Environment

BSW

Basic Software

AUTOSAR

Automotive Open System Architecture

LDF

LIN Description File

SOME/IP

Scalable Service-Oriented MiddlewarE over IP

Chapter 1

Introduction

In the last years, technology has become part of our lives by changing people's habits. The technological innovation has improved in every field in terms of functionality, although increasing the complexity during the product life cycle. One of the industries most affected by the technological revolution is the automotive industry. The modern cars are very different from the first cars produced, because nowadays the concept of the car has changed drastically; in the beginning, people saw cars as a means of replacing animals without any kind of intelligence, but nowadays we see cars as a connected, smart and also autonomous means of transportation. Alternatively, it is possible to think of the automobile as a platform for technology. [1]

A short list of features in a car are shown in the image below (fig. 1.1). The features are executed by microprocessor-controlled devices also called ECU; modern cars can have more than 100 ECUs. The large number of ECUs generates several problems with regard to cost, complexity and space [2]. By increasing the number of features inside a car, more people will have to work on its development. The car's development is a complex process in which thousands of people are involved and must collaborate together, and each function developed by group of engineers must be tested and traced for safety reasons.

This thesis work will be carried out through the use of PREEvision software. The aim of this thesis is not to create a manual for PREEvision software, but to discover and test a methodology for the system design in automotive projects that employs the service-oriented architecture and model-based approach and to show its potential. The thesis has seven chapters: Chapter 1 introduces the problem and complexity of modern vehicular architecture; chapter 2 explains the main automotive concepts covered in this thesis, such as AUTOSAR; chapter 3 explains the division and use of the various layers of PREEvision, which will then be implemented to create the network architecture for adaptive cruise control; chapter 4 introduces the main concepts of the CAN network protocol, useful in the

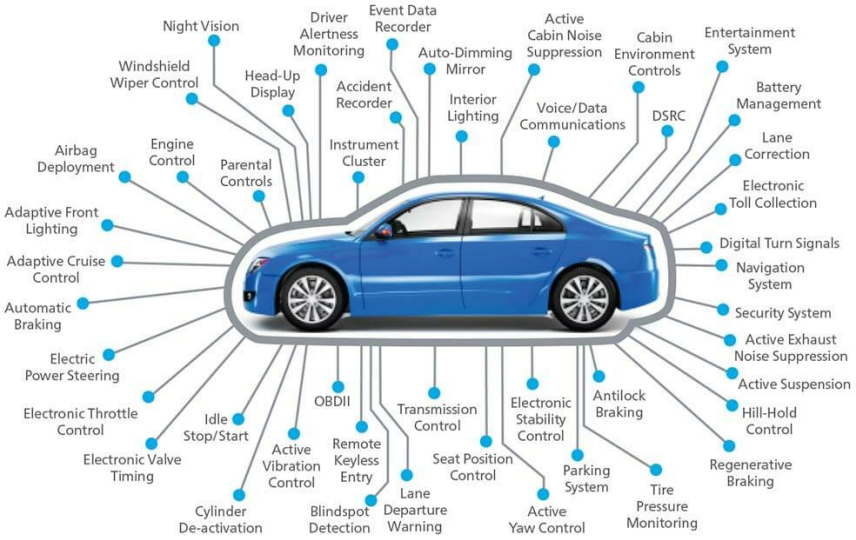


Figure 1.1: Car functionalities

following chapters since it is the only network protocol used in this thesis; in chapter 5, my architecture for adaptive cruise control functionality using the PREEvision software is created, explaining the strategy adopted; chapter 6 is devoted to the application to the previously created network architecture of the so-called metrics, i.e. functions which make it possible to interrogate the created architecture, in order to check its correct functioning; chapter 7 displays the pros and cons of this approach and the future developments of this thesis.

In image 1.2 the main features introduced over time in the automotive field are shown.

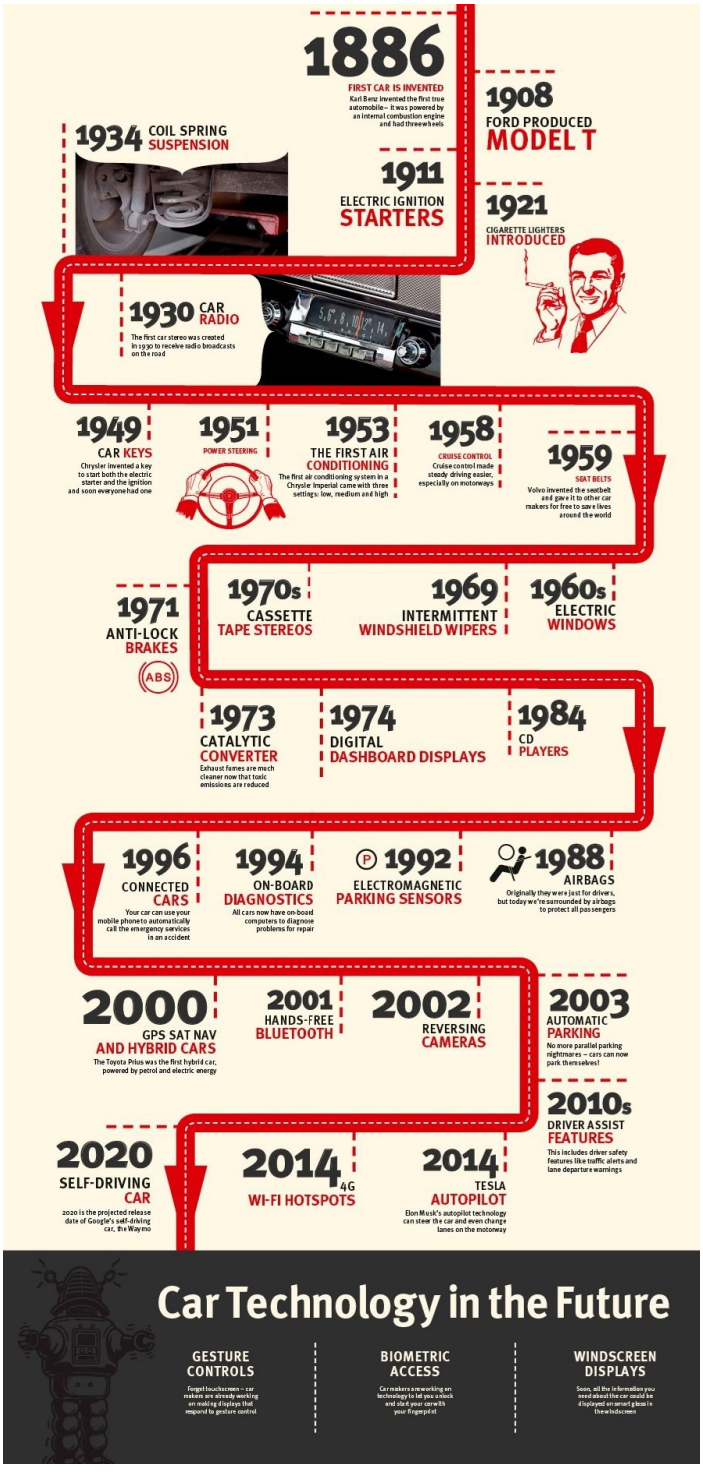


Figure 1.2: Car technology through the ages

Chapter 2

Automotive Background

In this chapter, the main automotive arguments discussed in the thesis are explained. The explanation will help understanding the aim and the strategies used in this thesis.

2.1 Electrical/Electronic Architecture

Modern vehicular architectures are composed of thousands of electronic components that make up the electrical and electronic (E/E) architecture. All the functionalities inside a vehicle, starting from the simplest ones such as windows or lights management, up to more advanced functionalities - such as autonomous driving or infotainment - are managed by electronic components that communicate with each other and also with sensors and actuators. The first introduction of electronic and mechanical components that interacted with each other to provide functionality dates back to the late 1950s, along with the introduction of the first version of cruise control. 70 years later, this functionality has changed drastically, by allowing the introduction of autonomous driving, and increasing driving safety through the ability to avoid collisions in case the driver is distracted. In order to create such complex and efficient functionality, the automobile became a platform for technology and created a very complex system of ECUs, actuators, sensors and connections between these components [3].

The E/E architecture is defined as the set of electronic components, software applications, network architecture and physical links that make up the entire automotive architecture. This is a generic definition of what is meant by E/E architecture. The E/E architecture could be viewed from different perspectives, which provide a better understanding of the utility of using such an architecture:

1. From the point of view of physical topology, it defines which are the components of the vehicle, such as sensors, ECUs, switches, power supply etc., and how they

are placed, including the network topology and the relative communication of all components.

2. From the logical topology point of view, it defines all the components and the relationships between them that make up the architecture from the logical point of view, leaving out the implementation and physical details of the elements.

3. The E/E architecture defines a set of rules for designing and building a vehicular system to meet functionality and performance requirements.

In the past, E/E architectures were very different from those of today. Early architectures had no domains, meaning each ECU performed a single function and there was no communication between ECUs. Over the years, the second type of E/E architecture has introduced the communication between ECUs and the concept of domain. The domain is a network formed by several ECUs that communicate with each other to perform functions inherent to that domain; communication is also possible between different domains. Initially, four domains were created: body/comfort, chassis, power train and infotainment. The problem with this architecture was the limited communication between domains. In modern architectures the central gateway has been added, i.e. a very powerful ECU connected to all domains, which allows communication between domains through itself. This architecture also allows to share resources between different domains through a direct communication that does not pass through the central gateway [4].

Today's central gateway-based architecture also has problems with inter-domain communication. Such a topology hides the domains' purposes and this can also be a problem for the software development of applications. For example, given several teams working on different functionalities, they could be running on the same ECU, risking to get very high load rates. From autonomous driving to connecting adjacent vehicles, the latest automotive innovations are leading to a new architecture design, which has to manage interaction between different domains in a more efficient way [5]. One solution could be to have a centralized domain that can manage the other domains.

The 2.1 image graphically shows the evolution of E/E architectures.

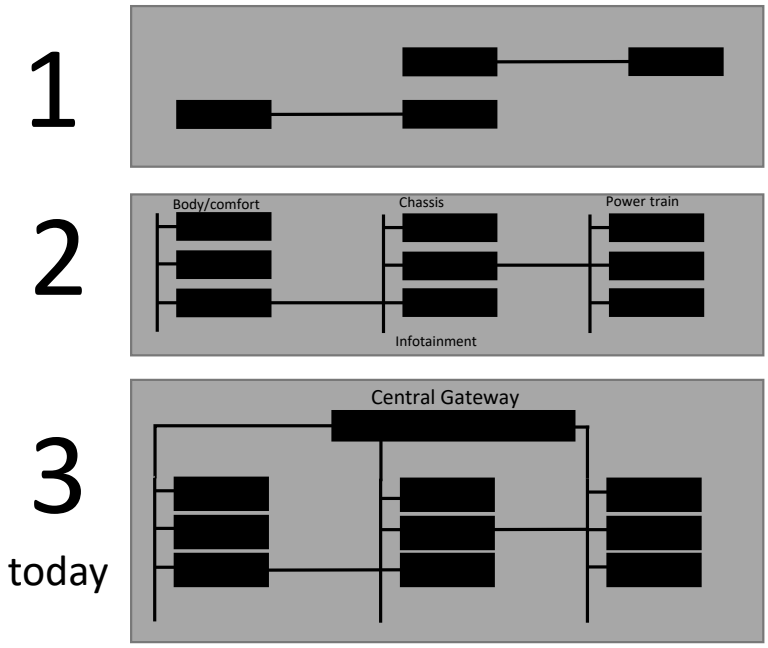


Figure 2.1: Evolution of E/E architecture

2.2 AUTOSAR

Until around 2004, the automotive E/E architecture was characterized by proprietary solutions, making the automotive sector a closed environment, where each automaker required components specific to Tier 1. Thanks to the creation and use of AUTOSAR, the automotive world has drastically changed the way cars are built. AUTOSAR stands for Automotive Open System Architecture, which is an open architecture for deploying software in ECUs. AUTOSAR is a consortium founded in 2003 by BMW, Robert Bosch GmbH, Continental AG, Daimler AG, Siemens VDO, and Volkswagen [6]. Before the advent of AUTOSAR, solutions in the automotive world were tailor-made, meaning OEMs required Tier 1 specific solutions. This was a very expensive solution, because a lot of resources were reserved for just one OEM. In order to reduce this cost, perhaps by sharing this cost among several OEMs, AUTOSAR was born. AUTOSAR allows to create an ECU by putting together different components chosen by the OEMs.

As the image 2.2 shows, AUTOSAR is composed of OEMs which are the car manufacturers, Tier 1 which are the suppliers of OEMs producing ECUs, software standards for ECU development, tools and services and semi-conductor companies. In this architecture, software standards that allow to open the market to all suppliers programming ECUs following these precise standards are defined. In

order to program these ECUs, the programmer employs different tools and services, which also allow him/her to hide the implementation choices. In the consortium there are also the companies that produce the semi-conductors. In short, the AUTOSAR consortium is composed of OEMs (the ones that make the requests for certain functionalities), the Tiers 1 (the ones who are in charge of providing a solution to the OEMs request). The Tier 1's task of giving a response to the OEMs request is possible thanks to tools and services, which allow them to generate standard codes. Finally there are the semi-conductor companies.

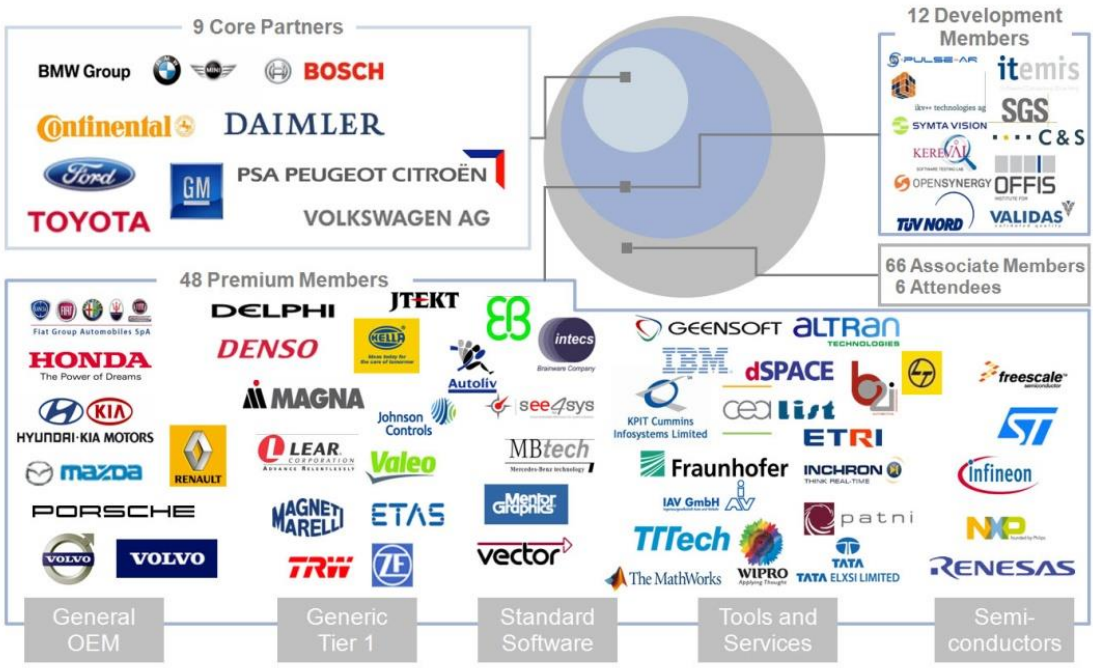


Figure 2.2: AUTOSAR partners

To make the ECU development process as generic as possible, ECUs are divided into standard components:

- Application Software
- Basic Software
- Hardware

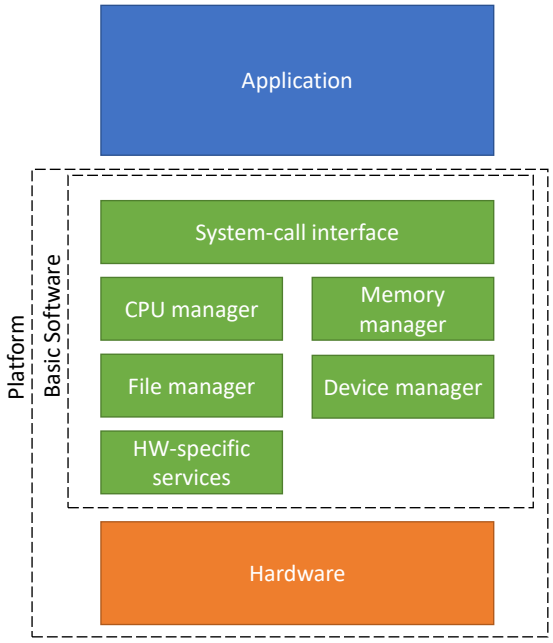


Figure 2.3: ECU division into standard components

Basic software fits on top of the hardware and exposes standardized set of services. Above the basic software is the application, where these standardized set of services are used. The advantage of having a standard is that it provides a well-defined interface, giving the ability to decouple the development of the application from the development of the platform, developing both parts in parallel, because in this way we know from the beginning how these two parts will communicate. This is a very important advantage for OEMs, because they can much better control the cost of platform and application development, by assigning different competitors to develop the components. Tiers 1 also have advantages, because one platform can be reused across multiple applications [7].

The image 2.4 shows how communication occurs across AUTOSAR components. Assuming we have a request that comes from an OEM to develop a feature in an application, developers start working in an abstract layer leaving out the hardware details. Developers will work with AUTOSAR SW-Cs that contain the functionality that needs to be developed. The SW-C descriptions define the properties of the application, in terms of how many building blocks need to be put together, how they communicate, and what functionality is assigned to that SW-C. AUTOSAR SW-Cs interact with each other through virtual buses, which are abstract interfaces. AUTOSAR provides a tool-chain to select which SW-Cs are to be put into the ECUs that make up the platform. The tool-chain (Deployment tools) in the center of the image takes the information about what the ECUs are capable of doing,

then takes the information about the functionality of the SW-Cs and produces the software code to integrate the different SW-Cs into the ECUs [7].

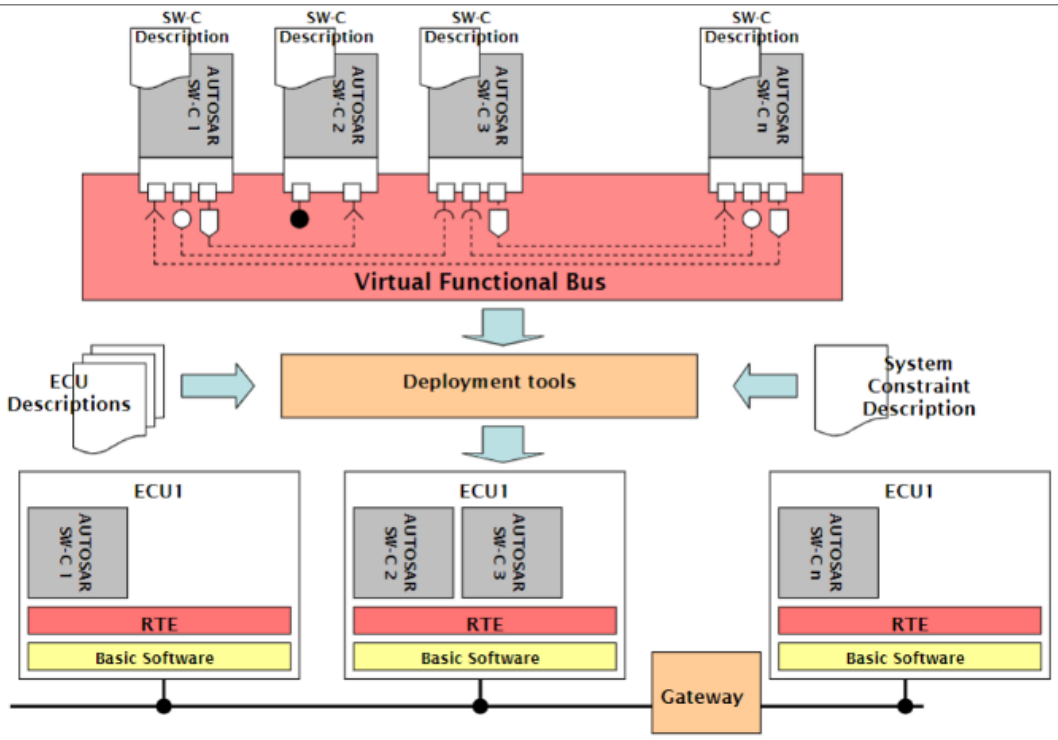


Figure 2.4: AUTOSAR components and their communication

Inside an ECU there is RTE (run time environment) that implements the communication described in the SW-C description, and allows different SW-C to communicate. For instance, in the second ECU, SW-C 3 requests a service from SW-C 2 and the communication will be handled by the RTE [7]. With this architecture, the application is platform-independent, so the same application can be used on different platforms using the deployment tools, i.e. the tool-chain to generate the code. Through this approach, it is possible to obtain a portable system. In summary, the components of the AUTOSAR architecture are as follows:

Software Component They are the atomic components of an application, i.e. software components which cannot be decomposed and deployed in different ECUs. The SW-Cs are always in one ECU.

Virtual Functional Bus It is a set of logical connections through which the SW-Cs communicate.

Runtime Environment It is the specific code for an ECU. It is the code that is

generated once the mapping between the component software and the ECU is defined.

Basic Software This is also an ECU specific code that provides the services used by the RTE to enable the execution of the software components.

AUTOSAR has two versions: classic and adaptive. AUTOSAR Classic allows the development of basic functions within a car, such as turning on lights, managing pedals, etc., and does not provide the ability to interact with components outside the vehicle to create more complex functionality. AUTOSAR Adaptive, on the other hand, supports complex applications with the ability to communicate with components outside the vehicle, by using a much more powerful hardware architecture. Through AUTOSAR Adaptive, it is possible to create applications for autonomous driving, where high hardware performances are required. In this thesis, only AUTOSAR Classic is used.

2.2.1 ARXML file

An ARXML file is a database of AUTOSAR configurations and is used to describe different aspects of the ECUs, such as software components, communication, etc. This file type was created to standardize data exchange between the various companies that make up the automotive market. ARXML files are classified in different types:

System Configuration It contains information related to the ECUs, especially messages, signals sent and received by the ECUs, ports and interfaces of the software components present in the ECUs.

ECU Extract It is a part of a system configuration file. OEMs prepare system configuration files, while Tiers 1, who are responsible for an individual ECU, are only interested in the ECUs they have to make, so they use ECU extract files for the development of individual ECUs. This file contains the messages, signals sent and received by an ECU, but also the details of SWCs like for example ports and interfaces of an ECU.

ECU Configuration The input of the ECU configuration file is a portion of the ECU extract file. The purpose of this file is to provide configuration parameters for the BSW modules within an ECU. Starting from this file, it is possible to generate and build the ECU executable. The relation between system configuration, ECU extract and ECU configuration file is shown in the following picture.

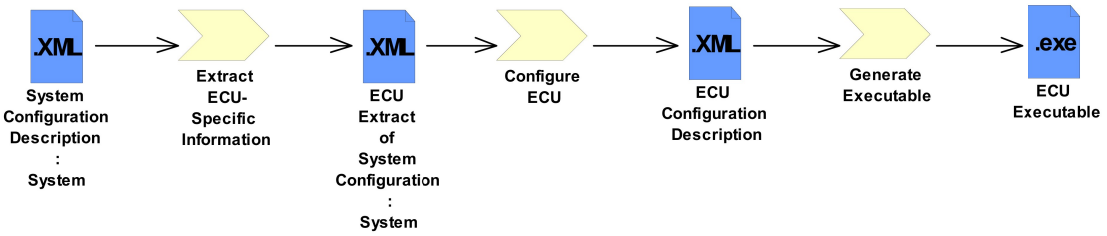


Figure 2.5: ARXML file chain

SWC Description It is specific to software components; it contains the ports and interfaces of a particular SWC. The difference with ECU extract files is that this file does not contain any message or signal.

BSW Module Description It contains the definition of the data structures of all configurable parameters of the BSW modules. These files are implementation specific and are part of the static code of the BSW modules.

2.3 Service Oriented Architecture

As the chapter 1 reports, modern vehicles are no longer just mechanical components for the purpose of moving the automobile, but they are a platform for the technology with their own intelligence. The automobile, seen as a technological platform, is dependent on the embedded systems, making their efficiency very relevant in order for its performance to be optimal. The high number of ECUs and features inside an automobile forces manufactures to create a very efficient system, that allows ECUs to be connected and to take advantage of functionalities. Some examples of advance functionalities are the autonomous driving, or the ability to install apps inside the car like a smartphone. In order to understand the problem with today's systems, it is necessary to understand the difference between complex system and traditional system; the difference is that the latter is homogeneous, bounded and static, while the first one is heterogeneous, unbounded, dynamic and undefined. Modeling and designing these new complexes engineered systems requires intern and alternative paradigms in system architecture.

Service Oriented Architecture (SOA) is the most used and efficient software design paradigm for high level systems in the automotive field. [8] SOA is an approach or a template software architecture for distributed systems architecture that employs coupled services, standard interfaces and protocols to deliver cross platform features. In this approach, the participants provide or consume services (features) using a defined protocol over a network. The goal is to distribute logic

over a number of atomic services that will interact with each other to provide more complex functions [9].

Services are the most important components of a service-oriented architecture. A service can be any type of functionality, any type of feature or any type of data. Services are loosely coupled to the system; this means that these different modules can join or leave, couple or decouple from the system, thus maintaining their independence and re-usability. A service [10]:

- represents functional unit
- is a black box for consumers
- is self-contained
- is stateless
- can consist of underlying services
- uses standardized interfaces to communicate
- re-usability

A service provides information and functions available to consumers when it acts as a provider, while when it acts as a consumer, it uses the data or functions of other services to provide its own functions. A complex service consists of multiple simple services.[9] The communication between services is made by service interface that consists of:

- method
- property (field, attribute)
- event

One of the possible methods is an operation called by a service, but executed by another service, which can expose result data through properties. The service that calls the method can also receive notification of changes of the field's value through event.

In the picture below 2.6 there is an example of a service oriented architecture, which can be used in navigator inside a car. The main service *LocalTraffic* gives information about the traffic for a specific city. The service plays both a provider and a consumer role. The communication between the service provider and the service consumer is made by the interface *LocalTrafficInterface*. The consumer requests the traffic for a certain city and the main service bundles the information and makes it available to consumers. Inorder to do this, the *LocalTraffic* gets information from other services. The *Location* service returns a city on the basis of the geographical coordinates. This allows to request the traffic status for a city from the *Traffic* service.

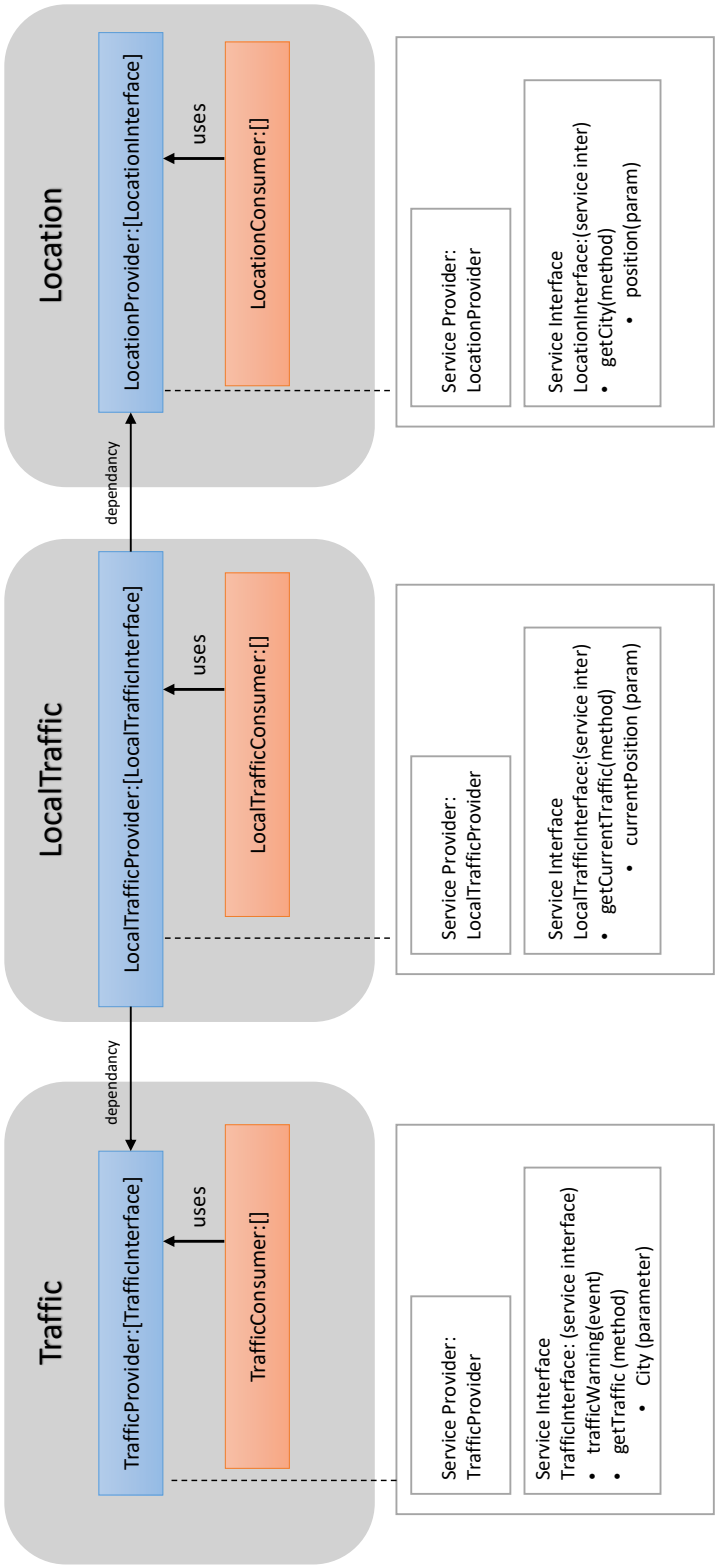


Figure 2.6: Service architecture for a traffic service

In conclusion, SOA describes the interaction between services made by service interfaces. The power of its abstract design is the creation of AUTOSAR compatible systems and the advantages can be listed as follows:

Loosely coupled Services are independent, changes in one service do not affect other services

Agility Reusable services allow to reuse code and reduce the number of lines written by developer

Scalability Services can run across multiple platforms. A service can be executed in all ECUs present in a car.

Reliability Services are small and independent, thus making it easier to test and debug application

Easy Maintenance Maintaining or updating the application is easy because the service-oriented architecture is an independent unit

2.4 V-Cycle

Software applications in the automotive world need a high degree of reliability because of their application field, but most of all, once installed in the car, they have to work well even without future updates. For example, the Adaptive Cruise Control feature is developed by the car manufacturer, and once it is installed in the car, it must work properly since the very beginning. This requires that the number of bugs within the programmed software is close to zero, and that many tests have been done to verify that the software works properly. Since the scope is critical to human safety, a slightly different approach than the waterfall approach must be used for programming the software. The traditional approach to software development is called waterfall. In this approach, software development phases are executed sequentially, because each phase depends on the previous one. The phases are:

Requirements All the specifications that the software needs are brought together. What the software needs to do is defined.

Design The programming languages and frameworks to be used are chosen, and the software architecture is defined.

Development The application is developed.

Testing & Integration Tests are performed to verify the correct functioning of the application

Deployment The application is released for use.

Maintenance The application is modified through updates based on customer requests.

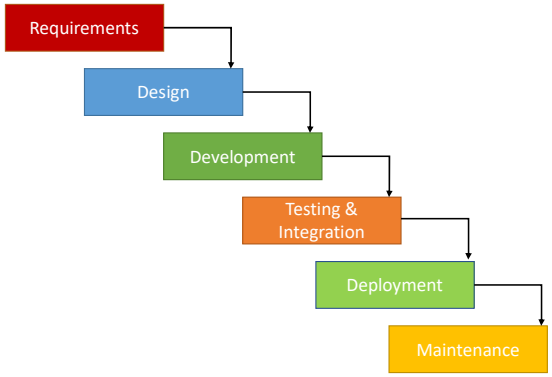


Figure 2.7: Waterfall approach

This approach performs testing only after the development phase is complete. In very complex systems, modifying parts in the previous phases in case of an error significantly increases the time to release the software, because each phase will have to be modified. In the automotive industry, there is a need to perform many more tests to avoid such situations, but also to create a more reliable system.

The solution is to adopt the V-Cycle model/approach. In V-cycle, "v" stands for verification and validation. In this approach, the development process is divided into two parts, one for each arm of the V. In this model, two macro phases are defined, the system definition phase and the test phase. Each macro phase has micro phases in which actions are performed; each micro phase belonging to one arm of the V communicates and interacts with the other micro phase in the other arm, so each definition phase (phase in the first arm) has a related verification and validation phase, that is defined before proceeding to the next phase. The individual phases are:

Requirements Customer requirements are defined to understand the goal of the application.

System Design The hardware and communication between the components that will be used for the application is defined.

Architecture Design The architecture of the application and the communication of the various modules that compose it are defined. The communication can be internal to the system or external.

Module Design The internal workings of all the modules that make up the system are defined in detail.

Coding The code is developed

Unit Test The operation of each module is verified.

Integration Test The operation of the finished product is verified once all modules are assembled together.

System Test It is verified that the created system meets the initial specifications.

Acceptance Test The application is validated by the client.

At each stage of system definition, tests for that stage are created and run when the system is completed.

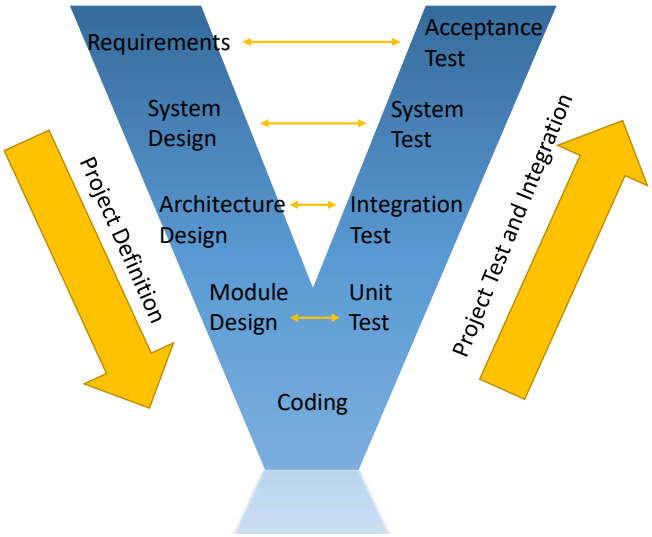


Figure 2.8: V-Cycle approach

All automotive applications are developed following this approach, because it allows to create very complex systems with an excellent quality and reliability of the final product.

Chapter 3

PREEvision software

In this chapter there is a short explanation about the use of PREEvision and its functionalities. More example are reported in the next chapter where there is the explanation of the project using this software.

3.1 Vector Group

Vector Informatik is a computer company specialized in software and component development for networking of electronic systems based on the serial bus systems. Vector developed software tools like CANalyzer to analyze network protocols, CANoe for development, testing and analysis of ECUs and entire ECU networks. For my thesis, the most important one is PREEvision and it will be explained in the next section.

Figure 3.1: Vector Informatik group logo

3.2 PREEvision

Given the large number of functionalities and hardware components inside a car, such as autonomous driving, the design of a car is a very complex and critical process. Nowadays the information needed by an ECU to work properly can be

received locally or from the cloud. Due to the large number of ECUs and their reliability in a car, an ECU must first be designed logically tracing all the signals that are sent and received, and then developed.

PREEvision is a tool for model-based development of distributed, embedded systems in the automotive industry; it supports the entire development process in a single and integrated application. PREEvision has a lot of features for developing classic or service-oriented architectures, managing architecture requirements, designing communication in an automotive architecture, designing vehicle safety, developing AUTOSAR-compatible software and architectures, and even managing wiring harnesses. [11]. Using this software all functions provided by E/E system of a modern vehicle are assigned in different domains; Using this software all functions provided by the E/E system of a modern vehicle are assigned in different domains; the functions are networked within the domains, but can also communicate with external domains. For example, advanced driver assistance functions communicate through the network with transmission management, engine management, steering, and braking systems. [12]

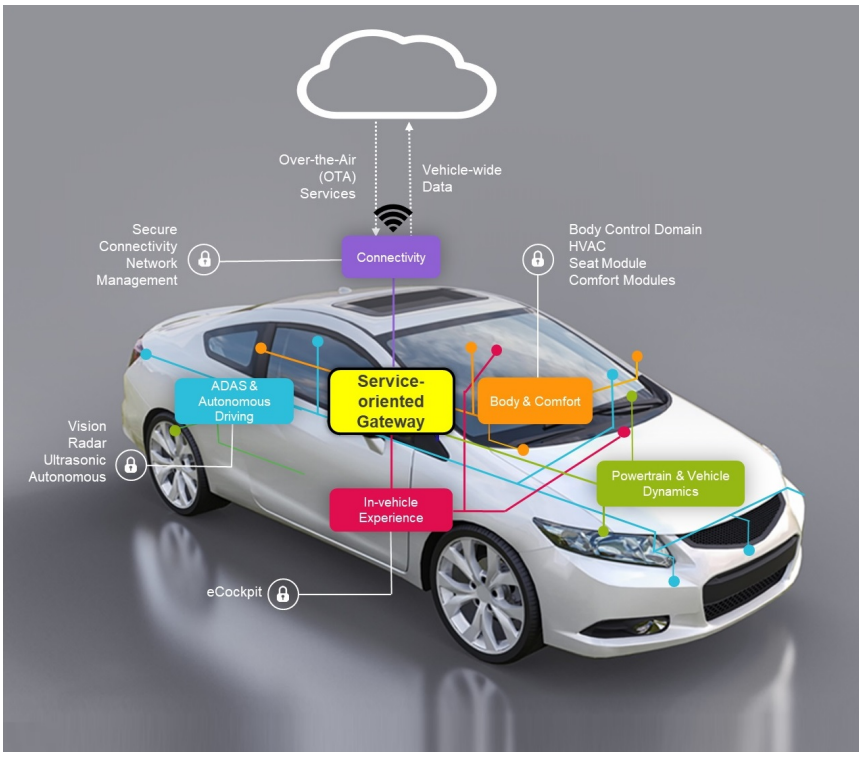


Figure 3.2: Vehicle domains and connectivity

Today new functionalities have influenced and changed E/E architectures drastically and they can be summarized as follows:

- New power-train concepts as electric and hybrid drives
- Cloud integration
- Advanced driver assistance systems
- New user interfaces such as gesture control.

PREEvision allows to design all these aspects and more, creating the best E/E architecture with a model-based approach, supporting the design not only for a single vehicle, but also supporting a complete product line of vehicles including many variant options.[12] It uses a wide, domain-specific graphical specification language and data model optimized for the development of embedded E/E systems; it has an architecture modeling layers:

- Product goals
  - Customer features
  - Use cases
  - Requirements
- Logical function architecture
- System software/service architecture (according to AUTOSAR)
  - Service-oriented architecture
  - Software library
  - System software architecture
  - Implementation
- Hardware architecture
  - Hardware component architecture
  - Hardware network topology (according to AUTOSAR)
  - Electric circuit
- Wiring harness
  - Wiring harness
  - Geometry (vehicle topology)
- Communication (according to AUTOSAR)

All these layer are linked together by mappings.

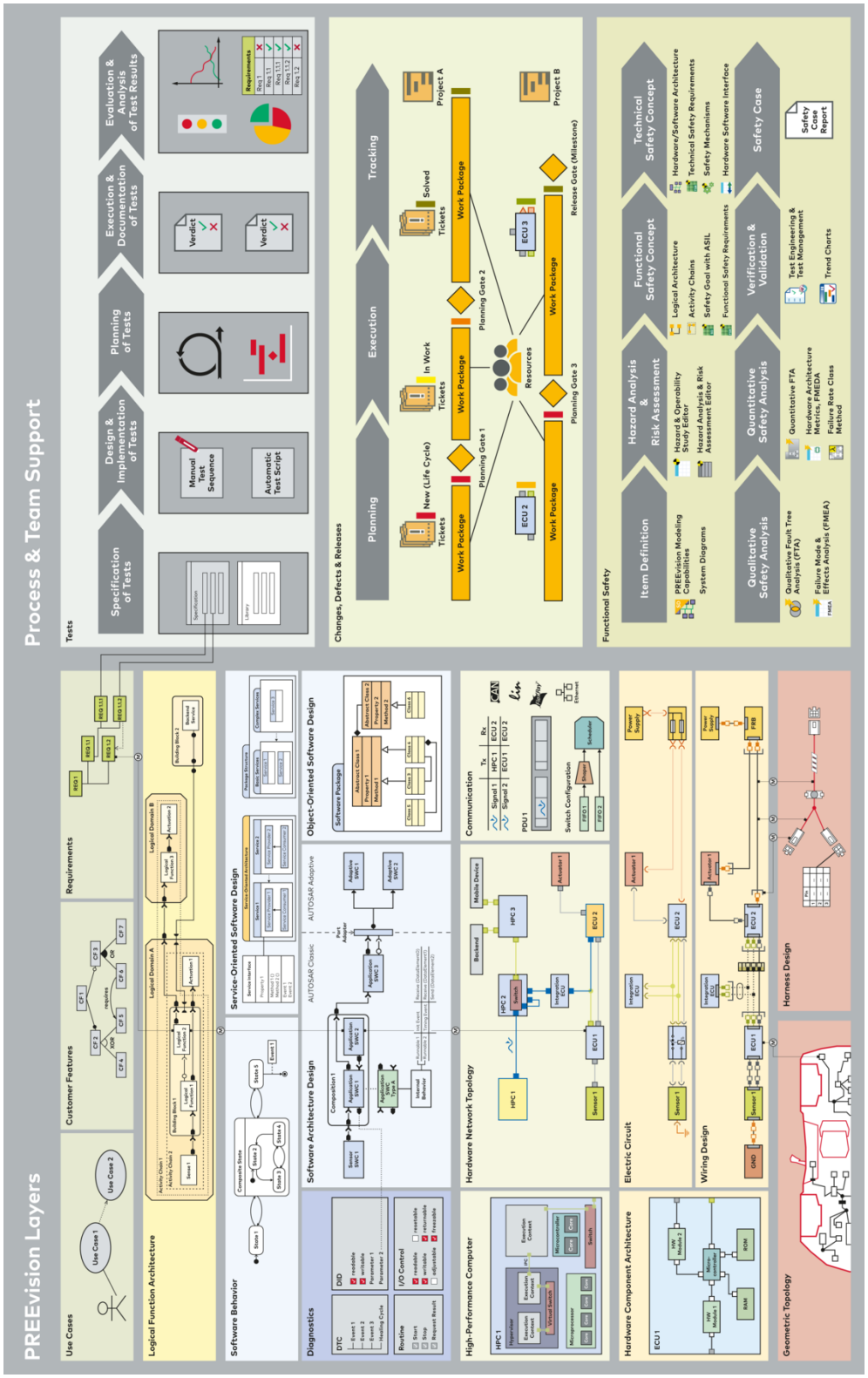


Figure 3.3: PREEvision layers

For this thesis I didn't use all layers, I just used customer features, requirements, logical function architecture, system software architecture, hardware network topology and communication. In the following sections there is a brief introduction to these layers, but the full explanation is in chapter 5.

3.2.1 Customer features and Requirements

The customer features describe the characteristics of a vehicle or a functionality, which are used to specify the features in a non-technical manner. A set of customer features can describe all or most of the functions of a vehicle. An example of a customer feature can be the description of the start-stop functionality or the car's air conditioning.

After writing the customer feature, one should write the requirements, i.e. a redefinition of customer features in a technical manner, where one describes in detail how those functionalities work. An example of requirement can be the description of an ECU that is in charge of managing the start-stop functionality. There are different types of requirements:

Requirement (shall) It is a requirement that engineers should meet.

> e.g Center of mass should be at center of the vehicle (top view) and 40 cm from bottom.
>
> It may not be exactly in the center.

Definition (must) It is a requirement that engineers must meet

> e.g The maximum weight of the vehicle is not to exceed 1.5 tons.
>
> This requirement must be satisfied.

Information (can) It is an additional requirement, not relevant to the functioning. It can also be used to write useful information.

> e.g Opening and closing of doors should make a nice sound.

Creating a link between customer features and requirements for traceability reasons is a good practice.

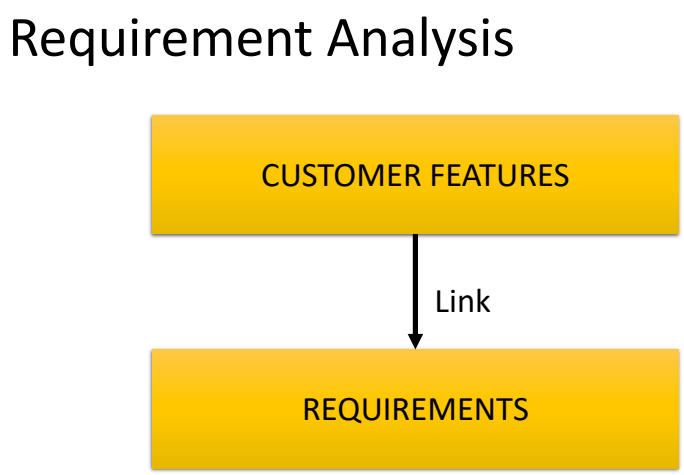


Figure 3.4: Customer Features and Requirements link

3.2.2 Logical, software and hardware architecture

The logical function architecture provides graphical block elements that allow to define an abstract graphical functional network; The graphical elements are used to describe logical functions, sensors, actuators, building blocks (container of blocks), etc.

In the software architecture, software components and their communication are designed graphically, meaning that no code is required. Even if you can add to graphical software components written code, PREEvision objective is to design the architecture. As for the development, you should use other tools. The software architecture supports the AUTOSAR methodology.

The hardware architecture allows the modelling of physical components, such as the electronic components inside an ECU, actuator and sensor but also their networking. The hardware architecture supports the AUTOSAR methodology. The mapping between hardware architecture components and logical or software components is required to use PREEvision functionalities such as the auto generation of signals.

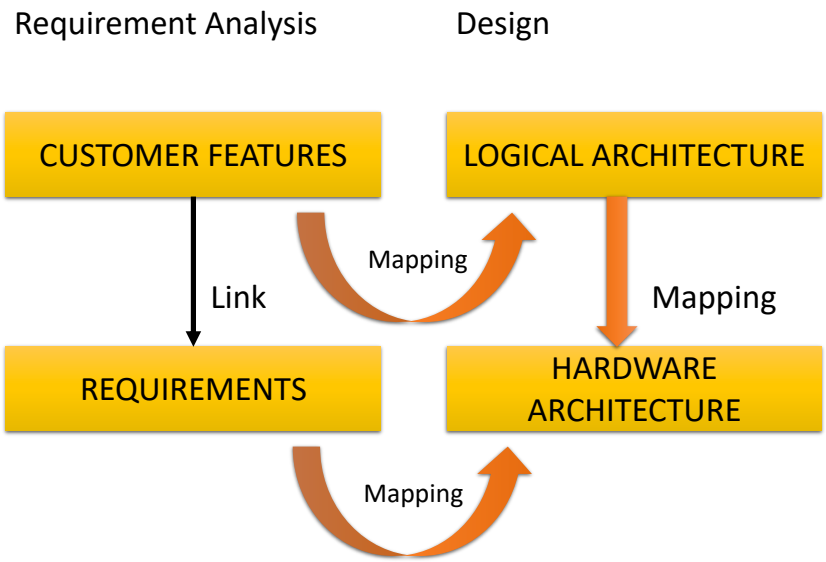


Figure 3.5: Relationship between customer features, requirements, logical architecture and hardware architecture mapping

3.2.3 Communication

The PREEvision communication design supports the AUTOSAR communication design for all bus systems, such as LIN, CAN, CAN FD, FlexRay and Ethernet. The communication layer contains all communication information such as signals, PDU, frame. PREEvision offers table editors and automations for signal, frame and PDU generation. In order to design network communication the *Logical Function Architecture*, *System Software Architecture* and *Hardware Architecture* must be defined, because specific components of these layers are needed. In PREEvision, all components in the various layers are called artifacts; in the communication layer, the main artifacts for this thesis are:

Signal IPDU It is a data packet that may contain one or more signals. PDUs are transmitted via frames.

Signal It is the data exchanged between ECUs.

PDU Transmission It's the transmissions of a PDU and its signals on a frame to which the PDU is assigned.

CAN Frame It contains and transmits PDUs

CAN Frame Transmission It is the representation of a CAN Frame, containing PDUs, on a bus.

CAN Signal Transmission It contains the information for the transmission of a CAN Signal.

Gateway A gateway ECU is an ECU that connects two or more ECUs. The frames between the two ECUs will pass through the gateway ECU.

Signal Gateway Routing Entry It indicates the incoming signals and outgoing signals transmissions over buses.

PDU Gateway Routing Entry It indicates the incoming PDUs and outgoing PDUs transmissions over buses.

Frame Gateway Routing Entry It indicates the incoming Frames and outgoing Frames transmissions over buses.

Channel Communication It groups relevant communication information like ECU Interfaces, Frame, PDU and Signal Transmissions.

(This information are taken from PREEvision's manual [11])
The relations between signal, PDU and frame are shown in the following figure.

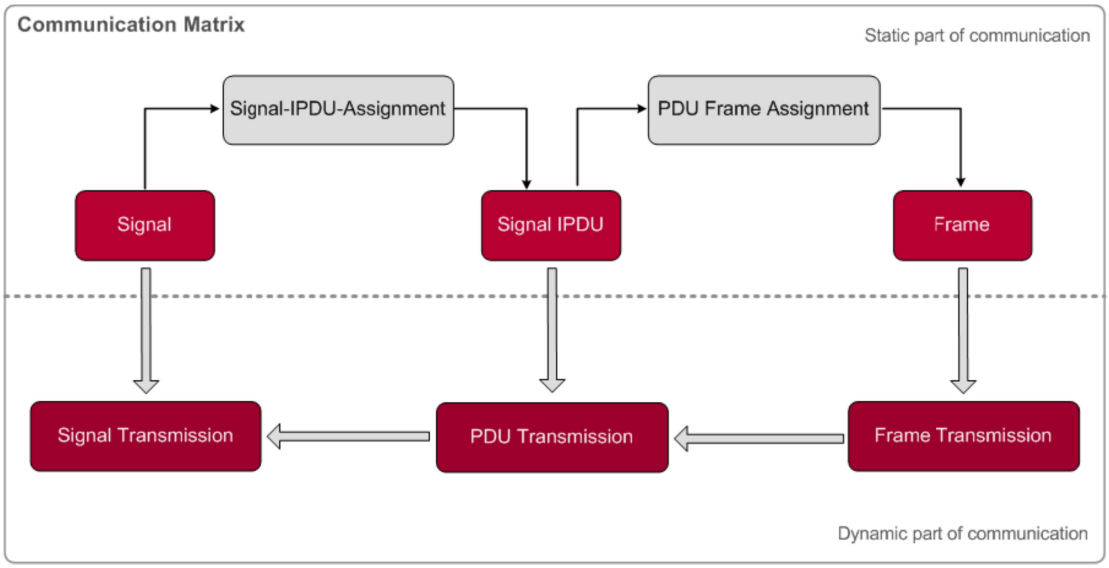


Figure 3.6: Signal, PDU and Frame relationship

3.3 Software and communication design process in PREEvision according to AUTOSAR

In order to design communication in PREEvision that allows export of ARXML files, there are several steps to follow represented in the figure 3.7. The process starts with the creation of software and hardware architecture. When these two architectures are finished, the software components must be mapped to the hardware components where they should be executed. The mapping between software and hardware refers to the assignment of each software component to an ECU, meaning that a particular software component runs in that ECU. Software components communicate with each other through data elements on a bus. The signal router creates signals and signals transmissions based on software-hardware mapping, and automatically maps data elements to corresponding signals. Running the signal router also creates network routing. At the end, it's possible to design the network communication for every bus such as CAN, CAN FD, LIN, FlexRay and Ethernet. With these design steps, PREEvision can export ARXML files useful for ECUs implementation using AUTOSAR the basic software modules. (This information are taken from PREEvision's manual [11])

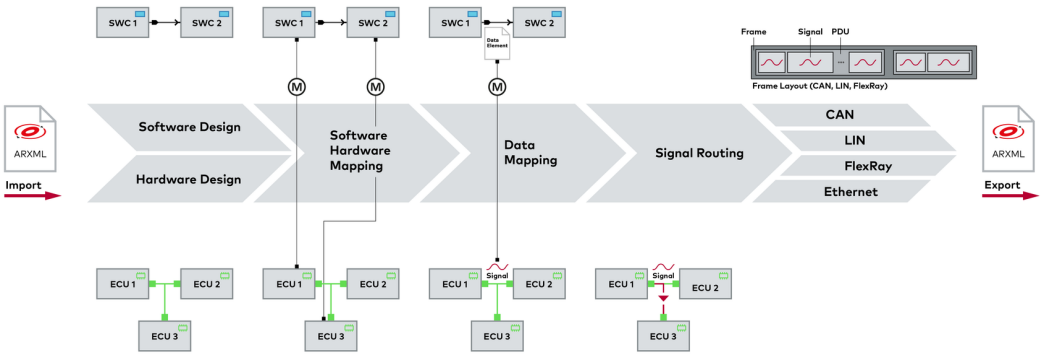


Figure 3.7: AUTOSAR system and software design process in PREEvision

"Signal routing is an automation mechanism which is able to automatically analyze the defined communication dataflow to determine the best signal routes, including, for communication bus systems, the optimal points at which to locate

inter-bus gateways. The signal router is also able to generate the signals and automatically carry out the required system signal mappings" [11].

After the signal routing the Frame-PDU synthesis allows the generation of Frame Transmissions and PDU Transmissions. "The frame-PDU synthesis calculates the transmissions based on the static communication information of the specific network communication CAN, LIN or FlexRay" [11].

Chapter 4

CAN protocol

The first cars with some electronic components had few independent ECUs in charge of executing few functionalities, there was no communication between them. In order to implement advanced functionality, several ECUs had to communicate with each other. At the beginning, the communication was carried out by physical channels for each transmitted signal, then network protocols were introduced. Due to its performance, upgrade-ability and flexibility in system design, the most widespread protocol in automotive is the CAN protocol.

This chapter shows a brief introduction of the CAN protocol.

4.1 Introduction

The CAN (controller area network) bus protocol was invented by Robert Bosch GmbH in 1986 for the automotive industry. Before CAN bus gained popularity, vehicle wiring harnesses could contain miles of wire. The CAN network allows to create a robust communication between ECUs with a single cable with a "high immunity to electrical interference and the ability to self-diagnose and repair data errors" [13].

The CAN protocol is implemented in hardware and defines the Data Link Layer and part of the Physical Layer in the OSI model, the other layers can either be implemented using existing non-proprietary higher layer protocols or they can be defined by the system designer.

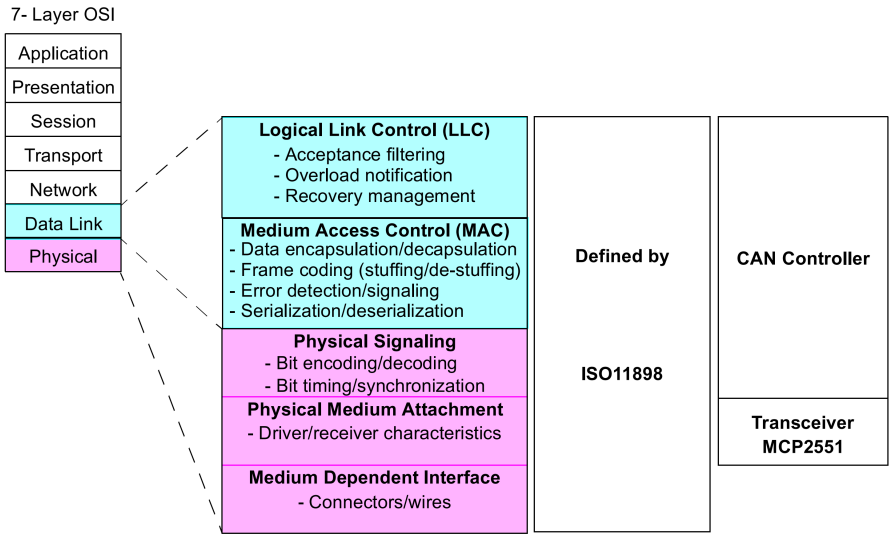


Figure 4.1: CAN and the OSI model

The two ISO documents ISO 11898-2 and ISO 11898-3 describe two different CAN physical layers:

- High-speed CAN physical layer
- Low-speed CAN physical layer

The differences between these two physical layers are the data transmission rates and the voltages. The maximum speed for High-speed CAN (ISO 11898-2) is 1 Mbit/second while Low-speed CAN (ISO 11898-3) can go up to 125 kbit/second. A CAN network has a set of CAN nodes linked via a CAN bus, a linear and shared bus. The maximum network extension is 40 m with a maximum of 32 nodes. "The CAN communication protocol is a carrier-sense, multiple-access protocol with collision detection and arbitration on message priority (CSMA/CD+AMP) "[13], so each node before sending data verifies if the bus is busy and in case of collisions, they will be resolved through a bit-wise arbitration in the identifier field of a message. The bus access will always be given to the higher priority identifier, that is the message with the smaller identifier.

4.2 Node

An ECU in a CAN network is called node and two or more nodes create a network. The nodes are connected between them through a can bus. Each node assigns a CAN identifier to the messages it sends that make them unique in the network and

allows the message arbitration, the process in which nodes decide who can access the bus. The message arbitration is explained in the following chapter 4.4.

A CAN node consists of:

- Microcontroller
- CAN controller
- Transceiver

The microcontroller decides which messages must be sent and it understands the content of received messages.

The CAN controller acts as a bridge between the microcontroller and the CAN transceiver; it receives serial bits sent by other nodes through the CAN transceiver, and it stores them in memory so that the microcontroller can retrieve them. It also sends serial bits from microcontroller to CAN transceiver.

The CAN transceiver converts the data stream received and sent by CAN bus, into a format useful for the controller and vice versa. Figure 4.2 shows the components of a CAN node and their relationships.

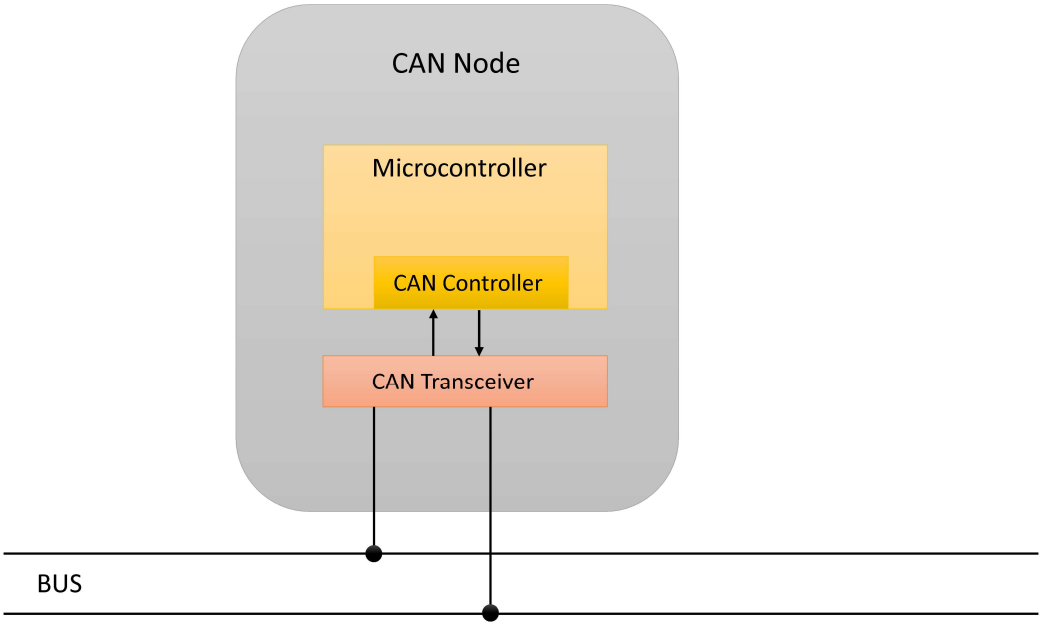


Figure 4.2: Can node components

4.3 Frame

A CAN network can be configured for two different frame formats:

- Base frame format
- Extended frame format

The main difference is the identifier length, because the base frame format has 11 bits for the identifier having at most $2^{11} = 2048$ identifiers, while the extended frame format reserves 29 bits having at most $2^{29} = 537$ million identifiers.

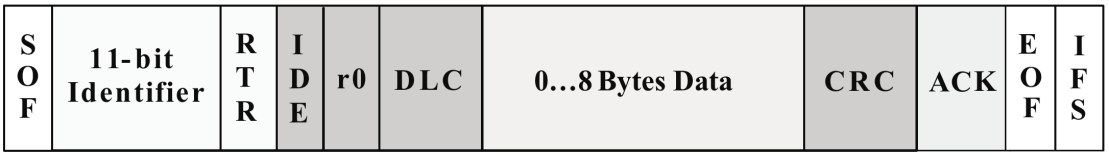


Figure 4.3: Base frame format

The image above shows the bit fields in the CAN base frame. The meaning of the frame format is the following:

- SOF (Start Of Frame), it indicates the start of frame and it is useful to synchronize the nodes.
  1 bit length
- 11-bit Identifier, it represents the message priority.
  11 bits length
- RTR (Remote Transmission Request), it indicates the frame type. It is 0 for data frame and it is 1 for remote frame.
  1 bit length
- IDE (Identifier Extension), it is 0 when it is a base frame with 11 bit identifier.
  1 bit length
- r0 (Reserved Bit), it is a space for future standard.
  1 bit length
- DLC (Data Length Code), it represents the number of data bytes transmitted.
  4 bits length
- Data Field, It contains the data
  from 0 to 8 bytes

- CRC (Cyclic Redundancy Check), it is an error-detecting code.
  16 bits length

- ACK, every node that receives a frame overwrites the 1 ACK field with 0, indicating an error-free message has been sent. ACK is 2 bits, one is the acknowledgment bit and the second is a delimiter.
  2 bits length

- EOF (End of Frame), it indicates the end of a frame.
  7 bits length

- IFS (Inter-Frame Spacing), it is the time that the controller needs to move a frame into position in the buffer area.
  3 bits length

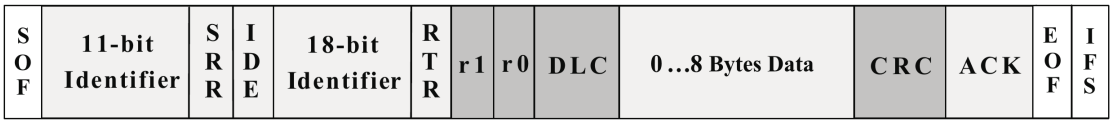


Figure 4.4: Extended frame format

The image above show instead the extended frame format; the differences are:

- SRR (Substitute Remote Request), it is always 1
  1 bit length

- IDE (Identifier Extension), it is 1 when the frame has an extended format
  1 bit length

- 18-bit Identifier, it represents the message priority
  18 bits length

- r1, it is a reserved bit typically set to 0
  1 bit length

In CAN protocol there is also a distinction between the message types that can be transmitted on a CAN bus; there are four different message types:

- Data Frame

- Remote Frame

- Error Frame

The data frame is used to send and receive user data, because it contains the data field.

The remote frame is a frame used to request a data frame from any CAN node. It is similar to a data frame but it has two differences: it has no data field and it has the RTR bit set to 1.

The error frame indicates the presence of an error in the communication. It is sent by a node that detects an error, and the other nodes that receive the error frame send another one. This process allows the data frame sender to re-transmit the message without errors.

4.4 Bus

In a CAN network physical signal transmission is based on the transmission of differential voltages, therefore the CAN bus has two lines, the CAN high line and the CAN low line. When the CAN bus is in idle mode, the two lines runs at 2.5V, when there is data transmission CAN high line runs at 3.75V and the CAN low line runs at 1.25V

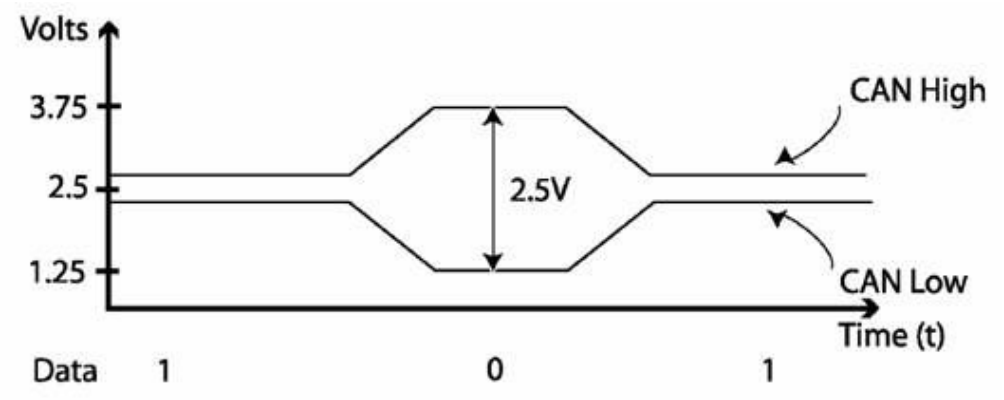


Figure 4.5: CAN voltage

The bus access has an event-driven approach: when there are two nodes which try to occupy the bus at the same time, bus access is done with bit-wise arbitration and in a non-destructive manner. Non-destructive means that after winning the arbitration, the winning node does not start sending the message again, but continues from the last bit sent. During arbitration, the node is never corrupted. [13]. The arbitration is won by the CAN frame with the lowest ID, because the lowest message identifier number has the highest priority. The CAN arbitration process is handled automatically by a CAN controller, because each node that sends data monitors its own transmission. When a node sends its identifier bits, it checks if the bus state contains the value it wrote. If the value has changed,

it means that there is another node with a lower identifier that wants to use the channel, so the node with higher identifier stops the transmission. Figure 4.6 shows an example of arbitration on a CAN bus: Node B can send its packet only at the end because when it wants to write on the bus, there is always a frame with higher priority on the bus. In the CAN bus there are idle moments between frames used to synchronize nodes.

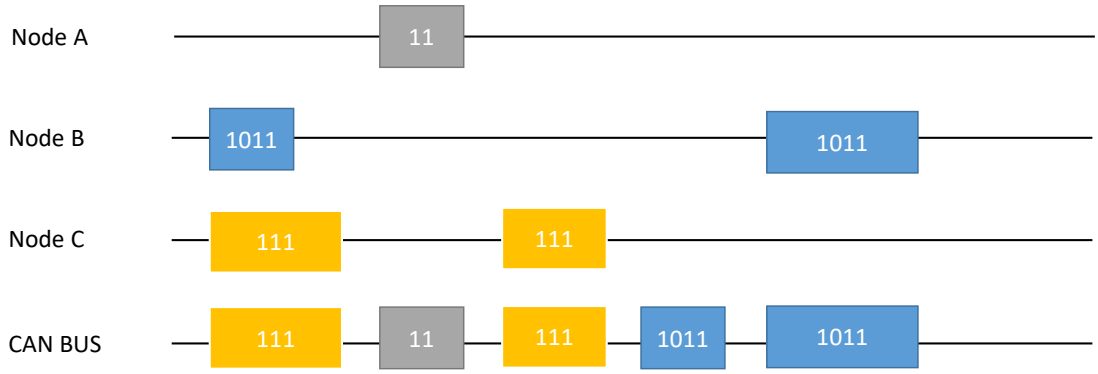


Figure 4.6: CAN arbitration

4.5 CAN DBC

A CAN DBC file (CAN bus database) is a text file that contains information such as message ID, message name, DLC, signal, sender and other information for decoding RAW CAN bus data to human readable values. It is very useful for tracking all the messages that populate a CAN network in the network design phase, and it is used with raw CAN frames to understand their content exchanged between can nodes. This section explains with examples taken from Csselectronics [14] the rules and the utility of the these files.

DBC file has a well-defined structure where each value in a position has its own meaning; the structure is shown in the picture below 4.7

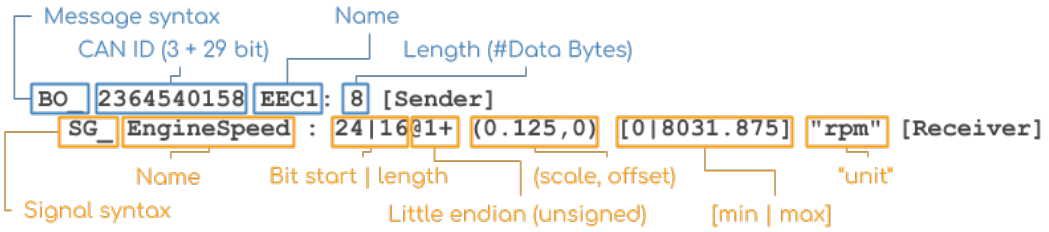


Figure 4.7: CAN DBC file with the explanation of the fields, from Csselectronics[14]

The explanation of the message syntax is as follows:

- A message starts with BO_ and the ID must be unique and in decimal
- The DBC ID adds 3 extra bits for 29 bit CAN IDs to serve as an 'extended ID' flag
- The name must be unique, 1-32 characters and may contain [A-z], digits and underscores
- The length (DLC) must be an integer between 0 and 1785
- The sender is the name of the transmitting node, or Vector___XXX if no name is available

The explanation of the signal syntax is as follows:

- Each message contains one or more signals that start with SG_
- The name must be unique, 1-32 characters and may contain [A-z], digits and underscores
- The start bit indicates the byte at which the data payload begins.
- The bit length is the signal length
- The @1 specifies that the byte order is little-endian/Intel. @0 for big-endian/Motorola
- + indicates that the value type is unsigned. - for signed signals
- The (scale,offset) values are used in the physical value linear equation
- The [min|max] and unit are optional meta information

- The receiver is the name of the receiving node (Vector__XXX is used as default)

Starting from a physical CAN frame and a CAN DBC file, we can read and understand the message inside.

Listing 4.1: Physical CAN frame

```
CAN ID     Data bytes
0CF00400   FF FF FF 68 13 FF FF FF
```

Listing 4.2: Small part of a CAN DBC file

```
BO_ 2364540158 EEC1: 8 Vector_XXX
 SG_ EngineSpeed : 24|16@1+ (0.125,0) [0|8031.875] "rpm" Vector_XXX
```

With some calculation, it is possible to obtain the decimal value of the engine speed, that is 621 rpm. For a complete explanation of the steps, follow Csselectronics's explanation [14]

Chapter 5

Adaptive Cruise Control Implementation

In this chapter I tested the efficacy and validity of a solution for the design of a network architecture of a car, by designing the adaptive cruise control (ACC) and related functionalities, such as car acceleration or deceleration, radar information, etc. The ACC architecture scheme presented in the following sections will be a simplified version, because the objective is not to improve the functionality, but to outline it in PREEvision, thereby taking advantage of the features of this software, for example the auto generation of **arxml** files. Some concepts of the functioning of a car are deliberately neglected to keep the project as simple as possible.

In the next sections there is a brief explanation of what is and how the adaptive cruise control works, and the explanation of my thesis.

5.1 What is the ACC

The adaptive cruise control is an intelligent system, capable of setting the car's speed, increasing or decreasing it according to traffic conditions. The main objective is to keep a safe distance from the vehicles in front without pilot intervention. This functionality detects the presence of vehicles or obstacles in the space in front of the car, through cameras, radar and frontal sensors. In this way, once the driver sets the speed and the distance he wants to keep from the car in front, the ACC will automatically perform all braking and acceleration based on traffic conditions.If the vehicle in front slows down, the ACC will reduce the speed to keep the safety distance previously set. When the driver wants to overtake a vehicle, he must insert direction indicator and the system will immediately understand the intention to overtake or change lanes. Typically, in the steering wheel there is a controller for the ACC, with which the driver can turn on the functionality and he can set the

speed and the distance to keep.

This feature allows to maintain a steady pace in a completely safe manner because in case of obstacles, the reaction time of the vehicle is less than the human one. Another advantage is the comfort of the trip because the driver must not intervene on the pedals.

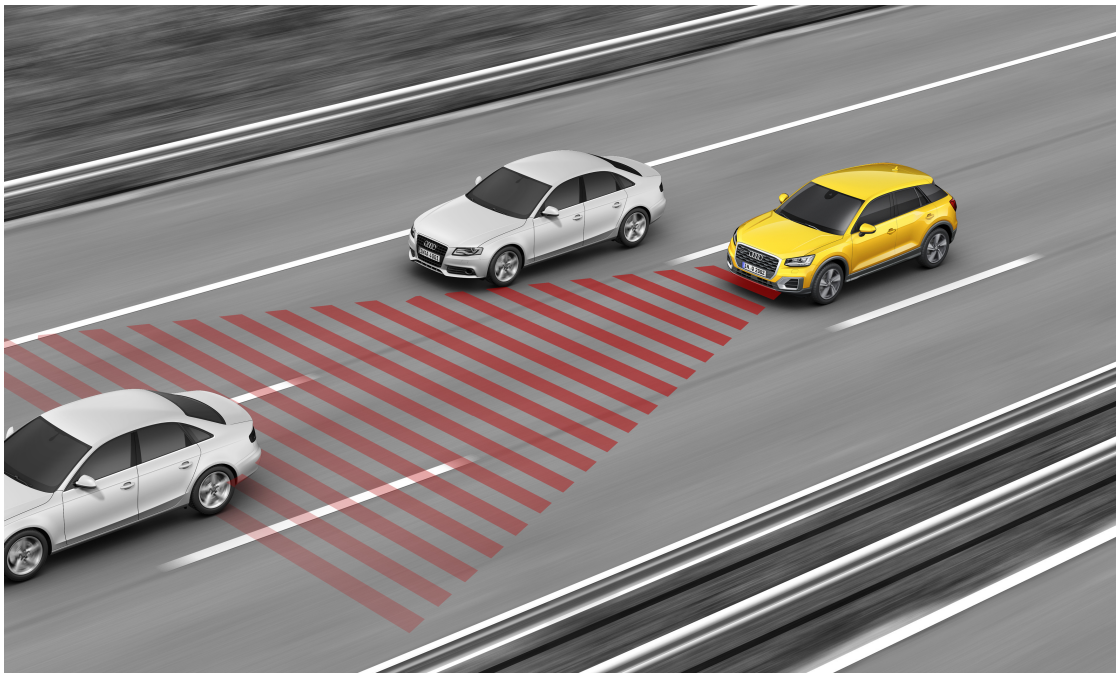


Figure 5.1: Adaptive Cruise Control

5.2 Customer Features and Requirements

In this section, there is the description of a simplified adaptive cruise control in technical and non-technical manner. Both customer features and requirements are schematized in table format (the table is simplified for graphic reasons), respecting a hierarchy that allows to create levels. Customer features and requirements are grouped according to the type of functionality, so the top level is the title and below is its description.

The functionalities I tried to implement are the following. The car has two driving modes:

Manual mode (classic) The driver plays the classic role of the pilot, steering, accelerating and decelerating based on its needs and the environment around it. On the one hand, in order to accelerate or decelerate the car, the driver must

press with his foot the pedal appropriate to the acceleration or deceleration of the car. On the other hand, in order to steer the vehicle, the driver must use the steering wheel that allows to steer the front wheels.

Advanced mode through ACC The driver has a controller in the steering wheel to take advantage of the autonomous driving functionality. In short, this feature allows the car to maintain the speed and distance set by the driver from the car in front, without driver intervention. If the surrounding environment does not allow to keep the set speed and distance, the car will automatically decrease the speed and brake the car. The pilot will not have to intervene on the pedal. In order to use this functionality, the driver must first turn on the feature using the appropriate button on the steering wheel. Once turned on, the commands present in the image below 5.2 will allow you to set the speed and distance to keep. There is also a button on the controller that temporarily disables the feature, thereby returning to manual driving mode, and a button that resumes operation of the feature.

Figure 5.2: ACC controller in steering wheel

With this brief introduction, the customer features can be schematized in this way:

Customer Features			
Level	Text	Type	Links
1-1	**1.1 Movement**	Heading	0
2-2	The car can decelerate in different ways: • When you press the decelerator • When the adaptive cruise control wants decelerate	Customer Feature	
3-2	The car can accelerate in different ways: • When you press the accelerator • When the adaptive cruise control wants accelerate	Customer Feature	
4-2	The speed indicator allows you to understand the current speed of the vehicle	Customer Feature	0

Customer Features			
Level	Text	Type	Links
5-1	**1.2 DriverAssistance**	Heading	
6-2	Adaptive Cruise Control (ACC) 1. The driver sets the speed and the distance (optional) to keep from the next car and the car will follow these parameters 2. (ACC off) To turn on the ACC you must hold down for 2 seconds the "cruise control" button 3. (ACC on) To turn off the ACC you must hold down for 2 seconds the "cruise control" button 4. To temporarily enable or disable the acc you must click the "Cruise control" button 5. Command scheme: • "volume up" speed increased: – Hold the button down; the speed will be increased by 10 km/h every 0.5 ms – One button click the speed will be increased by 1 km/h • "volume down" speed decreased – Hold the button down; the speed will be decreased by 10 km/h every 0.5 ms – One button click; the speed will be decreased by 1 km/h • "+ button" distance increased – One button click; the distance will be increased by 1 km/h	Customer Feature	8

Customer Features			
Level	Text	Type	Links
5-3	• "- button" distance decreased – One button click; the distance will be decreased by 1 km/h • "CNL button" – With this button you can deactive the adaptive cruise control functionality. The ACC is enabled but is not actived • "RES button" – This button allows to resume the adaptive cruise control functionality	Customer Feature	8
7-1	**1.3 Information**	Heading	
8-2	The cockpit is a display that shows information to the driver about the ACC and current speed	Customer Feature	0

Table 5.2: Customer Features table

The movement of the vehicle is a process that involves several sensors, actuators and ECUs that communicate with each other through the exchange of signals. Deceleration or acceleration of the vehicle can occur in two different ways:

The driver acts on the pedal The pedal pressure information is transformed into a value from 0 to 100, which will be sent to the unit that will act on the brake calipers or engine to deliver more power.

ACC wants to slow down or speed up the car The speed to be held is sent to the unit that will act on the brake calipers or the engine to deliver more power.

The operation and related criteria of the ACC are as follows:
before the acc is put into operation, it must pass a series of tests to verify the correct functioning of the sensors. Specifically, the radar, braking system and power delivery system are checked for their correct operation. If there are any problems,

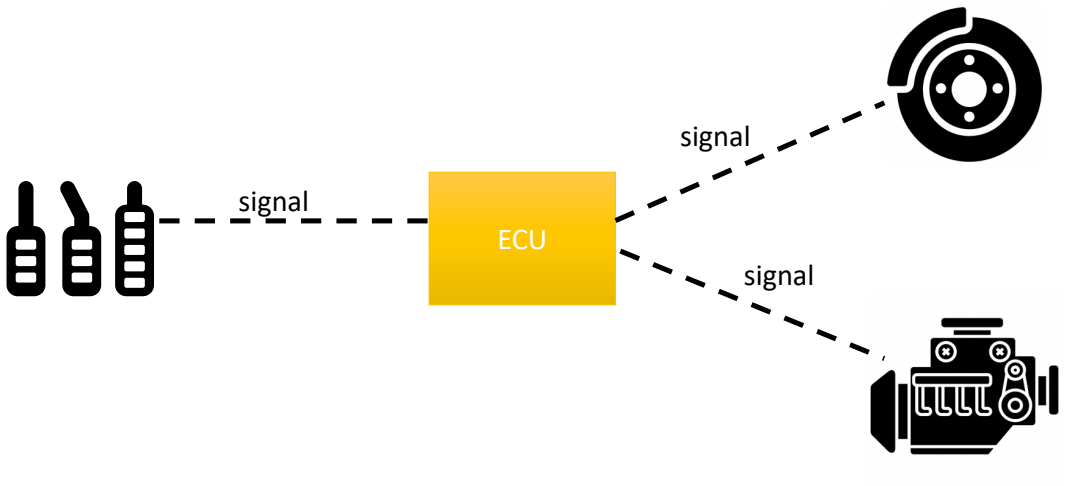


Figure 5.3: Sensors, ecu and actuators communication diagram

an error signal should be sent and the use of the autonomous driving functionality should be prohibited. ACC can be enabled only when the car speed is higher than 30 km/h. Once the feature is activated, the radar will produce information to build the environment surrounding the car. The driver has to set the speed and distance to be kept, which will then be sent to the unit that is in charge of moving the car. If the radar notices obstacles in front of the car, the ACC will calculate the speed to keep in order to maintain the distance set by the driver, and send the signal containing the speed the car should keep to the unit that will take care of slowing down or accelerating the car. If the driver does not set the distance, it will be used a value calculated by the safety distance formula using the current speed. (Formula for calculating the safety distance: The safety distance is the distance any vehicle must maintain from the one in front of it in order to stop, when necessary, without colliding with it. When evaluating the safety distance, it is important to take certain factors into account: the driver's reflexes; the type and state of efficiency of the vehicle; speed; visibility and atmospheric conditions; traffic conditions; the slope of the road and the characteristics and conditions of the road surface. A simple formula to remember in order to roughly calculate a good safety distance is as follows: divide your speed expressed in km/h by 10 and square the result; the resulting number is a good indicator, in meters, of the safety distance to be maintained. Example: at 50 km/h you should maintain a distance of 25 meters). If the radar does not notice any obstacles, the current speed is less than the speed set by the driver, and the current distance is greater than the distance set by the driver, the acc will send a signal containing the speed to be reached to the power unit.

Requirements			
Level	Text	Type	Links
2-2	**DriveTrain**	Req package	
3-3	**Brake**	Req package	
4-4	When the driver presses the brake pedal, the information is sent to the brake unit. The possible values are mapped on a scale from 0 to 100	Req (shall)	1
3-3	When the ACC needs to slow down, it sends that information to an ECU that is involved to give the information to the engine unit.	Req (shall)	1
6-3	**Accelerator**	Req package	
7-4	When the driver presses the accelerator pedal, the sensor gets information and gives those information to an ECU that is involved to give the information to the engine unit.	Req (shall)	1
8-4	When the Adaptive Cruise Controls needs more speed, it sends that information to an ECU that is involved to give the information to the engine unit.	Req (shall)	1
9-2	**DriverAssistance**	Req package	
10-3	**AdaptiveCruiseControl**	Req package	
11-4	The ACC works when the car speed is greater than 30 km/h	Definition (must)	1
12-4	Radar must pass the control test	Definition (must)	1
13-4	The driver sets the speed to be held during the trip	Req (shall)	1
14-4	Radar obtains information about the car's surroundings to see if there are cars or obstacles	Req (shall)	1
15-4	When Radar sees nearby obstacles the car must reduce the speed, braking automatically so the ECU must send an information to the Brake unit.	Req (shall)	1
16-4	If the distance is not set, the car uses a default value calculated with the car speed (safety distance)	Req (shall)	1

Requirements			
Level	Text	Type	Links
17-4	When radar doesn't see obstacles, and the current speed is lower than the set speed and the distance is greater than the set distance, the car must increase the speed; The ECU must send an information to the PowerTrain unit	Req (shall)	1
18-4	Formula for calculating the safety distance: The safety distance is the distance any vehicle must maintain from the one in front of it in order to stop, when necessary, without colliding with it. When evaluating the safety distance, it is important to take into account certain factors: the driver's reflexes; the type and state of efficiency of the vehicle; speed; visibility and atmospheric conditions; traffic conditions; the slope of the road and the characteristics and conditions of the road surface. A simple formula to remember to roughly calculate a good safety distance is as follows: divide your speed expressed in km/h by 10 and square the result; the resulting number is a good indicator, in meters, of the safety distance to be maintained. Example: at 50 km/h you should maintain a distance of 25 meters.	Info (can)	

Table 5.3: Requirements table

5.3 Logical Architecture

This section explains the logic diagram relating to the customer features and requirements written in the previous section. My approach has been to divide the individual functionalities into logic diagrams contained in building blocks, which are reusable containers to contain logic blocks. This allows you to break down a complex functionality, for instance the ACC in small logic blocks, that connected among them, compose the functionality in examination. The logical architecture has a Type/Instance approach. All created and instantiated blocks are based on types. Blocks of the same type will have identical behavior and the same interfaces; This makes it possible to reuse the same logic block for functionalities that differ only in the physical location of the vehicle. For example, car windows, brake calipers etc. The blocks are connected to each other through ports.

Ports must have assigned interfaces that contain one or more data elements describing the information exchanged by the logic blocks. A data element contains an application data type, which is a numeric value mapped in a computation method that indicates the range of possible values. Most of the choices made are aimed at simplifying the solution, because the goal is to focus on the methodology to create a vehicular architecture, rather than the correct functioning of the feature.

The speed indicator diagram (5.4) is a logical representation of how the speed indicator works. It is composed of two sensors and a function. The sensors *WheelRotationSpeedRearLT* and *WheelRotationSpeedRearRT* are placed in the rear wheels and are used to calculate the rotation of the wheel. The value is sent to the function *CalculateSpeed* that will calculate the value in km/h. The sensors

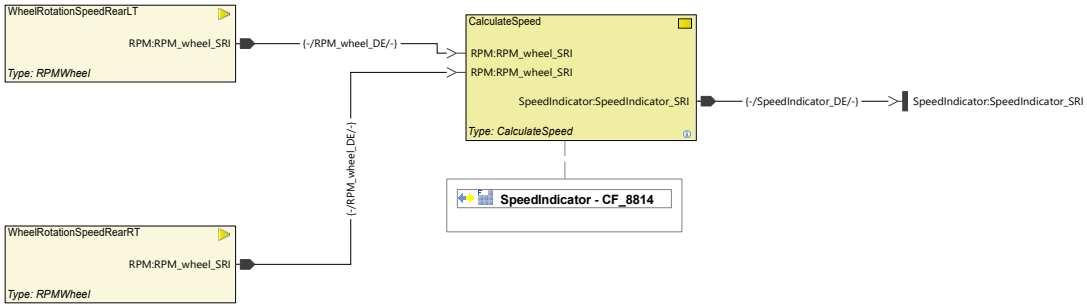


Figure 5.4: Speed indicator logic diagram

communicate with the function through the RPM_wheel interface sending the RPM_wheel_DE data element. Possible values range from 0 to 5000 are contained in the computation method RPM_wheel_CM. *CalculateSpeed* function has another port with another associated interface, to send the speed value expressed in km/h to the other logic blocks. The interface is SpeedIndicator_SRI, the data element is SpeedIndicator_DE and the computation method is SpeedIndicator_CM. Possible values range from 0 to 300. All this data will be useful in the end to understand the content of the arxml and dbc files that will be generated.

In the image above (5.4) it's also possible to see the mapping between the customer feature and the logical block; the mapping is not mandatory, but it is useful to indicate the type of functionality you are describing.

A summary of the interfaces, data elements and computation methods in the logic diagram is shown in the following table (5.4):

Speed Indicator		
Interface	**Data Element**	**Computation Method**
RPM_wheel_SRI	RPM_wheel_DE	RPM_wheel_CM
SpeedIndicator_SRI	SpeedIndicator_DE	SpeedIndicator_CM

Table 5.4: Speed indicator interfaces

The accelerator diagram (5.5) is composed of a sensor (AcceleratorByPedal), a function (DriveTrainFunction) and two signals (SpeedIndicator_DE and ACCOutputIncreaseSpeed_DE) coming from the building blocks of the *AdaptiveCruiseControl* and the *SpeedIndicator*. Since the car can accelerate in two ways:

AcceleratorByPedal The driver presses the accelerator pedal, the pedal displacement is mapped to a value between 0 and 100 and the value (AcceleratorByPedal_DE) is sent to the *DriveTrainFunction*.

AcceleratorByACC The *AdaptiveCruiseControl* sends a signal contained the speed in km/h (ACCOutputIncreaseSpeed_DE) to the *DriveTrainFunction*, while the *SpeedIndicator* sends the current speed.

The *DriveTrainFunction* receives this information, checks if there is a need to increase the speed of the car and, if necessary, it sends a signal (AcceleratorTraction_DE) to the traction building block; possible values range for this data element from 0 to 100.

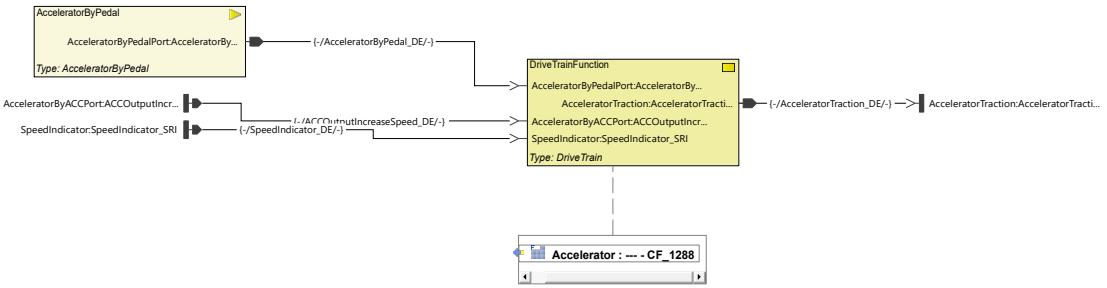


Figure 5.5: Accelerator logic diagram

Accelerator		
Interface	**Data Element**	**Computation Method**
AcceleratorByPedal_SRI	AcceleratorByPedal_DE	AcceleratorByPedal_CM
SpeedIndicator_SRI	SpeedIndicator_DE	SpeedIndicator_CM
ACCOutputIncrease Speed_SRI	ACCOutputIncrease Speed_DE	ACCOutputIncrease Speed_CM
AcceleratorTraction_SRI	AcceleratorTraction_DE	AcceleratorTraction_CM

Table 5.5: Accelerator interfaces

The traction diagram (5.6) is a simplified version containing only two actuators, which simulate the delivery of power to turn the wheels in a front wheel drive car. The *DriveTrainFunction* sends the signal concerning the power to be delivered to the wheels to make them turn, through the AcceleratorTraction_SRI interface. Possible values range for AcceleratorTraction_DE data element from 0 to 100.

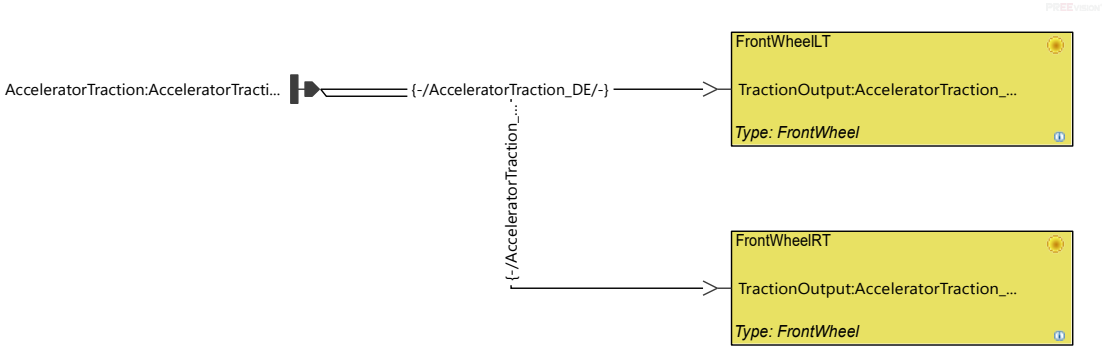


Figure 5.6: Traction logic diagram

Traction		
Interface	**Data Element**	**Computation Method**
AcceleratorTraction_SRI	AcceleratorTraction_DE	AcceleratorTraction_CM

Table 5.6: Traction interfaces

The cockpit is the driver display where useful information is shown, such as the current speed and the ACC status. Its diagram (5.7) is composed of one function (CockpitFunction), one actuator(CockpitDisplay) and two signals (SpeedIndicator_DE and ACCStatus_DE) coming from the building blocks of the *SpeedIndicator* and the *AdaptiveCruiseControl*. The *CockpitFunction* receives the current speed value (SpeedIndicator_DE) and the adaptive cruise control status (ACCStatus_DE), and finally processes them to send them (CockpitInfo_DE) to the display.

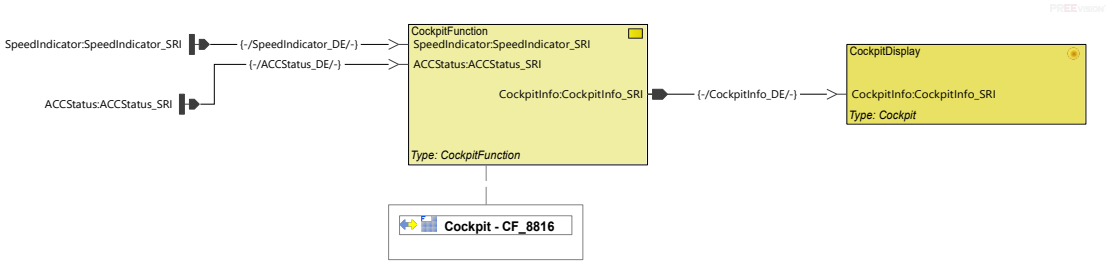


Figure 5.7: Cockpit logic diagram

Cockpit		
Interface	**Data Element**	**Computation Method**
SpeedIndicator_SRI	SpeedIndicator_DE	SpeedIndicator_CM
ACCStatus_SRI	ACCStatus_DE	ACCStatus_CM
CockpitInfo_SRI	CockpitInfo_DE	CockpitInfo_CM

Table 5.7: Cockpit interfaces

The brake diagram (5.8) is composed of one sensor (BrakeByPedal), four actuators (BrakeCaliperFronRT, BrakeCaliperFronLT, BrakeCaliperRearRT, BrakeCaliperRearRT), one function (BrakeFunction) and two signals (ACCOutputDecreaseSpeed_DE and SpeedIndicator_DE) coming from the building block of the *AdaptiveCruiseControl* and the *SpeedIndicator*. Since the car can decelerate in two ways:

BrakeByPedal The driver presses the brake pedal, the pedal displacemenet is mapped to a value between 0 and 100 and the value (BrakeByPedal_DE) is sent to the *BrakeFunction*.

BrakeByACC The *AdaptiveCruiseControl* sends a signal contained the speed in km/h (ACCOutputDecreaseSpeed) to the *BrakeFunction*, while the *SpeedIndicator* sends the current speed.

The *BrakeFunction* receives this information, checks if there is a need to decrease the speed of the car, and in case it sends a signal (BrakeCalipers_DE); possible value range for this data element from 0 to 100.

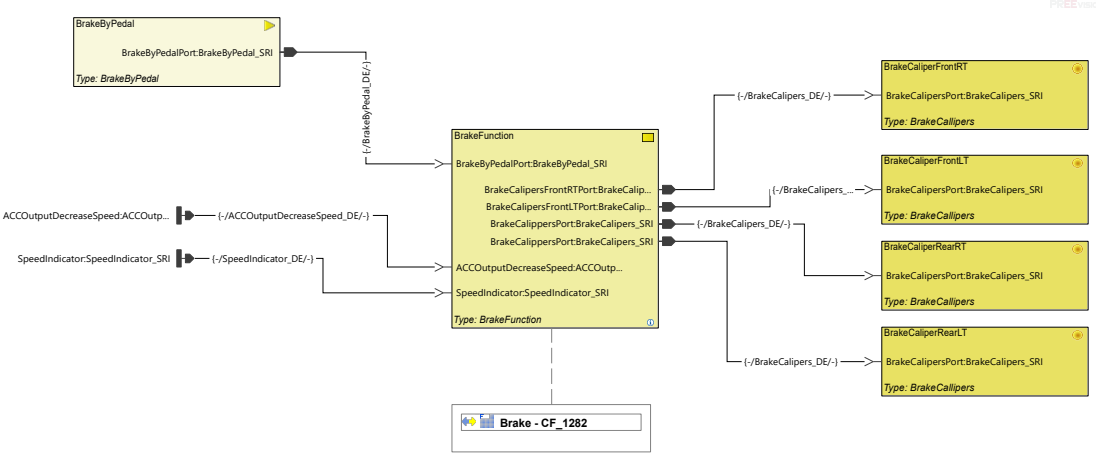


Figure 5.8: Brake logic diagram

Brake		
Interface	**Data Element**	**Computation Method**
BrakeByPedal_SRI	BrakeByPedal_DE	BrakeByPedal_CM
ACCOutputDecrease Speed_SRI	ACCOutputDecrease Speed_DE	ACCOutputDecrease Speed_CM
SpeedIndicator_SRI	SpeedIndicator_DE	SpeedIndicator_CM
BrakeCalipers_SRI	BrakeCalipers_DE	BrakeCalipers_CM

Table 5.8: Brake interfaces

The radar is a sensor that makes up the adaptive cruise control function. Without this sensor it would be impossible to adapt the speed and detect obstacles. Its logic diagram (5.9) is composed of a sensor (Radar) and a function (RadarFunction). The first one takes information from the external environment and sends it to the function through the RadarInfo_DE. To simplify the solution both data elements can assume values between 0 and 300, indicating the distance of the car in front. In a real solution, the sensor would send electromagnetic waves to detect and determine the position and possibly the speed of objects. The data would be formatted before being sent to the adaptive cruise control building block by the RadarFunction.

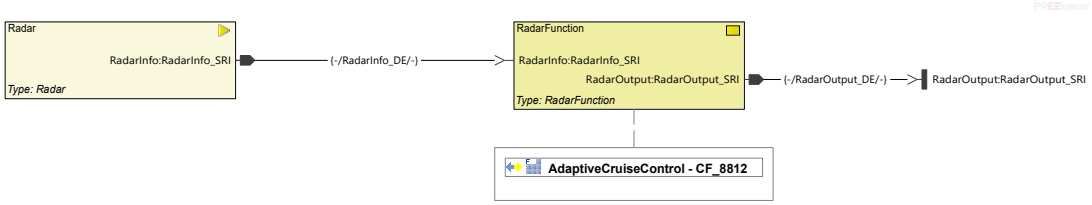


Figure 5.9: Radar logic diagram

Brake		
Interface	**Data Element**	**Computation Method**
RadarInfo_SRI	RadarInfo_DE	RadarInfo_CM
RadarOutput_SRI	RadarOutput_DE	RadarOutput_CM

Table 5.9: Radar interfaces

Adaptive cruise control is the main diagram (5.10) of this thesis. The goal is to take information from the driver and other sensors in order to decide whether to increase, decrease or maintain a constant speed. It is composed of six sensors, one function and two signals coming from the building block of the *SpeedIndicator* and the *Radar*. All sensors are buttons on the steering wheel that make up the acc controller; the controller is the one in the picture 5.2. Buttons are as follows:

ACCIncreaseSpeed It sends the ACCIncreaseSpeed_DE data element containing the value of the speed in km/h to be increased. The data can take the value of 1 (1 km/h) or 2 (10 km/h).

ACCStartButton It sends the ACCStartState_DE data element used to turn on or turn off the ACC. The values that it can assume are 0 (off), 1 (on).

ACC_CNL It sends the ACC_CNL_DE data element used to disable the autonomous driving functionality. When clicked it sends the value 1 which

indicates the intention to deactivate the cruise control.

ACCIncreaseDistance It sends the ACCIncreaseDistance_DE containing the value of the distance to be increased. The value can be 1 indicating 10 meters.

ACCDecreaseDistance It sends the ACCDecreaseDistance_DE containing the value of the distance to be decreased. The value can be 1 indicating -10 meters.

ACCDecreaseSpeed It sends the ACCDecreaseSpeed_DE data element containing the value of the speed in km/h to be decreased. The data can take the value of 1 (-1 km/h) or 2 (-10 km/h).

The *AdaptiveCruiseControlFunction* takes all the information generated by the radar and the driver, makes a decision and provides it to the other logic components through three signals: it sends the ACC status to the cockpit (ACCStatus_DE) consisting of speed to be held, distance to the car in front and whether the ACC is on or off. It also sends the value in km/h of the speed to be reached (ACCOutputIncreaseSpeed_DE or ACCOutputDecreaseSpeed_DE). The values can range from 30 to 250 and represent the speed to be held. If the car has to slow down this signal will be sent to the braking system, otherwise to the acceleration system.

AdaptiveCruiseControl		
Interface	**Data Element**	**Computation Method**
ACCIncreaseSpeed_SRI	ACCIncreaseSpeed_DE	ACCIncreaseSpeed_CM
ACCDecreaseSpeed_SRI	ACCDecreaseSpeed_DE	ACCDecreaseSpeed_CM
ACCIncreaseDistance _SRI	ACCIncreaseDistance _DE	ACCIncreaseDistance _CM
ACCDecreaseDistance _SRI	ACCDecreaseDistance _DE	ACCDecreaseDistance _CM
ACCStartButton_SRI	ACCStartButton_DE	ACCStartButton_CM
ACC_CNL_SRI	ACC_CNL_DE	ACC_CNL_CM
SpeedIndicator_SRI	SpeedIndicator_DE	SpeedIndicator_CM
RadarOutput_SRI	RadarOutput_DE	RadarOutput_CM
ACCOutputIncrease Speed_SRI	ACCOutputIncrease Speed_DE	ACCOutputIncrease Speed_CM
ACCOutputDecrease Speed_SRI	ACCOutputDecrease Speed_DE	ACCOutputDecrease Speed_CM
ACCStatus_SRI	ACCStatus_DE	ACCStatus_CM

Table 5.10: AdaptiveCruiseControl interfaces

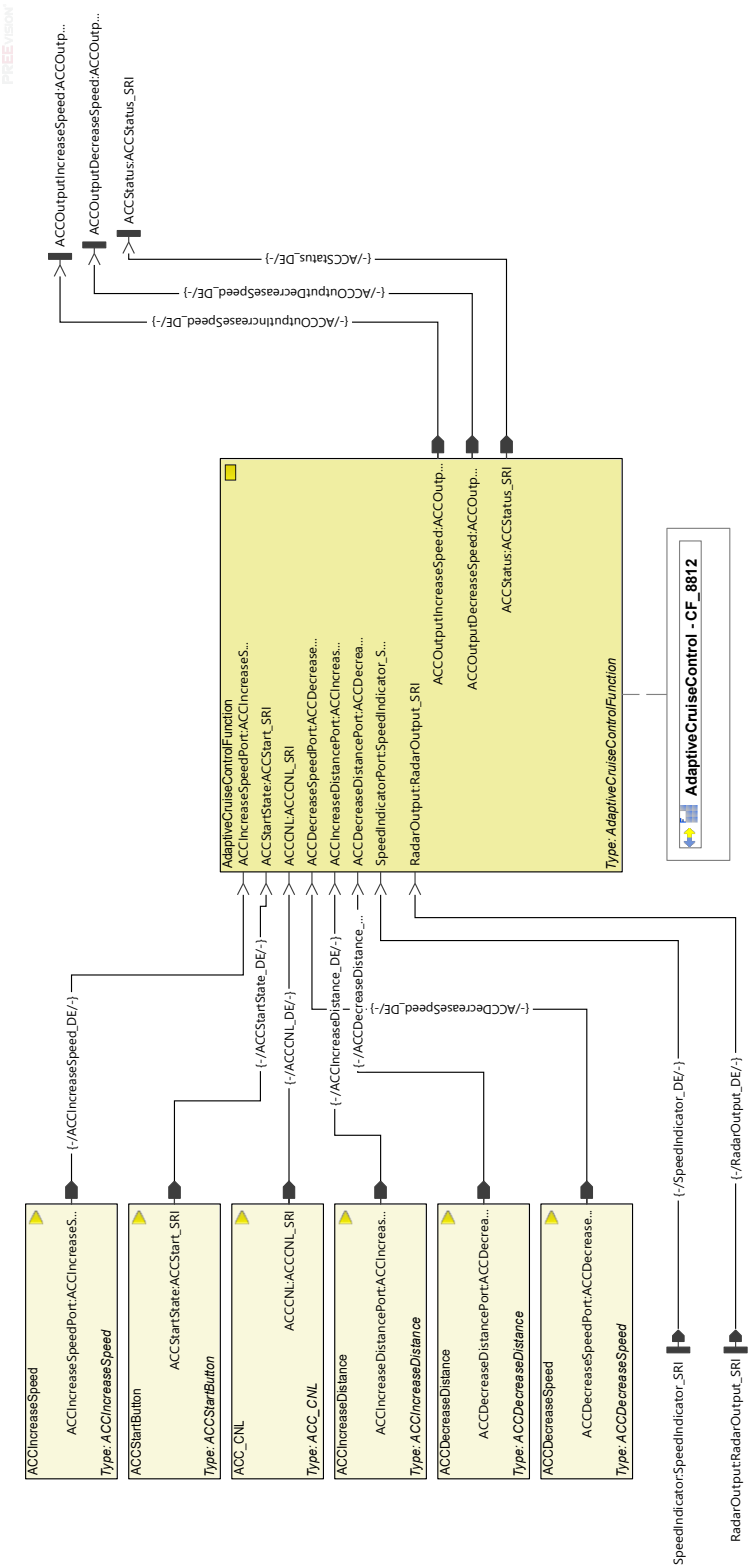


Figure 5.10: AdaptiveCruiseControl logic diagram

Each diagram created in PREEvision is schematized in a tree structure called model view. This model view shows the hierarchy of the various diagrams.For example, when a diagram is created that contains sensors, actuators and functions, these will be children of a package or a building block. In my case, having used the building blocks, the components of the diagram are children of a building block that is also child of a logical package. In this way it is possible to create two types of diagrams:

- Logical architecture diagram
- Logical architecture system diagram

The first one allows the creation of the diagrams explained before; it is used to describe a logical block that can represent a part of a functionality, always respecting the hierarchy of the packages. To understand how the hierarchy works, look at the image below (5.11). Inside the AdaptiveCruiseControl building block there are many sensors that cannot be used by other building blocks belonging to other packages. The second type of diagram allows instead to create models that do not respect the hierarchy. This makes it possible to connect the individual logical components in order to create a complex functionality such as, the complete functioning of the adaptive cruise control.

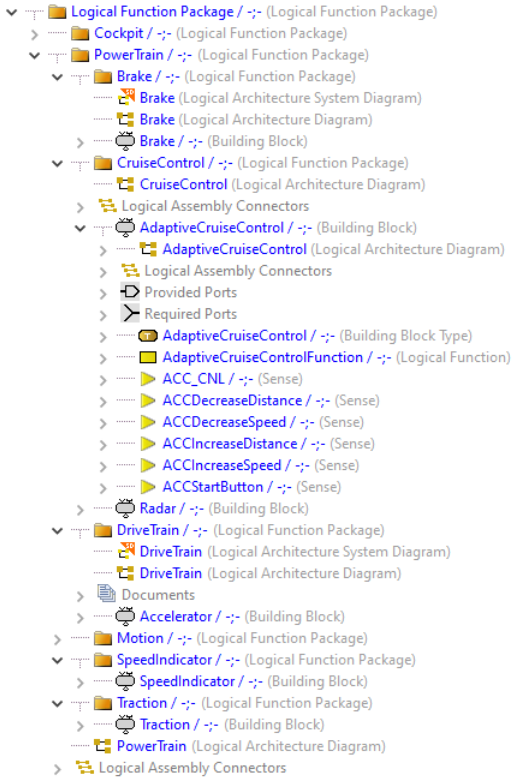


Figure 5.11: Model View in PREEvision

The image below(5.12) shows the logical architecture system diagram that I created to compose the adaptive cruise control functionality. Only the building blocks and the various connections between them are shown. This allows both to inspect them internally and to modify their internal workings without impacting other building blocks.

Communication between building blocks is done via the data elements and interfaces explained above.

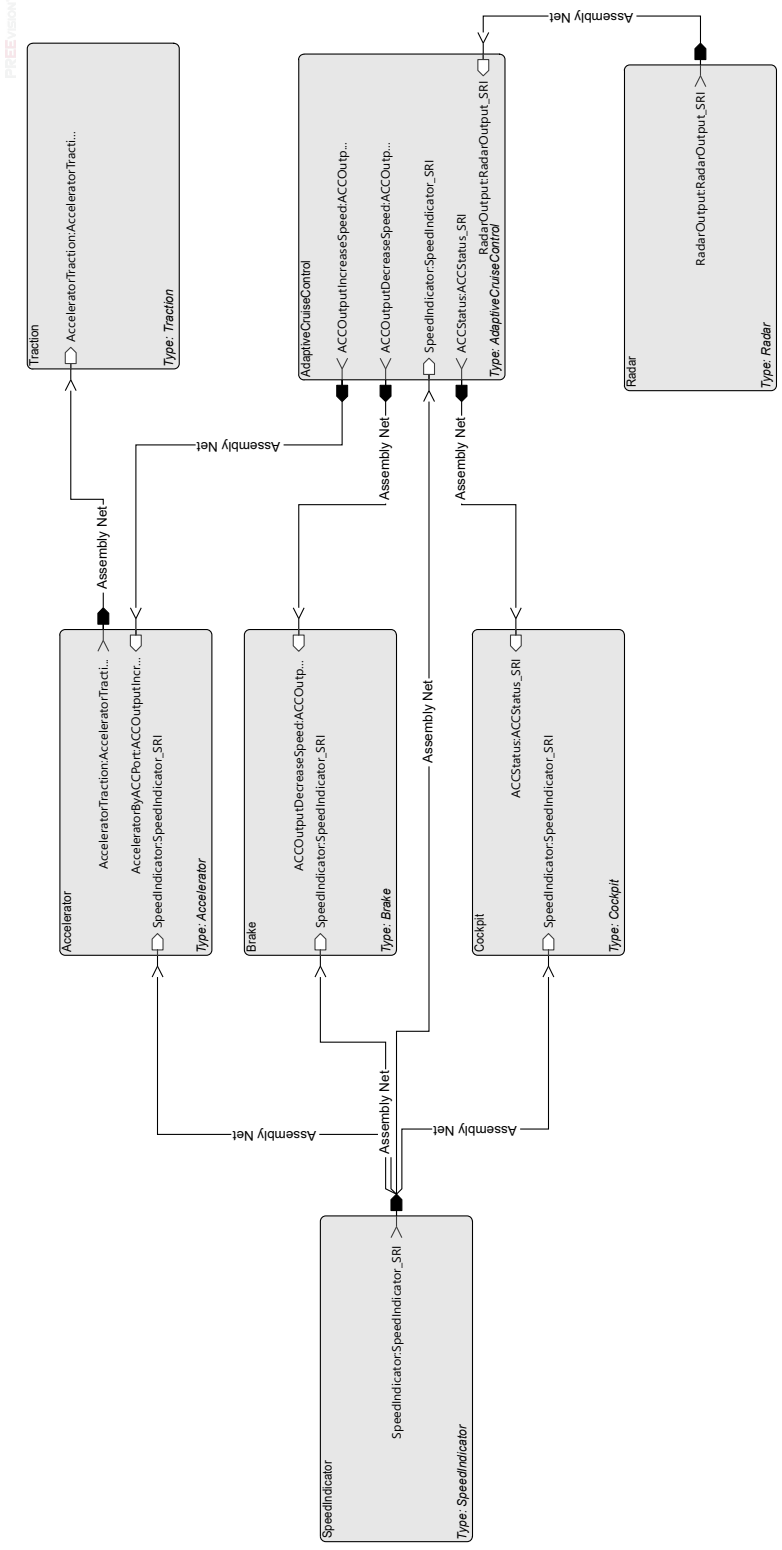


Figure 5.12: Logical architecture system diagram

5.4 System Software Architecture

This section explains the software architecture diagram. This layer is essential for generating ARXML files for future ECU development. In PREEvision, the logic layer and the software layer are very similar, and depending on the user's needs, it is possible to avoid creating either one. In this case, since the primary goal is not to program the ECUs using ARXML files, but to build a network architecture, this layer has not been studied that much. In any case, in case you want to program the ECUs, the software level and the hardware level are mandatory, because they describe the use of the ECUs (the steps to follow are shown in the image 3.7). However, in order to check how efficient this approach is, we tried to test this layer as well. The difference between the logical layer and the software layer can be very small, meaning that it is possible to have the same diagrams. By changing only the artifacts that compose it, and reusing the components for communication, such as interfaces, ports etc., it is then up to the user, according to the final goal to decide whether to create both with a different level of abstraction, or do without one of the two. In this thesis, two types of software diagrams were created.

The first type is the copy of the logic diagram, i.e. all the logic components were recreated in the software layer, using the specific components of this layer, so the building blocks were replaced with the Composition component, while the logic functions were replaced with the Application SW Component. All the communication that was created in the logical layer, i.e. the interfaces, data elements and computation methods, was reused without the need to recreate them. To avoid showing diagrams that are identical to the logic diagrams, they are not shown and explained in this section, since they were explained in the previous section. All components of the software diagrams have the same names as the components of the logic diagrams. The choice to create identical diagrams was forced by the fact that otherwise it would not have been possible to obtain the ARXML files. Since the goal of this thesis is not the development of ECUs, but to test a methodology for the creation of a vehicular architecture, the architecture created by the diagrams is generic and basic, but useful to achieve the proposed goal.

The second type of diagram tries to go into more detail about the single components, specifically parts are added to the diagram relative to Radar and Adaptive Cruise Control. The second type of diagram tries to go into more detail about the single components, specifically parts are added to the diagram relative to Radar and Adaptive Cruise Control. In spite of the addition in these two diagrams, the inserted parts will not be considered in the generation of the ARXML files.

The Radar software diagram (fig. 5.13) contains the Radar sensor to obtain information from the external environment, and two *Application Software Components* to manipulate the data obtained from the sensor. The first software

component is the *CreateEnvironment* that has the task to create the environment around the car, to verify the presence or not of obstacles. The second software component is the *Diagnostic* useful to verify the correctness of the data provided by the sensor. Both software components receive the data from the sensor through the *RadarInfo_SRI* interface but the *Diagnostic* component receives also further useful information for the diagnostics through the *RadarInfoDiagnostic_SRI* interface. Both software components will provide the ACC with the results of their executions via the *RadarInfoForACC_SRI* and *RadarFault_SRI* interfaces. In case *Diagnostic* detects an error, the ACC must abort the autonomous driving operation.

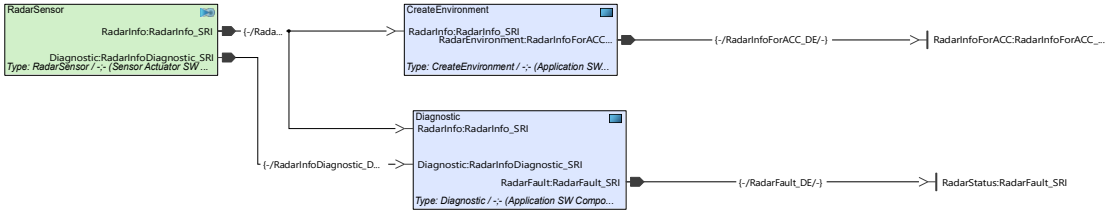


Figure 5.13: Radar software diagram

The software diagram related to the ACC (fig. 5.14) has three *Application Software Component*: *ButtonDetection* to manage the inputs of the controller placed in the steering wheel, *ACCController* to manage all the data useful for the functioning of the ACC, and a *FaultController* that has to manage the case of malfunction of some sensor, which can cause the incorrect functioning of the autonomous driving functionality. *ButtonDetection* communicates through the *ACCValues_SRI* interface with *ACCController*. This last one having all the necessary information for the operation, carries out the elaboration to calculate if to accelerate, to decelerate or to maintain the constant speed. This component software also receives through the interface *ACCAbort_SRI* the signal to abort the execution of ACC, in case there are problems in the system. This component software communicates directly with the radar component software in order to obtain information about the correct operation of the sensor. In case of problems, the default controller will send a signal through the interface to inform all interested ECUs in the car about the problem.

5.5 Hardware Architecture

The hardware layer allows you to create a hardware diagram of the car, showing all the sensors, control units and actuators. The connections between the various components are made through the bus system, which can be of all types, for example, CAN bus, LIN bus, ethernet bus etc. In this thesis, for time reasons,

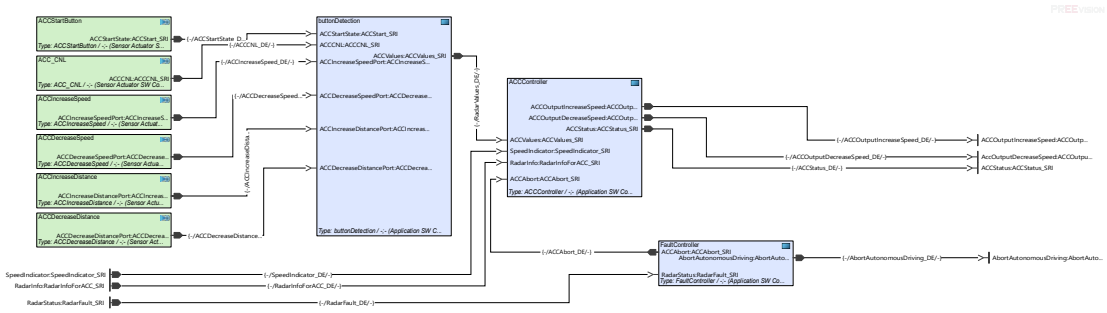


Figure 5.14: ACC software diagram

the CAN bus has always been used even when a more powerful bus should have been used, such as ethernet for radar communication. The use of the ethernet bus in addition to the CAN bus would have definitely taken more time, risking not being able to conclude the cycle of the architecture design. However, to keep in line with the goal of this thesis, the use of other buses will definitely be future work to continue this thesis. Future works will be discussed in the chapter 7.

The diagram I used was the network diagram (5.15), which allows you to create a network architecture, by reporting only the essential elements to manage an automotive network. PREEvision supplies ulterior diagrams in the hardware level, like as an example the electric circuit diagram for the description of the electric components, which make up the automotive architecture. To create the network diagram my idea was to report in the network diagram all sensors and actuators used in the logic model. The diagram I created is the as follows (5.15):

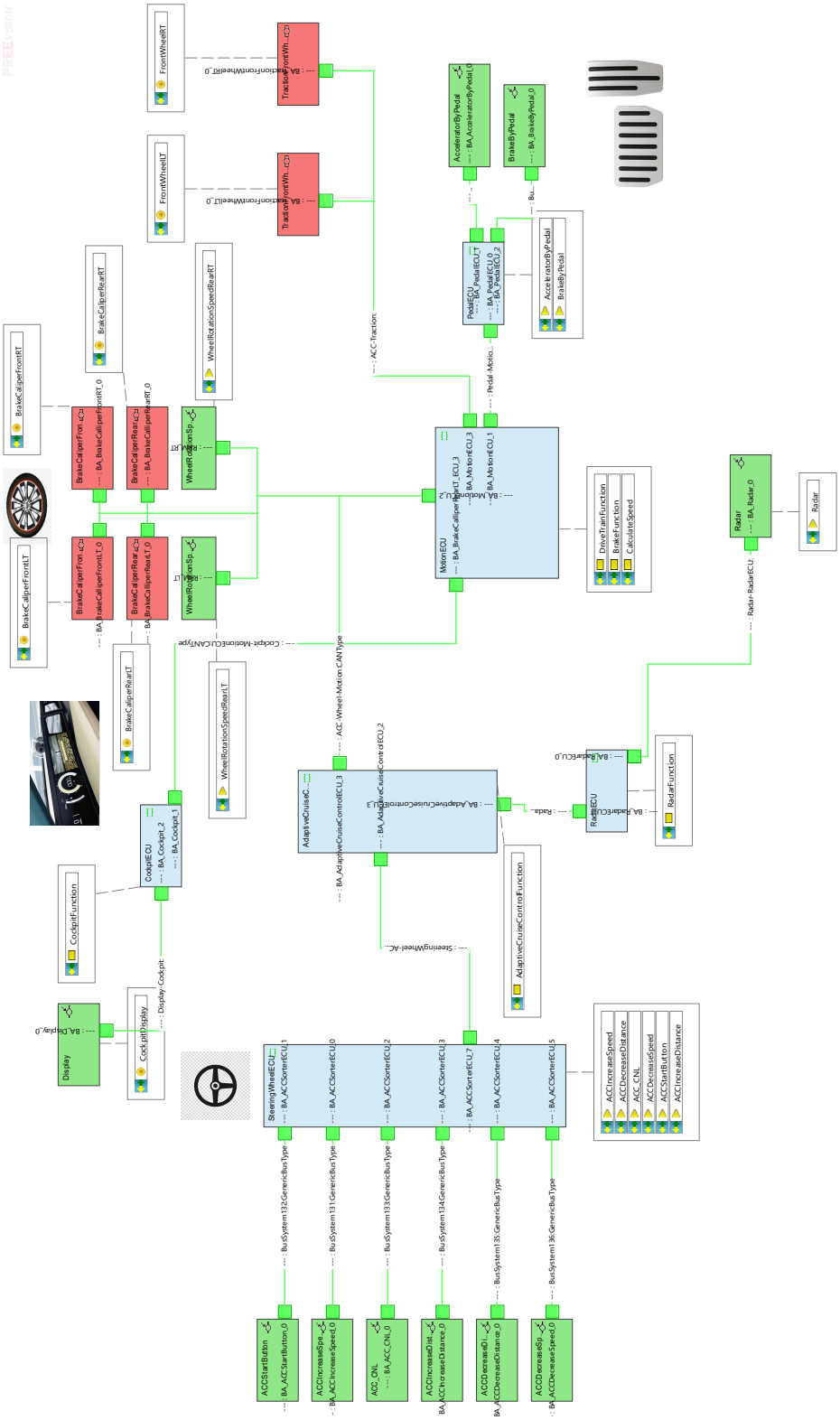


Figure 5.15: Network diagram

The sensors are:

ACC_CNL It's a button used to disable the adaptive cruise control functionality. It's an electric button, so when pressed, it allows current to flow in the electric circuit. the *SteeringWheelECU* receives the current flow and will make a decision. The same applies to the other ACC buttons.

ACCDecreaseDistance It's a button used to decrease the distance that the acc must maintain.

ACCDecreaseSpeed It's a button used to decrease the speed that the acc must maintain.

ACCIncreaseDistance It's a button used to increase the distance that the acc must maintain.

ACCIncreaseSpeed It's a button used to increase the speed that the acc must maintain.

AcceleratorByPedal It's is the accelerator pedal used to accelerate the car.

ACCStartButton It's a button used to turn on the acc functionality.

BrakeByPedal It's the brake pedal used to decelerate the car.

Display It is a driver display showing useful information such as current speed and adaptive. cruise control status.

Radar Radar is the key technology for the development of advanced driver assistance systems (ADAS), which can instantly measure distance, angle and speed and produce detailed images of the surrounding environment. It is the fundamental sensor for ACC.

WheelRotationSpeedRearLT It is the sensor in the left rear wheel for measuring RPM. In the wheel next to the disc brake there is a cogwheel. An electric field is emitted by the sensor perpendicular to the magnetic field produced by the cogwheel. This results in the production of an alternating current. The electronics of the sensor converts the analogue current signal into a numerical current into a numerical signal.

WheelRotationSpeedRearRT It is the sensor in the right rear wheel for measuring RPM. It is the same sensor as described above.

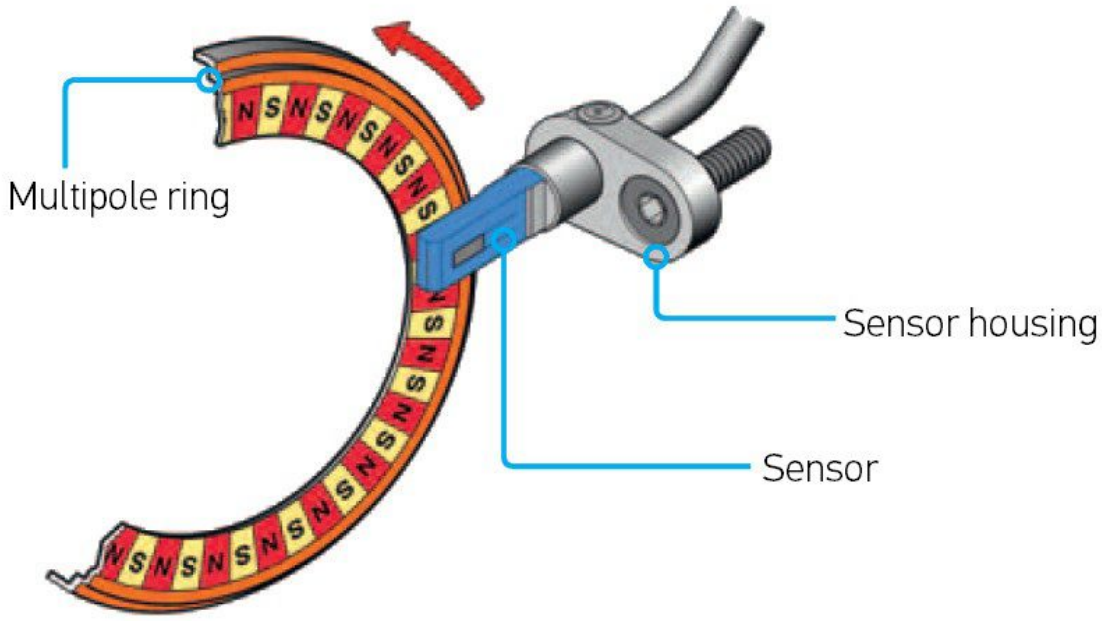


Figure 5.16: RPM sensor, from [15]

The actuators are:

BrakeCaliperFrontLT The brake caliper is one of the parts of the car's braking system. It is a metal part that allows you to adjust the intensity of your brakes. When you apply the brake pedal, the caliper pushes the pads against the disc, using the mechanical force generated by the flow of brake fluid into the system. It is positioned in the left front wheel.

BrakeCaliperFrontRT It is the brake caliper located in the right front wheel.

BrakeCaliperRearLT It is the brake caliper located in the right rear wheel.

BrakeCaliperRearRT It is the brake caliper located in the right rear wheel.

TractionFrontWheelLT This actuator tries to simulate traction for the left front wheel.

TractionFrontWheelRT This actuator tries to simulate traction for the right front wheel.

Next I tried to define the ECUs to be installed in the car for ACC functionality. As written before, all ECUs have only CAN interfaces. In the image (5.15) the interfaces are represented by a green square. The ECUs are:

AdaptiveCruiseControlECU Its purpose is to receive all the information necessary for the correct functioning of the autonomous driving functionality, so that it can make elaborations on them, and finally it will exchange these elaborations with the other ECUs such as *MotionECU*. It has three CAN interfaces: the first with the *SteeringWheelECU* and the bus name is *SteeringWheel-ACC*. The second with the *RadarECU* and the bus name is *Radar-ACC*. The third shared with *MotionECU*, *WheelRotationSpeedRearLT*, *WheelRotationSpeedRearRT*, *BrakeCaliperRearLT*, *BrakeCaliperRearRT*, *BrakeCaliperFrontLT*, *BrakeCaliperFrontRT* and the bus name is *ACC-Wheel-Motion*.

CockpitECU It is used to format information received from other ECUs appropriately to show on the driver display. It has only one CAN interface in a shared bus with *MotionECU* called *Cockpit-MotionECU*.

MotionECU It is the central ECU in this diagram because it communicates with several ECUs that are meant to make the car move. *MotionECU* receives parameters from *AdaptiveCruiseControlECU*, pedal pressure information from *PedalECU* and it will decide whether to accelerate or decelerate the car, sending signals to actuators such as traction or brake calipers. This ECU also communicates directly with the cockpitECU to send ACC status or current speed. It has four CAN interfaces: the first to communicate with *CockpitECU* through *Cockpit-MotionECU* bus. The second to communicate with *AdaptiveCruiseControlECU WheelRotationSpeedRearLT*, *WheelRotationSpeedRearRT*, *BrakeCaliperRearLT*, *BrakeCaliperRearRT*, *BrakeCaliperFrontLT*, *BrakeCaliperFrontRT* and the bus name is *ACC-Wheel-Motion*. The third to communicate with *TractionFrontWheelLT* and *TractionFrontWheelRT* with the bus *ACC-Traction*. The fourth to communicate with PedalECU through *Pedal-Motion* bus.

PedalECU It is the ECU used to get the value of the pedal pressure from the driver. Tt has only one CAN interface that is used to communicate with *motionECU*, the other two ports visible in the picture (5.15) are ports dedicated to electrical signals coming directly from the pedals;

RadarECU It obtains obstacle presence information from the radar sensor and exchanges it with the *AdaptiveCruiseControlECU*. It has two CAN interfaces: one for the bus *Radar-ACC* for the communication with *AdaptiveCruiseControlECU*, and the other for the bus *Radar-RadarECU* for the communication with *RadarECU*.

SteeringWheelECU It receives commands from the acc controller in the steering wheel and sends them to the *AdaptiveCruiseControlECU* which will process them for proper operation. It has a CAN interface for communication with *AdaptiveCruiseControlECU* through the *SteeringWheel-ACC* bus, while all buttons that are placed in the steering wheel are electrical signals that are sampled by the ECU itself, and then sent through the CAN interface.

More detailed information about signals, frames and other network information are in the next section 5.6.

As it is explained in chapter 3.3, in order to perform the auto generation of signals through the *run signal router* functionality, it is necessary first to perform the mapping between the logical model or software model and the hardware model, that is between the logical/software diagram and the network diagram. The mapping is done between logic or software components and hardware components. In this case, the mapping between logic and hardware is identical to the mapping between software and hardware because as explained in the previous chapter, the logic and software diagrams are identical. They can be seen in the network diagram (5.15) through rectangles connected to the hardware component by dotted lines, or it is outlined in the following table:

Mappings	
Logic/Software Component	**Hardware Component**
ACC_CNL	SteeringWheelECU
ACCDecreaseDistance	SteeringWheelECU
ACCDecreaseSpeed	SteeringWheelECU
AcceleratoraByPedal	PedalECU
ACCIncreaseDistance	SteeringWheelECU
ACCIncreaseSpeed	SteeringWheelECU
ACCStartButton	SteeringWheelECU
AdaptiveCruiseControlFunction	AdaptiveCruiseControlECU
BrakeByPedal	PedalECU
BrakeCaliperFrontLT	BrakeCaliperFrontLT
BrakeCaliperFrontRT	BrakeCaliperFrontRT
BrakeCaliperRearLT	BrakeCaliperRearLT
BrakeCaliperRearRT	BrakeCaliperRearRT
BrakeFunction	MotionECU
CalculateSpeed	MotionECU
CockpitDisplay	Display
CockpitFunction	CockpitECU
DriveTrainFunction	MotionECU
FrontWheelLT	TractionFrontWheelLT

Mappings	
Logic/Software Component	**Hardware Component**
FrontWheelRT	TractionFrontWheelRT
Radar	Radar
RadarFunction	RadarECU
WheelRotationSpeedRearLT	WheelRotationSpeedRearLT
WheelRotationSpeedRearRT	WheelRotationSpeedRearRT

Table 5.11: Mapping table

Through data mapping and signal routing, for each data element created in the logic diagram, a signal mapped to the data element can be created. After mapping, the next step is the creation of signals with the *run signal router* functionality, which allows the automatic creation of signals. This feature has several options to select for signal creation, such as automatically creating new gateways if needed, and running frame-PDU synthesis, i.e. the creation of frame transmissions and PDU transmission. With *run signal router* you have to choose the routing algorithm to calculate the frame path. The default algorithm is the Dijkstra's algorithm. With this algorithm, you have to assign a cost to gateways, bus systems etc. The routing algorithm will use these costs to calculate the cheapest path.

5.6 Communication

In the communication layer, all signals are generated manually or automatically by *run signal router* functionality. In this layer there is no diagrams and it is used to create all data needed for communication, such as frames and their bits. The explanation of the communication layer follows the order of procedures to be done using the methodology that PREEvision proposes. In this section there is an explanation of all the signals with their length relative to the data created in the previous layer. The reason for the length of the individual signals will be explained, and how the frames were created.

Once the *run signal router* is executed, the signals generated are as follows:

1. **ACCCNL_Signal**
   - Length: 1 bit
   - Path: *SteeringWheelECU* to *AdaptiveCruiseControlECU*
   - The *SteeringWheelECU* sends this signal when the driver clicks the "CNL button" in the steering wheel. It deactivates the ACC function.
2. **ACCDecreaseDistance_Signal**

- Length: 1 bit
- Path: *SteeringWheelECU* to *AdaptiveCruiseControlECU*
- The *SteeringWheelECU* sends this signal with value 1 when the driver clicks the "- button" in the steering wheel. It decreases the distance from the car in front. -10 km/h will be subtracted from the distance to be kept from the car in front.

3. **ACCDecreaseSpeed_Signal**

   - Length: 2 bits
   - Path: *SteeringWheelECU* to *AdaptiveCruiseControlECU*
   - The *SteeringWheelECU* sends this signal when the driver clicks the "volume down button" in the steering wheel. It decreases the current speed by 1 km/h if the signal value is 1 or 10 km/h if the signal value is 2.

4. **AcceleratorByPedal_Signal**

   - Length: 7 bits
   - Path: *PedalECU* to *MotionECU*
   - The *PedalECU* sends this signal when the driver presses the accelerator pedal. The signal contains the value of the pedal pressure mapped from 0 to 100.

5. **AcceleratorTraction_Signal**

   - Length: 7 bits
   - Path: *MotionECU* to *TractionFrontWheelLT* and *TractionFrontWheelRT*
   - The *MotionECU* sends this signal when the ACC wants to increase speed or the driver presses the accelerator pedal. It is the consequence of one of these two signals, namely AcceleratorByPedal_Signal or ACCOutputIncreaseSpeed_Signal.

6. **ACCIncreaseDistance_Signal**

   - Length: 1 bit
   - Path: *SteeringWheelECU* to *AdaptiveCruiseControlECU*
   - The *SteeringWheelECU* sends this signal with value 1 when the driver clicks the "+ button" in the steering wheel. It increases the distance from the car in front. 10 km/h will be added by the distance to be kept from the car in front.

7. **ACCIncreaseSpeed_Signal**
    - Length: 2 bits
    - Path: *SteeringWheelECU* to *AdaptiveCruiseControlECU*
    - The *SteeringWheelECU* sends this signal when the driver clicks the "volume up button" in the steering wheel. It increases the current speed by 1 km/h if the signal value is 1 or 10 km/h if the signal value is 2.
8. **ACCOutputDecreaseSpeed_Signal**
    - Length: 8 bits
    - Path: *AdaptiveCruiseControlECU* to *MotionECU*
    - The *AdaptiveCruiseControlECU* sends this signal to decrease the speed and it is sent when there is a change, which may be brake pedal pressure, the presence of an obstacle, or a decrease in speed or increase in distance from the ACC controller. This signal contains the value of the speed to hold.
9. **ACCOutputIncreaseSpeed_Signal**
    - Length: 8 bits
    - Path: *AdaptiveCruiseControlECU* to *MotionECU*
    - The *AdaptiveCruiseControlECU* sends this signal to increase the speed and it is sent when there is a change, which may be accelerator pedal pressure, the absence of an obstacle, or an increase in speed or decrease in distance from the ACC controller. This signal contains the value of the speed to hold.
10. **ACCStartState_Signal**
    - Length: 1 bit
    - Path: *SteeringWheelECU* to *AdaptiveCruiseControlECU*
    - The *SteeringWheelECU* sends this signal when the driver clicks the "cruise control button" in the steering wheel. It turns on or off the ACC function.
11. **ACCStatus_Signal**
    - Length: 18 Bits
    - Path: *AdaptiveCruiseControlECU* to *CockpitECU*. The *MotionECU* acts as a gateway between the *AdaptiveCruiseControlECU* and the *CockpitECU*.

- The *AdaptiveCruiseControlECU* sends this signal in cyclic mode containing the speed and distance set in the ACC and the active or inactive state of the ACC. This information will be used to show data on the cockpit display. Its length is 18 bits because it is composed of 8 bits for speed, 9 bits for distance and 1 bit for status.

12. **BrakeByPedal_Signal**

    - Length: 7 bits
    - Path: *PedalECU* to *MotionECU*
    - The *PedalECU* sends this signal when the driver presses the brake pedal. The signal contains the value of the pedal pressure mapped from 0 to 100.

13. **BrakeCalipersFrontLT_Signal**

    - Length: 7 bits
    - Path: *MotionECU* to *BrakeCaliperFrontLT*
    - The *MotionECU* sends this signal when the ACC wants to decrease speed or the driver presses the brake pedal. It is the consequence of one of these two signals, namely BrakeByPedal_Signal or ACCOutputDecreaseSpeed_Signal. BrakeCalipersFrontLT_Signal contains the value from 0 to 100 that indicates the tightness of the left front brake caliper.

14. **BrakeCalipersFrontRT_Signal**

    - Length: 7 bits
    - Path: *MotionECU* to *BrakeCaliperFrontRT*
    - The *MotionECU* sends this signal when the ACC wants to decrease speed or the driver presses the brake pedal. It is the consequence of one of these two signals, namely BrakeByPedal_Signal or ACCOutputDecreaseSpeed_Signal. BrakeCalipersFrontRT_Signal contains the value from 0 to 100 that indicates the tightness of the right front brake caliper.

15. **BrakeCalipersRearLT_Signal**

    - Length: 7 bits
    - Path: *MotionECU* to *BrakeCaliperRearLT*
    - The *MotionECU* sends this signal when the ACC wants to decrease speed or the driver presses the brake pedal. It is the consequence of one of these two signals, namely BrakeByPedal_Signal or ACCOutputDecreaseSpeed_Signal. BrakeCalipersRearLT_Signal contains the value from 0 to 100 that indicates the tightness of the left rear brake caliper.

16. **BrakeCalipersRearRT_Signal**

    - Length: 7 bits
    - Path: *MotionECU* to *BrakeCaliperRearRT*
    - The *MotionECU* sends this signal when the ACC wants to decrease speed or the driver presses the brake pedal. It is the consequence of one of these two signals, namely BrakeByPedal_Signal or ACCOutputDecreaseSpeed_Signal. BrakeCalipersRearLT_Signal contains the value from 0 to 100 that indicates the tightness of the right rear brake caliper.

17. **CockpitInfo_Signal**

    - Length: 27 bits
    - Path: *CockpitECU* to *Display*
    - The *CockpitECU* sends this signal in cyclic mode to show the ACC status and the current speed in the display. Its length is 27 bits because it is composed of 18 bits for ACC status and 9 bits for current speed.

18. **RadarInfo_Signal**

    - Length: 9 bits
    - Path: *Radar* to *RadarECU*
    - The *Radar* sends this signal in cyclic mode to to give information about the environment in front of the car. Being a simplified version it only provides the distance to an obstacle, from 0 to 300 meters.

19. **RadarOutput_Signal**

    - Length: 9 bits
    - Path: *RadarECU* to *AdaptiveCruiseControlECU*
    - The *RadarECU* received the RadarInfo_Signal, it carries out some elaborations and sends the RadarOutput_Signal to *AdaptiveCruiseControlECU*.

20. **RPM_wheel_Left_Signal**

    - Length: 13 bits
    - Path: *WheelRotationSpeedRearLT* to *MotionECU*
    - The *WheelRotationSpeedRearLT* sends this signal in cyclic mode containing the number of rotations per minute of the left rear wheel.

21. **RPM_wheel_Right_Signal**

- Length: 13 bits
- Path: *WheelRotationSpeedRearRT* to *MotionECU*
- The *WheelRotationSpeedRearRT* sends this signal in cyclic mode containing the number of rotations per minute of the right rear wheel.

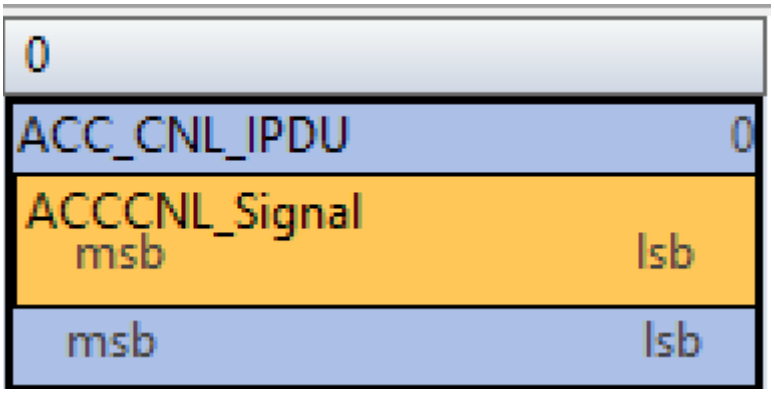
22. **SpeedIndicator_Signal**

- Length: 9 bits
- Path: *MotionECU* to *AdaptiveCruiseControlECU* and *CockpitECU*
- The *MotionECU* sends this signal containing the current speed after calculating it through the two signals RMP_wheel_right_signal and RMP_wheel_left_signal.

Each signal is transmitted in a frame. A frame can contain one or more signals. Below is an explanation of the frames in this car architecture, giving the CAN Frame ID, frame length, frame sending frequency, frame usage, and frame content. The frames are 18:

1. **ACC_CNL_Frame**

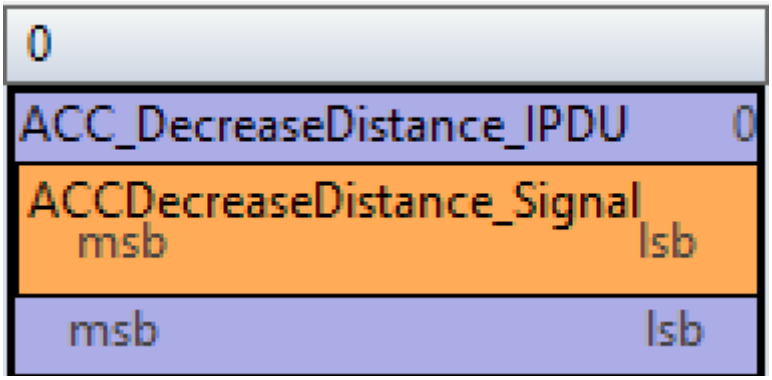
- ID: `0x6E`
- Path: *SteeringWheelECU* to *AdaptiveCruiseControlECU*
- It only carries the *ACCCNL_Signal*



2. **ACC_DecreaseDistance_Frame**

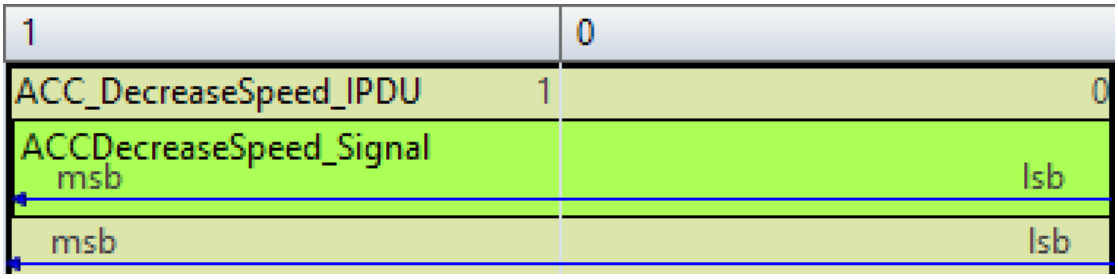
- ID: `0x70`
- Path: *SteeringWheelECU* to *AdaptiveCruiseControlECU*

- It only carries the *ACCDecreaseDistance_Signal*



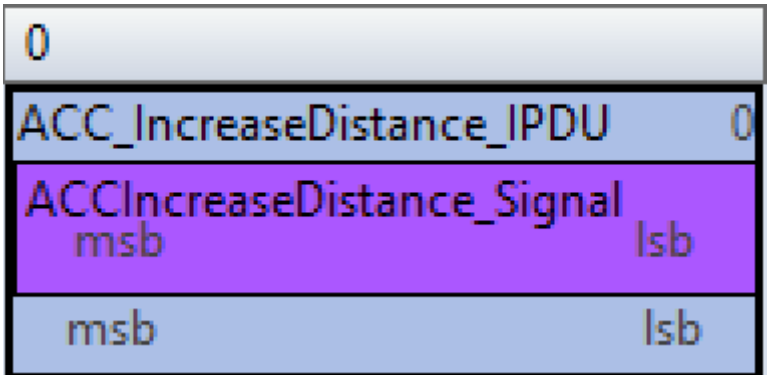
3. **ACC_DecreaseSpeed_Frame**

- ID: `0x6F`
- Path: *SteeringWheelECU* to *AdaptiveCruiseControlECU*
- It only carries the *ACCDecreaseSpeed_Signal*



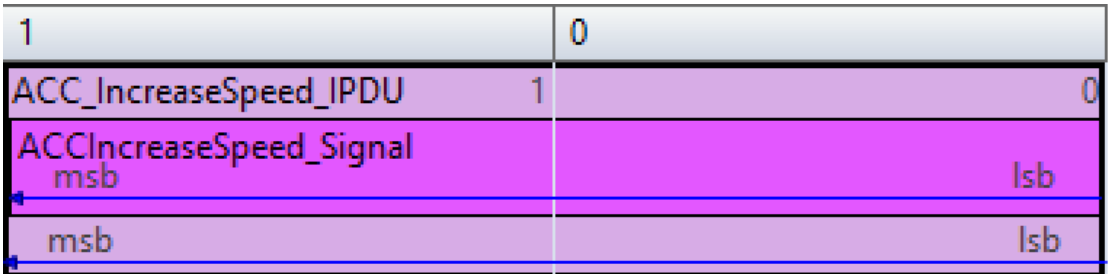
4. **ACC_IncreaseDistance_Frame**

- ID: `0x72`
- Path: *SteeringWheelECU* to *AdaptiveCruiseControlECU*
- It only carries the *ACCIncreaseDistance_Signal*



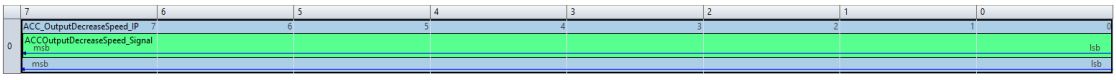
5. **ACC_IncreaseSpeed_Frame**

   - ID: `0x71`
   - Path: *SteeringWheelECU* to *AdaptiveCruiseControlECU*
   - It only carries the *ACCDecreaseSpeed_Signal*



6. **ACC_OutputDecreaseSpeed_Frame**

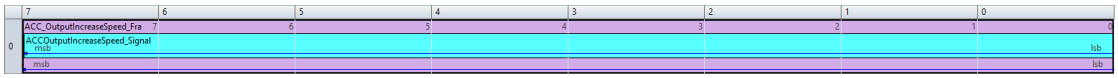
   - ID: `0x14`
   - Path: *AdaptiveCruiseControlECU* to *MotionECU*
   - It only carries the *ACC_OutputDecreaseSpeed_Signal*



7. **ACC_OutputIncreaseSpeed_Frame**

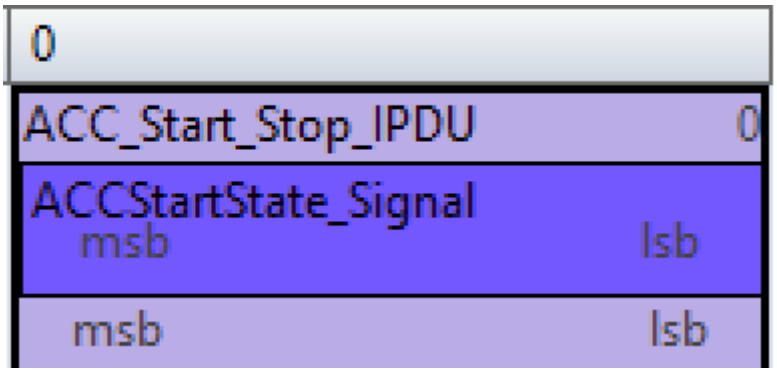
   - ID: `0x15`
   - Path: *AdaptiveCruiseControlECU* to *MotionECU*

- It only carries the *ACC_OutputIncreaseSpeed_Signal*



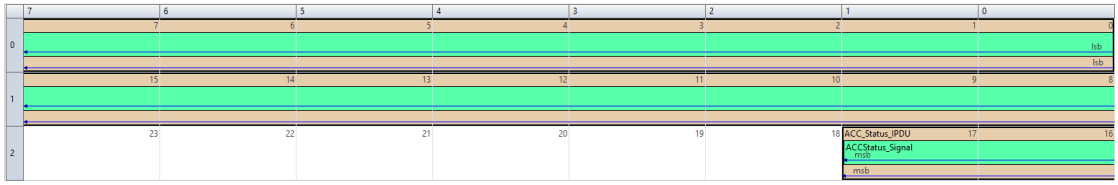
8. **ACC_Start_Stop_Frame**

- ID: `0x6D`
- Path: *SteeringWheelECU* to *AdaptiveCruiseControlECU*
- It only carries the *ACC_StartState_Signal*



9. **ACC_Status_Frame**

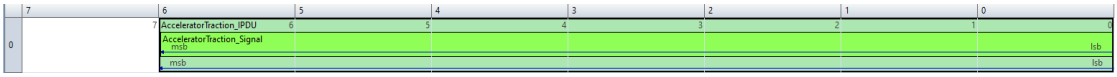
- ID: `0x73`
- Path: *SteeringWheelECU* to *AdaptiveCruiseControlECU*
- ID: `0x14`
- Path: *MotionECU* to *CockpitECU*
- It only carries the *ACC_Status_Signal* but it has two CAN ID, because it is sent on two different buses and in each bus has a different priority. It is sent every 20 ms.



10. **Accelerator_Traction_Frame**

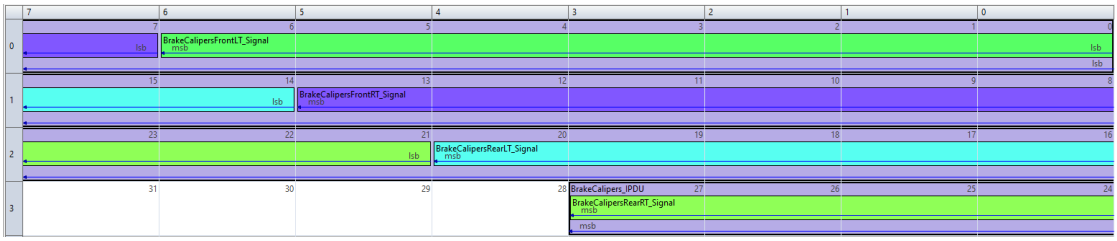
- ID: `0xB`

- Path: *MotionECU* to *TractionFrontWheelLT* and *TractionFrontWheelRT*
- It only carries the *ACC_AcceleratorTraction_Signal* but it is received by *TractionFrontWheelLT* and *TractionFrontWheelRT*



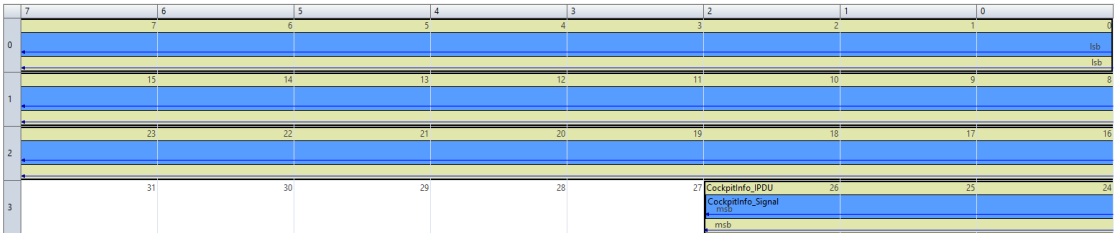
11. **BrakeCalipers_Frame**

- ID: `0xA`
- Path: *MotionECU* to *BrakeCaliperFrontLT*, *BrakeCaliperFrontRT*, *BrakeCaliperRearLT* and *BrakeCaliperRearRT*,
- It carries four signals, one for each brake caliper.They are in one frame because the four actuators are on the same bus, and with one frame I can assign the same priority for each signal. When the frame arrives at its destination, the actuator will read the bit assigned to it.



12. **CockpitInfo_Frame**

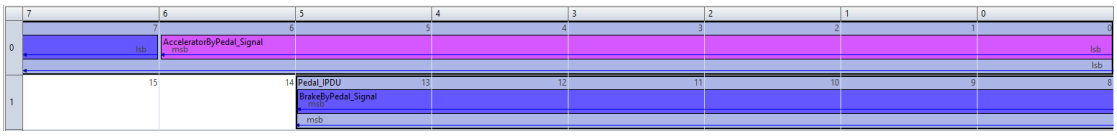
- ID: `0x96`
- Path: *CockpitECU* to *Display*
- It only carries the *CockpitInfo_Signal*. It is sent every 70 ms.



13. **Pedal_Frame**

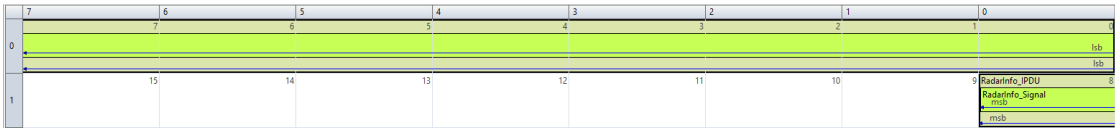
- ID: `0x5`

- Path: *MotionECU* to *PedalECU*
- It carries the *AcceleratorByPedal_Signal* and the *BrakeByPedal_Signal* because these signals must have the same priority, and the driver can also press the pedals at the same time.



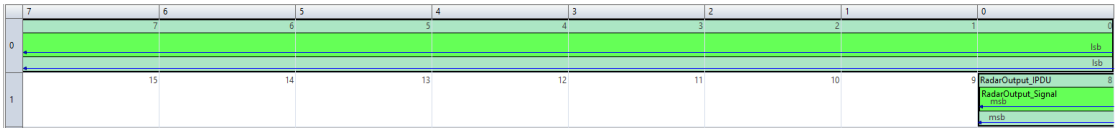
14. **RadarInfo_Frame**

- ID: `0xF`
- Path: *Radar* to *RadarECU*
- It only carries the *RadarInfo_Signal*. It is sent every 40 ms.



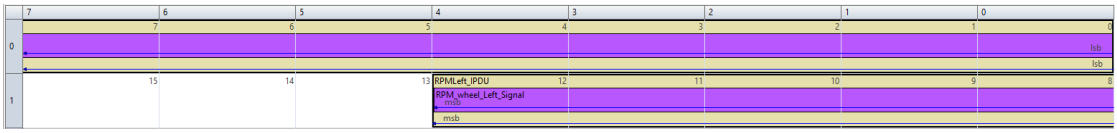
15. **RadarOutput_Frame**

- ID: `0xF`
- Path: *RadarECU* to *AdaptiveCruiseControlECU*
- It only carries the *RadarOutput_Signal*



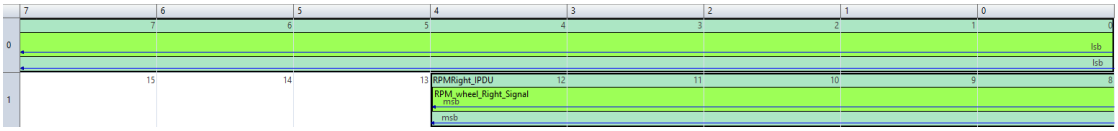
16. **RPMLeft_Frame**

- ID: `0x3C`
- Path: *WheelRotationSpeedRearLT* to *MotionECU*
- It only carries the *RPM_wheel_Left_Signal*. It is sent every 50 ms.



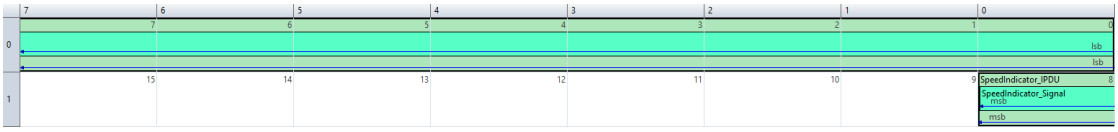
17. **RPMRight_Frame**

- ID: `0x3D`
- Path: *WheelRotationSpeedRearRT* to *MotionECU*
- It only carries the *RPM_wheel_Right_Signal*. It is sent every 50 ms.



18. **SpeedIndicator_Frame**

- ID: `0x60`
- Path: *MotionECU* to *AdaptiveCruiseControlECU*
- ID: `0x15`
- Path: *MotionECU* to *CockpitECU*
- It only carries the *SpeedIndicator_Signal* but it has two CAN ID, because it is sent on two different buses and in each bus has a different priority.



The ones explained above are all frames that make up the network architecture of this thesis. PREEvision offers different views for the frames, one for the description of the individual bits as is shown in the images above, or a table summarizing all the frames with frame information such as the CAN frame ID, frame length, timing period, and interfaces involved.

CAN Frame	CAN Frame ID	CAN Addre...	Byte Length	CAN Frame Transmission	Timing ...	R...	Routing	Sending Interfaces	Receiving Interfaces
ACC_CNL_Frame	0x6E	Standard	1	ACC_CNL_Frame			SteeringWheelToACC	SteeringWheelECU_SteeringWheelToACC	AdaptiveCruiseControlECU_SteeringWheelToACC
ACC_DecreaseDistance_Frame	0x70	Standard	1	ACC_DecreaseDistance_Fr...			SteeringWheelToACC	SteeringWheelECU_SteeringWheelToACC	AdaptiveCruiseControlECU_SteeringWheelToACC
ACC_DecreaseSpeed_Frame	0x6F	Standard	1	ACC_DecreaseSpeed_Frame			SteeringWheelToACC	SteeringWheelECU_SteeringWheelToACC	AdaptiveCruiseControlECU_SteeringWheelToACC
ACC_IncreaseDistance_Frame	0x72	Standard	1	ACC_IncreaseDistance_Fra...			SteeringWheelToACC	SteeringWheelECU_SteeringWheelToACC	AdaptiveCruiseControlECU_SteeringWheelToACC
ACC_IncreaseSpeed_Frame	0x71	Standard	1	ACC_IncreaseSpeed_Frame			SteeringWheelToACC	SteeringWheelECU_SteeringWheelToACC	AdaptiveCruiseControlECU_SteeringWheelToACC
ACC_OutputDecreaseSpeed_Fra...	0x14	Standard	1	ACC_OutputDecreaseSpe...			ACC-Wheel-Motion	AdaptiveCruiseControlECU_ACC-Wheel-Motion	MotionECU_ACC-Wheel-Motion
ACC_OutputIncreaseSpeed_Fra...	0x15	Standard	1	ACC_OutputIncreaseSpee...			ACC-Wheel-Motion	AdaptiveCruiseControlECU_ACC-Wheel-Motion	MotionECU_ACC-Wheel-Motion
ACC_Start_Stop_Frame	0x6D	Standard	1	ACC_Start_Stop_Frame			SteeringWheelToACC	SteeringWheelECU_SteeringWheelToACC	AdaptiveCruiseControlECU_SteeringWheelToACC
ACC_Status_Frame	0x73	Standard	3	ACC_Status_Frame	20.0		ACC-Wheel-Motion	AdaptiveCruiseControlECU_ACC-Wheel-Motion	MotionECU_ACC-Wheel-Motion
	0x14	Standard		ACC_Status_Frame			CockpitToMotionECU	MotionECU_CockpitToMotionECU	Cockpit_CockpitToMotionECU
		Standard							
AcceleratorTraction_Frame	0xB	Standard	1	AcceleratorTraction_Frame			ACC-Traction	MotionECU_ACC-Traction	TractionFrontWheelRT_ACC-Traction
									TractionFrontWheelLT_ACC-Traction
BrakeCalipers_Frame	0xA	Standard	4	BrakeCalipers_Frame			ACC-Wheel-Motion	MotionECU_ACC-Wheel-Motion	BrakeCalliperRearLT_ACC-Wheel-Motion
									BrakeCalliperRearRT_ACC-Wheel-Motion
									BrakeCalliperFrontRT_ACC-Wheel-Motion
									BrakeCalliperFrontLT_ACC-Wheel-Motion
CockpitInfo_Frame	0x96	Standard	4	CockpitInfo_Frame	70.0		Display-Cockpit	CockpitECU_Display-Cockpit	Display_Display-Cockpit
Pedal_Frame	0x5	Standard	2	Pedal_Frame			Pedal-Motion	PedalECU_PedalToMotion	MotionECU_PedalToMotion
RadarInfo_Frame	0xF	Standard	2	RadarInfo_Frame	44.0		Radar-RadarECU	Radar_Radar-RadarECU	RadarECU_Radar-RadarECU
RadarOutput_Frame	0xF	Standard	2	RadarOutput_Frame			Radar-ACC	RadarECU_Radar-ACC	AdaptiveCruiseControlECU_Radar-ACC
RPMLeft_Frame	0x3C	Standard	2	RPMLeft_Frame	50.0		ACC-Wheel-Motion	WheelRotationSpeedRearLT_ACC-Wheel-Motion	MotionECU_ACC-Wheel-Motion
RPMRight_Frame	0x3D	Standard	2	RPMRight_Frame	50.0		ACC-Wheel-Motion	WheelRotationSpeedRearRT_ACC-Wheel-Motion	MotionECU_ACC-Wheel-Motion
SpeedIndicator_Frame	0x60	Standard	2	SpeedIndicator_Frame			ACC-Wheel-Motion	MotionECU_ACC-Wheel-Motion	AdaptiveCruiseControlECU_ACC-Wheel-Motion
	0x15	Standard		SpeedIndicator_Frame			CockpitToMotionECU	MotionECU_CockpitToMotionECU	Cockpit_CockpitToMotionECU

Figure 5.17: Can Frame Overview

After creating all frames, through the run frame-PDU synthesis function, it is possible to create the CAN frame transmission and the related PDU transmission, in order to complete the communication within each channel or CAN Bus. The differences between CAN frame and CAN frame transmission, or between PDU and PDU transmission are reported in the chapter 3.3 . Once this function is executed, the network architecture through PREEvision is completed. Once a complete network architecture is obtained, it is then possible to perform calculations through metrics that are another level of PREEvision, or it is possible to export useful information of the architecture, such as CAN DBC or ARXML files.

Chapter 6

Metrics

PREEvision metrics are functions that can be applied to the various layers of PREEvision to obtain estimates or concrete results; for instance, it's possible to create a metric to calculate the bus load of the network architecture. The metrics are composed of graphical objects programmable in Java. Metrics are not associated with a single architecture but are reusable. They can be used for a small part of an architecture, for the entire architecture, or for a different architecture. PREEvision metrics:

- They are used to perform calculations on the data model for analysis and optimization.
- It is always executed on the full model, so the result is up to date.
- The output of metrics is streamed into reports or into GUI as lights, scales or values.
- They are based on a graphical notation and can be expanded by Java code.

There are different metric artifacts such as Objects, Connection, Report and Chart Block; they will be described in the next sections. (This information are taken from PREEvision's manual [11])

This chapter describes the metrics created through the use of the PREEvision software, associated to the network architecture created in the previous chapter. The objective is to understand the potentiality of the metrics, by making some interrogations on the architecture in order to verify the correct realization or to verify the presence of big critical points, or trying to understand which are the points that can be improved. The development of these metrics has been a bit extreme to understand the limit beyond which we can go in the realization of it, however any conclusions are discussed in the chapter 7. The metrics developed are:

- Hop Count
- Bus Load
- ECU Load Single Core
- ECU Load Dual Core

6.1 Hop count

The hop count metric calculates the number of ECU crossed by 18 CAN frames that make up the architecture. In a big car architecture, it is useful to understand if there is a frame that crosses many ECUs, since the big number of frames and the graphical visualization of the hardware model doesn't allow to get the data easily. The number of ECUs crossed by a frame is an important information since typically a frame does not cross more than 3 ECUs, and in case of network congestion, a frame could have important delays. The metric is composed by a *metric context block* artifact to insert the CAN frames to interrogate, a *calculation block* artifact where there are the java code to make the calculation and a *report result block* artifact to show the result in a table.

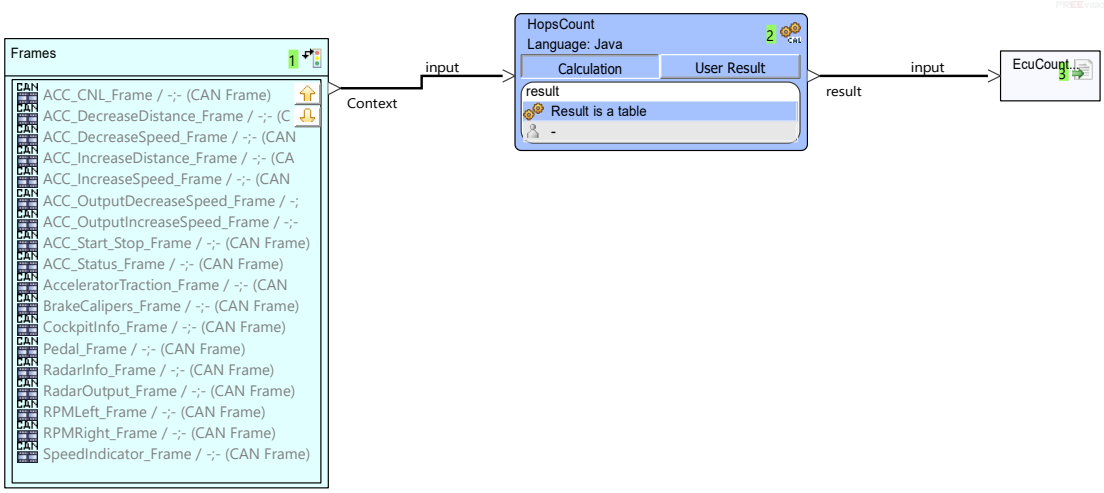


Figure 6.1: Hop count metric diagram

The calculation of the *calculation block* artifact is always performed in the *calculateResult* method. In this case, the method performs a loop for each input frame, obtains the number of frame transmission relative to the frame and increments it by one. For each input frame, the result is shown in the table through the *report result block* artifact.

```
1  private static final String IN_INPUT = "input"; //$NON-NLS-1$
2    private static final String OUT_RESULT = "result"; //$NON-NLS-1$
3
4    private ITempTable table = TempTableFactory.createTable();
5
6    private ITempTableColumn colFrame = TempTableFactory.createTableColumn(table, "
       Frame");
7
8    private ITempTableColumn colEcuCount = TempTableFactory.createTableColumn(table, "
       ECU Count");
9
10     @Override
11     public Object calculateResult() {
12       if (getInput(IN_INPUT) instanceof Collection<?>) {
13       Iterator<?> f = ((Collection<?>) getInput(IN_INPUT)).iterator();
14       while (f.hasNext()) {
15         Object frame = f.next();
16         if (frame instanceof MCANFrame) {
17           int count = ((MCANFrame) frame).getFrameTransmissions().size() + 1;
18           ITempTableRow row = TempTableFactory.createTableRow(table);
19           table.createTableCell(colFrame, row, frame.toString(), String.class);
20           table.createTableCell(colEcuCount, row, count, int.class);
21         } else {
22           printErrorMessage("INSERT A CAN FRAME", "");
23           return null;
24         }
25       }
26     }
27     setResult(OUT_RESULT, table);
28         return null;
29     }
```

Listing 6.1: Hop count metric

Each frame has associated at least 1 frame transmission, which means it passes through a bus. 2 frame transmission means that the frame crosses 2 buses. In a bus there is a sender and one or more receivers, but in the calculation of the crossed ECU, the number of hops will always be equal to 2. Only when a frame crosses a gateway the number can be higher than 2, because the gateway ECU is not interested in the information but must only forward the frame. This is the reason for the increase of one in the Java code. The result is:

Hop Count	
Frame	**ECU Count**
ACC_CNL_Frame	2
ACC_DecreaseDistance_Frame	2

Hop Count	
Frame	**ECU Count**
ACC_DecreaseSpeed_Frame	2
ACC_DecreaseSpeed_Frame	2
ACC_IncreaseDistance_Frame	2
ACC_IncreaseSpeed_Frame	2
ACC_OutputDecreaseSpeed_Frame	2
ACC_OutputIncreaseSpeed_Frame	2
ACC_Start_Stop_Frame	2
ACC_Status_Frame	3
AcceleratorTraction_Frame	2
BrakeCalipers_Frame	2
CockpitInfo_Frame	2
Pedal_Frame_Frame	2
RadarInfo_Frame	2
RadarOutput_Frame	2
RPMLeft_Frame	2
RPMRight_Frame	2
SpeedIndicator_Frame	3

Table 6.1: Hop count table

6.2 Bus load

The *bus load* metric calculates an estimate of the load of the buses entered as input. By defining a time interval, the metric allows you to calculate the percentage of bus load based on the amount of messages traversing that bus in that given time. The bus load depends on the baud rate which is the speed at which messages are transmitted on a bus; a baud rate of 125000 means that a maximum of 125000 bits per second can be transferred. The bus load depends not only on the baud rate, but also on the length of messages sent between nodes and the frequency of sending. Typically the value should never exceed 70% to avoid losing messages. [16]

The metric is composed of a *metric context block* artifact to insert the buses to be analyzed, a *calculation block* artifact where there is the java code that performs the calculation, two *parameter block* artifacts to perform the calculation counting also the stuffing bits, one to indicate the time interval to be analyzed, and finally a *report result block* artifact to show the result in the table. In order not to count any stuffing bits the *statisticStuffBitInterval* parameter must be equal to 0, otherwise they will be counted. In my case the time interval is set to 100 ms.

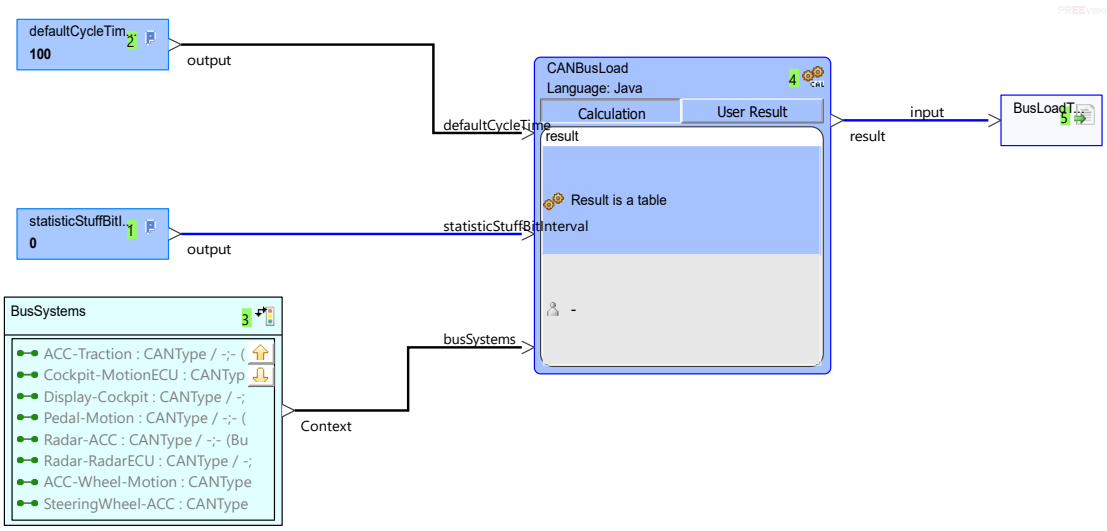


Figure 6.2: Bus load metric diagram

In this architecture the frames are all standard CAN frames and therefore are composed of:

- 1 bit start bit.
- 11 bit identifier
- 1 bit RTR
- 6 bit control field
- 0 to 64 bit data field
- 15 bit CRC
- Bit stuffing is possible in the first 34 bits, for every sequence of 5 consecutive bits of same level.
- 3 bit delimiter, ack etc.
- 7 bit end of frame
- 3 bit intermission field after frame.

```
1  public Object calculateResult() {
2      getInputs();
3      Iterator<MBusSystem> busIterator = busSystems.iterator();
4      while (busIterator.hasNext()) {
5        BusLoad cumulatedLoad = new BusLoad();
6        MBusSystem bus = busIterator.next();
7        if ((bus.getBelongsToCluster() != null && bus.getBelongsToCluster().getBaudrate() !=
         0.0)
8              || (bus.getBusType() != null && bus.getBusType().getBaudrate() != 0.0)) {
9          ArrayList<MSignalTransmission> signalTransmissions = new ArrayList<>();
10         Collection<MAbstractBusCommunication> communications = bus.
       getBusCommunications();
11         Iterator<MAbstractBusCommunication> busComIt = communications.iterator();
12         while (busComIt.hasNext()) {
13           MAbstractBusCommunication aBusCom = busComIt.next();
14           if (aBusCom instanceof MCANCommunication) {
15             MCANCommunication canCom = (MCANCommunication) aBusCom;
16             if (canCom.isIncludedInActiveVariant() && canCom.isPartOfActiveVariant()) {
17               if (canCom.getBusRouting() != null) {
18                 if (canCom.getBusRouting() instanceof MChannelCommunication) {
19                   MChannelCommunication routing = (MChannelCommunication) canCom.
       getBusRouting();
20                   signalTransmissions.addAll(routing.getSignalTransmissions());
21                 }
22               }
23             }
24           }
25         }
26         calculateLoad(signalTransmissions, bus);
27       }
28     }
29
30     setResult(OUT_RESULT, results);
31         return null;
32     }
```

Listing 6.2: calculateResult() method

With this method, the buses passed as input are saved in an iterator. For each bus the signals transmitted within that bus are obtained. Finally the method *private void calculateLoad(ArrayList<MSignalTransmission> signalTransmissions, MBusSystem bus)* is called.

```
1  private void calculateLoad(ArrayList<MSignalTransmission> signalTransmissions,
       MBusSystem bus) {
2      Iterator<MSignalTransmission> stIt = signalTransmissions.iterator();
3      BusLoad cumulatedLoad = new BusLoad();
4      while (stIt.hasNext()) {
5        MSignalTransmission st = stIt.next();
6        if (BusLoadUtil.getBusType(st) != null && BusLoadUtil.getBusType(st) instanceof
       MCANType) {
7          double transmissionRate = getTransmissionRateOfSigTrans(st);
8          if (transmissionRate > 0.0) {
9            double cycleTime = 0;
10           cycleTime = getCycleTime(st);
11           if(cycleTime > 0){
12             BusLoad load = calculateBusLoadPerTransmission(st, cycleTime,
       transmissionRate);
13             cumulatedLoad.addBusLoad(load);
14           }
15         }
16       }
17     }
18     ITempTableRow values = TempTableFactory.createTableRow(results);
19     TempTableFactory.createTableCell(busSystemColumn, values, bus, MBusSystem.class);
20     TempTableFactory.createTableCell(avgLoadColumn, values, round(cumulatedLoad.getAvg
       (), 2), Double.class);
21   }
```

Listing 6.3: calculateLoad() method

The *calculateLoad* method does not calculate the bus load yet, but prepares the data to be able to calculate it. Specifically, it obtains the frame by sending frequency from the signal attributes, then calls the method *private BusLoad calculateBusLoadPerTransmission(MSignalTransmission signalTransmission, double cycleTime, double transmissionRate)* to perform the bus load calculation, and finally inserts the obtained value in a row of the final table. Not all frames have a predefined transmission rate in a cyclic manner. For all frames that do not have a sending frequency, the value will be equal to the period under consideration or 100 ms. This allows the worst case to be considered for the bus load calculation.

```
1  private BusLoad calculateBusLoadPerTransmission(MSignalTransmission signalTransmission,
       double cycleTime, double transmissionRate) {
2      BusLoad busLoadPerTransmission = new BusLoad();
3      if (signalTransmission != null && transmissionRate > 0.0) {
4        int frameLength = BusLoadUtil.getFrameLength(signalTransmission, 1);
5        double bitTime = 1 / transmissionRate;
6        double frameTime;
7        if ( statisticStuffBitInterval  == 0) {
8          frameTime = ((47.0 + frameLength) * (defaultCycleTime / cycleTime)) * bitTime;
9        } else {
10         frameTime = (((44.0 + frameLength) + ((34 + frameLength - 1) / 4)) * (
       defaultCycleTime / cycleTime)) * bitTime;
11       }
12       double load = (frameTime / (defaultCycleTime / 1000)) * 100;
13       busLoadPerTransmission.setMax(load); signalLength, transmissionRate));
14     }
15     return busLoadPerTransmission;
16   }
```

Listing 6.4: calculateBusLoadPerTransmission() method

Bus load calculations can be summarized in 3 steps:

Bit time calculation It represents the transmission time of a single bit

Frame time calculation It represents the bus occupation time for the transmission of a frame

Bus load calculation Formula: $(FrameTime/DefaultCycleTime) * 100$

DefaultCycleTime is 100 ms and it is the time interval taken into account. For example, given a bit rate of 500 kbit/s, a frame length of 64 bits and a time interval of 100 ms:

1. bit time $= 1/bitrate = 1/(500 * 1000)s = 2 * 10^{-6}s = 2\mu s$
2. frame time $= 64 * 2\mu s = 128\mu s$
3. bus load $= (128\mu s/100ms) * 100 = 0.00128 * 100 = 0.128\%$

The baud rate in the network architecture of this project is 125000 bits per second for all buses. In case a frame is repeated several times within the DefaultCycleTime, it is considered as a frame with $n * FrameLength$, where n is the number of times it is sent. If there is no bit stuffing in any frame the result of the bus load is the following:

Bus Load	
BusSystem	**Load %**
ACC-Traction	0.44
ACC-Wheel-Motion	8.77
Cockpit-MotionECU	1.07
Display-Cockpit	0.9
Pedal-Motion	1.01
Radar-ACC	0.5
Radar-RadarECU	1.26
SteeringWheel-ACC	2.64

Table 6.2: Bus load table without bit stuffing

The CAN protocol is asynchronous and in order to maintain synchronization between CAN controllers it uses the bit stuffing technique. With the bit stuffing technique, a bit with opposite polarity is added every 5 consecutive bits of the same polarity. This increases the length of the CAN frame. CRC, ACK and end of frame fields that are of a fixed size cannot be stuffed while all other fields can have manipulation caused by this technique, while in the header only 34 bits of the 44 present can be influenced. In the data field varies according to its length. [17] To calculate the size of a frame after bit stuffing, the formula is as follows:

$$8n + 44 + (34 + 8n - 1)/4$$

n is the number of bytes used for the data field. In the worst case a bit is inserted every 4 original bits after the first one (-1 at numerator). The worst case example is this:

11111000011110000...

after the bit stuffing:

11111**0**0000**1**1111**0**0000**1**...

The result with the bit stuffing is the following:

Bus Load	
BusSystem	**Load %**
ACC-Traction	0.5
ACC-Wheel-Motion	10.1
Cockpit-MotionECU	1.23
Display-Cockpit	1.05
Pedal-Motion	1.15
Radar-ACC	0.58

Bus Load	
BusSystem	**Load %**
Radar-RadarECU	1.44
SteeringWheel-ACC	2.98

Table 6.3: Bus load table with bit stuffing

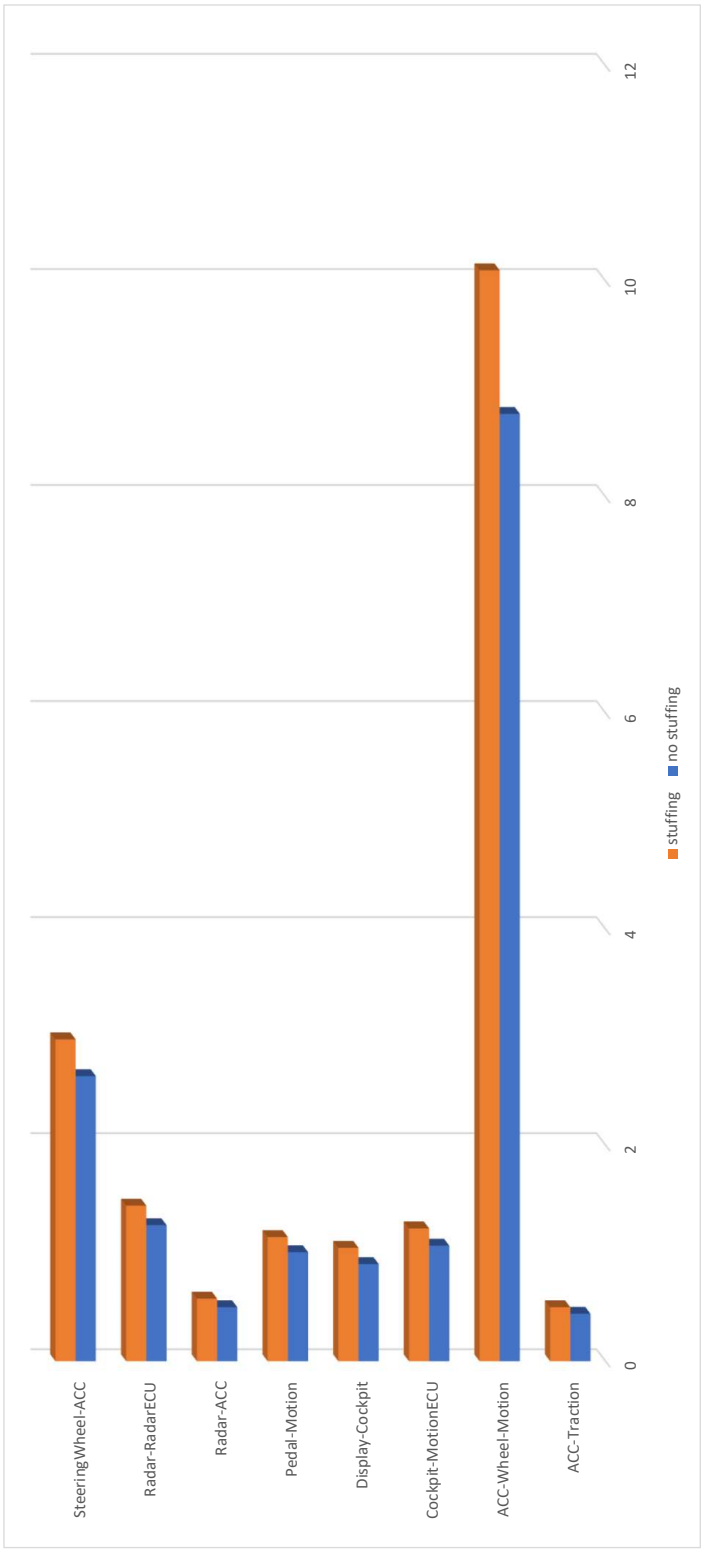


Figure 6.3: Bus load chart with or without bit stuffing

Being the architecture created small, even in the worst case the bit stuffing does not affect heavily the bus load. Obviously, the only bus that shows a slight increase is *ACC-Wheel-Motion* because it is the one where most data bits travel, so it is the one most affected by the increase of stuffed bits.

6.3 ECU load single core

The ecu load single core metric calculates an estimate of the load of the six ECUs that populate the architecture created, assuming all ECUs are single core. The calculation is based on the leaky bucket algorithm. Assuming to have a container with a hole at the bottom, the leaky bucket algorithm is used to understand if filling the container with water will leak from the top or or if it will remain in the container thanks to the constant leakage from the hole at the base of the container. Thus, it allows to understand if the average rate with which the water is poured is greater than the average rate with which the water comes out of the hole. [18] In this metric, water is replaced by credits. Each ECU has a defined number of credits (bucket size) it can handle in a defined time interval, which in this case amounts to 100 ms. Each signal has a defined weight in credits, and each time it passes through an ECU (sending or receiving a frame), it subtracts its value in credits from the total credits of the ECU. In this way it is possible to understand if the ECU is able to manage all the signals that cross it or not. The purpose of the credits is to simulate the complexity that requires the frame to be processed by the ecu. In this case they are assigned based on the length of the signal. Since it is not a simulation environment, we consider only the worst case, where all frames populate the network in the given time interval. The final value will be shown in percentage.

The metric is composed by a *metric context block* artifact where the ECUs to be interrogated are inserted, three *calculation block* artifacts to populate a first table with the useful data for the calculation, to perform the calculation and to create a last table for the visualization of the result. There are also 4 *parameter block* artifacts to set the operating parameters of the algorithm, which allow you to set the period to be interrogated, the frequency of sending default frames, the default weight of a frame that is used in case a frame does not have an assigned weight and the default credits for the ECU. There are two *report result block* artifacts, one to show the result in table format and the other for debugging.

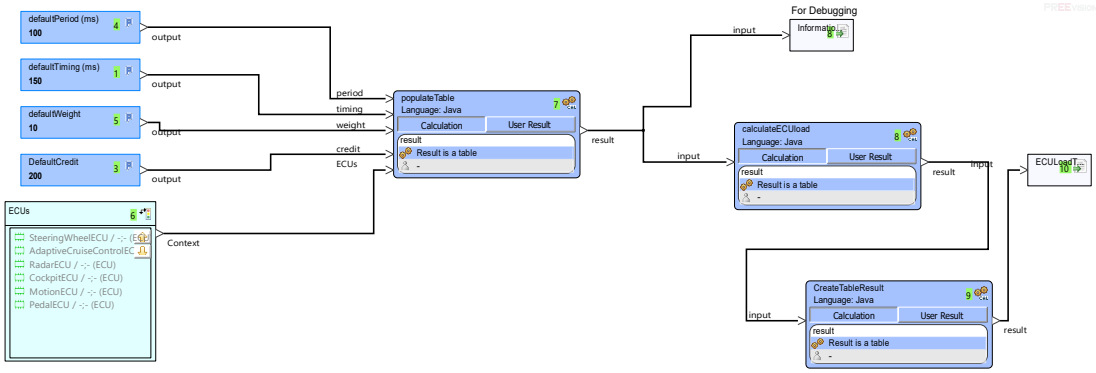


Figure 6.4: ECU load single core metric diagram

The credits of the ECU and the weight in credits of the signals are shown in the following two tables:

ECU credits	
ECU	**Credit**
AdaptiveCruiseControlECU	300
CockpitECU	400
MotionECU	600
PedalECU	100
RadarECU	200
SteeringWheelECU	200

Table 6.4: ECU credits table

Signal weights	
Signal	**Weight**
ACCCNL_Signal	5
ACCDecreaseDistance_Signal	5
ACCDecreaseSpeed_Signal	7
AcceleratorByPedal_Signal	15
AcceleratorTraction_Signal	15
ACCIncreaseDistance_Signal	5
ACCIncreaseSpeed_Signal	7
ACCOutputDecreaseSpeed_Signal	17
ACCOutputIncreaseSpeed_Signal	17
ACCStartState_Signal	5
ACCStatus_Signal	25

Signal weights	
Signal	**Weight**
BrakeByPedal_Signal	15
BrakeCalipersFrontLT_Signal	15
BrakeCalipersFrontRT_Signal	15
BrakeCalipersRearLT_Signal	15
BrakeCalipersRearRT_Signal	15
CockpitInfo_Signal	31
RadarInfo_Signal	20
RadarOutput_Signal	20
RPM_wheel_Left_Signal	27
RPM_wheel_Right_Signal	27
SpeedIndicator_Signal	20

Table 6.5: Signal weights table

```
1  void populateTable(MFrameTransmission frame, MECU ecu, MECUInterface ecuInt, String
       typeTransmission) {
2      frame.getContainedIPDUTransmissions().forEach(PDU -> {
3        PDU.getContainedSignalTransmissions().forEach(signal -> {
4          ITempTableRow row = TempTableFactory.createTableRow(table);
5          table.createTableCell(colECU, row, ecu.toString(), String.class);
6          table.createTableCell(colECUInterface, row, ecuInt.toString(), String.class);
7          Optional<MGenericAttribute> generic = ecu.getGenericAttributes().stream().filter(
       attr -> attr.getName().equals("credits")).findFirst();
8          if (generic.isPresent()) {
9            table.createTableCell(colECUCredits, row, Double.parseDouble(generic.get().
       getValue()), Double.class);
10         } else {
11           table.createTableCell(colECUCredits, row, defaultCredit, Double.class);
12         }
13         table.createTableCell(colSignal, row, signal.toString(), String.class);
14         table.createTableCell(colTypeTransmission, row, typeTransmission, String.class);
15         generic = signal.getGenericAttributes().stream().filter(attr -> attr.getName().
       equals("weight")).findFirst();
16         if (generic.isPresent()) {
17           table.createTableCell(colWeight, row, Integer.valueOf(generic.get().getValue()),
       int.class);
18         } else {
19           table.createTableCell(colWeight, row, defaultWeight, int.class);
20         }
21         double t=0;
22         if (PDU.getCyclicTimings().size() > 0) {
23           t = PDU.getCyclicTimings().get(0).getTimePeriod();
24         }else {
```

```
25            t = defaultTiming;
26          }
27          table.createTableCell(colTiming, row, t, Double.class);
28          if (t != 0) {
29            table.createTableCell(colNPeriod, row, (Double) (defaultPeriod / t), Double.class)
    ;
30          } else {
31            table.createTableCell(colNPeriod, row, (Double) 0.0, Double.class);
32          }
33        });
34      });
35    }
```

Listing 6.5: populateTable() method

The *populateTable* method is called for each signal sent or received by each ECU. Its purpose is to create an intermediate table containing all the information needed to perform the ECU load calculation. The necessary information are: ECUs, ECU interface, ECU credits, signals, type transmission (sending or receiving) , timing (signal sending period), signal weight, NPeriod (number of times the signal is sent or received in the predefined time interval).

```
1  public Object calculateResult() {
2      ITempTable table = getInput(IN_INPUT, ITempTable.class);
3      HashMap<String, Double> ECULoadMap = new HashMap<>();
4
5      double credit=0;
6      double creditTotal = 0;
7
8      double defaultPeriod = 0;
9      Object defaultPeriodInput = getInput("defaultPeriod");
10     if (defaultPeriodInput != null && defaultPeriodInput instanceof Double) {
11       defaultPeriod = ((Double) defaultPeriodInput).doubleValue();
12     }
13
14     String ECU = "";
15     for (int j = 0; j < table.getRowCount(); j++) {
16       double cost = 0;
17       for (int i = 0; i < table.getColumnCount(); i++) {
18         ITempTableCell cell = table.getCell(i, j);
19         if (cell.getColumnName().equals("ECU")) {
20           ECU = cell.getRepresentedObject().toString();
21         }
22         else if (cell.getColumnName().equals("ECUCredits")) {
23           ECULoadMap = populateECULoadMap(ECULoadMap, ECU, (double) cell.
    getRepresentedObject());
24           ECUCreditMap = populateECULoadMap(ECUCreditMap, ECU, (double) cell.
    getRepresentedObject());
```

```
25            creditTotal = (double) cell.getRepresentedObject();
26          }
27          else if (cell.getColumnName().equals("Weight")) {
28            cost = Double.parseDouble(cell.getRepresentedObject().toString());
29          } else if (cell.getColumnName().equals("NPeriod")) {
30            cost *= (double) cell.getRepresentedObject();
31            ECULoadMap.put(ECU, ECULoadMap.get(ECU) - cost);
32          }
33        }
34      }
35      ECULoadMap.forEach((k, v) -> {
36        ITempTableRow row = TempTableFactory.createTableRow(resultTable);
37        resultTable.createTableCell(colECU, row, k, String.class);
38        double result = ((ECUCreditMap.get(k) - v) / ECUCreditMap.get(k)) * 100;
39        result = result > 0 ? result : result * -1;
40        resultTable.createTableCell(colECULoad, row, round(result, 2) + "%", String.class);
41      });
42      setResult(OUT_RESULT, resultTable);
43          return null;
44      }
```

Listing 6.6: calculateResult() method

The *calculateResult()* method belonging to the *calculateECUload* artifact calculates for each entry of the previously created table, the number of credits for each signal expressed as the number of times the signal crosses an ECU, multiplied by its weight. The resulting value is subtracted from the credits of the affected ECU. Finally, for each ECU the load percentage is calculated and a table is created again which will be passed to another *calculation block* to perform the final visualization. The final result is the following:

ECU load single core	
ECU	**Load**
AdaptiveCruiseControlECU	65.67%
CockpitECU	18.57%
MotionECU	61.5%
PedalECU	20.0%
RadarECU	31.67%
SteeringWheelECU	11.33%

Table 6.6: ECU load table

6.4 ECU load dual core

The ECU load dual core metric estimates the load on the ECUs that populate the architecture, using the same algorithm as the ECU load single core metric, but assuming that all ECUs are dual core, and that frames are handled in the cores based on the frame id (The operation of the algorithm is explained in the previous section 6.3). On the basis of the filter on the frame id entered by the user, a frame is handled by one core rather than another.

The metric is composed by a *metric context block* artifact to insert the ECUs to interrogate, a *calculation block* artifact to populate a first table where are inserted all the useful data to perform the calculation, and a second *calculation block* artifact to perform the calculation of the ECUs load. There are also 5 *parameter block* artifacts to set the period of time to be interrogated, to define the frame sending frequency for all frames that do not have a defined cyclic sending frequency, to define the default weights and credits for frames and ECUs that do not have them defined, finally a last *parameter block* to set the frame id threshold beyond which the frame is managed by the second ECU core. To visualize the data there are two tables, a final one and an intermediate one, used for debugging purposes. In this case the frame id threshold is 100, so all frames with decimal frame id greater than 100 are handled by the second core.

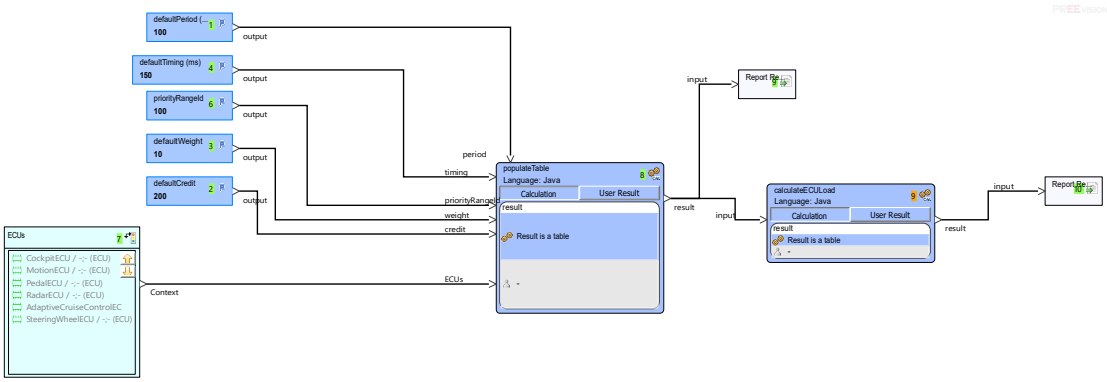


Figure 6.5: ECU load dual core metric diagram

Based on the created architecture, the result of the metric is as follows (all frame ids can be seen in the chapter 5.6):

ECU load dual core		
ECU	**Core 1**	**Core 2**
AdaptiveCruiseControl ECU	16.44 %	49.22 %
CockpitECU	7.5%	11.07 %
MotionECU	40.67 %	20.83 %
PedalECU	20.0%	0.0 %
RadarECU	31.67%	0.0 %
SteeringWheelECU	0.0 %	11.33%

Table 6.7: ECU load table

Chapter 7

Conclusion

In this final chapter, I discussed the approach used for the creation of the network architecture, by highlighting its positive and negative aspects. In this case, although the approach was strongly influenced by the PREEvision software, this chapter tries to discuss the importance of adopting an efficient strategy for the development of a complex system, thus avoiding a review of the software used. Instead, the last paragraph is used to introduce the future developments of this thesis.

The work of this thesis analyzes, verifies and produces an automotive network architecture following a methodology compatible with AUTOSAR. The model-based approach used by PREEvision allows to have well-defined graphical models for the definition of the architecture, thereby allowing not only the collaboration of several people in the same project, but also - and above all - a rapid design of a very complex system from scratch. The use of a model-based approach has become mandatory, especially given the complexity and the large number of ECUs present in a vehicle architecture. Although the creation of the architecture is done through graphical models, these models allow anyway the export of files and data needed for the ECU programming, speeding up the ECU development process, thanks to the definition of the communication and the functioning of the ECUs (more information about the ECU development are present in the next paragraph).

The design of the adaptive cruise control functionality has been deliberately simplified, in order to focus more on the aspects of the methodology to be followed for the design; therefore, although this created architecture cannot have an application in the real context, the initial goal is fully achieved, because it defines a step-based methodology which leads to the generation of the necessary files for the ECU development. The goal was to introduce and understand the approach to be followed, highlighting its positive and negative aspects. Although the PREEvision layers to be created for the ARXML file generation were only the software, hardware and communication layers, also the customer features, requirements and logic layers were very useful, since they allow to follow the v-cycle model that is used in the

design and development of automotive applications. The correct application of the v-cycle model surely allows to obtain a consistent architecture already at a first final version, without having to make big changes in the future.

In the project, the metrics created were useful in obtaining an estimate of how the architecture worked. Through the PREEvision software, a conceptual architecture is created in which true simulations cannot be done because it is not possible to influence the various sensors, creating situations that could occur. Despite this, being able to make static queries on the architecture created is very important because it allows the same to understand any inefficiencies before the development phase.

The work provides a complete architecture design, following the v-cycle process in figure 7.1. The first one called system design is practically completed and this allows to continue to the second step for the ECU development. In the figure only the software belonging to the Vector company are shown, but the fact of having ARXML files allows the use of any other tool, since being AUTOSAR specific files, the AUTOSAR consortium has many *Tools and Services* that can be used.

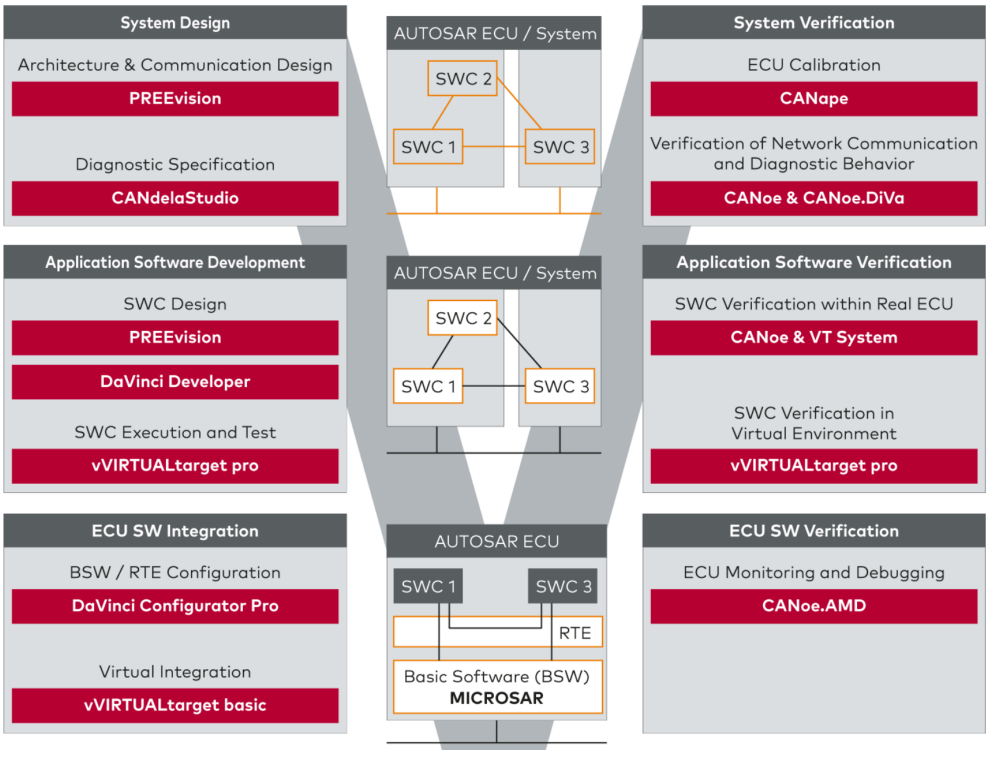


Figure 7.1: V-cycle for ECU development

Given the complexity of modern systems, the model-based approach is definitely a winning strategy compared to other approaches for designing a vehicle architecture.

This approach has no major disadvantages in the automotive domain, although it is not a methodology that allows to obtain a working system in a very short time. However, the time spent on model-based design allows for higher quality on the final product. Other approaches could obtain a final product in a shorter time but with a lower quality. Relevant positive aspects are:

- Communication between the team members who will develop the system is facilitated, allowing efficient analysis and verification of the system.
- Errors are easy to find. This benefits the production of the system because it perfectly delimits the system to be developed, without having to make any future changes.
- Models and their components are reusable.

7.1 Future work

In order to make the architecture created more complex and usable in a real-world context, the Ethernet protocol should be added to the design for managing the radar/cameras for autonomous driving. The sensors used for autonomous driving generate large amounts of data that must be processed in a very short time, and therefore cannot be sent through the simplest buses such as CAN or LIN. In order to have a highly performative communication, the Ethernet bus was recently introduced also in the automotive field. The creation of an ethernet network architecture is slightly more complex than the one seen in the previous chapters, because a service oriented design is used. The 7.2 shows the steps to follow for ethernet design in PREEvision. The final part is identical to the one seen for the architecture design of this thesis, while the initial part is about the definition and creation of services. Services are requested and received between ECUs. The ECUs communicate with each other through the SOME/IP protocol. Through this protocol, the communication takes place dynamically, following a request of the service and its response. In the ECUs, the service interfaces that are used to provide and receive services are defined. Thus, a future development is to add Ethernet to the architecture, delving into the SOME/IP protocol and service oriented architecture.

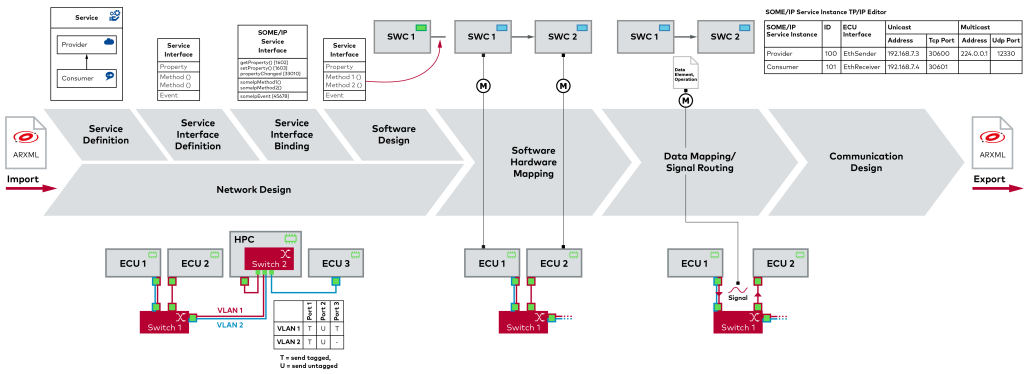


Figure 7.2: Ethernet design in PREEvision

Another future development for this thesis could be the programming of the ECUs, which stands as the second phase of the v-cycle shown in figure 7.1. Having

already defined the SW-Cs, the ports, the interfaces, the data and also the mapping between the SW-Cs and the ECUs, and having exported the ARXML file related to the whole system, we could start the development of the single ECUs. The basic functions of an AUTOSAR ECU such as communication and diagnostics are implemented by the Basic Software (BSW), while RTE handles data exchange and execution of the component software. Developing an AUTOSAR ECU therefore means implementing the BSW and RTE of each ECU. In order to do this, specific tools should be used, together with the generated ARXML files, because they allow the automatic generation of the files needed for the development.The files that will be generated can be for example, .c and .h files, with already present data that have been created in the design phase of the architecture.

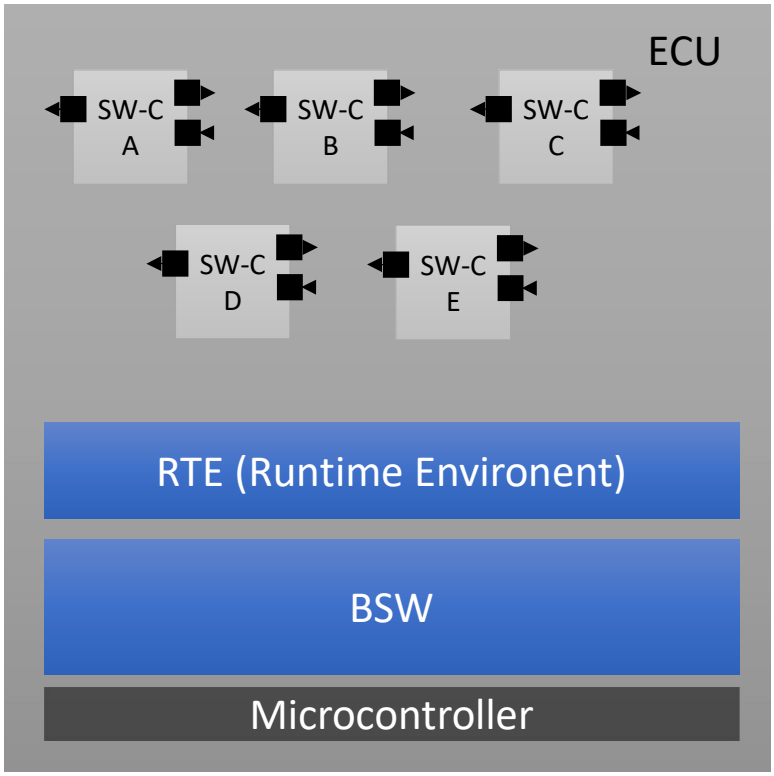


Figure 7.3: AUTOSAR ECU

Appendix A

Export CAN DBC

```
VERSION ""

NS_ :
    NS_DESC_
    CM_
    BA_DEF_
    BA_
    VAL_
    CAT_DEF_
    CAT_
    FILTER
    BA_DEF_DEF_
    EV_DATA_
    ENVVAR_DATA_
    SGTYPE_
    SGTYPE_VAL_
    BA_DEF_SGTYPE_
    BA_SGTYPE_
    SIG_TYPE_REF_
    VAL_TABLE_
    SIG_GROUP_
    SIG_VALTYPE_
    SIGTYPE_VALTYPE_
    BO_TX_BU_
    BA_DEF_REL_
    BA_REL_
    BA_DEF_DEF_REL_
    BU_SG_REL_
    BU_EV_REL_
    BU_BO_REL_
    SG_MUL_VAL_

BS_:
```

```
34
35 BU_: AdaptiveCruiseControlECU SteeringWheelECU
36
37
38 BO_ 109 ACC_Start_Stop_Frame: 1 SteeringWheelECU
39   SG_ ACCStartState_Signal : 0|1@1+ (1,0) [0|0] "" AdaptiveCruiseControlECU
40
41 BO_ 110 ACC_CNL_Frame: 1 SteeringWheelECU
42   SG_ ACCCNL_Signal : 0|1@1+ (1,0) [0|0] "" AdaptiveCruiseControlECU
43
44 BO_ 111 ACC_DecreaseSpeed_Frame: 1 SteeringWheelECU
45   SG_ ACCDecreaseSpeed_Signal : 0|2@1+ (1,0) [0|0] "" AdaptiveCruiseControlECU
46
47 BO_ 112 ACC_DecreaseDistance_Frame: 1 SteeringWheelECU
48   SG_ ACCDecreaseDistance_Signal : 0|1@1+ (1,0) [0|0] "" AdaptiveCruiseControlECU
49
50 BO_ 113 ACC_IncreaseSpeed_Frame: 1 SteeringWheelECU
51   SG_ ACCIncreaseSpeed_Signal : 0|2@1+ (1,0) [0|0] "" AdaptiveCruiseControlECU
52
53 BO_ 114 ACC_IncreaseDistance_Frame: 1 SteeringWheelECU
54   SG_ ACCIncreaseDistance_Signal : 0|1@1+ (1,0) [0|0] "" AdaptiveCruiseControlECU
55
56
57 VAL_ 112 ACCDecreaseDistance_Signal 1 "−10 m";
```

Listing A.1: CAN DBC file of SteeringWheel-ACC bus

Appendix B

Export ARXML file

```
1   ...
2   <APPLICATION-SW-COMPONENT-TYPE UUID="
    Obfa8380117e6793bc3e47a96XObfa8380117e6793bc3e47a9500">
3           <SHORT-NAME>AdaptiveCruiseControlFunction</SHORT-NAME>
4           <PORTS>
5             <P-PORT-PROTOTYPE UUID="
    Obfa8380117e6793bc3e47a96xObfa8380117e6793bc3e4a26300">
6               <SHORT-NAME>ACCOutputDecreaseSpeed</SHORT-NAME>
7             </P-PORT-PROTOTYPE>
8             <P-PORT-PROTOTYPE UUID="
    Obfa8380117e6793bc3e47a96xObfa8380117e6793bc3e4a31000">
9               <SHORT-NAME>ACCOutputIncreaseSpeed</SHORT-NAME>
10            </P-PORT-PROTOTYPE>
11            <P-PORT-PROTOTYPE UUID="
    Obfa8380117e6793bc3e47a96xObfa8380117e6793bc3e4943700">
12              <SHORT-NAME>ACCStatus</SHORT-NAME>
13            </P-PORT-PROTOTYPE>
14            <R-PORT-PROTOTYPE UUID="
    Obfa8380117e6793bc3e47a96xObfa8380117e6793bc3e47aa000">
15              <SHORT-NAME>ACCCNL</SHORT-NAME>
16            </R-PORT-PROTOTYPE>
17            <R-PORT-PROTOTYPE UUID="
    Obfa8380117e6793bc3e47a96xObfa8380117e6793bc3e482b200">
18              <SHORT-NAME>ACCDecreaseDistancePort</SHORT-NAME>
19            </R-PORT-PROTOTYPE>
20            <R-PORT-PROTOTYPE UUID="
    Obfa8380117e6793bc3e47a96xObfa8380117e6793bc3e4818600">
21              <SHORT-NAME>ACCDecreaseSpeedPort</SHORT-NAME>
22            </R-PORT-PROTOTYPE>
23            <R-PORT-PROTOTYPE UUID="
    Obfa8380117e6793bc3e47a96xObfa8380117e6793bc3e4821900">
24              <SHORT-NAME>ACCIncreaseDistancePort</SHORT-NAME>
25            </R-PORT-PROTOTYPE>
```

```
26              <R-PORT-PROTOTYPE UUID="
    Obfa8380117e6793bc3e47a96xObfa8380117e6793bc3e480f300">
27                <SHORT-NAME>ACCIncreaseSpeedPort</SHORT-NAME>
28              </R-PORT-PROTOTYPE>
29              <R-PORT-PROTOTYPE UUID="
    Obfa8380117e6793bc3e47a96xObfa8380117e6793bc3e4806300">
30                <SHORT-NAME>ACCStartState</SHORT-NAME>
31              </R-PORT-PROTOTYPE>
32              <R-PORT-PROTOTYPE UUID="
    Obfa8380117e6793bc3e47a96xObfa8380117e6793bc3e488c500">
33                <SHORT-NAME>RadarOutput</SHORT-NAME>
34              </R-PORT-PROTOTYPE>
35              <R-PORT-PROTOTYPE UUID="
    Obfa8380117e6793bc3e47a96xObfa8380117e6793bc3e4896d00">
36                <SHORT-NAME>SpeedIndicator</SHORT-NAME>
37              </R-PORT-PROTOTYPE>
38            </PORTS>
39          </APPLICATION-SW-COMPONENT-TYPE>
40        ...
41        <SENDER-RECEIVER-INTERFACE UUID="
    Obfa8380117cf9d1152e44fe7XObfa8380117cf9d1152e44fe600">
42            <SHORT-NAME>ACCCNL_SRI</SHORT-NAME>
43            <IS-SERVICE>false</IS-SERVICE>
44            <DATA-ELEMENTS>
45              <VARIABLE-DATA-PROTOTYPE UUID="
    Obfa8380117cf9d1152e4503cXObfa8380117cf9d1152e4503b00">
46                <SHORT-NAME>ACCCNL_DE</SHORT-NAME>
47                <TYPE-TREF DEST="APPLICATION-PRIMITIVE-DATA-TYPE">/
    DataTypes/ApplicationDataTypes/ACCCNL_ADT</TYPE-TREF>
48              </VARIABLE-DATA-PROTOTYPE>
49            </DATA-ELEMENTS>
50          </SENDER-RECEIVER-INTERFACE>
51          <SENDER-RECEIVER-INTERFACE UUID="
    Obfa8380117c73c6632d45806XObfa8380117c73c6632d4580500">
52            <SHORT-NAME>ACCDecreaseDistance_SRI</SHORT-NAME>
53            <IS-SERVICE>false</IS-SERVICE>
54            <DATA-ELEMENTS>
55              <VARIABLE-DATA-PROTOTYPE UUID="
    Obfa8380117c73c6632d45862XObfa8380117c73c6632d4586100">
56                <SHORT-NAME>ACCDecreaseDistance_DE</SHORT-NAME>
57                <TYPE-TREF DEST="APPLICATION-PRIMITIVE-DATA-TYPE">/
    DataTypes/ApplicationDataTypes/ACCDecreaseDistance_ADT</TYPE-TREF>
58              </VARIABLE-DATA-PROTOTYPE>
59            </DATA-ELEMENTS>
60          </SENDER-RECEIVER-INTERFACE>
61          <SENDER-RECEIVER-INTERFACE UUID="
    Obfa8380117c73c6632d440daXObfa8380117c73c6632d440d900">
62            <SHORT-NAME>ACCDecreaseSpeed_SRI</SHORT-NAME>
63            <IS-SERVICE>false</IS-SERVICE>
```

```
64              <DATA-ELEMENTS>
65                <VARIABLE-DATA-PROTOTYPE UUID="
    Obfa8380117c73c6632d44136XObfa8380117c73c6632d4413500">
66                  <SHORT-NAME>ACCDecreaseSpeed_DE</SHORT-NAME>
67                  <TYPE-TREF DEST="APPLICATION-PRIMITIVE-DATA-TYPE">/
    DataTypes/ApplicationDataTypes/ACCDecreaseSpeed_ADT</TYPE-TREF>
68                </VARIABLE-DATA-PROTOTYPE>
69              </DATA-ELEMENTS>
70            </SENDER-RECEIVER-INTERFACE>
71            <SENDER-RECEIVER-INTERFACE UUID="
    Obfa8380117c73c6632d45280XObfa8380117c73c6632d4527f00">
72              <SHORT-NAME>ACCIncreaseDistance_SRI</SHORT-NAME>
73              <IS-SERVICE>false</IS-SERVICE>
74              <DATA-ELEMENTS>
75                <VARIABLE-DATA-PROTOTYPE UUID="
    Obfa8380117c73c6632d452dcXObfa8380117c73c6632d452db00">
76                  <SHORT-NAME>ACCIncreaseDistance_DE</SHORT-NAME>
77                  <TYPE-TREF DEST="APPLICATION-PRIMITIVE-DATA-TYPE">/
    DataTypes/ApplicationDataTypes/ACCIncreaseDistance_ADT</TYPE-TREF>
78                </VARIABLE-DATA-PROTOTYPE>
79              </DATA-ELEMENTS>
80            </SENDER-RECEIVER-INTERFACE>
81            <SENDER-RECEIVER-INTERFACE UUID="
    Obfa8380117c73c6632d4394eXObfa8380117c73c6632d4394d00">
82              <SHORT-NAME>ACCIncreaseSpeed_SRI</SHORT-NAME>
83              <IS-SERVICE>false</IS-SERVICE>
84              <DATA-ELEMENTS>
85                <VARIABLE-DATA-PROTOTYPE UUID="
    Obfa8380117c73c6632d439a2XObfa8380117c73c6632d439a100">
86                  <SHORT-NAME>ACCIncreaseSpeed_DE</SHORT-NAME>
87                  <TYPE-TREF DEST="APPLICATION-PRIMITIVE-DATA-TYPE">/
    DataTypes/ApplicationDataTypes/ACCIncreaseSpeed_ADT</TYPE-TREF>
88                </VARIABLE-DATA-PROTOTYPE>
89              </DATA-ELEMENTS>
90            </SENDER-RECEIVER-INTERFACE>
91              ...
```

Listing B.1: A part of a system configuration arxml file

Bibliography

[1] *An Evolution Of Car Technology*. URL: https://mashable.com/archive/car-tech-ces (cit. on p. 1).

[2] Andr Hergenhan and Gernot Heiser. «Operating Systems Technology for Converged ECUs». In: (Jan. 2008) (cit. on p. 1).

[3] Shugang Jiang. «Vehicle E/E Architecture and Its Adaptation to New Technical Trends». In: 08 (Jan. 2019), p. 10 (cit. on p. 4).

[4] *Electrical/Electronic architecture is evolving toward a centralized setup*. URL: https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/automotive-software-and-electrical-electronic-architecture-implications-for-oems (cit. on p. 5).

[5] *Automotive E/E Architectures Are Key To Continued Innovation*. URL: https://semiengineering.com/automotive-e-e-architectures-are-key-to-continued-innovation/ (cit. on p. 5).

[6] *AUTOSAR*. URL: https://www.autosar.org (cit. on p. 6).

[7] Massimo Violante. *Model-Based Software Design Course*. 2019 (cit. on pp. 8, 9).

[8] G. L. Gopu, K. V. Kavitha, and James Joy. «Service Oriented Architecture based connectivity of automotive ECUs». In: *2016 International Conference on Circuit, Power and Computing Technologies (ICCPCT)*. 2016, pp. 1–4. DOI: 10.1109/ICCPCT.2016.7530358 (cit. on p. 11).

[9] Marcelino Varas. «Service-Oriented Software Architectures: Bridging the Gap Between AUTOSAR Classic and Adaptive Systems». In: (Nov. 2019). URL: https://cdn.vector.com/cms/content/know-how/_technical-articles/PREEvision/PREEvision_AUTOSAR_Integration_SOA_ElektronikAutomotive_PressArticle_201911_EN.pdf (cit. on p. 12).

[10] Marcelino Varas. *Service-Oriented Software Architectures and Ethernet design*. URL: https://cdn.vector.com/cms/content/events/2019/vAES19/vAES19_03_Varas_Vector.pdf (cit. on p. 12).

[11] Vector Informatik GmbH. *PREEvision manual*. Vector Informatik GmbH. Ingersheimer Straße 24 70499 Stuttgart, Germany (cit. on pp. 18, 24–26, 77).

[12] Jörg Schäuffele. *E/E architectural design and optimization using preevision*. Tech. rep. SAE Technical Paper, 2016 (cit. on pp. 18, 19).

[13] Steve Corrigan. «Introduction to the controller area network-texas instruments». In: *Tech. Rep. SLOA101, Texas Instruments* (2016) (cit. on pp. 27, 28, 32).

[14] *CAN DBC File Explained - A Simple Intro [+Editor Playground]*. URL: https://www.csselectronics.com/pages/can-dbc-file-database-intro (cit. on pp. 33–35).

[15] *SENSORE ABS*. URL: https://www.hella.com/techworld/it/Tecnica/Sensori-e-attuatori/Sensore-ABS-4074/ (cit. on p. 61).

[16] Samir Fassak, Younes El Hajjaji El Idrissi, Noureddine Zahid, and Mohamed Jedra. «A secure protocol for session keys establishment between ECUs in the CAN bus». In: *2017 International Conference on Wireless Networks and Mobile Communications (WINCOM)*. 2017, pp. 1–6. DOI: 10.1109/WINCOM.2017.8238149 (cit. on p. 80).

[17] *CAN bit stuffing*. URL: https://en.wikipedia.org/wiki/CAN_bus#Bit_stuffing (cit. on p. 85).

[18] Wikipedia contributors. *Leaky bucket — Wikipedia, The Free Encyclopedia*. [Online; accessed 22-February-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Leaky_bucket&oldid=1070590064 (cit. on p. 88).